

Берtrand Мейер

**Объектно-ориентированное конструирование
программных систем**

Переводчики: Владимир Биллиг, М. Дехтерь, Н. Супонев, Е. Павлова

Редактор: Владимир Биллиг

Языки: Русский

Издательство: Русская Редакция

ISBN 5-7502-0255-0, 0-13-62155-4; 2005 г.

Книга взята с intuit.ru, где она лежит в виде двух курсов: [Основы объектно-ориентированного программирования](#) и [Основы объектно-ориентированного проектирования](#)

Для конвертации были использованы wget, iPython, LibreOffice Writer, wkhtmltopdf.

Оглавление

Оглавление	2
Внешние и внутренние факторы	30
Обзор внешних факторов	30
Корректность (Correctness)	30
Устойчивость (Robustness)	31
Расширяемость (Extendibility)	32
Повторное использование (Reusability)	32
Совместимость (Compatibility)	33
Эффективность (Efficiency)	33
Переносимость (Portability)	35
Простота использования (Easy of Use)	35
Функциональность (Functionality)	35
Своевременность (Timeliness)	37
Другие качества	37
О документации	37
Компромиссы	38
Ключевые вопросы	38
О программном сопровождении	39
Ключевые концепции	40
О критериях	41
До какой степени мы должны быть догматичными?	41
Категории	41
Метод и язык	41
Бесшовность (seamlessness)	41
Классы	42
Утверждения (Assertions)	42
Классы как модули	42
Классы как типы	42
Вычисления, основанные на компонентах	42
Скрытие информации (information hiding)	43
Обработка исключений (Exception handling)	43
Статическая типизация (static typing)	43
Универсальность (genericity)	44
Единичное наследование (single inheritance)	44
Множественное наследование (Multiple inheritance)	44
Дублируемое наследование (Repeated inheritance)	44
Ограниченнная универсальность (Constrained genericity)	45
Переопределение (redefinition)	45
Полиморфизм	45
Динамическое связывание	46
Выяснение типа объекта в период выполнения	46
Отложенные (deferred) свойства и классы	46
Управление памятью (memory management) и сборка мусора (garbage collection)	47
Реализация и среда	47
Автоматическое обновление (automatic update)	47
Быстрое обновление (fast update)	47
Живучесть (persistence)	48
Документация	48

Быстрый просмотр (browsing)	48
Библиотеки	48
Базовые библиотеки	48
Графика и пользовательские интерфейсы	49
Механизмы эволюции библиотек	49
Механизмы индексации в библиотеках	49
Продолжение просмотра	49
Библиографические ссылки и объектные ресурсы	49
Пять критериев	51
Декомпозиция	51
Модульная Композиция	53
Модульная Понятность	54
Модульная Непрерывность	54
Модульная Защищенность	55
Пять правил	56
Прямое отображение	56
Минимум интерфейсов	57
Слабая связность интерфейсов	57
Явные интерфейсы	58
Скрытие информации	59
Пять принципов	61
Лингвистические Модульные Единицы	61
Самодокументирование	62
Унифицированный Доступ	63
Открыт-Закрыт	64
Единственный Выбор	67
Ключевые концепции	69
Библиографические замечания	69
Упражнения	69
УЗ.1 Модульность в языках программирования	69
УЗ.2 Принцип Открыт-Закрыт (для программистов Lisp)	70
УЗ.3 Ограничения на скрытие информации	70
УЗ.4 Метрики для модульности (отчетная исследовательская работа)	70
УЗ.5 Модульность существующих систем	70
УЗ.6 Управление конфигурацией и наследование	70
Цели повторного использования	72
Ожидаемые преимущества	72
Потребители и производители повторно используемых программ	73
Что следует повторно использовать?	73
Повторное использование персонала	73
Повторное использование проектов и спецификаций	73
Образцы проектов (design patterns)	74
Повторное использование исходного текста	75
Повторное использование абстрактных модулей	75
Повторяемость при разработке ПО	76
Нетехнические препятствия	76
Синдром NIH	76
Фирмы по разработке ПО и их стратегии	77
Организация доступа к компонентам	78
Несколько слов об индексировании компонентов	78

Форматы для распространения повторно используемых компонентов	79
Оценка	80
Техническая проблема	80
Изменения и постоянство	80
Повторно использовать или переделать? (The reuse-redo dilemma)	81
Пять требований к модульным структурам	82
Изменчивость Типов (Type Variation)	82
Группирование Подпрограмм (Routine Grouping)	82
Изменчивость Реализаций (Implementation Variation)	82
Независимость Представлений	83
Факторизация Общего Поведения	83
Традиционные модульные структуры	86
Подпрограммы	86
Пакеты	87
Пакеты: оценка	88
Перегрузка и универсальность	89
Синтаксическая перегрузка	89
Семантическая перегрузка (предварительное представление)	90
Универсальность (genericity)	91
Основные методы модульности: оценка	92
Ключевые концепции	92
Библиографические замечания	93
Ингредиенты вычисления	95
Базисный треугольник	95
Функциональная декомпозиция	96
Непрерывность	96
Проектирование сверху вниз	97
Не только одна главная функция	98
Обнаружение вершины	99
Функции и эволюция	100
Интерфейсы и проектирование ПО	100
Преждевременное упорядочение	101
Упорядочивание и ОО-разработка	101
Возможность повторного использования	102
Производство и описание	103
Проектирование сверху вниз: общая оценка	103
Декомпозиция, основанная на объектах	103
Расширяемость	104
Возможность повторного использования	104
Совместимость	104
Объектно-ориентированное конструирование ПО	104
Вопросы	105
Выявление типов объектов	105
Описания типов и объектов	106
Описание отношений и структурирование ПО	106
Ключевые концепции	106
Библиографические замечания	107
Критерии	108
Различные реализации	108
Представления стеков	109

Опасность излишней спецификации	110
Какова длина второго имени?	111
К абстрактному взгляду на объекты	111
Использование операций	111
Политика невмешательства в обществе модулей	112
Согласованность имен	112
Можно ли обойтись без абстракций?	113
Формализация спецификаций	113
Спецификация типов	114
Универсализация (Genericity)	114
Перечисление функций	115
Категории функций	117
Раздел АКСИОМЫ	117
Две или три вещи, которые мы знаем о стеках	118
Частичные функции	119
Предусловия	119
Полная спецификация	120
Ничего кроме правды	120
От абстрактных типов данных к классам	122
Классы	122
Как создавать эффективный класс	122
Роль отложенных классов	123
Абстрактные типы данных и скрытие информации	123
Переход к более императивной точке зрения	124
Назад к тому, с чего начали?	125
Конструирование объектно-ориентированного ПО	125
За пределами программ	125
Дополнительные темы	126
Еще раз о неявности	126
Соотношение спецификации и проектирования	127
Соотношение классов и записей	128
Альтернативы частичным функциям	128
Полна ли моя спецификация?	129
Доказательство достаточной полноты	131
Ключевые концепции	134
Библиографические замечания	134
Упражнения	134
У6.1 Точки	134
У6.2 Боксеры	135
У6.3 Банковские счета	135
У6.4 Сообщения	135
У6.5 Имена	135
У6.6 Текст	135
У6.7 Покупка дома	135
У6.8 Дополнительные операции для стеков	135
У6.9 Ограниченные стеки	135
У6.10 Очереди	135
У6.11 Распределители	136
У6.12 Булевский -- BOOLEAN	136
У6.13 Достаточная полнота	136
У6.14 Непротиворечивость	136

Классы, а не объекты - предмет обсуждения	137
Устранение традиционной путаницы	137
Роль классов	138
Модули и типы	138
Класс как модуль и как тип	138
Унифицированная система типов	138
Простой класс	139
Компоненты	139
Атрибуты и подпрограммы	140
Унифицированный доступ	142
Класс POINT	142
Основные соглашения	143
Распознавание вида компонент	143
Тело подпрограммы и комментарии к заголовку	144
Предложение indexing	144
Обозначение результата функции	144
Правила стиля	145
Наследование функциональных возможностей общего характера	145
Объектно-ориентированный стиль вычислений	146
Текущий экземпляр	146
Клиенты и поставщики	146
Вызов компонента	147
Принцип единственности цели	148
Слияние понятий модуль и тип	148
Роль объекта Current	148
Квалифицированные и неквалифицированные вызовы	149
Компоненты-операции	149
Селективный экспорт и скрытие информации	152
Неограниченный доступ	152
Ограничение доступа клиентам	152
Стиль объявления скрытых компонент	153
"Внутренний" экспорт	153
Собираем все вместе	154
Общая относительность	154
Большой Взрыв	154
Системы	155
Программа main отсутствует	155
Компоновка системы	156
Классическое "Hello"	158
Структура и порядок: программист в роли поджигателя	158
Обсуждение	159
Форма объявлений	159
Атрибуты или функции?	159
Экспорт атрибутов	160
Доступ клиентов к атрибутам	161
Оптимизация вызовов	162
Архитектурная роль селективного экспорта	162
Импорт листингов	163
Присваивание функции результата	163
Дополнение: точное определение сущности	165
Ключевые концепции	165

Библиографические замечания	166
Упражнения	166
У7.1 POINT как абстрактный тип данных	166
У7.2 Завершение реализации POINT	166
У7.3 Полярные координаты	166
Объекты	167
Что такое объект?	167
Базовая форма	167
Простые поля	168
Простое представление книги - класс BOOK	168
Писатели	169
Ссылки	170
Идентичность объектов	171
Объявление ссылок	172
Ссылка на себя	172
Взгляд на структуру объектов периода выполнения	173
Объекты как средство моделирования	174
Четыре мира программной разработки	174
Реальность: "седьмая вода на киселе"	175
Работа с объектами и ссылками	176
Динамическое создание и повторное связывание	176
Инструкция создания	176
Общая картина	177
Для чего необходимо явное создание объектов?	178
Процедуры создания	178
Перекрытие инициализации по умолчанию	179
Статус экспорта процедур создания	180
Правила, применимые к процедурам создания	180
Процедуры создания и перегрузка	180
Еще о ссылках	181
Состояния ссылок	181
Вызовы и пустые ссылки	181
Операции над ссылками	182
Присоединение ссылки к объекту	182
Сравнение ссылок	184
Значение void	184
Клонирование и сравнение объектов	185
Копирование объектов	186
Глубокое клонирование и сравнение	187
Глубокое хранилище: первый взгляд на сохраняемость	189
Составные объекты и развернутые типы	191
Ссылок не достаточно	191
Развернутые типы	192
Роль развернутых типов	193
Агрегирование	194
Свойства развернутых типов	195
Недопустимость ссылок на подобъекты	195
Присоединение: две семантики - ссылок и значений	196
Присоединение	197
Присоединение: ссылочное и копии	197
Гибридное присоединение	198

Проверка эквивалентности	198
Работа со ссылками: преимущества и опасности	199
Динамические псевдонимы	199
Семантика использования псевдонимов	199
Выработка соглашений для динамических псевдонимов	200
Псевдонимы в ПО и за его пределами	200
Инкапсуляция действий со ссылками	201
Обсуждение	202
Графические соглашения	202
Ссылки и простые значения	203
Форма операций клонирования и эквивалентности	204
Статус универсальных операций	205
Ключевые концепции	205
Библиографические замечания	205
Упражнения	206
У8.1 Книги и авторы	206
У8.2 Личности	206
У8.3 Проектирование нотации	206
Что происходит с объектами	207
Создание объектов	207
Использование динамического режима	209
Повторное использование памяти в трех режимах	209
Отсоединение	209
Недостижимые объекты	210
Достижимые объекты в классическом подходе	211
Достижимые объекты в ОО-модели	213
Проблема управления памятью в ОО-модели	215
Три ответа	215
Несерьезный подход (тривиальный)	216
Может ли быть оправдан несерьезный подход?	216
Надо ли заботиться о памяти?	216
Байт здесь, байт там, и реальные покойники	217
Восстановление памяти: проблемы	217
Удаление объектов, управляемое программистом	218
Проблема надежности	218
Проблема простоты разработки	218
Подход на уровне компонентов	219
Управление памятью связного списка	220
Работа с утилизированными объектами	221
Дискуссия	222
Автоматическое управление памятью	223
Необходимость автоматических методов	223
Что в точности понимается под восстановлением?	223
Подсчет ссылок	223
Сборка мусора	225
Механизм сборки мусора	225
Основа сборки мусора	226
Сборка по принципу "все-или-ничего"	226
Продвинутый (Advanced) подход к сборке мусора	227
Алгоритмы параллельной сборки мусора	227

Практические проблемы сборки мусора	228
Класс MEMORY	228
Механизм освобождения	228
Сборка мусора и внешние вызовы	229
Среда с управлением памятью	229
Основы	229
Сложные проблемы	230
Перемещение объектов	230
Механизм сборки мусора	230
Повышенное чувство голода и потеря аппетита (Bulimia and anorexia)	230
Операции сборщика мусора	231
Ключевые концепции	231
Библиографические заметки	231
Упражнения	232
У9.1 Модели создания объектов	232
У9.2 Какой уровень утилизации?	232
У9.3 Совместное использование стека достижимых элементов	232
У9.4 Совместное использование	232
Горизонтальное и вертикальное обобщение типа	233
Необходимость параметризованных классов	233
Родовые АТД	234
Проблема	234
Роль типизации	234
Родовые классы	235
Обявление родового класса	235
Использование родового класса	236
Терминология	236
Проверка типов	236
Правило типизации	236
Операции над сущностями родового типа	237
Типы и классы	238
Массивы	238
Массивы как объекты	238
Свойства массива	239
Размышления об эффективности	239
Синонимичная инфиксная операция	239
Стоимость универсализации	240
Обсуждение: что все-таки не сделано	240
Ключевые концепции	241
Библиографические замечания	241
Упражнения	241
У10.1 Ограниченная универсализация	241
У10.2 Двумерные массивы	241
У10.3 Использование своего формального родового параметра фактически как чужого	241
Базисные механизмы надежности	242
О корректности ПО	242
Выражение спецификаций	243
Формула корректности	243
Сильные и слабые условия	244

Введение утверждений в программные тексты	245
Предусловия и постусловия	245
Класс стек	245
Предусловия	246
Постусловия	247
Педагогическое замечание	247
Контракты и надежность ПО	247
Права и обязательства	247
Интуиция (Дзен) и искусство программной надежности: больше гарантий и меньше проверок	248
Утверждения не являются механизмом проверки вводимых данных	249
Утверждения это не управляющие структуры	250
Ошибки, дефекты и другие насекомые	250
Работа с утверждениями	251
Класс стек	251
Императив и аппликатив (применимость)	253
Замечание о пустоте структур	254
Проектирование предусловий: толерантное или требовательное?	254
Предусловия и статус экспорта	256
Толерантные модули	257
Инварианты класса	260
Определение и пример	260
Форма и свойства инвариантов класса	261
Инвариант в момент изменения	262
Кто должен обеспечить сохранность инвариантов	262
Роль инвариантов класса в программной инженерии	262
Инварианты и контракты	263
Когда класс корректен?	263
Корректность класса	264
Роль процедур создания	264
Ревизия массивов	265
Связывание с АТД	266
Не просто коллекция функций	266
Компоненты класса и АТД функции	266
Выражение аксиом	266
Функция абстракции	267
Инварианты реализации	267
Инструкция утверждения	269
Инварианты и варианты цикла	271
Трудности циклов	271
Сделаем циклы корректными	272
Ингредиенты доказательства корректности цикла	273
Синтаксис цикла	275
Использование утверждений	277
Утверждения как средство для написания корректного ПО	277
Использование утверждений для документирования: краткая форма класса	277
Мониторинг утверждений в период выполнения	279
Каков оптимальный уровень мониторинга?	280
Обсуждение	282
Нужен ли мониторинг в период выполнения?	282
Выразительная сила утверждений	283

Включение функций в утверждения	284
Инварианты класса и семантика ссылок	285
Что дальше	288
Ключевые концепции	288
Библиографические замечания	288
Упражнения	289
У11.1 Комплексные числа	289
У11.2 Класс и его АТД	289
У11.3 Полные утверждения для стеков	289
У11.4 Экспортирование размера	289
У11.5 Инвариант реализации	289
У11.6 Утверждения и экспорт	289
У11.7 Поиск жучков (bugs)	290
У11.8 Нарушение инварианта	290
У11.9 Генерация случайных чисел	290
У11.10 Модуль "очередь"	290
У11.11 Модуль "множество"	290
Постскриптуm: Катастрофа Ариан 5	290
Базисные концепции обработки исключений	292
Отказы	292
Исключения	292
Источники исключений	292
Ситуации отказа	293
Обработка исключений	293
Как не следует делать это - C-Unix пример	293
Как не следует делать это - Ada пример	294
Принципы обработки исключений	295
Цепочка вызовов	296
Механизм исключений	296
Спаси и Повтори (Rescue и Retry)	296
Как отказаться сразу	297
Таблица истории исключений	298
Примеры обработки исключений	299
Поломки при вводе	299
Восстановление при исключениях, сгенерированных операционной системой	300
Повторение программы, толерантной к неисправностям	301
N-версионное программирование	301
Задача предложения rescue	302
Корректность предложения rescue	302
Четкое разделение ролей	304
Когда нет предложения rescue	304
Продвинутая обработка исключений	304
Запросы при работе с классом EXCEPTIONS	305
Какой должна быть степень контроля?	306
Исключения разработчика	307
Обсуждение	307
Дисциплинированные исключения	307
Должны ли исключения быть объектами?	308
Методологическая перспектива	308
Ключевые концепции	308

Библиографические замечания	309
Упражнения	309
У12.1 Наибольшее целое	309
У12.2 Объект Exception	309
Взаимодействие с не объектным ПО	310
Внешние программы	310
Улучшенные варианты	310
Использование внешних программ	311
ОО-изменение архитектуры (re-architecturing)	311
Вопрос совместимости: гибридный программный продукт или гибридные языки?	312
Передача аргументов	313
Инструкции	315
Вызов процедуры	315
Присваивание (Assignment)	315
Создание (Creation)	316
Условная Инструкция (Conditional)	316
Множественный выбор	317
Циклы	318
Проверка	318
Отладка	318
Повторение вычислений	319
Выражения	319
Манифестные константы	319
Вызовы функций	319
Текущий объект	319
Выражения с операторами	320
Нестрогие булевые операторы	320
Строки	321
Ввод и вывод	322
Лексические соглашения	322
Ключевые концепции	323
Упражнения	323
У13.1 Внешние классы	323
У13.2 Избегая нестрогих операторов	323
Многоугольники и прямоугольники	324
Многоугольники	324
Прямоугольники	325
Основные соглашения и терминология	327
Наследование инварианта	328
Наследование и конструкторы	328
Пример иерархии	329
Полиморфизм	331
Полиморфное присоединение	331
Что на самом деле происходит при полиморфном присоединении?	331
Полиморфные структуры данных	332
Типизация при наследовании	334
Согласованность типов	334
Пределы полиморфизма	335
Экземпляры	336
Статический тип, динамический тип	336

Обоснованы ли ограничения?	337
Может ли быть польза от неведения?	337
Когда хочется задать тип принудительно	338
Полиморфное создание	338
Динамическое связывание	339
Использование правильного варианта	339
Переопределение и утверждения	340
О реализации динамического связывания	340
Отложенные компоненты и классы	341
Движения произвольных фигур	341
Отложенный компонент	342
Эффективизация компонента	342
Отложенные классы	343
Соглашения о графических обозначениях	344
Что делать с отложенными классами?	344
Задание семантики отложенных компонентов и классов	345
Способы изменения объявлений	347
Повторное объявление функции как атрибута	347
Обратного пути нет	348
Использование исходной версии при переопределении	348
Смысл наследования	349
Двойственная перспектива	349
Взгляд на класс как на модуль	350
Взгляд на класс как на тип	351
Наследование и децентрализация	352
Независимость от представления	353
Парадокс расширения-специализации	353
Роль отложенных классов	353
Назад к абстрактным типам данных	354
Отложенные классы как частичные интерпретации: классы поведения	355
Не вызывайте нас, мы вызовем вас	356
Программы с дырами	357
Роль отложенных классов при анализе и глобальном проектировании	358
Обсуждение	358
Явное переопределение	358
Доступ к предшественнику процедуры	358
Динамическое связывание и эффективность	359
Оценка накладных расходов	360
Статическое связывание как оптимизация	360
Кнопка под другим именем: когда статическое связывание ошибочно	361
Подход языка C++ к связыванию	363
Ключевые концепции	364
Библиографические замечания	365
Упражнения	365
У14.1 Многоугольники и прямоугольники	365
У14.2 Многоугольник с малым числом вершин	365
У14.3 Геометрические объекты с двумя координатами	365
У14.4 Наследование без классов	365
У14.5 Классы без объектов	365
У14.6 Отложенные классы и прототип	365
У14.7 Библиотека поиска в таблицах (семестровый проект)	366

У14.8 Виды отложенных компонентов	366
У14.9 Комплексные числа	366
Примеры множественного наследования	367
Пример, неподходящий для введения	367
Может ли самолет быть имуществом?	368
Числовые и сравнимые значения	369
Окна - это деревья и прямоугольники	370
Деревья - это списки и их элементы	371
Составные фигуры	372
Брак по расчету	375
Структурное наследование	376
Наследование функциональных возможностей	377
Лунка и кнопка	377
Оценка	378
Переименование компонентов	378
Конфликт имен	378
Результат переименования	379
Смена имен и переопределение	380
Подбор локальных имен	381
Играем в имена	381
Использование родительской процедуры создания	381
Плоские структуры	382
Плоская форма класса	382
Применение плоской формы	383
Краткая плоская форма	383
Дублируемое наследование	383
Общие предки	384
По обе стороны океана	384
Совместное использование и репликация	385
Ненавязчивое дублирующее наследование	387
Правило переименования	388
Конфликт переопределений	389
Конфликт при совместном использовании: отмена определения и соединение компонентов	
Конфликты при репликации: выделение	391
Выделение всех компонентов	390
Сохранение исходной версии при переопределении	392
Пример повышенной сложности	393
Дублируемое наследование и универсальность	394
Правила об именах	397
Обсуждение	398
Переименование	398
ОО-разработка и перегрузка	398
Ключевые концепции	399
Библиографические замечания	399
Упражнения	400
У15.1 Окна как деревья	400
У15.2 Является ли окно строкой?	400
У15.3 Завершение строительства	400
У15.4 Итераторы фигур	400
У15.5 Связанные стеки	400
У15.6 Кольцевые списки и цепи	400

У15.7 Деревья	400
У15.8 Каскадные или "шагающие" (walking) меню	400
У15.9 Плоский precursor (предшественник)	401
У15.10 Дублируемое наследование и репликация	401
Наследование и утверждения	402
Инварианты	402
Предусловия и постусловия при наличии динамического связывания	402
Как обмануть клиентов	404
Как быть честным	404
Пример	405
Устранение посредника	406
Субподряды	407
Абстрактные предусловия	407
Правило языка	408
Повторное объявление функции как атрибута	409
Замечание математического характера	409
Глобальная структура наследования	410
Универсальные классы	410
Нижняя часть иерархии	411
Универсальные компоненты	412
Замороженные компоненты	412
Запрет повторного объявления	412
Фиксированная семантика компонентов copy, clone и equality	412
Не злоупотребляйте замораживанием	413
Ограниченнная универсальность	414
Вектора, допускающие сложение	414
Не ОО-подход	415
Ограничение родового параметра	416
Игра в рекурсию	417
И снова неограниченная универсальность	417
Попытка присваивания	417
Когда правила типов становятся несносными	417
Проблема	418
Механизм решения	419
Правильное использование попытки присваивания	420
Типизация и повторное объявление	420
Устройства и принтеры	420
Одно- и двусвязные элементы	421
Правило повторного объявления типов	423
Закрепленные объявления	423
Несогласованность типов	423
Примеры из практики	424
Серьезное затруднение	425
Понятие опорного элемента	425
Опорный элемент Current	426
Еще раз о базовых классах	426
Правила о закрепленных типах	426
Когда не используются закрепленные объявления	427
Статический механизм	427
Наследование и скрытие информации	428
Кое-что о политике	428

Применение	428
Зачем нужна такая гибкость?	429
Интерфейс и повторное использование реализаций	430
Слово в защиту реализаций	430
Два стиля	430
Выборочный экспорт	431
Ключевые концепции	431
Библиографические замечания	431
Упражнения	431
У16.1 Наследование: простота и эффективность	431
У16.2 Векторы	432
У16.3 Экстракт?	432
Проблема типизации	433
Базисная конструкция	433
Статическая и динамическая типизация	433
Правила типизации	434
Реализм	434
Пессимизм	434
Статическая типизация: как и почему	435
Преимущества	435
Аргументы в пользу динамической типизации	436
Типизация: слагаемые успеха	436
"Типизирована ли кроха"?	437
Типизация и связывание	437
Ковариантность и скрытие потомком	439
Ковариантность	439
Параллельные иерархии	441
Своенравие полиморфизма	442
Скрытие потомком	443
Корректность систем и классов	443
Практический аспект	443
Корректность систем: первое приближение	444
Контравариантность и безвариантность	444
Использование родовых параметров	444
Типовые переменные	444
Полагаясь на закрепление типов	445
Глобальный анализ	447
Остерегайтесь полиморфных кэтколлов!	448
Назад, в Ялту	448
Одно правило и несколько определений	449
Оценка	450
Полное соответствие	450
Ключевые концепции	451
Библиографические замечания	451
Константы базовых типов	453
Атрибуты-константы	453
Использование констант	453
Константы пользовательских классов	454
Константы с манифестом для этого непригодны	454

Однократные функции	455
Применение однократных подпрограмм	456
Разделяемые объекты	456
Однократные функции с результатами базовых типов	457
Однократные процедуры	457
Параметры	458
Однократные функции, закрепление и универсальность	458
Константы строковых типов	459
Unique-значения	460
Обсуждение	461
Инициализация: подходы языков программирования	461
Строковые константы	462
Unique-значения и перечислимые типы	462
Ключевые концепции	463
Библиографические замечания	464
Упражнения	464
У18.1 Эмуляция перечислимых типов однократными функциями	464
У18.2 Однократные функции для эмуляции unique-значений	464
У18.3 Однократные функции в родовых классах	464
У18.4 Однократные атрибуты?	464
О методологии	465
Методология: что и почему	465
Как создавать хорошие правила: советы советчикам	466
Необходимость методологических руководств	466
Теория	466
Практика	466
Повторное использование	466
Типология правил	467
Абсолютная положительность	467
Абсолютная отрицательность	467
Рекомендации	467
Исключения	468
Абстракция и точность	469
Если это необычно, зафиксируй это	469
Об использовании метафор	470
Как важно быть скромным	471
Библиографические замечания	472
Упражнения	472
У1.1 Самоприменение правил	472
У1.2 Библиотека правил	472
У1.3 Применение правил	472
У1.4 Метафоры в сети	472
Многопанельные системы	473
Первая напрашивающаяся попытка	474
Функциональное решение: проектирование сверху вниз	475
Функция переходов	475
Архитектура программы	476
Критика решения	477
Статичность	477

Объектно-ориентированная архитектура	478
Закон инверсии	478
Состояние как класс	479
Наследование и отложенные классы	480
Описание полной системы	481
Класс приложения	483
Обсуждение	485
Библиографические замечания	485
Проделки дьявола	486
Откаты для пользы и для забавы	486
Многоуровневый откат и повтор: undo и redo	486
Практические проблемы	486
Требования к решению	487
Поиск абстракций	488
Класс Command	488
Основной интерактивный шаг	489
Сохранение последней команды	490
Действия системы	490
Как создается объект command	491
Многоуровневый откат и повтор: UNDO-REDO	491
Список истории	491
Реализация Undo	492
Реализация Redo	493
Выполнение обычных команд	493
Аспекты реализации	493
Аргументы команды	493
Предвычисленные командные объекты	494
Представление списка истории	495
Интерфейс пользователя для откатов и повторов	496
Обсуждение	497
Роль реализации	498
Небольшие классы	498
Библиографические замечания	499
Упражнения	499
У3.1 Небольшая интерактивная система (программистский проект)	499
У3.2 Многоуровневый Redo	499
У3.3 Undo-redo в Pascal	499
У3.4 Undo, Skip и Redo	499
У3.5 Сохранение командных объектов	499
У3.6 Составные команды	500
У3.7 Необратимые команды	500
У3.8 Библиотека команд (проектирование и реализация)	500
У3.9 Механизм истории	500
У3.10 Тестирование окружения	500
У3.11 Интегрируемые функции	500
Изучение документа "технические требования"	501
Существительные и глаголы	501
Как избежать бесполезных классов	501
Нужен ли новый класс?	502
Пропуск важных классов	503
Обнаружение и селекция	504
Сигналы опасности	504

Большое Заблуждение	505
Мой класс выполняет...	505
Императивные имена	505
Однопрограммные классы	506
Преждевременная классификация	506
Классы без команд	506
Смешение абстракций	507
Идеальный класс	508
Общие эвристики для поиска классов	508
Категории классов	508
Внешние объекты: нахождение классов анализа	508
Нахождение классов реализации	509
Отложенные классы реализации	510
Нахождение классов проектирования	510
Другие источники классов	510
Предыдущие разработки	510
Адаптация через наследование	510
Оценивание кандидатов декомпозиции	511
Находки других подходов	511
Файлы	511
Использование ситуаций	512
КОС (CRC) карты	513
Повторное использование	513
Подход снизу вверх	513
Сказка о поиске классов	513
Метод получения классов	514
Ключевые концепции	515
Библиографические замечания	515
Упражнения	516
У4.1 Floors как integers	516
У4.2 Инспектирование объектов	516
Побочные эффекты в функциях	517
Команды и запросы	517
Формы побочного эффекта	517
Ссыпочная прозрачность	518
Объекты как машины	519
Функции, создающие объекты	520
Чистый стиль для интерфейса класса	520
Генераторы псевдослучайных чисел: упражнение	521
Абстрактное состояние, конкретное состояние	522
Стратегия	523
Возражения	523
Законные побочные эффекты: пример	524
Много ли аргументов должно быть у компонента?	527
Важность числа аргументов	527
Операнды и необязательные параметры (опции)	528
Принцип	528
Преимущества, обеспечиваемые Принципом Операндов	529
Исключения из Принципа Операндов?	529
Контрольный перечень	530
Размер класса: Подход списка требований	530

Определение размера класса	530
Поддержка согласованности	531
Запреты и послабления	532
Активные структуры данных	532
Представление связного списка	532
Пассивные классы	533
Инкапсуляция и утверждения	536
Критика интерфейса класса	536
Простые, напрашивающиеся решения	536
Введение состояния	537
Поддержка согласованности: инвариант реализации	538
С точки зрения клиента	540
Взгляд изнутри	540
АТД и абстрактные машины	545
Отделение состояния	545
Слияние списка и стражей	545
Выборочный экспорт	548
Как справляться с особыми ситуациями	548
Априорная схема	549
Препятствия на пути априорной схемы	549
Апостериорная схема	550
Роль механизма исключений	550
Эволюция классов. Устаревшие классы	551
Документирование класса и системы	552
Показ интерфейса	552
Документирование на уровне системы	552
Ключевые концепции	553
Библиографические замечания	554
Упражнения	554
У5.1 Функция с побочным эффектом	554
У5.2 Операнды и опции	554
У5.3 Возможные аргументы	554
У5.4 Число элементов как функция	554
У5.5 Поиск в связных списках	554
У5.6 Теоремы в инварианте	554
У5.7 Двунаправленные списки	554
У5.8 Альтернативный проект связного списка	554
У5.9 Вставка в связный список	555
У5.10 Циклические списки	555
У5.11 Функции ввода, свободные от побочных эффектов	555
У5.12 Документация	555
У5.13 Самодокументированное ПО	555
Как не следует использовать наследование	556
Покупать или наследовать	558
Иметь и быть (To have and to be)	558
Правило изменений	559
Правило полиморфизма	561
Резюме	561
Приложение: техника описателей	561
Таксомания	563
Использование наследования: таксономия таксономии	564
Область действия правил	564

Ошибочное использование	565
Общая таксономия	565
Наследование подтипов	567
Наследование с ограничением	567
Наследование с расширением	567
Подходящая математическая модель	568
Наследование вариаций	569
Отмена эффективизации	569
Наследование с конкретизацией	570
Структурное наследование	570
Наследование реализации	571
Льготное наследование	571
Использование наследования с отложенными и эффективными классами	571
Один механизм, или несколько?	572
Наследование подтипов и скрытие потомков	573
Определение подтипа	573
Различные взгляды	573
Взгляд на подтипы	573
Необходимость скрытия потомком	574
Как избежать скрытия потомком	574
Приложения скрытия потомком	575
Таксономии и их ограничения	576
Использование скрытия потомком	577
Наследование реализации	577
Брак по расчету	577
Это выглядит привлекательно, но правильно ли это?	578
Как это делается без наследования	578
Льготное наследование	579
Использование кодов символов	579
Итераторы	580
Формы льготного наследования	581
Понимание льготного наследования	581
Множественные критерии и наследование видов	582
Классификация при множественных критериях	582
Наследование вида	583
Подходит ли нам наследование видов?	584
Критерии для наследования видов	585
Как разрабатываются структуры наследования	587
Специализация и абстракция	587
Произвольность классификации	588
Индукция и дедукция	588
Разнообразие абстракции	588
Независимость клиента	589
Совершенствование уровня абстракции	589
Итоговый обзор: используйте наследование правильно	589
Ключевые концепции	590
Библиографические замечания	590
Приложение: история таксономии	590
Упражнения	591
У6.1 Стек, основанный на массиве	591
У6.2 Метатаксономия	591

У6.3 Стеки Ханоя	591
У6.4 Являются ли многоугольники списками?	591
У6.5 Наследование функциональной вариации	591
У6.6 Примеры классификации	591
У6.7 Кому принадлежат итераторы?	591
У6.8 Наследование типа и модуля	591
У6.9 Наследование и полиморфизм	592
Философия проектирования	593
Общая схема разработки	593
Структура систем	593
Эволюция системы	593
Классы	593
Структура класса	593
Документация класса	593
Индексируйте классы.	594
Использование утверждений	594
Как обращаться со специальными ситуациями	594
Техника наследования	595
Повторные объявления	595
Отложенные классы	595
Полиморфизм	595
Формы наследования	595
Дела косметические!	596
Применение правил на практике	596
Кратко и явно	596
Роль соглашений	597
Самоприменение	597
Дисциплина и творчество	597
Выбор правильных имен	598
Общие правила	598
Локальные сущности и аргументы подпрограмм	599
Регистр	599
Грамматические категории	599
Стандартные имена	600
Преимущества согласованного именования	601
Использование констант	601
Манифестные и символические константы	601
Где размещать объявления констант	602
Заголовочные комментарии и предложения индексации	602
Комментарии в заголовках: упражнение на сокращение	602
Заголовочные комментарии предложений feature	604
Предложения индексирования	605
Не заголовочные комментарии	605
Форматирование и презентация текста	606
Форматирование	606
Высота и ширина	608
Детали отступов	608
Пробелы	609
Приоритеты и скобки	610
Война вокруг точек с запятыми	610
Утверждения	611
Шрифты	612

Основные правила	612
Другие соглашения	612
Цвет	613
Библиографические замечания	613
Упражнения	613
У8.1 Стиль заголовочных комментариев	613
У8.2 Неоднозначность точки с запятой	613
Цели анализа	614
Задачи	614
Требования	614
Облака и провалы	615
Изменчивая природа анализа	615
Вклад объектной технологии	615
Программирование телевизионного вещания	616
Графики вещания	616
Сегменты	617
Программы и реклама	618
Деловой регламент	619
Оценка	619
Представление анализа: разные способы	619
Методы анализа	622
Нотация BON (Business Object Notation)	623
Библиографические замечания	625
Кластеры	626
Параллельная разработка	626
Этапы и задачи	628
Кластерная модель жизненного цикла ПО	628
Обобщение	630
Бесшовность и обратимость	631
Бесшовная разработка	631
Обратимость: мудрость иногда расцветает слишком поздно	632
У нас все - лицо	633
Ключевые концепции	633
Библиографические замечания	633
Профессиональная подготовка (тренинг) в индустрии	635
Вводные курсы	636
Филогенез и онтогенез	636
Вымощенная дорога к другим подходам	636
Выбор языка	637
Другие курсы	638
Терминология	638
Среднее и высшее образование	639
Курсы для аспирантов	639
Полный учебный план (curriculum)	639
Вперед к новой педагогике для программистов	639
Стратегия "от потребителя к производителю"	640
Абстракция	640
Ученичество	641

Обращенный учебный план	641
Политика многих семестров	641
Объектно-ориентированный план	642
Ключевые концепции	643
Библиографические замечания	643
Предварительный просмотр	644
Возникновение параллельности	644
Мультипроцессорная обработка	645
Многозадачность	645
Посредники запросов объектов (брокеры объектных запросов - Object Request Broker)	645
Удаленное выполнение	646
От процессов к объектам	646
Сходство	647
Активные объекты	647
Конфликт активных объектов и наследования	648
Программируемые процессы	648
Введение параллельного выполнения	650
Процессоры	650
Природа процессоров	651
Операции с объектом	651
Дуальная семантика вызовов	652
Сепаратные сущности	652
Получение сепаратных объектов	653
Объекты здесь и там	654
Параллельная архитектура	654
Распределение процессоров: файл управления параллелизмом (Concurrency Control File)	
Библиотечные механизмы	655
Правила обоснования корректности: разоблачение предателей	656
Импорт структур объекта	658
Вопросы синхронизации	658
Синхронизация versus взаимодействия	658
Механизмы, основанные на синхронизации	658
Механизмы, основанные на взаимодействии	659
Синхронизация параллельных ОО-вычислений	660
Доступ к сепаратным объектам	661
Параллельный доступ к объекту	661
Резервирование объекта	662
Доступ к сепаратным объектам	663
Ожидание по необходимости	664
Мультипускатель	664
Оптимизация	665
Устранение блокировок (тупиков)	665
Условия ожидания	665
Буфер - это сепаратная очередь	666
Предусловия при параллельном выполнении	667
Парадокс предусловий	668
Параллельная семантика предусловий	668
Последовательные и параллельные утверждения	669
Ограничение проверки правильности	669
Состояния и переходы	670
Запросы специальных услуг	670

Экспресс сообщения	670
Дуэли и их семантика	671
Обработка исключений: алгоритм "Секретарь-регистратор"	671
О том, что будет дальше в этой лекции	672
Примеры	672
Обедающие философы	673
Полное использование параллелизма оборудования	675
Замки	676
Сопрограммы (Coroutines)	677
Система управления лифтом	679
Сторожевой механизм	681
Организация доступа к буферам	682
О правилах доказательств	683
Резюме параллельного механизма	684
Синтаксис	684
Ограничения	684
Семантика	685
Библиотечные механизмы	685
Обсуждение	686
Минимальность механизма	686
Полное использование наследования и других ОО-методов	686
Совместимость с Проектированием по Контракту	686
Поддержка различия между командами и запросами	686
Применимость ко многим видам параллельности	686
Адаптируемость с помощью библиотек	687
Поддержка программирования сопрограмм	687
Поддержка использования непараллельного ПО	687
Поддержка устранения блокировок	687
Допускается ли одновременный доступ?	687
Ключевые концепции	688
Библиографические замечания	688
Упражнения	689
У12.1 Принтеры	689
У12.2 Почему импорт должен быть глубоким	689
У12.3 "Аномалия наследования"	689
У12.4 Устранение тупиков (проблема для исследования)	689
У12.5 Приоритеты	689
У12.6 Файлы и парадокс предусловия	689
У12.7 Замки (Locking)	690
У12.8 Бинарные семафоры	690
У12.9 Целочисленные семафоры	690
У12.10 Контроллер сопрограмм	690
У12.11 Примеры сопрограмм	690
У12.12 Лифты	690
У12.13 Сторожа и принцип визитной карточки	690
У12.14 Однократные подпрограммы и параллельность	690
Сохраняемость средствами языка	691
Сохранение и извлечение структур объектов	691
Форматы сохранения	691
Вне рамок замыкания сохраняемости	692
Эволюция схемы	693
Наивные подходы	693
Преобразование объектов на лету	694
Выявление	694

Извещение	695
Исправление	695
От сохраняемости к базам данных	697
Объектно-реляционное взаимодействие	698
Определения	698
Операции	698
Запросы	698
Использование реляционных баз данных с ОО-ПО	699
Основания ОО-баз данных	699
На чем застопорились реляционные БД	699
Идентичность объектов	700
Пороговая модель	701
Дополнительные возможности	701
Версии объекта	702
Версии классов и эволюция схемы	702
Длинные транзакции	702
Блокировка	702
Запросы	702
ОО-СУБД: примеры	703
Matisse	703
Versant	703
Обсуждение: за пределами ОО-баз данных	704
Является ли "ОО-база данных" оксюмороном?	704
Неструктурированная информация	706
Ключевые концепции	706
Библиографические замечания	706
Упражнения	707
У13.1 Динамическая эволюция схем	707
У13.2 Объектно-ориентированные запросы	707
Необходимые средства	708
Конечные пользователи, разработчики приложений и разработчики инструментальных средств	708
Графические системы, оконные системы, инструментальные средства	708
Библиотека и конструктор приложений	709
Применение ОО-подхода	709
Переносимость и адаптация к платформе	709
Графические абстракции	711
Фигуры (изображения)	711
Координаты	712
Операции над окнами	713
Графические классы и операции	713
Механизмы взаимодействия	713
События	713
Контексты и объекты интерфейса пользователя	714
Обработка событий	714
Команды	714
Базисная схема	714
Состояния	714
Приложения	716
Контекст-Событие-Команда-Состояние: резюме	717

Математическая модель	717
Библиографические замечания	717
Немного контекста	718
Пакеты	719
Реализация стеков	719
Простой интерфейс	719
Использование пакета	720
Реализация	721
Универсальность	721
Скрытие представления: частная история	722
Исключения	723
Упрощение управляющей структуры	723
Возбуждение и обработка исключений	723
Обсуждение	724
Задачи	725
От Ada 83 к Ada 95	726
ОО-механизмы языка Ada 95: пример	726
Ada 95 и объектная технология: оценка	727
Обсуждение: наследование модулей и типов	728
Вперед к ОО-языку Ada	728
Ключевые концепции	729
Библиографические замечания	729
Упражнения	729
У15.1 Как выиграть, не используя скрытия	729
У15.2 Родовые параметры подпрограммы	729
У15.3 Классы как задачи (для программистов Ada)	729
У15.4 Добавление классов к Ada	730
У15.5 Пакеты-классы	730
Уровни языковой поддержки	731
ОО-программирование на языке Pascal?	731
Собственно Pascal	731
Модульные расширения языка Pascal	732
ОО-расширения языка Pascal	732
Fortran	732
Немного контекста	732
Техника COMMON	733
Техника подпрограммы с множественным входом	733
ОО-программирование и язык С	735
Немного контекста	735
Основные положения	735
Эмуляция объектов	736
Эмулирующие классы	737
ОО С: оценка	738
Библиографические замечания	738
Упражнения	738
У16.1 Графические объекты (для программистов на Fortran)	738
У16.2 Универсальность (для программистов на С)	738
У16.3 ОО-программирование на С (семестровый проект)	738
Simula	740
Основные понятия	740

Доступность	740
Основные черты языка	740
Пример	742
Концепции сопрограмм	742
Пример сопрограммы	743
Последовательное выполнение и наследование	744
Моделирование	745
Пример моделирования	747
Simula: оценка	747
Smalltalk	748
Языковой стиль	748
Сообщения	748
Окружение и производительность	749
Smalltalk: оценка	750
Расширения Lisp	750
Расширения С	750
Objective-C	751
C++	751
Сложность	752
C++: оценка	752
Java	753
Другие ОО-языки	754
Библиографические замечания	754
Simula	754
Smalltalk	754
Расширения С: Objective-C, C++	754
Расширения Lisp	754
Java	755
Другие языки	755
Упражнения	755
У17.1 Остановимся на коротких файлах	755
У17.2 Неявный вызов	755
У17.3 Эмулирующие сопрограммы	755
У17.4 Моделирование	755
У17.5 Ссылка на версию предка	755
Компоненты среды	756
Язык	756
Развитие	756
Открытость	756
Технология компиляции	756
Требования к компиляции	756
Технология тающего льда	757
Анализ зависимостей	757
Предкомпиляция	757
Удаленное выполнение	758
Оптимизация	758
Инструментальные средства	758
Bench и процесс разработки	758
Инструментальные средства высокого уровня	759
Библиотеки	761
Реализация интерфейса	762

Платформы	762
Инструментальные средства	762
Перенастройка и просмотр	765
Библиографические замечания	768

Основы объектно-ориентированного программирования

1. Лекция: Качество ПО

Качество - это цель инженерной деятельности; построение качественного ПО (software) - цель программной инженерии (software engineering). В данной книге рассматриваются средства и технические приемы, позволяющие значительно улучшить качество ПО. Прежде чем приступить к изучению этих средств и приемов, следует хорошо представлять нашу цель. Качество ПО лучше всего описывается комбинацией ряда факторов. В этой лекции мы постараемся проанализировать некоторые из них, покажем, где необходимы улучшения, и укажем дорогу в дальнейшем путешествии по лекциям этого курса.

Внешние и внутренние факторы

Все мы хотим, чтобы наше ПО было быстродействующим, надежным, легким в использовании, читаемым, модульным, структурным и т.д. Но эти определения описывают два разных типа качества. Наличие или отсутствие таких качеств, как скорость и простота использования ПО, может быть обнаружено его пользователями. Эти качества можно назвать **внешними** факторами качества.

Под словом "пользователи" нужно понимать не только людей, взаимодействующих с конечным продуктом, но и тех, кто их закупает, занимается администрированием. Такое свойство, например, как легкость адаптации продуктов к изменениям спецификаций - далее определенная в нашей дискуссии как расширяемость - попадает в категорию внешних факторов, поскольку она может представлять интерес для администраторов, закупающих продукт, хотя и не важна для "конечных пользователей", непосредственно работающих с продуктом.

Такие характеристики ПО, как модульность или читаемость, являются **внутренними** факторами, понятными только для профессионалов, имеющих доступ к тексту ПО.

В конечном счете, только внешние факторы имеют значение. Но ключ к достижению внешних факторов спрятан во внутренних факторах: для того, чтобы достичь видимого качества, проектировщики и конструкторы должны иметь внутренние приемы, позволяющие улучшать скрытые от пользователя качества.

Последующие лекции представляют описание набора современных технических средств достижения внутреннего качества. Однако за частностями не следует терять из вида общую картину; внутренние технические приемы не являются самоцелью - они лишь средство достижения внешних качеств нашего продукта.

Обзор внешних факторов

Рассмотрим самые важные внешние факторы качества, стремление к которым есть центральная задача ОО-построения ПО.

Корректность (Correctness)

Определение: корректность

Корректность - это способность ПО выполнять точные задачи так, как они определены их спецификацией.

Корректность является важнейшим качеством. Если система не делает того, что она должна делать, то все остальное - ее быстродействие, хороший пользовательский интерфейс - не имеет особого значения.

Но легче сказать, чем сделать. Даже первый шаг к корректности уже труден: необходимо в точной форме специфицировать технические требования к системе, что само по себе является тяжелой задачей.

Методы обеспечения корректности обычно **условны**. Серьезная система ПО, даже небольшая по нынешним меркам, использует столь многое, что невозможно гарантировать ее корректность, работая со всеми компонентами на одном уровне. Необходим многоуровневый подход:



Рис. 1.1. Слои в разработке ПО

В условном подходе к корректности мы заботимся только о том, чтобы обеспечить корректность каждого уровня, основываясь на **предположении**, что нижележащие уровни корректны. Это единственно реалистичный подход, поскольку он позволяет разделить проблему и на каждой ступени сконцентрироваться на ограниченном круге задач. Нельзя проверить, что программа на языке высокого уровня корректна, если не предположить, что используемый компилятор корректно реализует язык. Это не слепое доверие компилятору, а разделение проблемы на две: проверка корректности компилятора и проверка корректности программы относительно семантики языка.

В методе, описанном в нашей книге, слоев даже больше: разработка ПО будет основываться на библиотеках компонентов повторного использования, используемых во многих приложениях.

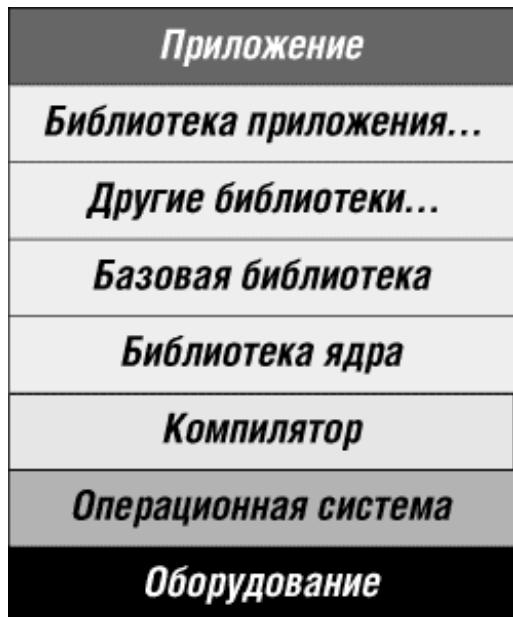


Рис. 1.2. Уровни в процессе разработки, включающем повторное использование

Здесь также применим условный подход: следует обеспечить корректность библиотек и корректность приложения при условии, что библиотеки корректны.

Многие практики полагают, что достижение корректности ПО связано с тестированием и исправлением ошибок. Мы же более амбициозны: в дальнейших лекциях исследуется ряд технических приемов, в частности типизация и метод утверждений, направленных на построение ПО, корректного с самого начала. Исправление ошибок и тестирование, конечно, остаются необходимыми как средства дополнительной проверки результата. Можно было бы пойти дальше и принять совсем формальный подход к построению ПО. Это не является целью наших лекций, как ясно из нескольких "робких" терминов - "проверять", "гарантировать", "обеспечивать", используемых выше вместо слова "доказывать". Все же многие из описанных ниже технических приемов происходят непосредственно от математических методов формальной спецификации и верификации программ, проходя длинный путь к обеспечению идеала корректности.

Устойчивость (Robustness)

Определение: устойчивость

Устойчивость - это способность ПО соответствующим образом реагировать на аварийные ситуации.

Устойчивость дополняет корректность. Корректность относится к поведению системы в случаях, определенных спецификацией; устойчивость характеризует то, что происходит за пределами этой спецификации.



Рис. 1.3. Устойчивость против корректности

Как видно из определения, устойчивость по своей природе более нечеткое понятие, чем корректность. Невозможно сказать, как в случае с корректностью, что в аварийных ситуациях система должна "выполнять свои задачи", поскольку ситуации выходят за пределы спецификации. Если бы эти задачи были известны, аварийный случай стал бы частью спецификации, и мы бы снова вернулись в область корректности.

Это определение "аварийной ситуации" нам еще понадобится при изучении обработки исключений (Об исключительных ситуациях см. [лекция 12](#)). Оно подразумевает, что понятия нормальной и аварийной ситуации всегда относительны по отношению к заданной спецификации; ситуация аварийна, если она выходит за рамки спецификации. Если расширить спецификацию, аварийные случаи становятся нормальными - даже если они соответствуют таким нежелательным событиям, как, например, ошибочный ввод пользователя.

Термин "нормальный" в этом смысле не означает "желательный", а просто "запланированный в проекте ПО". Хотя на первый взгляд может показаться парадоксальным, что ошибочный ввод может называться нормальным случаем, любой другой подход опирается на субъективные критерии и, таким образом, бесполезен.

Всегда будут существовать случаи, на которые спецификация явно не распространяется. Роль требования устойчивости - удостовериться, что и в таких случаях система не приводит к непоправимой ситуации; она должна выдать соответствующее сообщение об ошибке, гладко завершить работу или войти в так называемый режим "постепенного вывода из работы".

Расширяемость (Extendibility)

Определение: расширяемость

Расширяемость - это легкость адаптации ПО к изменениям спецификации.

Предполагается, что ПО должно быть **гибким (soft)**, и в принципе, оно такое и есть; ничего нет проще, чем изменить программу, если у вас есть доступ к ее исходному коду. Просто используйте свой любимый текстовый редактор.

Проблема расширяемости это проблема масштаба. Для маленьких программ изменение не является обычно большой проблемой, но по мере увеличения ПО адаптация становится все труднее. Большая программная система часто видится как огромный карточный дом, удаление одного элемента может привести к разрушению всего построения.

Нам нужна расширяемость, поскольку в основе ПО лежит человеческий феномен, склонный к изменчивости. Даже в научных расчетах, где можно ожидать, что законы физики неизменны, наше понимание этих законов и их моделирование будет изменяться.

Традиционные подходы к построению ПО не уделяли должного внимания изменениям. Они скорее исходили из идеального взгляда на жизненный цикл ПО, где требования замораживаются после завершения первоначальной ступени анализа. Последующий процесс посвящался проектированию и построению решения при фиксированных требованиях. Это вполне понятно: на том этапе развития дисциплины задача состояла в разработке надежных технических приемов для постановки и решения фиксированных проблем. Но сейчас стало возможным признать и рассмотреть центральный вопрос - что делать, если проблема изменяется в ходе ее решения. Изменения характерны для процесса разработки ПО: меняются требования, наше понимание требований, алгоритмы, представление данных, приемы реализации. Поддержка изменений является основной целью объектной технологии и постоянной темой нашей книги.

Хотя многие из технических приемов, улучшающих расширяемость, можно объяснить во вводных курсах и на небольших примерах, их значимость становится явной только для больших проектов. Для улучшения расширяемости важны два принципа:

- **Простота построения:** простая архитектура легче адаптируется к изменениям, чем сложная.
- **Децентрализация:** чем более автономны модули, тем выше вероятность того, что простое изменение затронет только один или небольшое количество модулей и не вызовет цепную реакцию изменений во всей системе.

ОО-метод - это, прежде всего, метод создания архитектуры системы, позволяющий проектировщику производить системы с простой и децентрализованной структурой даже для больших систем. Простота и децентрализация будут в следующих лекциях постоянными темами обсуждений, ведущих к ОО-принципам.

Повторное использование (Reusability)

Определение: повторное использование

Повторное использование есть способность элементов ПО служить для построения многих различных приложений.

Необходимость и возможность повторного использования возникает из наблюдений сходства систем - системы ПО часто имеют похожую схему. Следует использовать это сходство и не изобретать велосипед заново. Понимание этой схемы даст возможность повторно применять созданный элемент ПО во многих других разработках.

Повторное использование влияет на все остальные аспекты качества ПО. Поскольку решение проблемы повторного использования в сущности означает, что нужно писать меньше программ, следовательно, можно прилагать больше усилий (при той же общей стоимости) к улучшению других факторов, таких как, например, корректность и устойчивость.

При создании индустрии ПО необходимость повторного использования становится насущной проблемой.

Повторное использование будет играть важную роль в обсуждениях в последующих лекциях, одна из которых ([лекция 4](#)) фактически полностью посвящена углубленному рассмотрению этого фактора качества, его конкретной пользе и связанным с ним возникающим проблемам.

Совместимость (Compatibility)

Определение: совместимость

Совместимость - это легкость сочетания одних элементов ПО с другими.

Совместимость важна, поскольку мы не разрабатываем элементы ПО в вакууме: им необходимо взаимодействовать друг с другом. Но при этом слишком часто возникают проблемы, поскольку суждения разных элементов об остальном мире противоречивы. Простейшим примером может служить широкое разнообразие несовместимых файловых форматов, из-за чего, например, одна программа не может непосредственно использовать результат работы другой программы.

Ключ к совместимости находится в однородности построения и в стандартных соглашениях на коммуникации между программами. Эти подходы включают:

- Стандартные форматы файлов, как в системе Unix, где каждый текстовый файл - это просто последовательность символов.
- Стандартные структуры данных, как в системе Lisp, где все данные, а также программы, представлены бинарными деревьями (называемыми списками).
- Стандартные пользовательские интерфейсы, как в различных версиях Windows, OS/2 и MacOS, где все инструменты опираются на единую парадигму для коммуникации с пользователем, основанную на стандартных компонентах, таких как окна, значки, меню и т. д.

Большая общность достигается при определении стандартных протоколов доступа ко всем важным элементам, управляемым программами. Такова идея, лежащая в основе абстрактных типов данных и ОО-подхода, а также так называемого связующего программного обеспечения (middleware), например CORBA и Microsoft's OLE-COM (ActiveX).

Эффективность (Efficiency)

Определение: эффективность

Эффективность - это способность ПО как можно меньше зависеть от ресурсов оборудования: процессорного времени, пространства, занимаемого во внутренней и внешней памяти, пропускной способности, используемой в устройствах связи.

Почти синонимом эффективности является слово "производительность" (**performance**). В программистском сообществе есть два типичных отношения к эффективности:

- Некоторые разработчики одержимы проблемами производительности, что заставляет их прилагать много усилий к предполагаемой оптимизации.
- Существует общая тенденция недооценки вопросов эффективности, вытекающая из справедливых убеждений, существующих в промышленности: "сделай правильно, прежде чем сделать быстро" и "модель компьютера будущего года все равно будет на 50% быстрее".

Часто один и тот же человек в разное время выказывает разные типы отношения и является то доктором Abstract, то мистером Microsecond - происходит раздвоение личности, как в известной истории про доктора Джекила и мистера Хайда.

Где же истина? Разработчики часто явно излишне заботятся о микрооптимизации. Как уже отмечалось,

эффективность не дорогое стоит, если ПО некорректно. Можно привести новое изречение: "не беспокойтесь о быстродействии ПО, если оно к тому же и неверно". Забота об эффективности должна сопоставляться с другими целями, такими как расширяемость и возможность повторного использования. Оптимизация может сделать ПО настолько специализированным, что оно не будет годно для повторного использования и в случаях изменения спецификации. Более того, постоянно растущая мощь компьютерного оборудования позволяет нам слегка расслабиться и не стараться использовать последний байт или микросекунду.

Все это, однако, не умаляет важности эффективности. Никому не нравится, когда приходится ждать ответа от интерактивной системы или покупать дополнительную память для работы программы. Поэтому необдуманное отношение к производительности неприемлемо. Если конечная система медленно работает или громоздка, то начинают жаловаться и те, кто заявлял, что "скорость не так уж важна".

В этом вопросе отражается то, что я считаю главной характеристикой создания ПО. Построение ПО трудно именно потому, что оно требует принятия во внимание многих различных требований, часть из которых, например корректность, абстрактны и концептуальны, в то время как другие, например эффективность, конкретны и связаны со свойствами компьютерного оборудования.

Некоторые ученые считают разработку ПО отраслью математики, для некоторых инженеров - это отрасль прикладной технологии. На самом деле это и то, и другое. Разработчик ПО должен соединить абстрактные понятия с их конкретными реализациями, математику корректных вычислений с временными и пространственными ограничениями, возникающими из физических законов и ограниченности оборудования. Необходимость ублажать и ангелов, и чудищ - центральная проблема создания ПО.

Постоянное увеличение компьютерной мощи, каким бы оно ни было впечатляющим, не может заменить эффективность, по крайней мере, по трем причинам:

- Тот, кто покупает больший и более быстрый компьютер, хочет видеть действительные выгоды от дополнительной мощности - решать новые задачи, более быстро работать со старыми задачами, решать более важные версии старых задач за то же время. Если новый компьютер решает старые задачи за то же самое время - это нехорошо!
- Явный эффект повышения мощности компьютера сказывается тогда, когда велика доля "хороших" алгоритмов по отношению к плохим. Предположим, что новая машина работает в два раза быстрее, чем старая. Пусть n - размер решаемой задачи, а N - максимальный размер, при котором удается решить задачу на старом компьютере за приемлемое время. Если используется линейный алгоритм, временная сложность которого $O(n)$, то новый компьютер даст возможность решить задачу вдвое большего размера - 2^*N . Для квадратичного алгоритма со сложностью $O(n^2)$ увеличение N составит только 41%. Переборный алгоритм со сложностью $O(2n)$ добавит к N только единицу - небольшое улучшение за такие деньги.
- В некоторых случаях эффективность может влиять на корректность. Спецификация может устанавливать, что ответ компьютера на определенное событие должен произойти не позже, чем за определенное время, например, бортовой компьютер должен быть готов определить и обработать сообщение с сенсора рычага управления двигателя достаточно быстро, чтобы сработало корректирующее действие. Эта связь между эффективностью и корректностью не ограничивается приложениями, работающими "в реальном времени"; немногие люди заинтересуются моделью предсказания погоды, которой требуется 24 часа, чтобы предсказать погоду на завтра.

Приведу еще один пример, хотя возможно менее важный, но постоянно вызывавший у меня досаду. Система управления окнами моего компьютера, используемая мной какое-то время, иногда слишком медленно определяла, что курсор мыши передвинулся из одного окна в другое, так что набираемые на клавиатуре символы, предназначенные для одного окна, попадали в другое. В этом случае ограничение эффективности приводило к нарушению спецификации, то есть корректности, которая даже, казалось бы, в безобидном повседневном применении может привести к плохим последствиям: подумайте о том, что может случиться, если два окна используются для пересылки сообщений электронной почты двум различным корреспондентам. Даже более незначительные причины приводили к расторжению браков или к началу войн.

Поскольку эта книга сосредоточивается на **концепциях** создания ОО-ПО, а не на вопросах **реализации**, только немногие разделы явным образом имеют дело с производительностью. Но проблема эффективности присутствует везде. Когда представляется ОО-решение некоторой проблемы, рассматривается не только элегантность решения, но и его эффективность. Когда вводится новый ОО-механизм, будь это сборка мусора, динамическое связывание, параметризация или повторное наследование, за этим стоит знание того, что затраты на реализацию механизма будут приемлемы по времени и памяти. Всегда, по возможности, будут упоминаться последствия изучаемых технических приемов на производительность.

Эффективность - только один из факторов качества; мы не должны (как некоторые специалисты) позволять ему главенствовать в наших разработках. Но это один из важных факторов, и он должен приниматься во внимание и в построении систем ПО, и в создании языков программирования. Если вы забудете о производительности, производительность забудет о вас.

Переносимость (Portability)

Определение: переносимость

Переносимость - это легкость переноса ПО в различные программные и аппаратные среды.

Переносимость имеет дело с разнообразием не только физического оборудования, но чаще **аппаратно-программного механизма**, того, который мы действительно программируем, включающего операционную систему, систему окон, если она применяется, и другие основные инструменты. В дальнейшем в нашей книге будет использоваться слово "платформа" для обозначения аппаратно-программного механизма; примером платформы может служить "Intel X86 + Windows NT" (известная как "Wintel").

Существующие сегодня несовместимости различных платформ неоправданы. Для наивного наблюдателя единственным объяснением, кажется, заговор с целью ввести в заблуждение человечество вообще, и программистов в частности. Однако каковы бы ни были причины, разнообразие платформ делает переносимость главной заботой и разработчиков, и пользователей ПО.

Простота использования (Easy of Use)

Определение: простота использования

Простота использования - это легкость, с которой люди с различными знаниями и квалификацией могут научиться использовать ПО и применять его для решения задач. Сюда также относится простота установки, работы и текущего контроля.

Определение подчеркивает наличие различных уровней опыта потенциальных пользователей. Это требование ставит одну из важных проблем перед проектировщиками ПО, занимающимися простотой использования: как обеспечить подробное руководство и объяснения начинающим пользователям, не мешая умелым пользователям, которые сразу хотят заняться за работу?

Как и для многих других качеств, описанных в этой лекции, ключ к легкости использования - это структурная простота. Хорошо спроектированная система, построенная в соответствии с ясной хорошо продуманной структурой, будет более простой для изучения и использования, чем построенная беспорядочно. Выполнение этого условия способствует простоте системы, но его, конечно, недостаточно. То, что просто и ясно для проектировщиков, может быть трудным и неясным для пользователей, особенно если объяснение дается в терминах проектировщика, а не в терминах доступных пользователю.

Простота использования - одна из областей, где ОО-метод особенно продуктивен; многие приемы, появившиеся вначале для решения вопросов проектирования и реализации, дали новые яркие идеи для построения интерфейса, ориентированного на конечного пользователя. В последних лекциях приводятся примеры на эту тему.

Желательно, чтобы проектировщики ПО, озабоченные простотой использования, с некоторым недоверием рассматривали принцип "**зной пользователя**". Изложенный в статье Хансена¹, он часто цитируется в литературе, посвященной пользовательским интерфейсам. Подразумевается, что хороший проектировщик должен приложить усилие для понимания того, для каких пользователей предназначена система. Этот взгляд игнорирует одно из свойств успешной системы: она всегда выходит за пределы предполагаемого круга пользователей. Напомню два старых известных примера - язык Fortran разрабатывался как инструмент для решения задачи небольшого сообщества инженеров и ученых, программирующих на IBM 704, операционная система Unix предназначалась для внутреннего использования в Bell Laboratories. Система, изначально спроектированная для особой группы людей, исходит из предположений, которые просто не будут работать для более широкой группы.

Хорошие проектировщики пользовательского интерфейса придерживаются более осмотрительной политики. Они делают как можно меньше предположений относительно своих пользователей. При проектировании интерактивной системы можно считать, что пользователи просто люди и что они умеют читать, двигать мышью, нажимать кнопки и набирать текст (медленно), и не более. Если ПО создается для специализированной области приложения, вероятно, можно, предположить, что пользователи знакомы с ее основными концепциями. Но даже это рискованно. Если перевернуть и перефразировать совет Хансена, то получим следующий принцип:

Принцип построения пользовательского интерфейса

Не делайте вид, что вы знаете пользователя - это не так.

Функциональность (Functionality)

Определение: функциональность

Функциональность - это степень возможностей, обеспечиваемых системой.

Одна из самых трудных проблем, с которой сталкивается руководитель проекта, - определение достаточной функциональности. Всегда существует желание добавлять в систему все новые и новые свойства. Желание, известное на языке индустрии как **фичеризм (featurism)**, часто **ползучий фичеризм (creeping featurism)**. Его последствия плачевны для внутренних проектов, где давление исходит от разных групп пользователей внутри одной и той же компании. Они еще хуже для коммерческих продуктов, испытывающих давление, например от журналистских сравнительных обзоров, представляющих чаще всего таблицу, включающую одновременно свойства разных конкурирующих продуктов.

Расширение свойств системы приводит к двум проблемам, одна сложнее другой. Более простая проблема - потеря непротиворечивости, которая может возникнуть при добавлении новых свойств, затрагивающих простоту использования. Известно, что пользователи жалуются, что все украшения новой версии продукта делают его ужасно сложным. Однако таким комментариям не стоит слишком доверять. Новые свойства не возникают из ничего - в основном они возникают из спроса пользователей, **других** пользователей. Что для меня выглядит ненужной безделушкой, может для вас быть необходимым свойством.

Каково же решение проблемы? Необходимо снова и снова работать над состоянием всего продукта, пытаясь привести его в соответствие с общим замыслом. Хорошее ПО основывается на небольшом количестве сильных идей. У него может быть много специальных свойств - все они должны быть следствиями основных положений. "Великий план" должен быть виден, и в нем всему должно отводиться свое место.

Более сложная проблема - слишком большое внимание к одним свойствам в ущерб другим качествам системы. В проектах часто встречается ошибка, ситуацию, которую описал Роджер Осмонд в виде двух возможных путей работы над проектом:

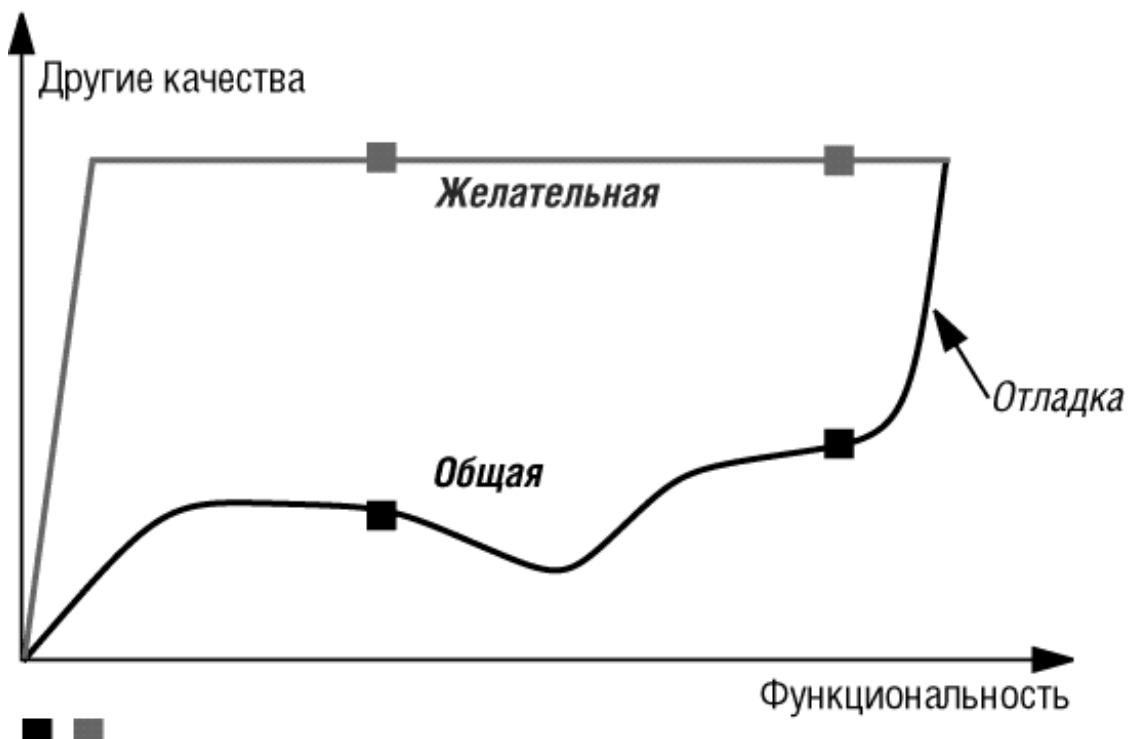


Рис. 1.4. Кривые Осмонда; по [Osmond 1995]

Нижняя кривая описывает фичеризм: в лихорадочной погоне за дополнительными свойствами теряется нить общего качества. Завершающая фаза такого проекта, предполагающая общую корректировку всех свойств, может быть долгой и напряженной. Если под давлением пользователей или конкурентов вы вынуждены выпустить продукт достаточно быстро - на стадиях, отмеченных на рисунке квадратами, - результат может повредить вашей репутации.

Осмонд предлагает (верхняя кривая) во время создания проекта поддерживать на высоком постоянном уровне качество всех факторов, кроме функциональности. Никаких компромиссов по надежности, расширяемости и прочим факторам: вы просто отказываетесь от добавления новых свойств до тех пор, пока вас удовлетворяют существующие.

Этот метод трудно осуществить в повседневной практике из-за упомянутого давления, но он в конечном итоге дает более эффективный процесс создания качественного ПО. Даже если окончательный результат тот же, что

показан на рисунке, он достигается быстрее (хотя на рисунке время не отражено). Решение выпустить "скорую" версию становится если не легче, то проще: оно будет основываться на вашей оценке того, имеете ли вы уже достаточную долю полного набора свойств, способных привлечь, но не отвратить возможных клиентов. Может возникнуть вопрос: "достаточно ли это хорошо?", но не будет стоять вопрос: "будет ли это работать?"

Как знает любой читатель, который возглавлял проект создания ПО, легче одобрить такой совет, чем его использовать. Но каждый проект должен стараться следовать подходу, соответствующему лучшей кривой Осмонда. Этот подход соответствует **клUSTERНОЙ МОДЕЛИ**, вводимой в одной из лекций книги в качестве общей схемы для дисциплинированной ОО-разработки. (См. [лекцию 10](#) курса "Основы объектно-ориентированного проектирования" Кластерная модель жизненного цикла ПО)

Своевременность (Timeliness)

Определение: своевременность

Своевременность - это выпуск ПО в нужный момент, то есть тогда или незадолго до того, как у пользователей появилась соответствующая потребность в подобной системе.

Несвоевременность - одно из больших разочарований нашей промышленности. Прекрасное ПО, появляющееся слишком поздно, может совсем не достичь своей цели. Так обстоит дело и в других отраслях промышленности, разница в том, что немногие продукты появляются так же быстро как программные.

Своевременность - до сих пор необычное явление для больших проектов. Когда Microsoft объявил, что его операционная система, находящаяся в разработке несколько лет, будет выпущена на месяц раньше, это была новость, достойная заголовка первой полосы Computer World²¹ (в статье упоминались значительные задержки в предыдущих проектах).

Другие качества

Другие качества, кроме тех, которые до сих пор обсуждались, затрагивают пользователей систем ПО и людей, покупающих эти системы или заказывающих их разработки. В частности:

- Верифицируемость (Verifiability) - это легкость подготовки процедур приемки, особенно тестовых данных, процедур обнаружения неполадок и трассировки ошибок на этапах заключительной проверки и введения проекта в действие.
- Целостность (Integrity) - это способность ПО защищать свои различные компоненты (программы, данные) от несанкционированного доступа и модификации.
- Восстанавливаемость (Repairability) - это способность облегчать устранение дефектов.
- Экономичность (Economy) сочетается с своевременностью - это способность системы завершиться, не превысив выделенного бюджета или даже не истратив его.

О документации

Казалось бы, наличие хорошей документации это тоже один из факторов качества ПО. Но это не так - напротив, необходимость документации является следствием других факторов качества, рассмотренных выше. Выделим три вида документации:

- Внешнюю, дающую пользователям возможность понять сильные стороны системы и удобство их использования. Необходимость в ней является следствием простоты использования системы.
- Внутреннюю, дающую разработчикам ПО возможность понять структуру и реализацию системы, - следствие требования расширяемости.
- Описывающую интерфейс модулей. Она дает возможность разработчикам понять функции, реализованные модулем, без изучения его реализации. Этот вид документации является следствием требования повторного использования и расширяемости, поскольку документация позволяет определить, будет ли данное изменение влиять на определенный модуль.

Документацию не следует считать независимой частью проекта. Предпочтительнее в максимально возможной степени создавать самодокументируемое ПО. Это справедливо для всех трех видов документации:

- Включение возможности получения справки проясняет соглашения пользовательского интерфейса. Тем самым облегчается задача авторов руководств пользователей и других форм внешней документации.
- Хороший язык реализации устраивает необходимость большой части внешней документации. Это будет одним из главных требований ОО-нотации, разработанной в этой книге.
- Нотация будет поддерживать скрытие информации и другие технические приемы (такие как утверждения), позволяющие отделить интерфейс модуля от его реализации. При этом становится возможным из текстов автоматически извлекать документацию интерфейса модулей. Эта тема подробно изучается в лекциях книги. Все эти приемы уменьшают роль традиционной документации,

хотя, конечно, не следует ожидать, что они полностью ее заменят.

Компромиссы

В данном обзоре внешних факторов качества ПО мы встретились с требованиями, которые могут конфликтовать друг с другом.

Как можно достичь **целостности**, если не вводить защиты различного рода, что неизбежно затруднит **простоту использования**? Экономичность часто конфликтует с **функциональностью**.

Оптимальная **эффективность** требует полной адаптации к определенному оборудованию и программной среде, что является противоположностью **переносимости**. **Повторное использование** требует решения общих задач, что расширяет границы, заданные спецификацией. Давление **своевременности** может склонить нас к технике RAD - быстрой разработки приложения (Rapid Application Development), что может повредить **расширяемости**. Хотя во многих случаях удается найти решение, примиряющее явно конфликтующие факторы, иногда приходится идти на компромисс.

Разработчики слишком часто и без колебаний идут на компромисс, не давая себе труда рассмотреть соответствующие вопросы и имеющиеся варианты. В таких молчаливых решениях доминирующим фактором обычно является эффективность. По-настоящему инженерный подход к созданию ПО подразумевает работу по ясной формулировке критериев и осознанного выбора вариантов.

Как бы ни были необходимы компромиссы между факторами качества, один из факторов стоит в стороне от остальных - корректность. Нет никакого оправдания тому, что корректность подвергается опасности ради других факторов, таких как эффективность. Если ПО не выполняет свою функцию, все остальное не имеет смысла.

Ключевые вопросы

Все описанные выше факторы важны. Но при современном состоянии индустрии ПО четыре фактора имеют особую важность:

- **Корректность и устойчивость**: все еще слишком трудно создавать ПО без ошибок (bugs), и слишком сложно исправлять ошибки, когда они появляются. Разновидности технических приемов для улучшения корректности и устойчивости одни и те же: более систематические подходы к построению ПО; более формальные спецификации; встроенный контроль в течение всего процесса построения ПО (не просто испытания и отладка после создания); более совершенные языковые механизмы, такие как статическая типизация, утверждения, автоматическое управление памятью и упорядоченное управление исключительными ситуациями, обеспечение возможности разработчикам устанавливать требования корректности и устойчивости в сочетании с возможностью инструментов обнаруживать случаи несостоятельности до того, как они приведут к ошибкам. Близость вопросов корректности и устойчивости делает удобным введение общего термина для обозначения обоих факторов - **надежность (reliability)**.
- **Расширяемость и повторное использование**: ПО должно быть легко изменяемым; компоненты создаваемого ПО должны быть широко применимы, и должен существовать больший перечень общечелевых компонентов, которые можно повторно использовать при разработке новой системы. Здесь также одни и те же идеи полезны для улучшения обоих качеств: любая идея, помогающая производить продукт с более децентрализованной архитектурой, компоненты которой автономны и взаимодействуют только через ограниченные и ясно определенные каналы, будет полезной. Термин **модульность (modularity)** включает повторное использование и расширяемость.

ОО-метод, детально изучаемый в последующих лекциях, может значительно улучшить четыре основных фактора качества, вот почему он так привлекателен. Он также может внести значительный вклад в другие аспекты, в частности:

- **Совместимость**: метод обеспечивает общий стиль проектирования и стандартизацию интерфейсов модулей и систем, что помогает совместно работать разным системам.
- **Переносимость**: уделяя особое внимание абстракции и скрытию информации, объектная технология способствует тому, что проектировщики начинают отделять спецификацию от особенностей реализации, что и облегчает перенос. Полиморфизм и динамическое связывание делает возможным создание системы, автоматически адаптируемой к аппаратно-программному механизму, например, различным системам окон или различным системам управления базами данных.
- **Простота использования**: вклад ОО-инструментов в современные интерактивные системы, и особенно их пользовательские интерфейсы, так хорошо известен, что иногда он затмевает другие аспекты (люди, создающие рекламу - не единственные, кто называет "объектно-ориентированной" любую систему, использующую значки, окна и ввод с помощью мыши).
- **Эффективность**: как отмечалось выше, повторное использование компонентов профессионального качества часто может значительно улучшить производительность.

- **Своевременность, экономичность и функциональность:** ОО-техника дает возможность тем, кто ее освоил, производить ПО быстрее и по более низкой стоимости; она облегчает добавление функций и даже сама может предложить новые функции.

Несмотря на все эти успехи, мы должны помнить, что ОО-метод - это не панацея, и что многие обычные вопросы проектирования ПО остаются нерешенными. Помощь в решении проблемы - это не то же самое, что ее решение.

О программном сопровождении

Приведенный список факторов не включил обычно приводимое качество: возможность сопровождения (maintainability). Чтобы понять почему, мы должны поближе взглянуть на лежащее в его основе понятие: **сопровождение (maintenance)**. Сопровождение начинается с момента поставки ПО пользователям.

Обсуждения методологии создания ПО обычно сосредоточиваются на фазе разработки; то же находим и во вводных курсах по программированию. Но широко известно, что 70% стоимости ПО приходится на его сопровождение. Никакое изучение качества ПО не может быть удовлетворительным, если оно игнорирует этот аспект.

Что означает сопровождение для ПО? Если на минуту задуматься, то становится ясно, что этот термин употребляется неправильно: ПО не изнашивается от постоянного использования, и ему не требуется такое "обслуживание", как автомобилю или телевизору. Специалисты по программным продуктам используют это слово для описания уважаемых (noble) и не очень уважаемых функций сопровождения. К уважаемой, достойной части работы можно отнести модификацию системы. Поскольку спецификации компьютерных систем меняются, отражая изменения во внешнем мире, должны меняться и сами системы. Наименее уважаемая часть - это запоздалая отладка: удаление ошибок, которых не должно было быть в начале.



Рис. 1.5. Распределение расходов на сопровождение. Источник: [Лиенц, 1980]

Вышеприведенная диаграмма, взятая из ключевого исследования Лиенца и Свонсона, проливает некоторый свет на то, что на самом деле значит включающий разнообразные понятия термин "сопровождение". Исследование рассмотрело 487 систем, разрабатывающих ПО разного рода; возможно, оно немного устарело, но более поздние публикации подтверждают те же общие результаты. Оно показывает долю стоимости, приходящуюся на каждый идентифицированный авторами вид работ по сопровождению.

Более двух пятых стоимости идет на расширения и модификации, требующиеся пользователям. Это то, что мы выше назвали уважаемой частью сопровождения, без которой работающая система обойтись не может. Неразрешенный вопрос в том, какую долю общей работы промышленность может сэкономить, если с самого начала она будет строить ПО, уделяя больше внимание расширяемости. Мы можем законно ожидать, что объектная технология здесь будет полезна.

Второй значимый фактор в распределения стоимости сопровождения особенно интересен: изменение формата данных. При изменении физической структуры файлов и других элементов данных приходится адаптировать программы. Например, американская почтовая служба несколько лет назад ввела почтовый код "5+4",

использующий девять цифр вместо пяти. Пришлось переписывать многочисленные программы, имеющих дело с адресами и "знающими", что почтовый код состоит точно из пяти цифр. По сообщениям прессы, затраты оценивались в сотни миллионов долларов.

Другая известная проблема - Millenium - переход компьютеров на даты нового тысячелетия.

Вопрос не в том, что некоторая часть программы знает физическую структуру данных: это неизбежно, поскольку доступ к данным необходим. Но при традиционных методах построения это знание распространяется слишком на многие части системы, приводя к неоправданно большим программным изменениям при изменении физической структуры. Другими словами, если почтовые коды изменяются с пяти до девяти цифр или даты требуют еще одной цифры, то резонно ожидать, что программа, манипулирующая кодами и датами, будет требовать адаптации. Недопустимо лишь, чтобы изменения в программе были несоизмеримы по сравнению с концептуальным размером изменения спецификации.

Теория абстрактных типов данных даст ключ к этой проблеме (Лекция 6 подробно описывает абстрактные типы данных), позволяя программам иметь доступ к данным с помощью внешних свойств, а не физической реализации.

Следующие пункты в списке Лиенца и Свонсона также интересны, но не так непосредственно связаны с темами этой книги. Аварийная отладка (производимая в спешке, когда пользователь сообщает, что программа не дает ожидаемых результатов или ведет себя катастрофически) стоит больше, чем обычные плановые исправления. Это так не только потому, что она производится в короткие сроки, но и потому, что она прерывает плановый процесс выпуска новых (безошибочных) вариантов и может дать новые ошибки.

Еще одно интересное наблюдение в распределении затрат по видам деятельности - это сравнительно низкая доля (5,5%) стоимости документации. Помните, что это - стоимость задач, решаемых в период эксплуатации. Наблюдение здесь - или скорее, догадка, при отсутствии более точных данных - таково: проект должен либо заботиться о том, чтобы создание документации стало частью разработки, либо совсем не делать этого. Мы научимся использовать стиль построения, в котором большая часть документации действительно встроена в ПО, и есть специальные инструменты для ее извлечения.

Последние два вида работ дают очень малую долю:

- Первый - это улучшение эффективности; похоже, предполагается, что когда система работает, менеджеры проекта и программисты неохотно прерывают ее работу с целью улучшения производительности, предпочитая не трогать довольно хорошую систему. (При рассмотрении принципа "сначала сделай ее хорошо, а потом сделай ее быстрой" многие проекты, возможно, вполне довольствуются первым шагом.)
- Небольшие средства тратятся и на "переход к новой аппаратной среде". Из-за отсутствия более детальных данных можно высказать лишь некоторое предположение. Все системы относятся к двум крайним случаям, промежуточные варианты практически отсутствуют. В первом случае системы изначально строятся как переносимые, и потому для них этот вид затрат невелик. Другие настолько тесно привязаны к своей первоначальной платформе и перенос был бы так труден, что разработчики даже не пытаются делать что-то в этом направлении.

Ключевые концепции

- Целью программной инженерии является нахождение путей построения ПО высокого качества.
- Качество ПО лучше всего видится как компромисс между целым рядом различных целей, а не как единый фактор.
- Внешние факторы, понятные пользователям и клиентам, следует отличать от внутренних факторов, понятных проектировщикам и конструкторам.
- Действительное значение имеют внешние факторы, но управление системой возможно только через внутренние факторы, благодаря которым достигается нужный эффект.
- Список основных внешних факторов качества приведен выше. ОО-метод направлен на улучшение качества тех факторов, которые прежде всего нуждаются в лучших подходах. К ним относятся факторы корректности и устойчивости, связанные с безопасностью, вместе известные как надежность, и факторы, требующие децентрализованной архитектуры ПО, - повторное использование и расширяемость, вместе известные как модульность.
- Сопровождение ПО, потребляющее большую долю его стоимости, находится в невыгодном положении из-за трудности реализации изменений в ПО и из-за слишком большой зависимости программ от физической структуры данных, которыми они манипулируют.

¹⁾ См. Wilfred Hansen, "User Engineering Principles for interactive Systems", Proceedings of FJSS 39, AFIPS Press, Montvale (NJ), 1971. pp 523-532

²⁾ "NT 4.0 Beats Clock", Computer World, vol.30, №30, 22 july 1996

Основы объектно-ориентированного программирования

2. Лекция: Критерии объектной ориентации

В предыдущей лекции исследовались цели ОО-метода. Готовясь к чтению технических деталей метода в следующих лекциях, полезно быстро, но с широких позиций рассмотреть ключевые аспекты ОО-разработки ПО. Такова цель этой лекции. Прежде всего, здесь будет дано лаконичное пояснение того, что делает систему объектно-ориентированной. Уже в этом есть определенная польза, поскольку этот термин используется так неразборчиво, что необходим список точных свойств; имея их, мы сможем оценить метод, язык или инструмент, претендующие на звание объектно-ориентированных. Ограничимся минимумом объяснений, поэтому при первом чтении нельзя надеяться на понимание деталей в всех перечисленных критериях; объяснение их - задача остальных разделов книги. Можно считать это обсуждение предваряющим просмотром - не настоящим кино, а ансамблем. В отличие от анонса, эта лекция скорее является так называемым спойлером (*spoiler*) - она пересказывает сюжет, нарушая порой общий план книги. Этим она отличается от других лекций, в особенности лекций 3-6, терпеливо выстраивавших объектную технологию, рассматривавших проблему за проблемой на пути к получению и обоснованию решения. Если вам нравится идея обзора, предшествующая глубокому изучению вопросов, эта лекция для вас. Но если вы предпочитаете не портить удовольствия, открывая решения одно за другим, то просто пропустите ее.

О критериях

Рассмотрим выбор критериев, позволяющих оценить объектную ориентированность системы (*objectness*).

До какой степени мы должны быть догматичными?

Список, представленный ниже, включает все свойства, кажущиеся существенными для создания высококачественного ПО ОО-методом. Наш список может показаться бескомпромиссным и даже догматичным. Какие заключения следует делать, если среда удовлетворяет некоторым, но не всем этим критериям? Следует ли считать ее полностью неадекватной?

Только вы, мой читатель, можете ответить на этот вопрос применительно к собственному контексту. Вот несколько причин, по которым может быть необходим компромисс:

- Быть ОО-системой - это не булево условие. Из двух сред А и В первая может быть более объектно-ориентированной, хотя и не является таковой на 100%. Поэтому если внешние ограничения сводят ваш выбор только к А и В, следует выбрать А как наименьшее из двух зол.
- Не каждому нужны всегда все свойства.
- Объектная ориентация может быть просто одним из факторов, определяющих наше решение, поэтому придется соблюдать баланс между критериями, приведенными здесь, и другими соображениями.

Все это не меняет очевидного: для обоснованного выбора, даже если практические ограничения навязывают далеко не совершенные решения, необходимо видеть полную картину.

Категории

Набор критериев делится на три части:

- **Метод и язык (Method and Language)** : эти два почти не различимые аспекта охватывают мыслительные процессы и нотацию, использующуюся для анализа, проектирования и программирования ПО. Заметьте, что (особенно в объектной технологии) термин "язык" относится не только к языку программирования в строгом смысле, но также и к языкам анализа и проектирования и используемой в них нотации, текстовой или графической.
- **Реализация (Implementation) и Среда (Environment)** : критерии в этой категории описывают основные свойства инструментария, позволяющего разработчикам применять ОО-идеи.
- **Библиотеки (Libraries)** : объектная технология основана на повторном использовании компонентов ПО. Критерии в этой категории описывают как наличие базовых библиотек, так и механизмы, необходимые для их использования и создания новых библиотек.

Такое деление удобно, но не абсолютно, поскольку некоторые критерии относятся к двум или трем категориям. Например критерий, помеченный "управление памятью", относится к категории языка, поскольку язык может поддерживать или не допускать автоматическую сборку мусора. Этот же критерий относится к категории реализации и среды.

Метод и язык

Первый набор критериев относится к методу и поддерживающей его нотации.

Бесшовность (seamlessness)

ОО-подход амбициозен: он включает весь жизненный цикл ПО. При рассмотрении ОО-решений следует

проверить, что метод, язык и поддерживающие их инструменты, применимы к анализу и проектированию, а также к реализации и сопровождению. Язык, в частности, должен служить средством мышления, помогающим на всех стадиях работы.

В результате получается бесшовный процесс разработки, где общность концепций и нотации помогает сгладить переходы между последовательными ступенями жизненного цикла.

Эти требования исключают два часто встречающихся случая - оба неудовлетворительных:

- Использование ОО-концепций на этапе анализа и проектирования с такой нотацией, которая не может использоваться на этапе программирования.
- Использование ОО-языка программирования, неподходящего для этапа анализа и проектирования.

ОО-язык и ОО-среда, вместе с поддерживающим их методом, должны быть применимы ко всему жизненному циклу, минимизируя сложность переходов между последовательными шагами.

Классы

ОО-метод основан на понятии класса. Неформально, класс - элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию. Абстрактный тип данных - множество объектов, определяемое списком компонентов (**features**) - операций, применимых к этим объектам, и их свойств.

Понятие класса должно быть центральной концепцией метода и языка.

Утверждения (Assertions)

Компоненты абстрактного типа данных имеют формально специфицированные свойства, отражаемые в соответствующих классах.

Утверждения - предусловия и постусловия программ класса и инварианты классов - играют эту роль.

Утверждения имеют три основных применения: помогают создать надежное ПО, обеспечивают систематическую документацию и являются инструментом тестирования и отладки ПО.

Язык должен давать возможность: поставлять класс и его компоненты вместе с утверждениями (предусловиями, постусловиями и инвариантами); включать инструментарий для получения документации из этих утверждений; осуществлять мониторинг утверждений во время выполнения программы.

В сообществе программных модулей, где классы являются городами, а инструкции - исполнительной ветвью власти, утверждения представляют законодательную власть. Ниже мы увидим, что играет роль судебной системы в таком сообществе.

Классы как модули

Объектная ориентация - в первую очередь архитектурная техника: она в основном затрагивает модульную структуру системы.

Здесь опять велика роль классов. Класс описывает не только тип объектов, но и модульную единицу. В чистом ОО-подходе:

Классы должны быть единственным видом модулей.

В частности, исчезает понятие главной программы, а подпрограммы не существуют как независимые модульные единицы (они могут появляться только как часть классов). Нет необходимости в "пакетах", используемых в таких языках, как Ada. Хотя удобно в целях управления группировать классы в административные единицы, называемые **кластерами**.

Классы как типы

Понятие класса достаточно мощное, чтобы избежать необходимости любого другого механизма типизации:

Каждый тип должен быть основан на классе.

Даже базовые типы, такие как INTEGER и REAL, можно рассматривать как классы; обычно такие классы являются встроенными.

Вычисления, основанные на компонентах

В ОО-вычислениях существует только один базовый вычислительный механизм. Есть некоторый объект, всегда являющийся (в силу предыдущего правила) экземпляром некоторого класса, и вычисление состоит в том, что данный объект вызывает некоторый компонент этого класса. Например, для показа окна на экране вызывается компонент `display` объекта, представляющего окно, - экземпляра класса `WIND0W`. Компоненты могут иметь аргументы: для увеличения зарплаты работника `e` на дату `d`, на `n` долларов, вызывается компонент `raise` объекта `e`, с аргументами `n` и `d`.

Базисные типы рассматриваются как предопределенные классы, и основные операции (например, сложение чисел) рассматриваются как специальные предопределенные случаи вызова компонентов - общий механизм вычислений:

Вызов компонента должен быть основным механизмом вычисления.

Класс, содержащий вызов компонента класса `C`, называют клиентом **класса C**.

Вызов компонента иногда называют **передачей сообщения (message passing)**; по этой терминологии вышеупомянутый вызов будет описываться как передача объекту `e` сообщения: "повысить вашу плату" с аргументами `d` и `n`.

Скрытие информации (information hiding)

При создании класса зачастую в него приходится включать компонент, необходимый только для внутренних целей, являющейся частью реализации класса, но не его интерфейса. Другие компоненты этого класса, - возможно, доступные клиентам, - могут вызывать этот внутренний компонент для собственных нужд. Но не следует такую возможность давать клиенту.

Механизм, делающий определенные компоненты недоступными для клиентов, называется скрытием информации.

На практике бывает недостаточно того, чтобы механизм скрытия информации поддерживал экспортруемые компоненты (доступные для всех клиентов) и скрытые компоненты (не доступные ни одному клиенту).

Создатели классов должны также иметь возможность избирательно экспортировать компоненты для выбранных клиентов.

Автор класса должен иметь возможность указать, что компонент доступен: всем клиентам, ни одному клиенту или выбранным клиентам.

Прямое следствие этого правила - строгая ограниченность взаимодействия классов. В частности, хороший ОО-язык не должен включать понятие глобальной переменной. Классы обмениваются информацией исключительно через вызовы компонентов и механизм наследования.

Обработка исключений (Exception handling)

В процессе выполнения программ могут встречаться различные аномалии. В ОО-вычислениях они соответствуют вызовам, которые не могут быть выполнены надлежащим образом: например в результате сбоя в оборудовании, переполнения при выполнении арифметических операций или ошибок ПО.

Для создания надежного ПО необходимо иметь возможность восстановления нормального хода вычислений. Это является целью механизма обработки исключений.

Язык должен обеспечивать механизм восстановления в неожиданных аварийных ситуациях.

В сообществе программных модулей механизм обработки исключений - третья ветвь власти, судебная система и поддерживающие ее силы полиции.

Статическая типизация (static typing)

Когда в системе происходит вызов некоторого компонента определенным объектом, как узнать, что объект способен обработать вызов? (В терминологии сообщений: как узнать, что объект может обработать сообщение?)

Чтобы гарантировать корректное выполнение, язык должен быть типизирован. Это означает, что он отвечает нескольким правилам совместимости:

- Каждая сущность (**entity**) объявляется явным образом с указанием определенного типа, порожденного классом. Под сущностью понимается имя, используемое в тексте ПО для ссылки на объекты времени выполнения.

- Каждый вызов компонента - это вызов **доступного** компонента соответствующего класса.
- Присваивание и передача аргументов подчиняются **правилам согласования**, требующим совместимости исходного типа и целевого типа.

В языке, включающем такую политику, возможен **статический контроль типов**. Тогда еще на этапе компиляции подобные ошибки будут обнаружены, и во время выполнения гарантируется отсутствие ошибок типа: "компонент недоступен объекту".

Хорошо определенная система типов гарантирует безопасность работы с объектами во время выполнения программной системы.

Универсальность (genericity)

Для того чтобы типизация была практической, необходимо иметь возможность определять классы с параметрами, задающими тип. Такие классы известны как родовые. Родовой класс LIST [G] описывает списки элементов произвольного типа G - "формальным родовым параметром".

Классы, задающие специальные списки, будут его производными, например LIST [INTEGER] и LIST [WINDOW] используют типы INTEGER и WINDOW в качестве "фактических родовых параметров". Все производные классы разделяют один и тот же текст родового класса.

Должна существовать возможность создания классов с формальными родовыми параметрами, представляющими произвольные типы.

Эта форма параметризации типа называется **неограниченной** универсальностью. Дополнительной возможностью, описанной ниже, является **ограниченная** универсальность, использующая понятие наследования.

Единичное наследование (single inheritance)

Разработка ПО включает создание большого числа классов, многие из которых являются вариантами ранее созданных классов. Для управления потенциальной сложностью такой системы необходим механизм классификации, известный как наследование. Класс А будет наследником (heir) класса В, если он встраивает (наследует) компоненты класса В в дополнение к своим собственным. Потомок (descendant)- это прямой или непрямой наследник; обратное понятие - предок (ancestor).

Должно быть возможным объявить класс наследником другого класса.

Наследование - одно из центральных понятий ОО-метода; оно оказывает большое влияние на процесс разработки ПО.

Множественное наследование (Multiple inheritance)

Часто необходимо сочетать различные абстракции. Рассмотрим класс, моделирующий понятие "младенец". Его можно рассматривать как класс "человек" с компонентами, связанными с этим классом. Его же можно рассматривать и более прозаично - как класс "элемент, подлежащий налогообложению", которому положены скидки при начислении налогов. Наследование оправдано в обоих случаях. Множественное наследование (multiple inheritance) - это гарантия того, что класс может быть наследником не только одного класса, но многих, если это концептуально оправдано.

При множественном наследовании возникает несколько технических проблем, например разрешение конфликта имен (компоненты, наследованные от разных классов, имеют одно и то же имя). Любая нотация, предлагающая множественное наследование, должна обеспечить адекватное решение этих проблем.

Класс должен иметь возможность быть наследником нескольких классов.

Конфликты имен при наследовании разрешаются адекватным механизмом.

Решение, разработанное в этой книге, основано на переименовании конфликтующих компонентов у класса наследника.

Дублируемое наследование (Repeated inheritance)

При множественном наследовании возникает ситуация дублируемого наследования (repeated inheritance), когда некоторый класс многократно становится наследником одного и того же класса, проходя по разным ветвям наследования:

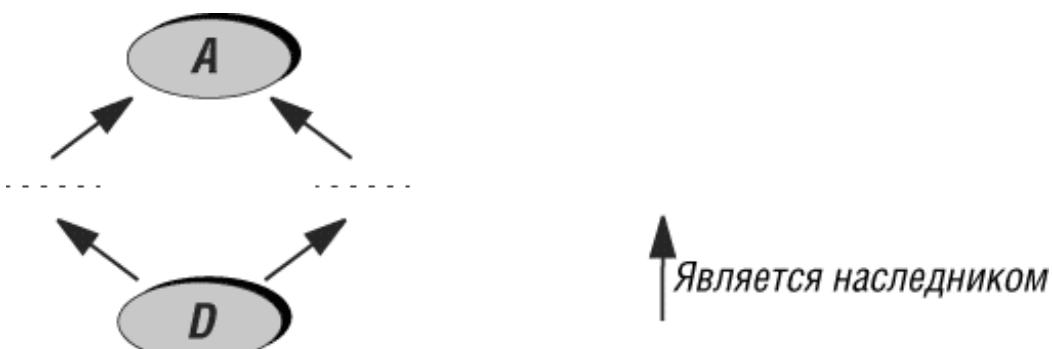


Рис. 2.1. Дублируемое наследование

В этом случае язык должен обеспечить точные правила, определяющие, что происходит с компонентами, наследованными повторно от общего предка (на рисунке - это А). В некоторых случаях желательно, чтобы компонент из А создавал только один компонент в D (**разделение**), а в других - нужно, чтобы он создавал два (**дублирование**). Разработчики должны обладать гибкими средствами, позволяющими предписывать одну из возможностей независимо для каждого компонента.

При дублируемом наследовании судьбой компонентов должны управлять точно определенные правила, позволяющие разработчикам выбирать для каждого такого компонента разделение, либо дублирование.

Ограниченнная универсальность (Constrained genericity)

Сочетание универсальности и наследования дает полезную технику - ограниченную универсальность (constrained genericity). Теперь вы можете определить класс с родовым параметром, представляющим не произвольный тип, а лишь тип, являющийся потомком некоторого класса.

Родовой класс SORTABLE_LIST описывает списки; он содержит компонент sort, сортирующий элементы списка в соответствии с заданным отношением порядка. Параметр этого родового класса задает тип элементов списка. Но этот тип не может быть произвольным: он должен поддерживать отношение порядка. Фактический родовой параметр должен быть потомком библиотечного класса COMPARABLE, описывающего объекты, снабженные отношением порядка. Ограниченнная универсальность позволяет объявить наш родовой класс следующим образом: SORTABLE_LIST [G - " COMPARABLE] .

Механизм универсальности должен поддерживать форму ограниченной универсальности.

Переопределение (redefinition)

Когда класс является наследником другого класса, может потребоваться изменить реализацию или другие свойства некоторых наследованных компонент. Класс SESSION, управляющий сессиями пользователей в операционной системе, может иметь компонент terminate, выполняющий чистку в конце сеанса. Его наследником может быть класс REMOTE_SESSION, управляющий сеансом удаленного компьютера в сети.

Если завершение удаленного сеанса требует дополнительных действий, таких как, например, уведомление удаленного компьютера, класс REMOTE_SESSION переопределит компонент terminate .

Переопределение может повлиять на реализацию компонента, его сигнатуру (тип аргументов и результата) и спецификацию.

Должно быть возможным переопределить спецификацию, сигнатуру и реализацию наследованного компонента.

Полиморфизм

При наследовании, требование статической типизации, о котором говорилось выше, становится ограничивающим, если бы оно означало, что каждая сущность типа С может быть связана только с объектом точно такого же типа С. Например в системе управления навигацией сущность типа BOAT нельзя было использовать для объектов класса MERCHANT_SHIP или SPORTS_BOAT, хотя оба класса являются потомками класса BOAT.

Как уже отмечалось, "сущность" - это имя, к которому во время выполнения могут присоединяться различные значения. Сущность - это обобщение традиционного понятия переменной.

Полиморфизм (polymorphism) - способность присоединять к сущности объекты различных возможных типов. В статически типизированной среде полиморфизм не будет произвольным, а будет контролироваться

наследованием.

Должна иметься возможность в период выполнения присоединять к сущности объекты различных возможных типов под управлением наследования.

Динамическое связывание

Сочетание последних двух механизмов, переопределения и полиморфизма, непосредственно предполагает следующий механизм. Допустим, есть вызов, целью которого является полиморфная сущность, например сущность типа BOAT вызывает компонент turn. Различные потомки класса BOAT, возможно, переопределили этот компонент различными способами. Ясно, что должен существовать автоматический механизм, гарантирующий, что версия turn всегда соответствует фактическому типу объекта, вне зависимости от того, как объявлена сущность. Эта возможность называется динамическим связыванием (dynamic binding).

Вызов сущностью компонента всегда должен запускать тот компонент, который соответствует типу присоединенного объекта, а не типу сущности.

При различных выполнениях одного и того же вызова могут запускаться разные компоненты.

Динамическое связывание оказывает большое влияние на структуру ОО-приложения, поскольку дает возможность разработчикам писать простые вызовы, например объект my_boat вызывает компонент turn. В действительности, данный вызов означает несколько возможных вызовов, зависящих от соответствующих ситуаций времени выполнения. Это упраздняет необходимость многих повторных проверок (является ли объект merchant_ship? Является ли он sports_boat?), наводняющих программные продукты, создаваемые при обычных подходах.

Выяснение типа объекта в период выполнения

Разработчики ОО-ПО вскоре вырабатывают здоровую неприязнь к любому стилю вычислений, основанному на явном выборе между различными типами объекта. Полиморфизм и динамическое связывание намного предпочтительнее. Однако в некоторых случаях объект приходит извне, так что автор ПО не имеет возможности с определенностью предсказать его тип. В частности, это случается, если объект извлекается из внешних хранилищ, получен по сети или передан некоторой другой системой.

Тогда ПО нуждается в механизме, обеспечивающем безопасный способ доступа к объекту без нарушения ограничений статической типизации. Такой механизм должен проектироваться с большой аккуратностью, так чтобы не утратить пользы от полиморфизма и динамического связывания.

Операция **попытка присваивания (assignment attempt)** удовлетворяет этим требованиям. Это условная операция: она пытается присоединить объект к сущности; если при выполнении операции тип объекта соответствует типу сущности, то она действует как нормальное присваивание; в противном случае сущность получает специальное значение **void**. Итак, можно управлять объектами, тип которых не известен наверняка, не нарушая безопасности системы типов.

Необходимо иметь возможность определять во время выполнения, соответствует ли тип объекта статически заданному типу.

Отложенные (deferred) свойства и классы

В некоторых случаях, для которых динамическое связывание дает элегантное решение, устранивая необходимость явных проверок, не существует начальной версии компонента, подлежащего переопределению. Например, класс BOAT может быть настолько общим, что не может обеспечить реализацию turn по умолчанию. Все же, мы хотим иметь возможность вызывать компонент turn сущностью типа BOAT, если мы уверены, что во время выполнения она будет получать значение объектов таких полностью определенных типов как, например, MERCHANT_SHIP и SPORTS_BOAT.

В таких случаях BOAT может объявляться как отложенный класс (класс, который не полностью реализован) и с отложенной реализацией компонента turn. Отложенные свойства и классы все же могут иметь утверждения, описывающие их абстрактные возможности, но их реализация откладывается для классов потомков. Если класс не является отложенным, - он считается **эффективным**.

Необходимо иметь возможность написания класса или компонента как отложенного, то есть специфицированного, но не полностью реализованного.

Отложенные классы (также называемые абстрактными классами) особенно важны для ОО-анализа и высокоуровневого проектирования, поскольку они делают возможным задать основные аспекты системы, оставляя детали до более поздней стадии.

Управление памятью (memory management) и сборка мусора (garbage collection)

Может показаться, что этот критерий метода и языка должен принадлежать к следующей категории - реализации и среде. На самом деле он принадлежит к обеим категориям. Важнейшие требования предъявляются к языку, остальное - это вопрос хорошей инженерии.

ОО-системы даже в большей степени, чем традиционные системы, за исключением, быть может, Lisp, имеют тенденцию создания большого числа объектов, иногда со сложными взаимозависимостями. Политика, возлагающая на разработчиков ответственность за управление памятью, вредит и эффективности процесса разработки, и безопасности полученной системы. Трудно утилизировать память, занятую более не нужными объектами, усложняются программы, все это требует времени разработчиков, увеличивается риск некорректной обработки областей памяти. В хорошей ОО-среде управление памятью будет автоматическим, под контролем сборщика мусора (garbage collector) - компонента системы периода выполнения (runtime system).

Автоматическая сборка мусора - это проблема языка, так же как и реализации. Если язык явно не спроектирован для автоматического управления памятью, то зачастую реализация становится невозможной. Это справедливо для языков, где, например, указатель на объект определенного типа может быть преобразован (используя кастинг - **cast**) в указатель другого типа или даже в целое число, - такие средства делают невозможным создание надежного сборщика мусора.

Язык должен давать возможность надежного автоматического управления памятью, а реализация должна обеспечить наличие автоматического менеджера, управляющего памятью, в функцию которого входит сборка мусора.

Реализация и среда

Мы подошли к важным свойствам среды разработки, поддерживающей создание ОО-ПО.

Автоматическое обновление (automatic update)

Разработка ПО - процесс нарастающий. Разработчики обычно не пишут тысячи строк за один раз; они работают, добавляя и модифицируя, начиная чаще всего с системы, уже имеющей значительный размер.

При выполнении такого обновления важно иметь гарантию, что полученная в результате система будет согласованной. Например, если вы меняете некоторый компонент *f* класса *C*, то вы должны быть уверены, что любой потомок *C*, который не переопределяет *f*, получит новую версию *f*, и что каждое обращение к *f* клиента *C* или потомка *C* будет запускать эту новую версию.

Традиционные подходы к этой проблеме предполагают работу вручную, заставляя разработчиков записывать все зависимости и прослеживать их изменения, используя специальные механизмы, известные как "создавать файлы" и "включать файлы". Это неприемлемо в современных разработках программных продуктов, особенно в ОО-мире, где взаимозависимости между классами, вытекающие из отношений наследования, часто сложны, но могут быть выведены из систематического рассмотрения текста ПО.

Обновление системы после изменения должно быть автоматическим, а анализ межклассовых зависимостей выполняться инструментарием, а не вручную разработчиками.

Это требование можно удовлетворить в компилируемой среде (где компилятор будет работать вместе с инструментарием, выполняющим анализ зависимостей), в интерпретируемой среде или в среде, сочетающей обе эти техники реализации языка.

Быстрое обновление (fast update)

На практике механизм обновления системы должен быть не только автоматическим, но и быстрым. Более точно, он должен быть пропорциональным размеру изменений, а не размеру системы в целом. Без этого свойства метод и среда могут быть применимыми только к небольшим системам, а применять их нужно к большим.

Время обработки ряда изменений в системе, создающих обновленную версию, должно быть функцией размера измененных компонентов и не зависит от размера системы в целом.

И компилируемая, и интерпретируемая среда могут удовлетворять этому критерию, хотя в последнем случае компилятор должен быть инкрементным (он не должен все компилировать заново). Наряду с инкрементным компилятором, среда может, конечно, включать глобальный оптимизирующий компилятор, работающий на всей системе. При условии, что глобальный компилятор нужен только для выпуска конечного продукта, разработка будет в основном использовать инкрементный компилятор.

Живучесть (persistence)

Многие приложения, вероятно, большинство, требуют сохранения объектов от одного сеанса до следующего. Среда должна обеспечивать механизм выполнения этого простым способом.

Объект часто содержит ссылки на другие объекты, тоже содержащие, в свою очередь, ссылки на объекты. Поэтому каждый объект может иметь большое количество **зависимых** объектов с возможно сложным графом зависимости (который может содержать циклы). Обычно не имеет смысла сохранять или восстанавливать объект без всех его прямых и непрямых зависимых объектов. Говорят, что механизм живучести поддерживает замыкание живучести (persistence closure), если он может автоматически сохранять зависимые объекты наряду с самим объектом.

Должен существовать механизм хранения, поддерживающий замыкание живучести. Он сохраняет объект вместе со всеми зависимыми объектами на внешних устройствах и восстанавливает их в течение того же или другого сеанса.

Для некоторых приложений простой поддержки живучести недостаточно; такие приложения **нуждаются** в полной **поддержке баз данных (database support)**. Понятие ОО-базы данных объясняется в одной из дальнейших лекций, где также исследуются другие вопросы живучести, такие как **эволюция схемы**, способность безопасного восстановления объектов, даже если изменились соответствующие классы.

Документация

Разработчики классов и систем должны обеспечивать руководство, заказчиков и других разработчиков ясными высокоуровневыми описаниями создаваемого ПО. Им необходим инструментарий, помогающий в этой работе. Большая часть документации должна автоматически создаваться на основе текстов ПО. Утверждения, как уже отмечено, помогают сделать такие документы, извлекаемые из ПО, точными и информативными.

Должны быть в наличии инструментальные средства для автоматического получения документации о классах и системах.

Быстрый просмотр (browsing)

При работе с классом часто необходимо получить информацию о других классах; в частности, компоненты данного класса часто могут определяться не в самом классе, а в его различных предках. Среда должна обеспечить разработчиков инструментами для исследования текста класса, нахождения зависимых классов и быстрого переключения с текста одного класса на другой.

В этом и состоит задача просмотра. Типичные хорошие возможности просмотра включают: поиск классов - клиентов, поставщиков, потомков, предков; поиск всех переопределений компонента; поиск исходного объявления переопределенного компонента. (Определение: S - поставщик C, если C - клиент S. Термин "клиент класса" пояснен выше.)

Средства интерактивного просмотра должны давать возможность разработчикам ПО быстро и удобно прослеживать зависимости между классами и компонентами.

Библиотеки

Один из характерных аспектов разработки ПО ОО-способом - возможность создавать его на основе существующих библиотек. ОО-среда должна обеспечивать хорошие библиотеки и механизмы создания новых библиотек.

Базовые библиотеки

Изначально в информатике изучаются фундаментальные структуры данных - множества, списки, деревья, стеки; связанные с ними алгоритмы - сортировки, поиска, обхода, сопоставления с образцом. Эти структуры и алгоритмы вездесущи в разработках ПО. Нередко, когда в своей системе очередной разработчик повторно их реализует. Это не только расточительно, но и пагубно отражается на качестве ПО, поскольку вряд ли отдельный разработчик, реализующий структуру данных не как цель саму по себе, а в качестве компонента некоторого приложения, достигнет оптимальной надежности и производительности.

ОО-среда разработки должна обеспечить повторно используемые классы, удовлетворяющие общим потребностям.

Должны быть доступны повторно используемые классы, реализующие фундаментальные структуры данных и алгоритмы.

Графика и пользовательские интерфейсы

Многие современные системы ПО интерактивны. При взаимодействии с пользователем широко используется графика и удобный, чаще всего графический интерфейс. Это одна из областей, где ОО-модель оказалась наиболее впечатляющей и полезной. Разработчики должны иметь возможность использовать графические библиотеки для быстрого и эффективного построения интерактивных приложений.

Должны быть доступны повторно используемые классы для разработки приложений, обеспечивающих пользователей приятными графическими пользовательскими интерфейсами.

Механизмы эволюции библиотек

Разработка высококачественных библиотек - долгая и трудная задача. Невозможно гарантировать, что построенные библиотеки сразу будут совершенными. Следовательно, важной проблемой является обеспечение разработчиков библиотеки возможностью обновлять и модифицировать их проекты, не нанося вреда существующим системам, основанным на библиотеках. Этот важный критерий эволюции мы отнесли к категории библиотек, но он относится также и к категории метода и языка.

Должны быть доступны механизмы, облегчающие эволюцию библиотек с минимальными нарушениями работы ПО клиентов.

Механизмы индексации в библиотеках

Еще одна насущная проблема библиотек - это необходимость механизмов идентификации классов для удовлетворения определенных нужд. Этот критерий затрагивает все три категории: библиотеки, язык (поскольку должен быть способ вводить индексирующую информацию в текст каждого класса) и инструментарий (для обработки запросов для классов, удовлетворяющих определенным условиям).

Библиотечные классы должны быть снабжены индексирующей информацией, допускающей поиск, основанный на свойствах.

Продолжение просмотра

Чтобы глубоко понять концепции, предпочтительно читать эту книгу последовательно, однако читатели, желающие дополнить данный теоретический обзор, могут, прежде чем идти дальше, посмотреть, как работает метод на практическом примере. Для этого следует обратиться к [лекции 2](#) курса "Основы объектно-ориентированного проектирования", где рассматривается конкретная задача и сравнивается ОО-решение с решением, использующим традиционную технику.

Изучение этой лекции в основном самодостаточно, так что вы сможете понять решение, не читая промежуточные лекции. Но если вы заглянете туда, то должны обещать вернуться назад, чтобы продолжить изучения материала последовательно, начиная с [лекции 3](#).

Библиографические ссылки и объектные ресурсы

Введение в ОО-критерии - это то место, где стоит привести список работ, дающих хорошие введения в объектную технологию в целом.

[Walden 1995] обсуждает самые важные проблемы объектной технологии, обращая особое внимание на анализ и проектирование, эта книга является, вероятно, лучшим справочным руководством по этому вопросу.

[Page-Jones 1995] дает отличный обзор метода.

[Cox 1990] (первое издание относится к 1986 году) основывается на несколько другом взгляде на объектную технологию; книга послужила распространению ОО-концепций среди широкой публики.

[Henderson-Sellers 1991] (второе издание готовится) дает краткий обзор ОО-идей. Книга предназначена для людей, которых их компания просит "пойти и посмотреть, что это такое объектное программирование", содержит готовые для копирования диапозитивные оригиналы, в некоторых случаях очень ценные. Еще один обзор - это [Eliens 1995].

Словарь Объектной Технологии [Firesmith 1995] дает обширный справочный материал по многим аспектам метода.

Все эти книги в различной степени адресованы людям с техническими наклонностями. Существует также необходимость обучать менеджеров.

Книга [M 1995] выросла из лекции, первоначально предназначеннной для данной книги, и стала полноправной

книгой, в которой объектная технология обсуждается с позиций управляющего персонала. Она начинается небольшой технической презентацией, использующей профессиональные термины, и далее дает анализ вопросов менеджмента (жизненный цикл, управление проектами, политика повторного использования). Еще одна книга с управленческим уклоном, [Goldberg 1995], дает дополнительную перспективу многих важных тем. [Baudoin 1996] делает акцент на вопросах жизненного цикла и важности стандартов.

Возвращаясь к техническим презентациям, три важных книги по ОО-языкам, написанные создателями этих языков, содержат общие методологические обсуждения, делающие их интересными даже для тех читателей, которые не используют эти языки или даже, возможно, неодобрительно к ним относятся. История языков программирования и книг о них показывает, что создатели не всегда наилучшим образом пишут о своих созданиях, но в этом случае они сделали это хорошо. Это книги:

- Simula BEGIN [Birtwistle 1973] (еще два автора являются создателями языка - Nygaard и Dahl.)
- Smalltalk-80: Язык и его реализация [Goldberg 1983].
- Язык программирования C++, второе издание [Stroustrup 1991].

Совсем недавно некоторые начальные учебники по программированию стали использовать ОО-идеи с самого начала, поскольку нет причин позволять "онтогенезу повторять филогенез". Нет необходимости, чтобы бедные студенты, как их предшественники, прошли через всю историю колебаний и ошибок, пока не доберутся до правильных идей. Первый такой текст (насколько я знаю) был [Rist 1995]. Другая хорошая книга, отвечающая тем же потребностям - это [Wiener 1996]. На следующем уровне - учебники для второго курса по программированию. Обсуждение структур данных и алгоритмов, основанное на нотации этой книги - вы найдете в [Gore 1996] и [Wiener 1997]; [Jezequel 1996] представляет принципы ОО-инженерии ПО. Преподавание технологии обсуждается также в [лекции 11](#) курса "Основы объектно-ориентированного проектирования".

Группа новостей Usenet **comp.object**, на нескольких сайтах сети, является естественной площадкой обсуждения многих вопросов объектной технологии. Как и все подобные форумы, это смесь хорошего, плохого и ужасного. Раздел Объектной Технологии в **Computer** (IEEE), который я редактирую с его начала в 1995 году, часто помещает колонки ведущих экспертов.

Журналы, посвященные Объектной Технологии:

- **Journal of Object-Oriented Programming** (первый журнал в этой области, в центре которого технические обсуждения, но они предназначены для широкой публики), **Object Magazine** (более общего диапазона, с некоторыми статьями для менеджеров), **Objekt Spektrum** (на немецком языке), **Object Currents** (онлайн), адрес <http://www.sigs.com>.
- Theory and Practice of Object Systems, архивный журнал.
- L'OBJET (на французском языке), адрес <http://www.tools.com/lobjet>.

Основные международные конференции по объектному ориентированию:

OOPSLA (ежегодная, USA или Canada, см. <http://www.acm.org>); Object Expo (различное время и разные места, см. <http://www.sigs.com>); и TOOLS (Технология ОО-языков и систем), организуемая ISE три раза в год (USA, Europe, Pacific), см. материалы по адресу: <http://www.tools.com>, также является общим ресурсом объектной технологии и вопросов, обсуждаемых в этой книге.

Основы объектно-ориентированного программирования

3. Лекция: Модульность

В лекциях 3-6 будут рассмотрены требования к разработке программного продукта, которые почти наверняка приведут нас к объектной технологии. Чтобы обеспечить расширяемость (extendibility) и повторное использование (reusability), двух основных факторов качества, предложенных в [лекции 1](#), необходима система с гибкой архитектурой, состоящая из автономных программных компонент. Именно поэтому в [лекции 1](#) введен термин модульность (modularity), сочетающий оба фактора.

Модульное программирование ранее понималось как сборка программ из небольших частей, обычно подпрограмм. Но такой подход не может обеспечить реальную расширяемость и повторное использование программного продукта, если не гарантировать, что элементы сборки - модули - являются самодостаточными и образуют устойчивые структуры. Любое достаточно полное определение модульности должно обеспечивать реализацию этих свойств. Таким образом, метод проектирования программного продукта является модульным, если он помогает проектировщикам создать систему, состоящую из автономных элементов с простыми и согласованными структурными связями между ними. Цель этой лекции - детализация этого неформального определения и выяснение того, какими конкретно свойствами должен обладать метод, заслуживающий название "модульного". Наше внимание будет сосредоточено на этапе проектирования, но все идеи применимы и к ранним этапам - анализа и спецификации, также как и этапам разработки и сопровождения. Рассмотрим модульность с разных точек зрения. Введем набор дополнительных свойств: пять критериев (criteria), пять правил (rules) и пять принципов (principles) модульности, обеспечивающих при их совместном использовании выполнение наиболее важных требований, предъявляемых к методу модульного проектирования. Для практикующего разработчика ПО принципы и правила не менее важны, чем критерии. Различие лишь в причинной связи: критерии являются взаимно независимыми (метод может удовлетворять одному из них и в тоже время противоречить оставшимся), в то время как правила следуют из критериев, а принципы следуют из правил. Можно было бы ожидать, что эта лекция начнется с подробного описания того, как выглядит модуль. Но это не так, и для этого есть серьезные основания. Задача этой и двух следующих лекций - анализ свойств, которыми должна обладать надлежащим образом спроектированная модульная структура. Вопросом о виде модулей мы займемся в конце нашего обсуждения, а не в его начале. И пока мы не дойдем до этой точки, слово "модуль" будет означать компонент разбиения рассматриваемой системы. Если вы знакомы с не ОО-методами, то, вероятно, вспомните о подпрограммах, имеющихся в большинстве языков программирования и проектирования, или, быть может, о пакетах (packages) языка Ada и (правда, под другим названием) языка Modula. Наконец, в последующих лекциях наше обсуждение приведет к ОО-виду модуля - классу. Даже если вы уже знакомы с классами и ОО-методами, все же следует прочитать эту лекцию для понимания требований, предъявляемых к классам, - это поможет правильному их конструированию.

Второе [из правил, которые я решил твердо соблюдать] - делить каждую из рассматриваемых мною трудностей на столько частей, сколько потребуется, чтобы лучше их разрешить.

Третье - располагать свои мысли в определенном порядке, начиная с предметов простейших и легко познаваемых, и восходить мало-помалу, как по ступеням, до познания наиболее сложных, допуская существование порядка даже среди тех, которые в естественном ходе вещей не предшествуют друг другу.

Рене Декарт, "Рассуждения о методе" (1637)

Пять критериев

Метод проектирования, который можно называть "модульным", должен удовлетворять пяти основным требованиям:

- Декомпозиции (decomposability).
- Композиции (composability).
- Понятности (understandability).
- Непрерывности (continuity).
- Защищенности (protection).

Декомпозиция

Метод проектирования удовлетворяет критерию Декомпозиции, если он помогает разложить задачу на несколько менее сложных подзадач, объединяемых простой структурой, и настолько независимых, что в дальнейшем можно отдельно продолжить работу над каждой из них.

Такой процесс часто будет циклическим, поскольку каждая подзадача может оказаться достаточно сложной и потребует дальнейшего разложения.

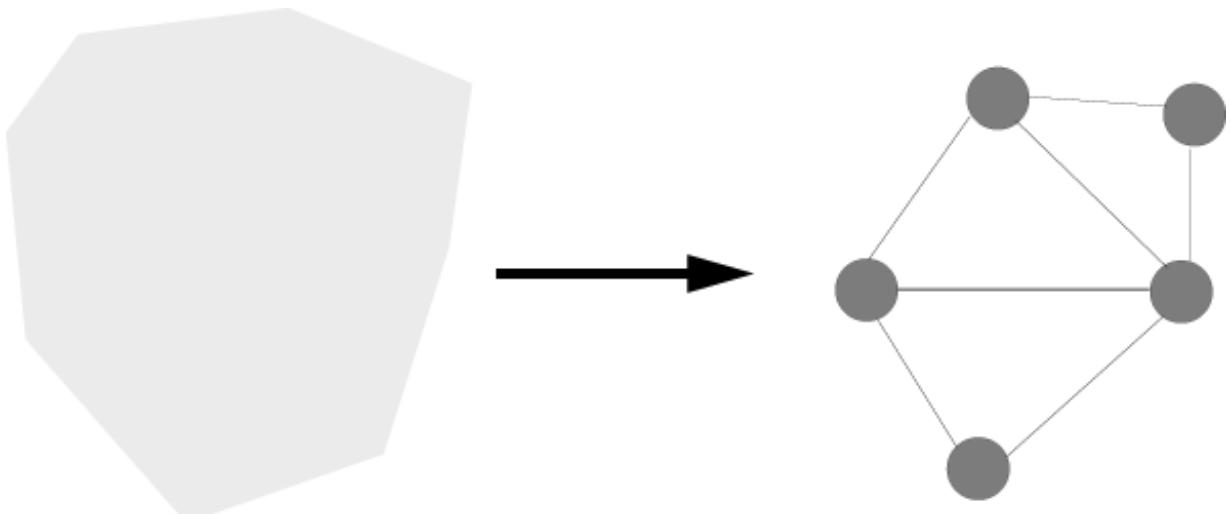


Рис. 3.1. Декомпозиция

Следствием требования декомпозиции является **разделение труда (division of labor)**: как только система будет разложена на подсистемы, работу над ними следует распределить между разными разработчиками или группами разработчиков. Это трудная задача, так как необходимо ограничить возможные взаимозависимости между подсистемами:

- Необходимо свести такие взаимозависимости к минимуму; в противном случае разработка каждой из подсистем будет ограничиваться темпами работы над другими подсистемами.
- Эти взаимозависимости должны быть известны: если не удастся составить перечень всех связей между подсистемами, то после завершения разработки проекта будет получен набор элементов программы, которые, возможно, будут работать каждая в отдельности, но не смогут быть собраны вместе в завершенную систему, удовлетворяющую общим требованиям к исходной задаче.

Наиболее очевидным **примером** обсуждаемого метода¹¹, удовлетворяющим критерию декомпозиции, является метод **нисходящего (сверху вниз) проектирования (top-down design)**. В соответствии с этим методом разработчик должен начать с наиболее абстрактного описания функций, выполняемой системой. Затем последовательными шагами детализировать это представление, разбивая на каждом шаге каждую подсистему на небольшое число более простых подсистем до тех пор, пока не будут получены элементы с настолько низким уровнем абстракции, что становится возможной их непосредственная реализация. Этот процесс можно представить в виде дерева.

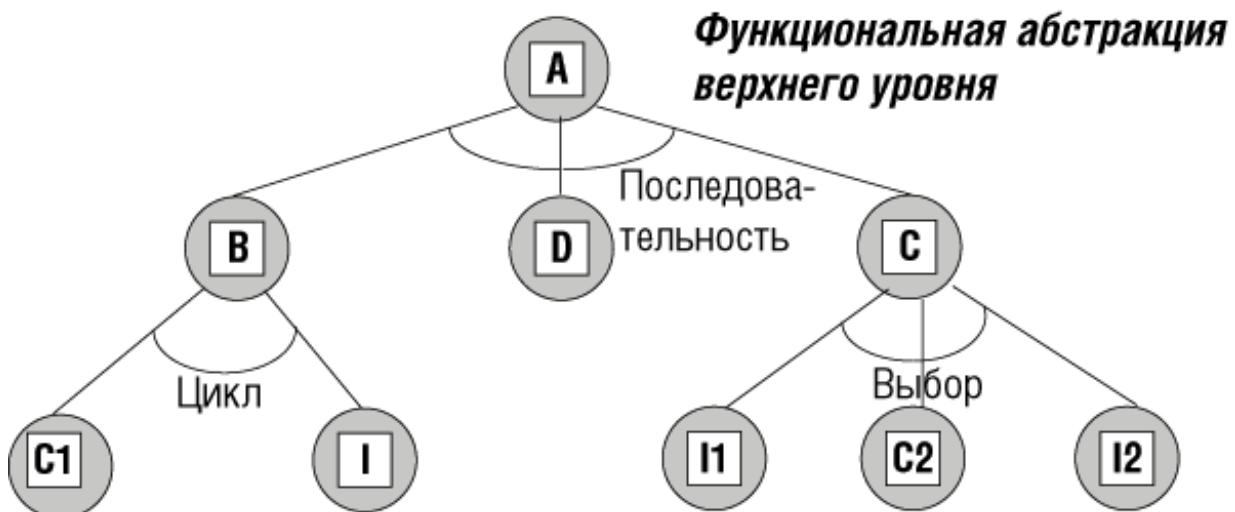


Рис. 3.2. Иерархия нисходящего проектирования

Типичным **контрпримером (counter-example)** является любой метод, предусматривающий включение в разрабатываемую систему модуля глобальной инициализации. Многие модули системы нуждаются в инициализации - открытии файлов или инициализации переменных.

Каждый модуль должен произвести эту инициализацию до начала выполнения непосредственно возложенных на него операций. Могло бы показаться, что все такие действия для всех модулей системы неплохо сосредоточить в одном модуле, который проинициализирует сразу все для всех. Подобный модуль будет обладать хорошей "согласованностью во времени" (temporal cohesion) в том смысле, что все его действия выполняются на одном этапе работы системы. Однако для получения такой "согласованности во времени", придется нарушать автономию других модулей. Придется модулю инициализации дать право доступа ко многим структурам данных, принадлежащим различным модулям системы и требующим специфических

действий по их инициализации. Это означает, что автор модуля инициализации должен будет постоянно следить за структурами данных других модулей и взаимодействовать с их авторами. А это несовместимо с критерием декомпозиции.

Термин "согласованность во времени" пришел из метода, известного как структурное проектирование (см. комментарии к библиографии).

В объектно-ориентированном методе каждый модуль должен самостоятельно инициализировать свои структуры данных.

Модульная Композиция

Метод удовлетворяет критерию Модульной Композиции, если он обеспечивает разработку элементов программного продукта, свободно объединяемых между собой для получения новых систем, быть может, в среде, отличающейся от той, для которой эти элементы первоначально разрабатывались.

Композиция определяет процесс, обратный декомпозиции: элементы программного продукта извлекаются из того контекста, для которого они были первоначально предназначены, для использования их вновь в ином контексте.

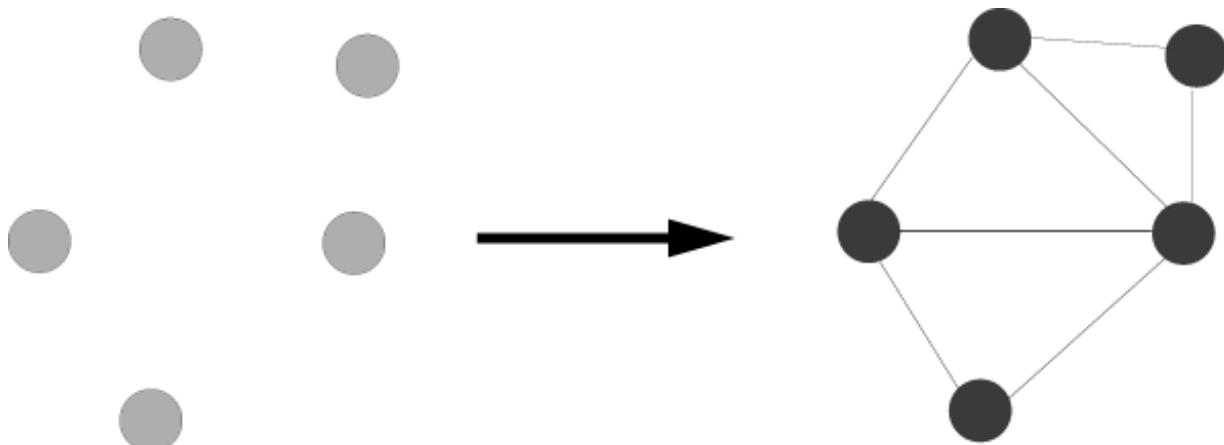


Рис. 3.3. Композиция

Метод модульного проектирования облегчает этот процесс, создавая автономные элементы программного продукта достаточно независимыми от первоначально поставленной задачи, что делает такое извлечение возможным.

Композиция непосредственно связана с повторным использованием. Этот критерий отражает старую мечту - превратить процесс конструирования программного продукта в работу по складыванию кубиков так, чтобы строить программы из фабрично изготовленных элементов.

- **Пример 1: Библиотеки подпрограмм.** Библиотеки подпрограмм создаются как наборы компонуемых элементов. Одной из областей, где они успешно используются, являются численные вычисления, основанные на тщательно подготовленных библиотеках подпрограмм для решения задач линейной алгебры, метода конечных элементов, дифференциальных уравнений и др.
- **Пример 2: Соглашения, принятые в командном языке Shell операционной системы UNIX.** Основные команды системы UNIX оперируют с входным потоком последовательных символов и выдают результат, имеющий такую же стандартную структуру. Потенциальная возможность композиции поддерживается оператором | командного языка "Shell". Запись A | B означает композицию программ. Вначале запускается программа A, ее результаты поступают на вход программы B, начинающей свою работу по завершении работы программы A. Такое системное соглашение благоприятствует композиции программных средств.
- **Контрпример: Препроцессоры.** Общепринятым способом расширения языка программирования, а иногда и преодоления его недостатков, является использование "препроцессора", принимающего входные данные в расширенном синтаксисе и отображающего их в стандартной для этого языка форме. Типичные препроцессоры для Fortran'a и С поддерживают графические примитивы, расширенные управляющие структуры или операции над базами данных. Однако обычно такие расширения не являются взаимно совместимыми; что не позволяет сочетать два таких препроцессора, и приходится выбирать между, например, графикой или базой данных.

Композиция не зависит от декомпозиции. Фактически эти критерии часто противоречат друг другу. Например, метод исходящего проектирования, удовлетворяющий, как уже было показано, критерию декомпозиции, обычно приводит к созданию таких модулей, которые **нелегко** сочетать с модулями, полученными из других источников. При такой декомпозиции модули обычно тесно связаны с теми специфическими требованиями,

которые привели к их разработке, и не могут быть приспособлены к использованию в других условиях. Метод исходящего проектирования не дает рекомендаций по разработке модулей, удовлетворяющих общим требованиям. В нем нет средств такой разработки, он не позволяет ни избежать, ни хотя бы обнаружить программную избыточность модулей, получаемых в различных частях иерархии.

Как композиция, так и декомпозиция являются частью требований к модульному методу проектирования. Неизбежна смесь двух подходов к проектированию: сверху-вниз и снизу-вверх. На этот **принцип дополнительности** обратил внимание Рене Декарт почти четыре столетия тому назад, как видно из сопоставления двух правил его **Рассуждений**, приведенных в эпиграфе этой лекции.

Модульная Понятность

Метод удовлетворяет критерию Модульной Понятности, если он помогает получить такую программу, читая которую можно понять содержание каждого модуля, не зная текста остальных, или, в худшем случае, ознакомившись лишь с некоторыми из них.

Важность этого критерия следует из его влияния на процесс сопровождения программного продукта. Почти все действия по сопровождению программы, как неизбежные, так и не столь неизбежные, связаны с глубоким пониманием ее элементов. Метод едва ли может называться модульным, если тот, кто читает программный текст, не в состоянии понять его смысл.

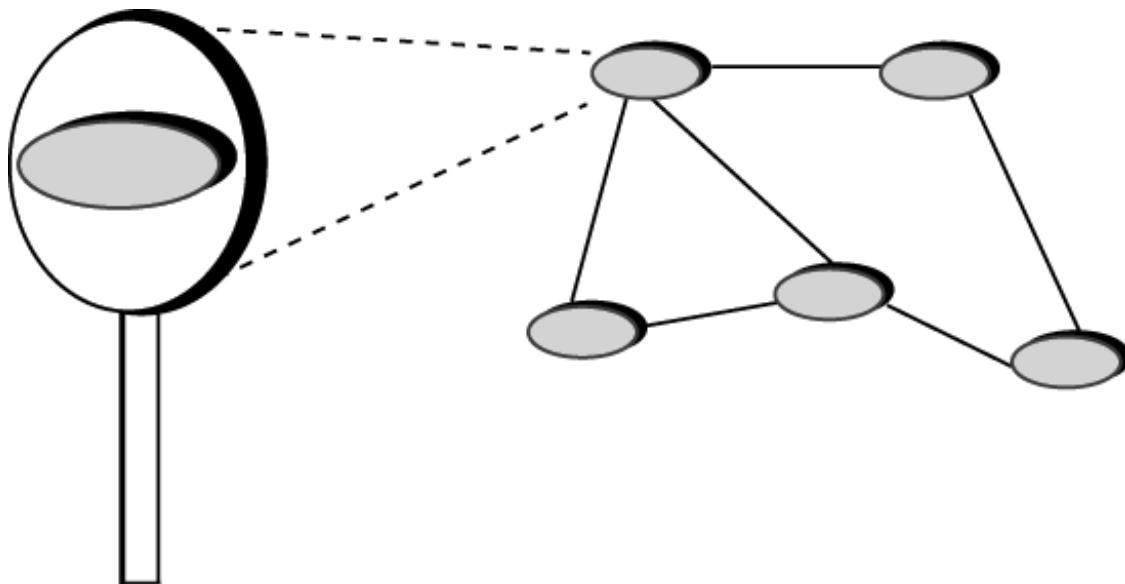


Рис. 3.4. Понятность

Этот критерий, подобно четырем остальным, применим к модулям при описании системы на любом уровне: анализа, проектирования, реализации.

- **Контрпример: последовательные зависимости.** Предположим, что некоторые модули спроектированы таким образом, что они будут правильно функционировать лишь при их запуске в определенном заранее предписанном порядке. Например, B может работать надлежащим образом лишь при запуске его после A и перед C, возможно потому, что эти модули предназначены для использования в "конвейере" Unix, упоминавшемся ранее: A | B | C. В таком случае, по-видимому, трудно понять как работает B, не понимая работу A и C.

В последующих лекциях критерий модульной понятности поможет при рассмотрении двух важных вопросов: как документировать многократно используемые компоненты и как их индексировать, чтобы разработчики программного продукта могли без труда обращаться к ним путем соответствующего запроса. В соответствии с этим критерием информация о компоненте, полезная для документирования или поиска, должна, насколько это возможно, содержаться в тексте самого компонента, тогда средства документирования, индексации или поиска смогут обработать этот компонент и получить требуемую информацию.

Наличие нужной информации в каждом компоненте предпочтительнее хранения ее где-либо в другом месте, например в базе данных для хранения информации о компонентах.

Модульная Непрерывность

Метод удовлетворяет критерию Модульной Непрерывности, если незначительное изменение спецификаций разработанной системы приведет к изменению одного или небольшого числа модулей.

Этот критерий непосредственно связан с критерием расширяемости. Как подчеркивалось в предыдущей

лекции, внесение изменений является неотъемлемой частью процесса разработки программного продукта. Соответствующие требования к программе будут неминуемо изменяться в ходе разработки. Непрерывность означает, что небольшие изменения будут воздействовать только на отдельные модули в структуре системы, а не на всю систему.

Термин "непрерывность" предлагается по аналогии с понятием непрерывной функции в математическом анализе. Математическая функция является непрерывной, если (неформально) малое изменение аргумента приводит к пропорционально малому изменению результата. В нашем случае роль функции играет метод конструирования программного продукта, который может рассматриваться как механизм, получающий на входе спецификации и возвращающий в качестве результата систему, удовлетворяющую заданным требованиям:

Метод_конструирования_ПО: Спецификации -> Система

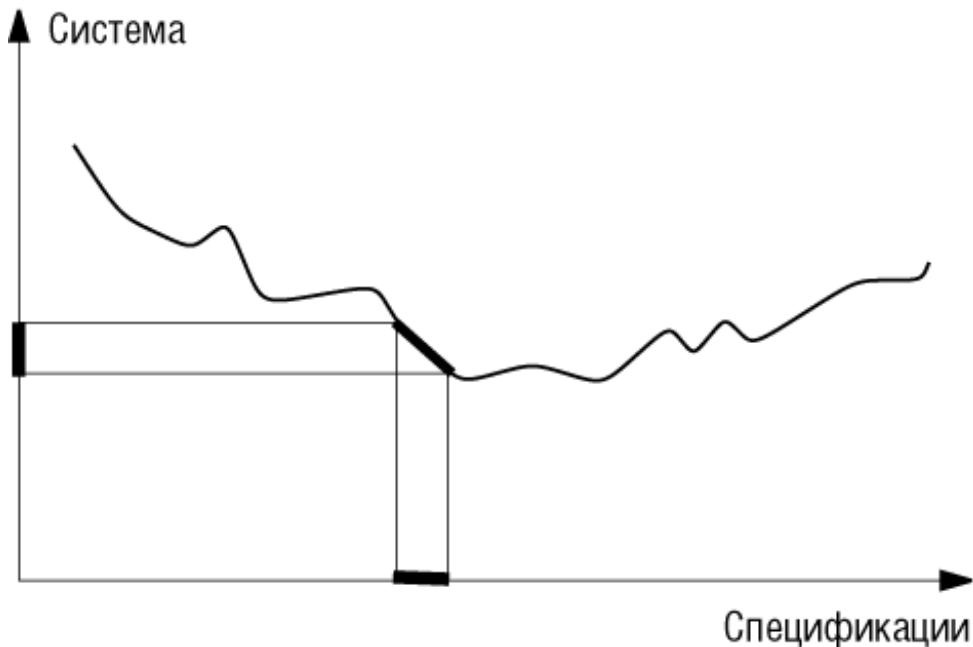


Рис. 3.5. Непрерывность

Этот математический термин введен здесь лишь по аналогии, поскольку не существует формального понятия размера спецификации и программы. Можно было бы ввести приемлемую меру для определения "небольших" или "больших" изменений программы, но дать подобное определение для спецификаций к программе это уже настоящая проблема. Однако если не претендовать на строгость, то такое интуитивно понятное определение будет соответствовать необходимому требованию к любому модульному методу.

- **Пример 1:именованные константы²⁾.** Разумный стиль не допускает в программе констант, заданных литералами. Вместо этого следует пользоваться именованными константами, значения которых даются в их определениях (**constant** в языках Pascal или Ada, макрокоманды препроцессоров в языке C, **PARAMETER** в языке Fortran 77, атрибуты констант в обозначениях этого курса). Если значение изменяется, то следует лишь внести единственное изменение в определение константы. Это простое, но важное правило является разумной мерой обеспечения непрерывности, потому что значения констант, несмотря на их название, довольно часто могут изменяться.
- **Пример 2: принцип Унифицированного Доступа.** Еще одно правило требует единой нотации при вызове свойств объекта независимо от того, представляют они обычные или вычислимые поля данных.
- **Контрпример 1: использование физического представления информации.** Метод, в котором разрабатываемые программы согласуются с физической реализацией данных, будет приводить к конструкциям, весьма чувствительным к незначительным изменениям окружения.
- **Контрпример 2: статические массивы.** Такие языки, как Fortran или стандартный Pascal, в которых не допускаются динамические массивы, границы которых становятся известными лишь во время выполнения программы, существенно усложняют развитие системы.

Модульная Защищенность

Метод удовлетворяет критерию Модульной Защищенности, если он приводит к архитектуре системы, в которой аварийная ситуация, возникшая во время выполнения модуля, ограничится только этим модулем, или, в худшем случае, распространится лишь на несколько соседних модулей.

Вопрос об отказах и ошибках является основным в программной инженерии. Сейчас речь идет об ошибках периода исполнения программы, связанных с аппаратными прерываниями, ошибочными входными данными

или исчерпанием необходимых ресурсов (например, из-за недостаточного объема памяти). Критерий защищенности направлен не на предотвращение или исправление ошибок, а на проблему, непосредственно связанную с модульностью - распространением ошибок в модульной системе.

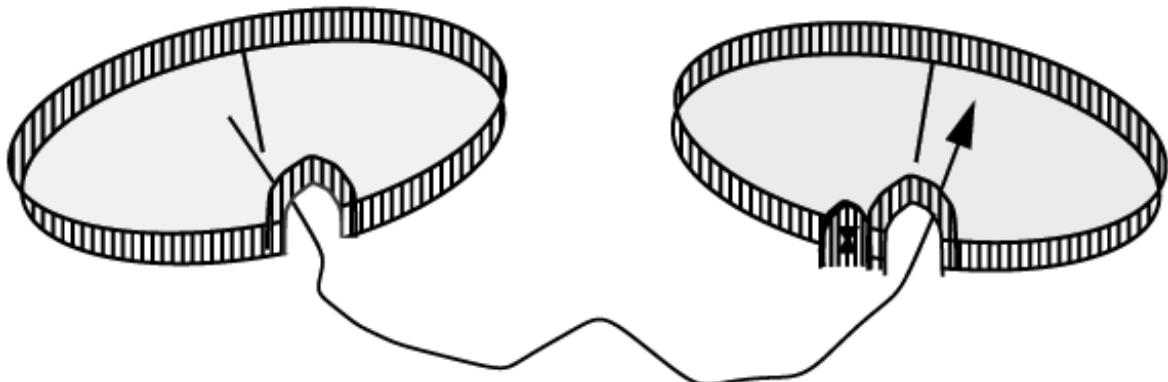


Рис. 3.6. Нарушение защищенности

- **Пример: проверка достоверности входных данных в источнике.** Метод, требующий от каждого модуля, вводящего данные, проверку их достоверности, пригоден для реализации модульной защищенности.³⁾
- **Контрпример: недисциплинированные (undisciplined) исключения.** (Об обработке исключений см. [лекцию 12](#)) Такие языки как PL/I, CLU, Ada, C++ и Java поддерживают понятие исключения (exception). Исключение это ситуация, при которой программа не может нормально выполняться. Исключение "возбуждается" ("raised") некоторой командой модуля, и в результате операционной системе посыпается специальный сигнал. Обработчик исключения (exception handler) может находиться в одном или нескольких модулях, расположенных в, возможно, удаленной части системы. Детали этого механизма отличаются в разных языках программирования; Ada или CLU являются более строгими в этом отношении, чем PL/I. Такие средства контроля ошибок позволяют отделить алгоритмы для обычных случаев от алгоритмов обработки ошибок. Но ими следует пользоваться осторожно, чтобы не нарушить модульную защищенность. В [лекции 12](#), посвященной исключениям, рассматривается проектирование дисциплинированного (disciplined) механизма исключений, удовлетворяющего критерию защищенности.

Пять правил

Из рассмотренных критериев следуют пять правил, которые должны соблюдаться, чтобы обеспечить модульность:

- Прямое отображение (Direct Mapping).
- Минимум интерфейсов (Few Interfaces).
- Слабая связность интерфейсов (Small interfaces - weak coupling).
- Явные интерфейсы (Explicit Interfaces).
- Скрытие информации (инкапсуляция) (Information Hiding).

Первое правило касается отношения между внешней системой и ПО. Следующие четыре правила касаются общей проблемы - как модули общаются между собой. Для получения хорошей модульной архитектуры необходим управляемый и строгий метод обеспечения межмодульных связей.

Прямое отображение

Любая прикладная система стремится удовлетворить потребности некоторой проблемной области. Если имеется хорошая модель для описания этой проблемной области, то желательно обеспечить четкое отображение структуры проблемы, описываемой моделью, на структуру системы. Из этого следует первое правило:

Модульная структура, создаваемая в процессе конструирования ПО, должна оставаться совместимой с модульной структурой, создаваемой в процессе моделирования проблемной области.

Эта рекомендация следует, в частности, из двух критериев модульности:

- Непрерывность: отслеживание модульной структуры проблемы в структуре решения облегчит оценку и ограничит последствия изменений.
- Декомпозиция: если уже была проделана некоторая работа по анализу модульной структуры проблемной области, то это может явиться хорошей отправной точкой для разбиения программы на модули.

Минимум интерфейсов

Правило Минимума Интерфейсов ограничивает общее число информационных каналов, связывающих модули системы:

Каждый модуль должен поддерживать связь с возможно меньшим числом других модулей.

Связь между модулями может осуществляться различными способами. Модули могут вызывать друг друга (если они являются процедурами), совместно использовать структуры данных и так далее. Правило Минимума Интерфейсов ограничивает число таких связей.

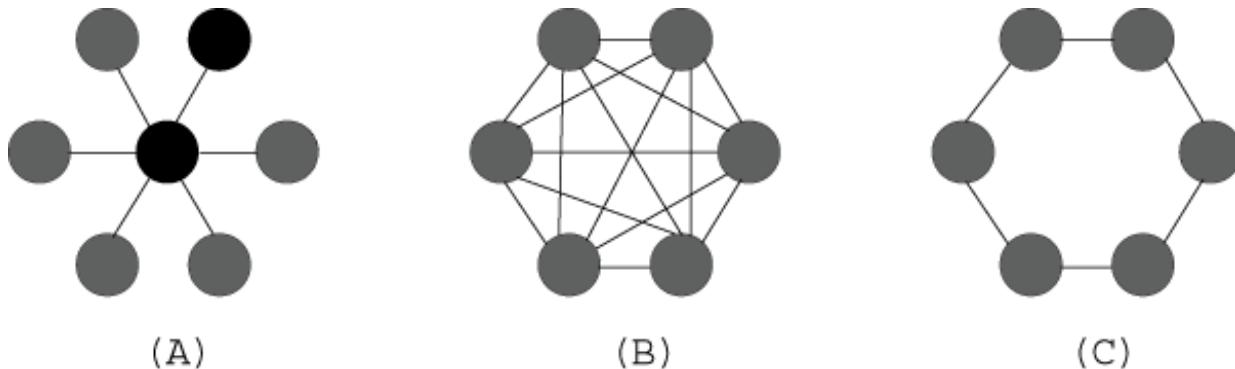


Рис. 3.7. Виды структур межмодульных связей

В системе, составленной из n модулей, число межмодульных связей должно быть намного ближе к минимальному значению $n - 1$, как показано на рисунке (A), чем к максимальному $\frac{(n - 1)n}{2}$, как показано на рисунке (B).

Это правило следует, в частности, из критериев непрерывности и защищенности: если между модулями имеется слишком много взаимосвязей, то влияние изменения или ошибки может распространяться на большое число модулей. Оно также имеет отношение к критериям композиции (чтобы модуль мог использоваться в новой программной среде, он не должен зависеть от слишком большого числа других модулей), понятности и декомпозиции.

Вариант (A) на последнем рисунке показывает, как добиться минимального числа связей, $n - 1$, с помощью весьма централизованной структуры: один основной модуль, а все остальные общаются только с ним. Но имеются намного более "демократические" структуры, такие как (C), содержащие почти такое же число связей. В этой схеме каждый модуль непосредственно общается с двумя ближайшими соседями, центральной власти здесь нет. Такой подход к конструированию программы кажется сначала немного неожиданным, поскольку он не согласуется с традиционной моделью исходящего проектирования. Но он может приводить к надежным, расширяемым решениям. Это именно такой вид структуры, к созданию которой будет стремиться ОО-метод при его разумном применении.

Слабая связность интерфейсов

Правило Слабой связности интерфейсов относится к размеру передаваемой информации, а не к числу связей:

Если два модуля общаются между собой, то они должны обмениваться как можно меньшим объемом информации.

Инженер-электрик сказал бы, что каналы связи между модулями должны иметь ограниченную полосу пропускания:



Рис. 3.8. Канал связи между модулями

Требование Слабой связности интерфейсов следует, в частности, из критериев непрерывности и защищенности.

Особо примечательным **контрпримером** является конструкция из языка Fortran, знакомая некоторым читателям как "общий блок для мусора" ("garbage common block"). Общим блоком в Fortran'e является

директива вида:

```
COMMON /общее_имя/ переменная1 : переменнаяn.
```

Переменные, перечисленные в блоке, доступны во всех модулях, содержащих директиву COMMON с тем же общим_именем. Нередко встречаются программы на Fortran'e, в которых каждый модуль содержит одну и ту же огромную директиву COMMON с перечислением всех существенных переменных и массивов, так что каждый модуль может непосредственно обращаться к любым данным программы.

Возникающие здесь затруднения состоят в том, что любой из модулей может неправильно использовать общие данные, а модули тесно связаны между собой; поэтому проблемы реализации непрерывности (распространение изменений) и защищенности (распространение ошибок) являются чрезвычайно трудно разрешимыми. Тем не менее, эта освященная годами техника все еще остается любимой многими программистами, хотя и ведет к длительным ночных отладочным бдениям.

Разработчики, пользующиеся языками с вложенными структурами, испытывают такие же затруднения. При наличии блочной структуры, введенной в языке Algol и поддерживаемой, в более ограниченной форме, в языке Pascal, можно "вкладывать" блоки, содержащиеся внутри пар begin ... end, внутрь других блоков. К тому же каждый блок может вводить свои собственные переменные, которые имеют смысл лишь в синтаксическом контексте (syntactic scope) этого блока. Например:

```
local -- Начало блока B1
      x, y: INTEGER
do
  ... Команды блока B1 ...
  local -- Начало блока B2
        z: BOOLEAN
  do
    ... Команды блока B2 ...
  end -- Конец блока B2
  local -- Начало блока B3
        y, z: INTEGER
  do
    ... Команды блока B3 ...
  end -- Конец блока B3
  ... Команды блока B1 (продолжение) ...
end -- Конец блока B1
```

Переменная x доступна для всех команд в этом фрагменте программы, в то время как области действия двух переменных с именем z (одна типа BOOLEAN, другая типа INTEGER) ограничены блоками B2 и B3 соответственно. Подобно x, переменная y объявлена на уровне блока B1, но ее область действия не включает блока B3, где другая переменная с тем же именем и тем же типом локально имеет приоритет над самой ближней внешней переменной y. В Pascal'e этот вид блочной структуры существует лишь для блоков, связанных с подпрограммами (процедурами и функциями).⁴⁾

При наличии блочной структуры, эквивалентом "мусорного" общего блока Fortran'a является объявление всех переменных на самом верхнем (глобальном) уровне. В языках на основе языка С таким эквивалентом является объявление всех переменных внешними (external). (О кластерах см. [лекции 10](#) курса "Основы объектно-ориентированного проектирования". Альтернатива вложеннности рассматривается в разделе "Архитектурная роль выборочного экспорта (selective exports)".)

Использование блочной структуры является оригинальной идеей, но это может приводить к нарушению правила Слабой связности Интерфейсов. По этой причине мы будем воздерживаться от применения ее в объектно-ориентированной нотации, развиваемой далее в этом курсе. Язык Simula - объектно-ориентированная производная от Algol'a - поддерживает блочную структуру классов. Опыт работы с ним показал, что способность создавать вложенные классы является излишней при наличии некоторых возможностей, обеспечиваемых механизмом наследования. Структура объектно-ориентированного программного обеспечения содержит три уровня: система является набором кластеров; кластер является набором классов; класс является набором компонент (атрибутов (attributes) и методов (routines)). Кластеры скорее организационное средство, чем лингвистическая конструкция, могут быть вложенными, что позволяет руководителю проекта структурировать большую систему на любое необходимое число уровней; но классы, как и компоненты, имеют одноуровневую плоскую (flat) структуру, поскольку вложенность на любом из этих уровней приведет к излишнему усложнению.

Явные интерфейсы

Четвертое правило является еще одним шагом к укреплению тоталитарного режима в обществе модулей: требуется не только, чтобы любые переговоры ограничивались лишь несколькими участниками и были

немногословными; необходимо, чтобы такие переговоры были публичными и гласными!

Всякое общение двух модулей А и В между собой должно быть очевидным и отражаться в тексте А и/или В.

За этим правилом стоят критерии:

- Декомпозиции и композиции. Если нужно разложить модуль на несколько подмодулей или компоновать его с другими модулями, то любая внешняя связь должна быть ясно видна.
- Непрерывности. Должно быть очевидно, какие элементы могут быть затронуты возможным изменением.
- Понятности. Как можно истолковывать действие модуля А, если на его поведение может косвенным образом влиять модуль В?

Одной из проблем, возникающих при применении правила Явных Интерфейсов, является то, что межмодульная связь может осуществляться не только через вызовы процедур; источником косвенной связи может быть, например, совместное использование данных (data sharing):

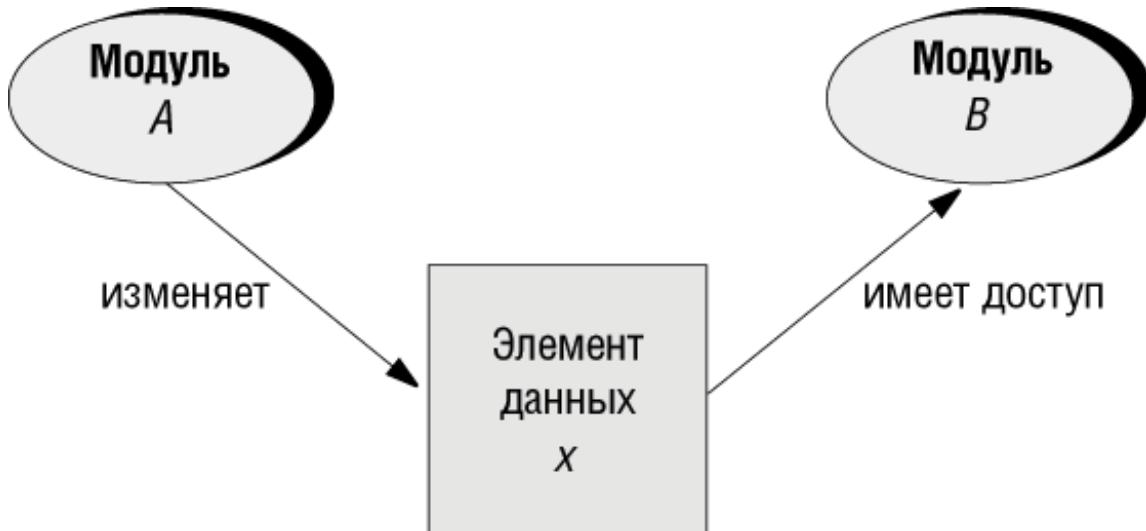


Рис. 3.9. Совместное использование данных

Предположим, что модуль А изменяет данные, а модуль В использует тот же элемент данных х. Тогда А и В оказываются фактически связанными через х, хотя между ними может и не быть явной взаимосвязи, например, вызова процедуры.

Скрытие информации

Правило Скрытия Информации можно сформулировать следующим образом:

Разработчик каждого модуля должен выбрать некоторое подмножество свойств модуля в качестве официальной информации о модуле, доступной авторам клиентских модулей.

Применение этого правила означает, что каждый модуль известен всем остальным (то есть разработчикам других модулей) через некоторое официальное описание, или так называемые **общедоступные (public)** свойства.

Конечно, таким описанием может быть весь текст модуля (текст программы, текст проекта): он и обеспечивает правильное представление о модуле, поскольку **это и есть** модуль! Но правило Скрытия Информации устанавливает, что в общем случае это не обязательно: описание должно включать лишь **некоторые** из свойств модуля. Остальные свойства должны оставаться не общедоступными, или **закрытыми (secret)**. Вместо терминов - общедоступные и закрытые свойства - используются также термины: экспортируемые и частные (скрытые) (private) свойства. Общедоступные свойства модуля известны также как **интерфейс (interface)** модуля (не следует путать с пользовательским интерфейсом системы программирования).

В основе правила Скрытия Информации лежит критерий непрерывности. Предположим, что в некотором модуле происходят изменения, касающиеся лишь его скрытых элементов и не затрагивающие общедоступных свойств; тогда на другие обращающиеся к нему модули, называемые его **клиентами**, эти изменения не действуют. Чем меньше общедоступная часть, тем больше шансов на то, что изменения в модуле будут содержаться в его скрытой части.

Можно изобразить модуль, поддерживающий правило Скрытия Информации, в виде айсберга; лишь его верхушка - интерфейс - видна клиентам.



Рис. 3.10. Модуль в условиях скрытия информации

В качестве характерного примера рассмотрим процедуру поиска по ключу атрибутов, хранящихся в таблице, такой как картотека личного состава или таблица идентификаторов компилятора. Эта процедура существенно зависит от способа представления таблицы - последовательный массив или файл, хэш-таблица, двоичное или индексное (B-Tree) дерево и т.д. Скрытие информации означает, что выбранный способ реализации таблицы не влияет на использование такой процедуры. Модули-клиенты не должны страдать от каких-либо изменений в реализации программы.

Правило скрытия информации придает особое значение отделению описания функции от ее реализации, - что делает функция и как она это делает - разные вещи. Помимо критерия непрерывности, это правило связано также с критериями декомпозиции, композиции и понятности. Нельзя независимо разрабатывать модули системы, комбинировать существующие модули или понимать действие отдельных модулей, если неизвестно в точности, что каждый из них может (или не может) ожидать от других модулей.

Какие же из свойств модуля должны быть общедоступными, а какие - скрытыми? Как правило, в общедоступную часть следует включать функциональность, заданную спецификацией модуля, а все, что связано с реализацией этих функциональных возможностей, должно быть скрыто, предохраняя другие модули от последующих изменений реализации программы.

Однако эта рекомендация является нечеткой, так как не дано определение спецификации (specification) и реализации (implementation). Действительно, можно поддаться искушению, изменив определение на прямо противоположное, и утверждать, что спецификация состоит из общедоступных свойств модуля, а реализация - из его скрытых свойств! ОО-подход обеспечит намного более точные рекомендации на основе теории абстрактных типов данных.(См. [лекцию 6](#), в частности "Абстрактные типы данных и скрытие информации".)

Для понимания смысла скрытия информации и применения этого правила должным образом, важно избежать широко распространенного неверного толкования. Несмотря на свое название, скрытие информации не означает защиты информации в смысле обеспечения секретности - запрещения авторам модулей-клиентов доступа к тексту модуля-поставщика (supplier module). На самом деле авторы модулей-клиентов имеют доступ ко всем интересующим их подробностям. В некоторых случаях было бы разумно запретить им это, но такое решение, которое, конечно, может принять руководство проекта, не следует из правила скрытия информации. Скрытие информации, как техническое требование, означает лишь, что модули-клиенты (независимо от того, разрешен ли их авторам доступ к скрытым свойствам модулей-источников) должны рассчитывать только на общедоступные свойства модуля-поставщика. Точнее говоря, должно быть невозможным создание клиентских модулей, правильное функционирование которых зависело бы от скрытой информации.

При формальном подходе к разработке программного обеспечения это определение можно было бы сформулировать следующим образом. Для доказательства корректности модуля необходимо сделать некоторые допущения о свойствах его модулей-поставщиков. Скрытие информации означает, что

доказательство может основываться лишь на общедоступных свойствах поставщиков и никоим образом - на их скрытых свойствах.

Рассмотрим вновь пример модуля, обеспечивающего реализацию алгоритма поиска в таблице. Некоторый модуль-клиент, который может быть частью системы, реализующей работу с электронными таблицами, обращается к нашему модулю для поиска в таблице определенного элемента. Предположим далее, что наш алгоритм поиска основан на реализации дерева двоичного поиска, но это его свойство является скрытым - и не отражено в интерфейсе. Автор модуля поиска в таблице может сам решать, сообщать ли автору программы электронных таблиц то, как реализован алгоритм поиска. Это решение относится к управлению проектом или, возможно (в случае серийно выпускаемого программного обеспечения), является решением на уровне маркетинга; так или иначе, это не связано с вопросами скрытия информации.

Скрытие информации означает нечто другое: **даже если автор программы электронных таблиц знает** о том, что поиск основан на дереве двоичного поиска, ему не следует составлять такой модуль-клиент, который правильно функционирует лишь при этой реализации поиска - и перестанет работать при замене алгоритма поиска на какой-либо другой, например, на поиск с хешированием.

Одной из причин вышеупомянутого недопонимания является сам термин "скрытие информации" ("information hiding"), который наводит на мысль о защите физического характера. В этом смысле предпочтительным, по-видимому, мог бы явиться термин "инкапсуляция" ("encapsulation"), иногда используемый в качестве синонима скрытию информации, однако в нашем обсуждении будет по-прежнему использоваться общий термин "скрытие информации".

Из этого обсуждения следует, что ключом к скрытию информации являются не решения по организации доступа к исходному тексту модуля в рамках управления проектом или маркетинговой политики, а строгие языковые правила, определяющие, какие права на доступ к модулю следуют из свойств его источника. В следующей лекции показано, что первые шаги в этом направлении реализованы в таких "языках с инкапсуляцией" как Ada и Modula-2. Объектно-ориентированная технология программирования приведет к более полному решению проблемы.⁵⁾

Пять принципов

Из предыдущих правил и, косвенным образом, из критериев следуют пять принципов конструирования ПО:

- Принцип Лингвистических Модульных Единиц (Linguistic Modular Units).
- Принцип Самодокументирования (Self-Documentation).
- Принцип Унифицированного Доступа (Uniform Access).
- Принцип Открыт-Закрыт (Open-Closed).
- Принцип Единственного выбора (Single Choice).

Лингвистические Модульные Единицы

Принцип Лингвистических Модульных Единиц утверждает, что формализм описания ПО на различных уровнях (спецификации, проектирования, реализации) должен поддерживать модульность:

Принцип Лингвистических Модульных Единиц

Модули должны соответствовать синтаксическим единицам используемого языка.

Упомянутым выше языком может быть язык программирования, язык проектирования, язык оформления технических требований и т. д. В случае языка программирования модули должны независимо компилироваться.

Этот принцип на любом уровне (анализа, проектирования, реализации) не допускает объединения метода, исходящего из концепции модульности, и языка, не содержащего соответствующих модульных конструкций. В самом деле, нередко встречаются фирмы, которые на этапе проектирования применяют некие методологические подходы, например используя модули языка Ada, но затем реализуют свои замыслы в таком языке программирования, как Pascal или C, не поддерживающим эти подходы. Такой подход нарушает некоторые из критериев модульности:

- Непрерывность: если границы модуля в окончательном тексте программы не соответствуют логической декомпозиции спецификации или проекта, то при сопровождении системы и ее эволюции будет затруднительно или даже невозможно поддерживать совместимость различных уровней. Изменение спецификации можно считать небольшим, если оно затрагивает спецификацию лишь небольшого числа модулей. Для обеспечения "непрерывности" должно иметь место прямое соответствие между спецификацией, проектом и модулями реализации.
- Прямое отображение: необходимо поддерживать явное соответствие между структурой модели и структурой решения. Для этого необходимо иметь явную синтаксическую идентификацию

концептуальных единиц модели и решения, отражающее разбиение, предусмотренное методом разработки.

- Декомпозиция: для разбиения системы на отдельные задачи необходимо быть уверенным, что результатом решения каждой из задач явится четко ограниченная синтаксическая единица; на этапе реализации эти программные компоненты должны быть раздельно компилируемыми.
- Композиция: что же, кроме модулей с однозначно определенными синтаксическими границами, можно объединять между собой?
- Защищенность: лишь в случае, если модули синтаксически разграничены, можно надеяться на возможность контроля области действия ошибок.

Самодокументирование

Подобно правилу Скрытия Информации, принцип Самодокументирования определяет, как следует документировать модули:

Принцип Самодокументирования

Разработчик модуля должен стремиться к тому, чтобы вся информация о модуле содержалась в самом модуле.

Обычно реализации этого принципа мешает общепринятое положение, согласно которому информацию о модуле помещают в отдельные проектные документы.

Документация, рассматриваемая здесь, является внутренней документацией о компонентах ПО.

Пользовательская документация о выпущенном программном продукте может быть отдельным документом, реализованном в виде печатного текста, либо размещенном на CD-ROM или страницах в Интернете. Как отмечалось при обсуждении вопроса о качестве программного обеспечения, следствием общего принципа самодокументирования является наблюдаемая сейчас тенденция к большему использованию средств диалоговой оперативной подсказки. (См."О документировании" [лекция 1](#))

Наиболее очевидным обоснованием необходимости принципа Самодокументирования является критерий модульной понятности. По-видимому, однако, более важным является то, что этот принцип помогает реализации критерия непрерывности. Если программное обеспечение и документацию к нему рассматривать как отдельные объекты, то трудно гарантировать, что они будут оставаться совместимыми - будут синхронно изменяться при всех изменениях системы. Однако если хранить все в одном месте, то это, хотя и не дает полную гарантию, но все же поможет поддерживать совместимость.

Этот принцип, безусловно, на первый взгляд, противоречит многому из того, что обычно рекомендуется к практическому применению в литературе по разработке ПО. Преобладает мнение, что разработчик ПО - инженер-программист - должен делать то, чем, по-видимому, обязаны заниматься остальные инженеры: производить килограмм бумаги на каждый грамм фактически создаваемой продукции. Предложение вести запись процесса разработки ПО является неплохим советом, но из этого вовсе не следует, что программа и документация к ней являются разными продуктами.

Такой подход игнорирует характерное свойство ПО, которое здесь неоднократно обсуждается: возможность его изменения. Если рассматривать программу и документацию к ней как два разных продукта, то вскоре можно оказаться в ситуации, когда в документации утверждается одно, а программа делает нечто иное. А ведь наличие неправильной документации намного хуже, чем ее отсутствие.

Главным достижением последних нескольких лет явилось появление стандартов качества ПО. Разработаны сертификаты ISO, стандарт "2167" и его преемники, Модель Полноты Потенциала (Capability Maturity Model), предложенная Институтом программной инженерии (Software Engineering Institute). Но поскольку они брали начало из моделей, используемых в других отраслях знания, они наделены обширным "хвостом" бумажной документации. Некоторые из этих стандартов могли бы оказать значительно большее влияние на качество ПО, (помимо того, что они дают администраторам программного продукта средство для оправданий в случае последующих эксплуатационных неполадок) если бы они включали принцип Самодокументирования.

В этом курсе следствием принципа Самодокументирования является метод документирования классов - модулей при ОО-конструировании ПО, предусматривающий включение документации в сам модуль. Это вовсе не означает, что **сам модуль является** своей документацией: текст программы обычно содержит слишком много подробностей (это и явилось доводом в пользу скрытия информации). Просто модуль должен **содержать** свою документацию. (См. "Использование утверждений класса (assertions) для документирования" в [лекции 11](#). См. также [лекция 5](#) курса "Основы объектно-ориентированного проектирования" и последние два упражнения в ней.)

При таком подходе ПО превращается в единственный программный продукт, обеспечивающий его различные **представления или облики (views)**. Один облик, пригодный для компиляции и выполнения, - полный исходный текст модуля. Другой - документация, задающая абстрактный интерфейс модуля, позволяющий

разработчикам программного обеспечения создавать модули-клиенты, не знакомясь с содержанием исходного модуля, что соответствует правилу Скрытия Информации. Возможны и другие представления.

Унифицированный Доступ

Хотя вначале может показаться, что принцип Унифицированного Доступа направлен лишь на решение проблем, связанных с принятой нотацией, в действительности он задает правило проектирования, влияющее на многие аспекты ОО-разработки ПО. Принцип следует из критерия Непрерывности; его можно рассматривать и как частный случай правила Скрытия Информации.⁶⁾

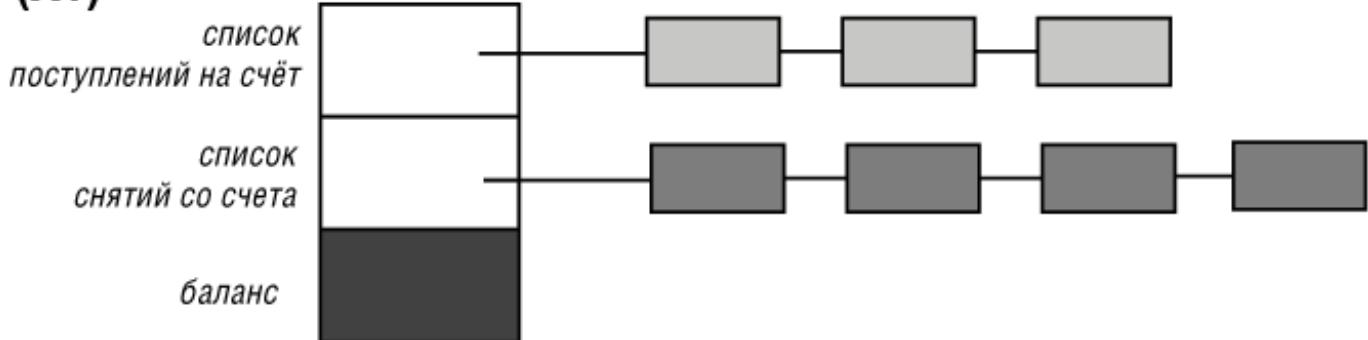
Пусть x - имя, используемое для доступа к некоторому элементу данных, который в последующем будем называть объектом. Пусть f - имя компонента (feature), применимого к x . Под компонентом понимается некоторая операция; далее этот термин будет определен подробнее. Например, x может быть переменной, представляющей счет в банке, а f - компонент, задающий текущий баланс этого счета (account's current balance). Унифицированный Доступ направлен на решение вопроса о том, какой должна быть нотация, задающая применение f к x , не содержащая каких-либо преждевременных обязательств по способу реализации f .

Во многих языках проектирования и программирования выражение, описывающее применение f к x , зависит от реализации f , выбранной разработчиком. Это может быть свойство, хранимое вместе с x , или метод, вызываемый всякий раз, когда это требуется. В примере с банковскими счетами и остатками на счетах возможно использование обоих подходов:

- **A1** Можно представить баланс банковского счета в виде одного из полей записи, описывающей каждый счет. При использовании такого подхода каждая банковская операция, изменяющая баланс, должна предусматривать корректировку соответствующего поля.
- **A2** Можно определить функцию, вычисляющую баланс на основании других полей этой записи, например полей, представляющих списки денежных сумм, снятых со счета и внесенных на счет. При использовании такого подхода значение баланса не сохраняется, а вычисляется по запросу.

В общепринятой нотации таких языков, как Pascal, Ada, C, C++ и Java используется обозначение $x.f$ для случая **A1** и $f(x)$ для случая **A2**.

(A1)



(A2)

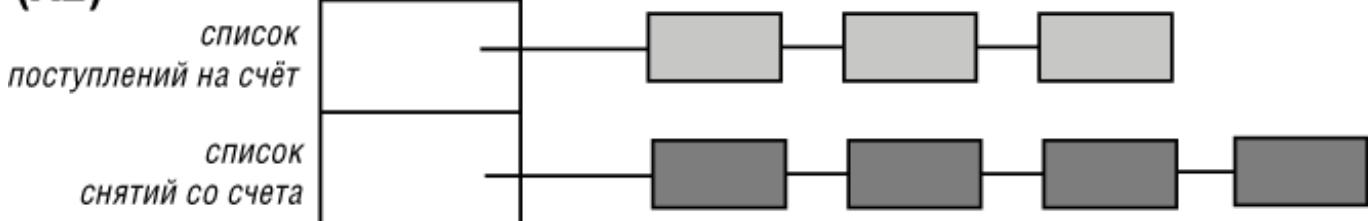


Рис. 3.11. Два представления банковского счета

Выбор между представлениями **A1** и **A2** это компромисс между "памятью и временем": первое экономит на вычислениях, а второе - на памяти. Решение о выборе одного из вариантов является типичным примером решения, изменяемого разработчиком, по крайней мере один раз за время существования проекта. Поэтому с целью поддержания непрерывности желательно иметь нотацию для доступа к компоненту, не зависящую от выбора одного из двух представлений. Если способ реализации x 'ов на некотором этапе разработки проекта будет изменен, то это не потребует изменений в модулях, использующих вызов f .

Мы рассмотрели пример принципа Унифицированного Доступа. В общем виде принцип можно сформулировать так:

Принцип Унифицированного Доступа

Все службы, предоставляемые модулем, должны быть доступны в унифицированной нотации, которая не подведет вне зависимости от реализации, использующей память или вычисления.

Этому принципу удовлетворяют немногие языки. Старейшим из них был Algol W, в котором как вызов функции, так и доступ к полю записывались в виде $a(x)$. Первым из ОО-языков, удовлетворяющих Принципу Унифицированного Доступа, был язык Simula 67, использовавший обозначение $x.f$ в обоих случаях. Нотация, предлагаемая в лекциях 7-18 этого курса, будет поддерживать такое соглашение.

Открыт-Закрыт

Любой метод модульной декомпозиции должен удовлетворять принципу семафора: Открыт-Закрыт:

Принцип Открыт-Закрыт

Модули должны иметь возможность быть как открытыми, так и закрытыми.

Противоречие является лишь кажущимся, поскольку термины соответствуют разным целевым установкам:

- Модуль называют открытым, если он еще доступен для расширения. Например, имеется возможность расширить множество операций в нем или добавить поля к его структурам данных.
- Модуль называют закрытым, если он доступен для использования другими модулями. Это означает, что модуль (его интерфейс - с точки зрения скрытия информации) уже имеет строго определенное окончательное описание. На уровне реализации закрытое состояние модуля означает, что модуль можно компилировать, сохранять в библиотеке и делать его доступным для использования другими модулями (его **клиентами**). На этапе проектирования или спецификации закрытие модуля означает, что он одобрен руководством, внесен в официальный репозиторий утвержденных программных элементов проекта - **базу проекта** (project **baseline**), и его интерфейс опубликован в интересах авторов других модулей.

Необходимость закрывать модули и необходимость оставлять их открытыми вызываются разными причинами. Для разработчиков ПО естественным состоянием модуля является его открытость, поскольку почти невозможно заранее предусмотреть все элементы - данные, операции - которые могут потребоваться в процессе создания модуля. Поэтому разработчики стараются сохранять гибкость ПО, допускающую последующие изменения и дополнения. Но необходимо, особенно с точки зрения руководителя проекта, закрывать модули. В системе, состоящей из многих модулей, большинство модулей зависимы. Например, модуль интерфейса пользователя может зависеть от модуля синтаксического разбора (parsing module) - синтаксического анализатора и от модуля графики. Синтаксический анализатор может зависеть от модуля лексического анализа, и так далее. Если не закрывать модуль до тех пор, пока не будет уверенности, что он уже содержит все необходимые компоненты, то невозможно будет завершить разработку многомодульной программы: каждый из разработчиков будет вынужден ожидать, когда же завершат свою работу все остальные.

При использовании традиционной методики, две рассмотренные целевые установки оказываются несовместимыми. Либо модуль остается открытым, что не позволяет пользоваться им всем остальным, либо он закрывается, и тогда любое изменение или дополнение может дать начало неприятной цепной реакции трудоемких изменений во многих других модулях, непосредственно или косвенно зависящих от этого исходного модуля.

Два рисунка, приведенные ниже, иллюстрируют ситуацию, в которой трудно согласовать потребности в открытых и закрытых состояниях модуля. На первом рисунке модуль А используется модулями-клиентами В, С, D, которые сами могут иметь своих клиентов - Е, F и так далее.

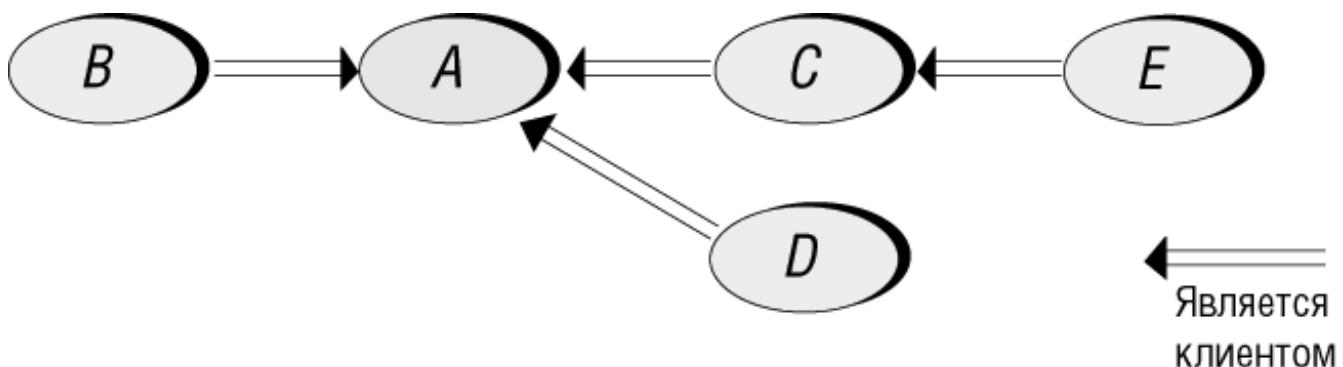


Рис. 3.12. Модуль А и его клиенты

В процессе течения времени ситуация изменяется и появляются новые клиенты - F и другие, которым

требуется расширенная или приспособленная к новым условиям версия модуля А, которую можно назвать А':

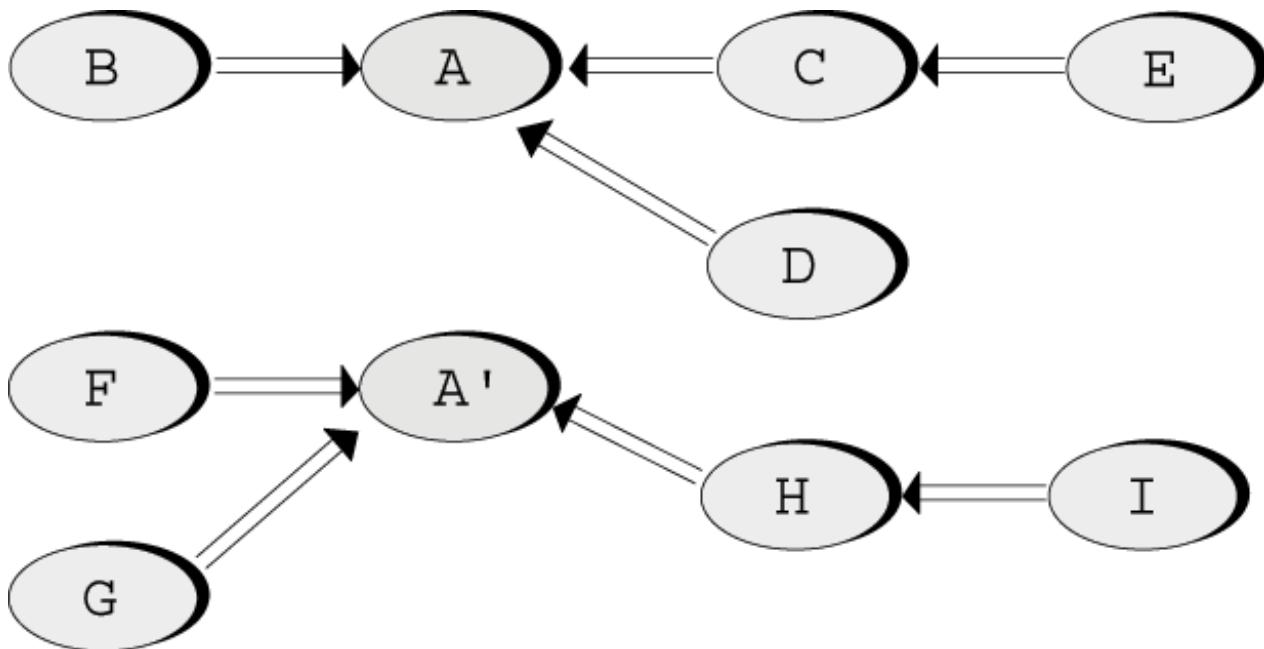


Рис. 3.13. Старые и новые клиенты

При использовании не ОО-методов, возможны лишь два решения этой проблемы, в равной степени неудовлетворительные:

- N1 Можно переделать модуль А так, чтобы он обеспечивал расширенную или видоизмененную функциональность, требуемую новым клиентам.
- N2 Можно сохранить А в прежнем виде, сделать его копию, изменить имя копии модуля на А', и выполнить все необходимые переделки в новом модуле. При таком подходе новый модуль А' никак не будет связан со старым модулем А.

Возможные катастрофические последствия решения N1 очевидны. Модуль А мог использоваться длительное время и иметь многих клиентов, таких как В, С и Д. Переделки, необходимые для удовлетворения потребностей новых клиентов, могут нарушить предположения, на основе которых старые клиенты использовали модуль А; в этом случае изменения в А могут "запустить" катастрофическую цепочку изменений у клиентов, у клиентов этих клиентов, и так далее. Для руководителя проекта это будет настоящим кошмаром: внезапно целые части ПО, считавшегося давным-давно завершенным и сданным в эксплуатацию, окажутся заново открытыми, что "запустит" новый цикл разработки, тестирования, отладки и документирования. Многие ли из руководителей проектов ПО захотят видеть себя в роли Сизифа - быть приговоренными вечно катить камень на вершину горы лишь для того, чтобы видеть, как он всякий раз вновь скатывается вниз - и все из-за проблем, вызванных необходимостью заново открывать ранее закрытые модули.

На первый взгляд решение N2 кажется лучшим: оно позволяет избежать синдрома Сизифа, поскольку не требует модификации уже существующих программных средств (показанных в верхней части последнего рисунка). Но в действительности, это решение может иметь еще худшие последствия, поскольку оно лишь отодвигает час расплаты. Экстраполируем воздействие этого решения на множество модулей, - потребуется множество модификаций, занимающих длительное время. В конечном счете, последствия оказываются ужасными: бурный рост числа вариантов исходных модулей, многие из которых очень похожи, хотя и не вполне идентичны.

Для многих организаций по разработке ПО такое изобилие модулей, не согласующееся с количеством выполняемых функций (многие из вариантов, кажущихся различными, оказываются, по существу, клонами), создает серьезную проблему управления конфигурацией ПО. И эту проблему обычно пытаются преодолеть путем использования сложных инструментальных средств. Полезные сами по себе, эти инструментальные средства пытаются "лечить" программу в ситуациях, когда предпочтительней было бы первое из рассмотренных решений. Ведь лучше избежать избыточности, чем создавать ее.

Несомненно, управление конфигурацией окажется полезным, но лишь в случае, если удастся найти модули, нуждающиеся в повторном открытии после возникших изменений, и в то же время избежать повторной компиляции модулей, не нуждающихся в этом. (В упражнении УЗ.6 предлагается выяснить, какова будет необходимость управления конфигурацией в объектно-ориентированной среде программирования.)

Но как можно получить модули, которые были бы одновременно и открытыми и закрытыми? Можно ли сохранить неизмененным модуль А и всех его клиентов в верхней части рисунка, и в то же время

предоставить модуль A' клиентам в нижней части, избегая дублирования программных средств? Благодаря механизму наследования (inheritance), ОО-подход обеспечивает особенно изящный вклад в решение этой проблемы.

Механизм наследования подробно рассматривается в последующих лекциях, а здесь дается лишь общее представление об этом. Для разрешения дилеммы, - изменять или повторно выполнять - наследование позволяет определить новый модуль A' на основе существующего модуля A, констатируя лишь различия между ними. Опишем A' как

```
class A' inherit
  A
    redefine f, g, ... end
feature
  f is ...
  g is ...
  ...
  u is ...
  ...
end
```

где предложение **feature** содержит как определение новых компонент, характерных для A', например **u**, так и переопределение тех компонент (таких как **f**, **g**,**:**), представление которых в A' отличается от того, которое они имели в A.

Для графической иллюстрации наследования используется стрелка от "наследника" (heir) (нового класса A') к "родителю" (parent) (классу A):

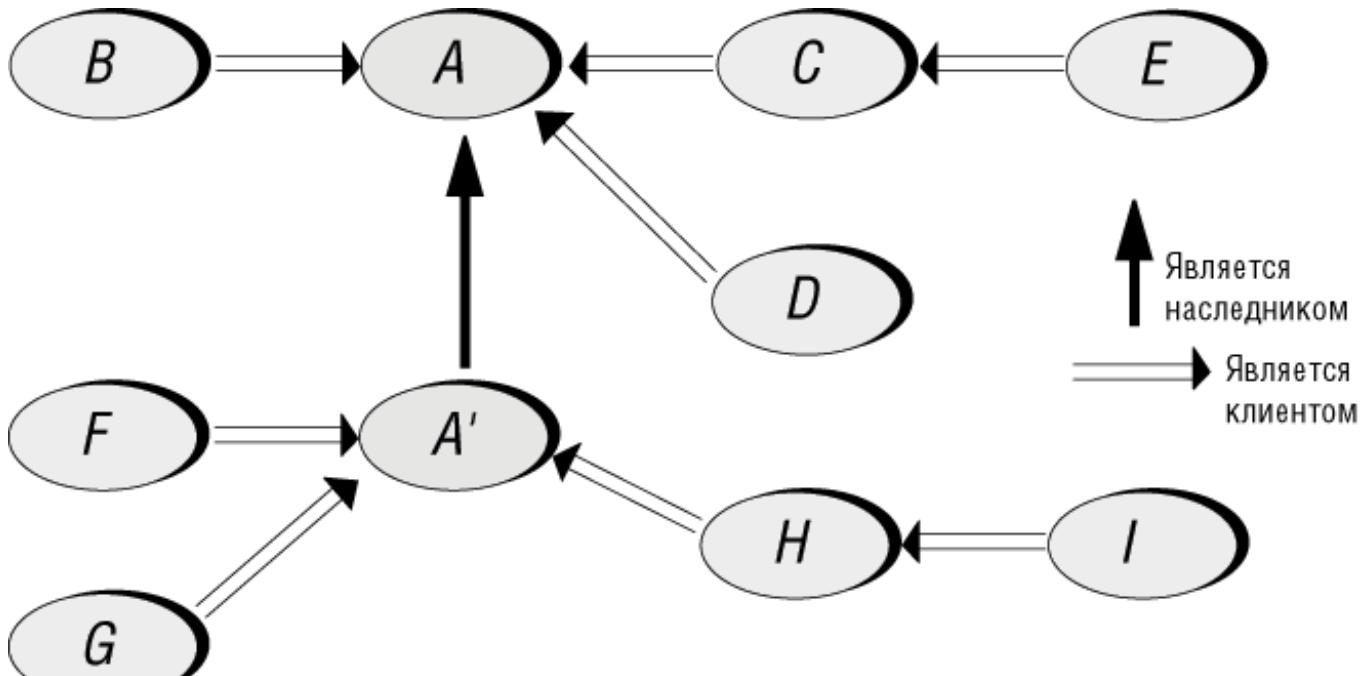


Рис. 3.14. Адаптация модуля к новым клиентам

Благодаря механизму наследования ОО, разработчики могут осуществлять гораздо более последовательный подход к разработке ПО, чем это было возможно при использовании прежних методов. Один из способов описания принципа Открыт-Закрыт и следующих из него ОО-методов состоит в рассмотрении их как **организованного хакерства**. Под "хакерством" здесь понимается небрежный (slipshod) подход к компоновке и модификации программы (а вовсе не несанкционированное и, конечно, недопустимое проникновение в компьютерные сети). Хакера можно считать плохим человеком, но часто намерения его чисты. Он может разглядеть полезный фрагмент программы, который почти пригоден для реализации текущих потребностей, намного превосходящих потребности, предусмотренные при первоначальной разработке программы. Вдохновленный похвальным желанием не создавать повторно то, что можно повторно использовать, наш хакер начинает модифицировать исходный текст программы, дополняя его средствами для выполнения новых задач. Конечно, такой порыв неплох, но результатом часто оказывается "засорение" программы многочисленными выражениями вида: `if (этот_частный_случай) then`. После нескольких повторений, возможно, осуществляемыми разными хакерами, программа начинает походить на ломть швейцарского сыра, оставленного слишком долго на августовской жаре (безвкусность этой метафоры оправдывается тем, что она хорошо воспроизводит появление в такой программе как "дырок", так и "наростов").

Организованная форма хакерства дает возможность приспосабливаться к изменяющейся структуре решаемых задач, не нарушая непротиворечивости исходной версии.

Небольшое предупреждение: здесь не предлагается неорганизованное хакерство. В частности:

- Если имеется возможность переписать исходную программу так, чтобы она, без излишнего усложнения, смогла удовлетворять потребности нескольких разновидностей клиентов, то следует это сделать.
- Как принцип Открыт-Закрыт, так и переопределение в механизме наследования не позволяют справиться с дефектами разработки, не говоря уже об ошибках в программе. **Если в модуле что-то не в порядке, то следует это сразу исправить в исходной программе**, не пытаясь разбираться с возникающей проблемой в производном модуле. Возможным исключением из этого правила является случай некорректной программы, которую не разрешено модифицировать. Принцип Открыт-Закрыт и связанные с ним методы программирования, предназначены для адаптации "здоровых" модулей, то есть модулей, которые хотя и не могут решать некоторые новые задачи, однако отвечают строго определенным требованиям в интересах своих клиентов.

Единственный Выбор

Последний из пяти принципов модульности можно считать следствием как принципа Открыт-Закрыт, так и правила Скрытия Информации.

Прежде чем подробно ознакомиться с принципом Единственного Выбора, рассмотрим типичный пример. Предположим, что создается система для работы с библиотекой (в не-программистском смысле слова: с множеством книг и других изданий, а не модулей программы). Эта система будет обрабатывать структуры данных, представляющие различные публикации. Можно объявить соответствующий тип в синтаксисе языков Pascal-Ada:

```
type PUBLICATION =
  record
    author, title: STRING;
    publication_year: INTEGER
  case pubtype:(book, journal, conference_proceedings) of
    book:(publisher: STRING);
    journal:(volume, issue: STRING);
    proceedings:(editor, place: STRING) -- Conference proceedings
  end
```

Здесь использован "тип записи с вариантами" (record type with variants) для описания наборов структур данных с полями, одни из которых (в этом примере author, title, publication_year) являются общими во всех случаях, а другие - характерны для частных вариантов данных.

Использование конкретной синтаксической конструкции здесь не является существенным. Языки программирования Algol 68 и C обеспечивают такую же возможность с помощью типа "объединение" (union). Тип union это тип T, определен как объединение ранее существовавших типов A, B,: значение типа T это либо значение типа A, либо значение типа B,: . Достоинством типов записей с вариантами является то, что в них с каждым вариантом явно связан некоторый ярлык (tag), например book, journal, conference_proceedings.

Пусть А - модуль, который содержит описанное выше объявление типа. Пока модуль А считается открытым, к нему можно добавлять поля или вводить в него новые варианты. Но когда модуль А передается клиентам, следует закрыть его, а это по умолчанию означает, что в нем уже перечислены все существенные поля и варианты. Итак, пусть В это типичный клиент модуля А. В будет манипулировать с публикациями через некоторую переменную, например:

```
p: PUBLICATION
```

Чтобы с помощью p осуществлять какие-либо полезные действия, необходимо явно выделить различные случаи:

```
case p of
  book:... Instructions which may access the field p.publisher...
  journal:... Instructions which may access fields p.volume, p.issue...
  proceedings:... Instructions which may access fields p.editor, p.place...
end
```

Здесь оказалась удобной команда выбора case из языков Pascal и Ada; ее синтаксис воспроизводит определение типа записи с вариантами. В Fortran'e и C это может имитироваться многократным использованием команды безусловного перехода goto (switch в языке C). В этих и других языках такой же

результат можно получить, используя вложенные команды условного перехода (`if ... then ... elseif ... else ... end`).

Следует отметить, что, независимо от используемой синтаксической конструкции, для осуществления такого выбора каждый модуль-клиент должен знать полный список вариантов представления для публикации, поддерживаемых модулем A. Последствия этого нетрудно предвидеть. Наступит момент, когда потребуется новый вариант, например технические отчеты фирм и университетов. Тогда необходимо расширить определение типа PUBLICATION в модуле A, учитывая новый случай. Это вполне логично и неизбежно: если было изменено определение понятия публикации, то следует обновить и соответствующее объявление типа. Однако значительно труднее найти оправдание другому следствию: любой клиент модуля A, такой как B, также будет требовать обновления, если в нем использовалась рассмотренная выше структура, основанная на полном списке случаев для р. А это, очевидно, будет иметь место для большинства клиентов.

Итак, наблюдаются очень опасные изменения в программе: простое и естественное дополнение может вызвать цепную реакцию изменений во многих модулях-клиентах.

Эта проблема возникнет всякий раз, когда некоторое понятие допускает множество вариантов. Здесь таким понятием было "публикация" ("publication"), а его начальными вариантами были: книга (book), журнальная статья (journal article), труды конференции (conference proceedings); другими типичными примерами могут быть:

- В системе работы с графикой: понятие фигуры (figure), с такими вариантами как многоугольник (polygon), окружность (circle), эллипс (ellipse), отрезок (segment) и другие основные виды фигур.
- В текстовом редакторе: понятие команды пользователя (user command), с такими вариантами как вставка строки (line insertion), удаление строки (line deletion), удаление символа (character deletion), глобальная замена (global replacement) одного слова другим.
- В компиляторе для языка программирования: понятие языковой конструкции (language construct), с такими вариантами как команда (instruction), выражение (expression), процедура (procedure).

В любом таком случае необходимо допускать возможность того, что список вариантов, заданных и известных на некотором этапе разработки программы, может в последующем быть изменен путем добавления или удаления вариантов. Чтобы обеспечить реализацию такого подхода к процессу разработки программного обеспечения, нужно найти способ **защитить** структуру программы от воздействия подобных изменений. Отсюда следует принцип Единственного Выбора:

Принцип Единственного Выбора

Всякий раз, когда система программного обеспечения должна поддерживать множество альтернатив, их полный список должен быть известен только одному модулю системы.

Требование того, чтобы список выбора был известен лишь одному модулю, обеспечивает подготовку к последующим изменениям: при добавлении вариантов понадобится произвести обновление только того модуля, в котором содержится эта информация - такова сущность единственного выбора. А все остальные модули, в частности - его клиенты, смогут продолжать свою работу как обычно.

Таким образом, как показывает пример с библиотекой публикаций, традиционные методы не обеспечивают решения проблемы, в то время как объектные технологии позволяют получить ее решение благодаря двум методическим приемам, связанным с наследованием: полиморфизмом (polymorphism) и динамическим связыванием (dynamic binding). Однако приведенного здесь предварительного обсуждения недостаточно; эти методические приемы можно будет понять лишь в контексте всего метода наследования. (См. "Динамическое связывание" [лекция 4](#))

Принцип Единственного Выбора нуждается еще в нескольких комментариях:

- В соответствии с этим принципом, список возможных выборов должен быть известен одному и только одному модулю. Из целей модульного программирования следует, что желательно иметь **не более чем один** модуль, располагающий этой информацией; но очевидно также, что ему должен обладать **хотя бы один** модуль. Невозможно составить программу текстового редактора, если, по крайней мере один из компонентов не будет иметь списка всех поддерживаемых этой программой команд, для графической программы - списка всех типов фигур, для компилятора - списка всех языковых конструкций.
- Подобно другим правилам и принципам, обсужденным в этой лекции, принцип Единственного Выбора касается **распределения знаний (distribution of knowledge)** в системе ПО. Этот вопрос является действительно решающим при поиске расширяемых, многократно используемых программных средств. Чтобы получить цельную, надежную архитектуру ПО, следует предпринять строго обдуманные шаги по ограничению объема информации, доступной каждому модулю. По аналогии с методами, используемыми некоторыми общественными организациями, можно назвать это **принципом необходимого знания (need-to-know)**: запретить каждому модулю доступ к любой информации, которая не является безусловно необходимой для его надлежащего функционирования.

- Можно рассматривать принцип Единственного Выбора как прямое следствие принципа Открыт-Закрыт. Обсудим пример с библиотекой публикаций в свете рисунка, иллюстрирующего необходимость в открытых и закрытых модулях: А это модуль, содержащий первоначальное описание типа PUBLICATION; клиенты В, С это модули, зависящие от исходного списка вариантов; А' это усовершенствованная версия А, предлагающая дополнительный вариант - технические отчеты (technical reports). (См. второй рисунок в разделе "Открыт-Закрыт")
- Можно также понимать этот принцип как сильную форму принципа Скрытия Информации. Разработчик модулей-поставщиков, таких как А и А', стремится скрыть информацию (относительно точного списка вариантов для некоторого понятия) от модулей-клиентов.

Ключевые концепции

- Выбор надлежащей структуры модуля является ключом к достижению целей его возможного повторного использования и расширяемости.
- Модули служат как для декомпозиции программного обеспечения (проектирование сверху вниз), так и для его композиции (снизу-вверх).
- Принципы модульности применимы как к спецификации и проектированию, так и к реализации ПО.
- Всеобъемлющее определение модульности должно объединять различные точки зрения; разные требования иногда оказываются взаимно противоречивыми, например декомпозиция (стимулирующая методы проектирования сверху-вниз) и композиция (способствующая использованию метода снизу-вверх).
- Управление количеством и формой связей между модулями является основой разработки хорошей модульной архитектуры.
- Для долгосрочной целостности структур модульной системы требуется скрытие информации, что приводит к необходимости строгого разделения интерфейса и реализации.
- Унифицированный доступ освобождает клиентов от знания выбора внутренних представлений, реализованных в модулях-поставщиках.
- Закрытым является такой модуль, который может использоваться, благодаря знанию его интерфейса, модулями-клиентами.
- Открытым является такой модуль, который еще можно расширять.
- Для эффективного руководства проектом следует поддерживать модули, являющиеся одновременно как открытыми, так и закрытыми. Но традиционные подходы к разработке и программированию не дают такой возможности.
- Принцип Единственного Выбора предписывает ограничивать распространение полной информации обо всех вариантах некоторого понятия.

Библиографические замечания

В методе проектирования, известном как "структурное проектирование" [Yourdon 1979], особое значение придается важности использования модульных структур. Этот метод был основан на анализе "цепления" и "связности" модулей. Но неявно выраженное представление модулей в структурном проектировании было основано на традиционном понятии подпрограммы, что ограничило рамки обсуждения. Принцип унифицированного доступа был первоначально предложен (под названием "унифицированная ссылка") в работе [Geschke 1975]. При обсуждении унифицированного доступа упоминался язык Algol W, преемник языка Algol 60 и предшественник языка Pascal (в котором были предложены некоторые интересные механизмы, не сохранившиеся в Pascal'e), разработанный Виртом и Хоаром, и описанный в работе [Hoare 1966].

Скрытие информации было предложено в двух основополагающих статьях Дэвида Парнаса [Parnas 1972] [Parnas 1972a].

Средства управления конфигурацией, которые будут перекомпилировать модули, затронутые изменениями в других модулях, исходя из подробного списка зависимостей между модулями, основаны на концепциях сервисной программы Make, первоначально разработанной для Unix [Feldman 1979]. Современные сервисные программы - а их имеется много на рынке программных средств - существенно дополнили функциональность основных идей.

В некоторых из приводимых ниже упражнений предлагается разработать метрики для количественной оценки различных неформальных критериев модульности, сформулированных в этой лекции. Некоторые результаты, относящиеся к ОО-метрикам, содержатся в работах Кристине Минджинс (Christine Mingins) [Mingins 1993] [Mingins 1995] и Брайана Хендerson-Селлерса (Brian Henderson-Sellers) [Henderson-Sellers 1996a].

Упражнения

УЗ.1 Модульность в языках программирования

Рассмотрите модульные структуры в любом хорошо знакомом вам языке программирования и оцените,

насколько они удовлетворяют критериям и принципам, изложенным в этой лекции.

УЗ.2 Принцип Открыт-Закрыт (для программистов Lisp)

Многие реализации Lisp'a связывают конкретные функции с их именами не статически, а во время выполнения программы. Означает ли это, что язык Lisp лучше поддерживает принцип Открыт-Закрыт, чем статические языки?

УЗ.3 Ограничения на скрытие информации

Представляете ли вы себе обстоятельства, при которых скрытие информации не должно применяться к связям между модулями?

УЗ.4 Метрики для модульности (отчетная исследовательская работа)

Критерии, правила и принципы модульности были описаны в этой лекции с помощью качественных определений. Однако некоторые из них поддаются количественному анализу. Это могут быть:

- Модульная непрерывность.
- Минимум интерфейсов.
- Слабая связность интерфейсов.
- Явные интерфейсы.
- Скрытие информации.
- Единственный выбор.

Выясните возможность разработки метрик модульности, чтобы оценить, насколько модульной является архитектура системы программного обеспечения в соответствии с некоторыми из этих понятий. Метрики должны быть размерно-независимыми: увеличение размера системы без изменения ее модульной структуры не должно приводить к изменению мер ее сложности (см. также следующее упражнение).

УЗ.5 Модульность существующих систем

Примените критерии, правила и принципы модульности из этой лекции для оценки системы, к которой у вас есть доступ. Если вы решили предыдущее упражнение, примените любую из предложенных вами метрик модульности.

Можете ли вы установить какие-нибудь взаимозависимости между результатами этого анализа (качественными, количественными, или теми и другими) и оценками структурной сложности исследуемой системы, основанными либо на ее неформальном анализе, либо, если это возможно, на реальных замерах затрат на ее отладку и сопровождение?

УЗ.6 Управление конфигурацией и наследование

Это упражнение предполагает знание механизма наследования, описанного далее в этом курсе. Его не стоит пока что выполнять, если вы дошли до этой лекции, изучая курс последовательно.

Обсуждение принципа Открыт-Закрыт показало, что отсутствие наследования в не ОО-методах вызывает чрезмерные расходы на разработку средств управления конфигурацией, поскольку желание избежать повторного открытия закрытых модулей может приводить к созданию слишком большого числа модульных вариантов. Выясните, какая роль остается за средствами управления конфигурацией в ОО-среде, где имеется механизм наследования, и вообще - как использование объектной технологии влияет на управление конфигурацией.

Если вы знакомы с конкретными средствами управления конфигураций, выясните, как они взаимодействуют с механизмом наследования и другими принципами ОО-разработки ПО.

¹⁾ Дальнейшее обсуждение метода нисходящего проектирования показывает, что этот метод не вполне согласуется с другими критериями модульности.

²⁾ Это будет одним из принципов нашего стиля программирования: Принцип именованной константы (Symbolic Constant Principle.)

³⁾ Более подробно этот вопрос рассмотрен в разделе "Формальные утверждения (assertions) не являются механизмом контроля входа данных"

⁴⁾ Тело блока это последовательность команд. Примененный здесь синтаксис совместим с нотацией, используемой в последующих лекциях и несколько отличается от синтаксиса языка Algol. "--" означает начало комментария.

⁵⁾ По умолчанию, "Ada" всегда означает не более новую версию Ada 95, а наиболее распространенную

форму этого языка (версия 1983 года.). Обе версии рассмотрены в лекции 15 курса "Основы объектно-ориентированного проектирования".

⁶⁾ Он известен также как принцип Унифицированных Ссылок

Основы объектно-ориентированного программирования

4. Лекция: Подходы к повторному использованию

"Последуйте примеру проектирования компьютерных технических средств! Это неверно, что каждая новая программная разработка должна начинаться с чистого листа. Должны существовать каталоги программных модулей, такие же, как каталоги сверхбольших интегральных схем СБИС (VLSI devices). Создавая новую систему, мы должны заказывать компоненты из этих каталогов и собирать систему из них, а не изобретать каждый раз заново колесо. Создавая меньше новых программ, мы, возможно, найдем лучшее применение своим усилиям. И, может быть, исчезнут некоторые из трудностей, на которые все жалуются - большие затраты, недостаточная надежность. Разве не так?" Вы, вероятно, слышали или даже сами высказывали такого рода замечания. Еще в 1968 г. на известной конференции НАТО по проблемам разработки ПО, Дуг Мак-Илрой (Doug McIlroy) пропагандировал идею "серийного производства компонентов ПО". Таким образом, мечта о возможности повторного использования программных компонентов не является новой. Было бы нелепо отрицать, что повторное использование имеет место в программировании. Фактически одним из наиболее впечатляющих результатов развития индустрии ПО с тех пор, как в 1988 г. появилось первое издание этой книги, явилось постепенное появление повторно используемых компонентов. Они получали все большее распространение, начиная от небольших модулей, предназначенных для работы с Visual Basic (VBX) фирмы Microsoft и OLE 2 (OCX, а сейчас ActiveX), до обширных библиотек классов, известных также как "каркасы приложений" ("framework applications"). Еще одним замечательным достижением является развитие Интернета: причество общества, охваченного Сетью (wired society), облегчило или в ряде случаев устранило некоторые из логических препятствий, казавшихся почти непреодолимыми еще несколько лет назад. Но это только начало. Мы далеки от предвидения Мак-Илроя о превращении программной инженерии в отрасль промышленности, основанную на использовании программных компонентов. Однако методология конструирования ОО-ПО впервые дала возможность представить себе практическую реализацию этого предвидения. И это может принести весьма значительную пользу не только разработчикам ПО, но, что еще важнее, тем, кто нуждается в их продукции, своевременно появляющейся и высококачественной. В этой лекции будут рассмотрены некоторые из проблем, направленных на широкомасштабное внедрение повторного использования программных компонентов.

Цели повторного использования

Прежде всего, следует понять, почему так важно улучшать возможности повторного использования ПО. Здесь незачем обращаться к доводам типа "любовь к матери и яблочному пирогу". Как мы увидим, наша борьба за повторное использование преследует надлежащие цели, позволит избежать миражей, и принесет хороший доход от соответствующих инвестиций.

Ожидаемые преимущества

Повторное использование может обеспечить прогресс на следующих направлениях:

- **Своевременность (timeliness)** (в том смысле, который определен при обсуждении показателей качества: быстрота доведения проектов до завершения и продукции до рынка). При использовании уже существующих компонентов нужно **меньше** разрабатывать, а, следовательно, ПО создается **быстрее**.
- **Сокращение объема работ по сопровождению ПО (decreased maintenance effort)**. Если кто-то разработал ПО, то он же отвечает и за его последующее развитие. Известен парадокс компетентного разработчика ПО: "чем больше вы работаете, тем больше работы вы себе создаете". Довольные пользователи вашей продукции начнут просить добавления новых функциональных возможностей, переноса на новые платформы. Если не надеяться "на дядю", то единственное решение парадокса - стать некомпетентным разработчиком, - чтобы никто больше не был заинтересован в вашей продукции. В этой книге подобное решение не поощряется.
- **Надежность**. Получая компоненты от поставщика с хорошей репутацией, вы имеете определенную гарантию, что разработчики предприняли все нужные меры, включая всестороннее тестирование и другие методы контроля качества. В большинстве случаев можно ожидать, что кто-то уже испытал эти компоненты до вас и обнаружил все возможно остававшиеся ошибки. Заметьте, вовсе не предполагается, что разработчики компонентов умнее вас. Для них создаваемые компоненты - будь то графические модули, интерфейсы баз данных, алгоритмы сортировки - это служебная обязанность, цель работы. Для вас это лишь второстепенная, рутинная работа, поскольку **вашей** целью является создание некоторой прикладной системы в вашей собственной области деятельности.
- **Эффективность**. Факторы, способствующие возможности повторного использования ПО, побуждают разработчиков компонентов пользоваться наилучшими алгоритмами и структурами данных, известными в их конкретной сфере деятельности. Однако в команде, разрабатывающей большой прикладной проект, трудно ожидать наличия специалистов по **каждой** проблеме, затрагиваемой в этом проекте. При разработке большого проекта невозможно оптимизировать все его детали. Следует стремиться к достижению наилучших решений в своей области знаний, а в остальном использовать профессиональные разработки.
- **Совместимость**. Если использовать хорошую современную ОО-библиотеку, то ее стиль повлияет, за счет естественного "процесса диффузии", на стиль разработки всего ПО. Это существенно помогает повысить качество программного продукта.
- **Инвестирование**. Создание повторно используемого ПО позволяет сберечь плоды знаний и открытый лучшим разработчикам, превращая временные ресурсы в постоянные.

Многие из тех, кто признает повторное использование желательным, имеют в виду лишь первый из факторов в этом списке, - повышение производительности. Но это не всегда самый важный вклад повторного

использования в процесс разработки ПО. Повышение надежности, например, является не менее существенным фактором. Тоже можно сказать и об эффективности.

В этом отношении повторное использование можно рассматривать как особый показатель, отличающийся от других факторов, обсуждавшихся в лекции 1. Его улучшение дает возможность улучшить **почти все** остальные факторы качества ПО. А причина чисто экономическая: если элемент ПО служит не для одного, а для многих проектов, то экономически разумно использовать лучшие методы создания высококачественного ПО - формальную верификацию, всестороннюю оптимизацию. В обычных разработках от таких приемов зачастую отказываются как от ненужного излишества. Однако для повторно используемых компонентов аргументация существенно изменяется - улучшение всего лишь одного элемента может оказаться выгодным для тысяч разработок.

Конечно, эти рассуждения не являются совсем новыми - они отчасти представляют собой перенос на производство ПО тех идей, которые уже существенно затронули другие отрасли деятельности, когда они перешли от индивидуально изготавляемых изделий к индустрии массового производства. Изготовление чипа СБИС обходится значительно дороже, чем серийное изготовление простой специализированной схемы, но если он хорошо выполнен, то он проявит себя в бесчисленных компьютерных системах и повысит их качество благодаря всей вложенной в него раз и навсегда работе его конструкторов.

Потребители и производители повторно используемых программ

В приведенном выше списке преимуществ можно выделить две ситуации - использование профессиональных или собственных компонентов. Первые четыре элемента списка описывают ситуацию использования существующих, профессионально разработанных компонентов. Последний элемент списка характеризует повторное использование собственного программного продукта. Элемент списка - совместимость - относится к обоим случаям.

Такое разграничение достоинств отражает два аспекта повторного использования: точку зрения **потребителя**, пользующегося продукцией разработчиков компонент, и точку зрения **производителя**, обеспечивающего возможность повторного использования своих разработок.

Для разработчиков ПО, еще не имеющих большого опыта, следует быть **потребителями** компонентов. Принципиально невозможно сразу приступить к производству повторно используемых программ. Единственно возможный путь **стать производителем** - состоит в изучении и копировании уже существующих хороших образцов. Такой подход сразу принесет свои полезные плоды, поскольку в своих разработках вы воспользуетесь достоинствами этих компонентов.

Дорога к Повторному использованию

Станьте потребителем повторного использования, прежде чем пытаться стать его производителем.

Что следует повторно использовать?

Убедив себя в том, что Повторное использование - Это Хорошо, осталось выяснить, как же этого добиться?

Первый возникающий вопрос - на каком уровне следует осуществлять повторное использование: персонала, спецификаций, проектов, их образцов, исходного кода, компонентов или абстрактных модулей.

Повторное использование персонала

Наиболее просто повторно использовать разработчиков, что широко практикуется в промышленности. Переводя разработчиков ПО с одного проекта на другой, фирмы избегают потери накопленного ими ранее опыта и обеспечивают его достойное применение в новых разработках.

Ввиду высокой текучести программистских кадров возможности такого подхода ограничены.

Повторное использование проектов и спецификаций

Этот подход является, по существу, более организованной версией предыдущего - повторного использования знаний, умений и опыта. Как показало обсуждение вопроса о документации, само представление проекта как независимого программного продукта, имеющего собственный жизненный цикл, независимый от соответствующей реализации, кажется сомнительным, поскольку трудно гарантировать, что проект и его реализация будут оставаться совместимыми в процессе изменения системы ПО.

Таким образом, если повторно использовать только проект, то возникает риск повторного использования неправильно работающих или уже вышедших из употребления элементов.

Эти замечания можно отнести и к другому смежному виду повторного использования: повторному использованию спецификаций.

Среди разработчиков ПО долгое время бытowała идея о том, что единственno заслуживающим внимания является повторное использование лишь проектов и спецификаций. Эта идея весьма существенно препятствовала продвижению вперед, поскольку означала, что создание компонентов направленно на удовлетворение лишь несущественных потребностей и не решает истинно трудных проблем. Прежде эта точка зрения была преобладающей; преодолеть ее удалось благодаря объединенному воздействию теоретических доводов (соображений ОО-технологии) и практических достижений (успешной реализации повторно используемых компонентов).

Термин "преодолеть" здесь является, пожалуй, слишком сильным, поскольку, как это часто бывает в подобных спорах, свою долю в достижение полезного результата внесли обе стороны. Идея повторного использования проектов становится намного более интересной при использовании подхода (такого, как точка зрения на ОО-технологию, развивающаяся в этой книге), который существенно устраняет разрыв между проектом и его реализацией. Тогда разница между модулем и проектом модуля (*design for a module*) является не принципиальной, а лишь количественной: проект модуля это просто модуль, отдельные фрагменты которого еще не полностью реализованы; а полностью реализованный модуль можно использовать, благодаря средствам абстрактного представления, в качестве проекта модуля. При таком подходе различие между повторным использованием модулей (рассматриваемым ниже) и повторным использованием проектов постепенно исчезает.

Образцы проектов (design patterns)

В середине девяностых годов специалистов привлекла идея **образцов** (или **шаблонов**) **проектов**. Образец - это архитектурный принцип, применимый во многих прикладных областях; следуя образцу можно построить решение некоторой проблемы.(Образец проекта с историей команд рассмотрен в [лекции 3](#) курса "Основы объектно-ориентированного проектирования").

Вот типичный пример, подробно обсуждаемый в одной из последующих лекций. **Проблема:** как снабдить интерактивную систему механизмом, позволяющим ее пользователям отменить ранее выполненную команду, если они решат, что она была нецелесообразной, и повторить выполнение отмененной команды, если они передумают. **Образец:** использовать класс COMMAND определенной структуры (которую мы в последующем рассмотрим) и связанный с ней "список истории". Будут рассмотрены и многие другие образцы проектов.

Одной из причин успешного внедрения идеи образца проекта явилось то, что это была не просто идея: книга¹¹, в которой впервые было предложено это понятие, и последовавшие за ней издания содержали каталог непосредственно применимых образцов, которые читатели могли изучать и использовать.

Образцы проектов уже внесли существенный вклад в развитие ОО-технологии, и по мере публикации все новых образцов они помогут разработчикам пользоваться опытом своих предшественников и современников. Как же этот общий принцип приложить к проблеме повторного использования? Образцы проектов не должны внушать надежду на возвращение к уже упоминавшейся ранее мысли о том, что "**все что нужно - это только повторно использовать проекты**". Образец, который по-существу представляет собой лишь сценарий образца (book pattern), пусть даже самый лучший и универсальный, является только "учебным пособием", а не инструментальным средством повторного использования. Как-никак, а в течении трех последних десятилетий учебники по компьютерным наукам рассказывают об оптимизации реляционных баз данных, AVL-деревьях (сбалансированных деревьях Адельсона-Вельского и Ландиса), алгоритме быстрой сортировки (Quicksort) Хоара, алгоритме Дейкстры для поиска кратчайшего пути в графе, без какого-либо упоминания о том, что эти методы совершили прорыв в решении проблемы повторного использования. В определенном смысле, образцы, разработанные за последние несколько лет, являются лишь очередными дополнениями к набору стандартных приемов, используемых специалистами по разработке ПО. При таком понимании новым вкладом в ОО-технологию следует считать не идею образца, а сами предлагаемые образцы.

Обстоятельное рассмотрение проблемы образцов показывает, что эта точка зрения оказывается излишне ограниченной (См. "Программы с дырами", [лекция 14](#)). По-видимому, само понятие образца является действительно новым вкладом, даже если это еще не вполне осознанно. Но требуется дополнительная работа над образцами, чтобы выйти за пределы их чисто педагогической ценности. Удачный образец не может быть представлен лишь некоторым текстовым описанием - это должен быть **компонент ПО** или набор таких компонентов. На первый взгляд такая цель может представляться довольно отдаленной, поскольку многие из образцов являются настолько универсальными и абстрактными, что кажется невозможным реализовать их в виде программных модулей.

Но использование ОО-технологии обеспечивает радикальный вклад - она позволяет создавать повторно используемые модули, обладающие способностью изменяться. Они не будут "замороженными" элементами, а служат общими схемами, образцами, - здесь действительно уместен термин **образец** в полном смысле этого слова, они могут быть адаптированы к различным конкретным ситуациям. Это новое понятие мы называем **классом, определяющим поведение (behavior class)** (более образным является термин **программы с дырами (programs with holes)**). Это понятие, основанное на понятии отложенного (абстрактного) класса (deferred class), будет рассмотрено в последующих лекциях. Объединяя его с идеей о группе компонентов,

предназначенных для совместного функционирования - часто называемых **каркасами (frameworks)** или просто **библиотеками** - получаем замечательное средство, сочетающее повторное использование и способность к адаптации.

Повторное использование исходного текста

Несмотря на полезность повторного использования персонала, проектов и спецификаций, здесь не реализуется ключевая цель повторного использования. Если мы хотели бы найти программистский эквивалент повторно используемых деталей из других технических дисциплин, то это означало бы необходимость повторно использовать тот "хлам", из которого фактически состоит наша программная продукция: исполняемые программы.

Если так, то в какой же форме следует их использовать? Естественный ответ - в первоначальной форме: в виде исходного текста. В некоторых случаях такой подход оказался весьма эффективным. Например, совершенствование операционной системы Unix, первоначально распространявшейся по университетам и исследовательским лабораториям, стало возможным в основном благодаря наличию исходного кода, получаемого в режиме онлайн. Это позволило пользователям изучать, копировать и расширять сферу использования системы. То же справедливо и для круга пользователей Lisp.

Существуют экономические и психологические препятствия на пути к распространению исходных кодов. Более серьезными ограничениями являются:

- Отождествление повторно используемого ПО с повторно используемым исходным текстом (*source*) исключает возможность скрытия информации. Следует иметь в виду, что повторное использование действительно больших проектов невозможно, если не предпринять систематических усилий по защите повторных пользователей от необходимости знания бесчисленных деталей.
- В сложных системах многие ее части могут не очевидным образом зависеть от других. Это часто затрудняет повторное использование отдельных элементов, приводя к необходимости повторно использовать и все остальное.

Удовлетворяющая требованиям модульности форма повторного использования должна устраниć эти ограничения, поддерживая абстракцию и обеспечивая "мелкоструктурную" реализацию повторного использования.

Повторное использование абстрактных модулей

Все предыдущие подходы, несмотря на их ограниченную применимость, осветили важные аспекты проблемы повторного использования:

- Повторное использование персонала необходимо, но недостаточно. Наилучшие повторно используемые компоненты бесполезны при отсутствии хорошо подготовленных разработчиков, которые обладают достаточным опытом, чтобы распознать ситуацию, в которой может помочь использование уже существующих компонентов.
- Для повторного использования проектов необходимы не только готовые решения конкретных задач, но и достаточно высокий концептуальный уровень и универсальность повторно используемых компонентов. Классы, с которыми мы встретимся при обсуждении ОО-технологии, могут рассматриваться и как модули-проекты, так и как модули-реализации.
- Возможность повторного использования исходного кода служит напоминанием о том, что ПО в конечном счете определяется текстами программ. Разумная политика в области повторного использования должна приводить к созданию повторно используемых программных элементов.

Обсуждение позволило сузить область поиска подходящих единиц повторного использования. Такой единицей должен быть программный элемент (**коллекция** элементов). Он должен быть **модулем** приемлемого размера, удовлетворяющим требованиям модульности из предыдущей лекции. В частности, его связи должны быть строго ограничены, чтобы облегчить возможность независимого повторного использования. Информация, характеризующая возможности модуля, и составляющая первичную документацию для программистов, повторно его использующих (*reusers*), должна быть **абстрактной**: в соответствии с принципом Скрытия Информации она должна освещать лишь свойства, существенные для клиентов, а не описывать все детали модуля (как это делается в исходном коде).

Термин абстрактный модуль будет применяться к таким повторно используемым единицам (*units of reuse*), входящим в состав непосредственно применяемых ПО, доступ из внешнего мира к которым может осуществляться через описание, содержащее лишь подмножество свойств каждой единицы.

Далее в лекциях 3-6 этого курса предлагается строгое определение таких абстрактных модулей; а затем в лекциях 7-18 будут рассмотрены их свойства.

Акцент на понятии абстрактности и отказ от использования исходного кода в качестве средства для повторного использования, вовсе не препятствует распространению модулей в виде исходных текстов (*source form*).

Противоречие здесь только кажущееся: в данном обсуждении речь идет не о том, как будут поставляться модули программистам, повторно их использующим, а о том, что они будут использовать в качестве первоисточника информации о модулях. Может оказаться приемлемым, чтобы модуль распространялся в виде исходного текста, но повторно использовался на основе абстрактного описания его интерфейса.

Повторяемость при разработке ПО

В поиске идеала абстрактного модуля следует рассмотреть суть процесса конструирования ПО. Наблюдая за разработкой, нельзя не обратить внимания на периодически повторяющиеся действия в этом процессе. Вновь и вновь программисты "сплетают" программу из множества стандартных элементов: сортировка, поиск, считывание, запись, сравнение, обход по дереву, - все повторяется. Опытным разработчикам знакомо это ощущение **de_ja vu** (дежавю - ощущение, что настоящее уже встречалось в прошлом), столь характерное для их профессии.

Чтобы оценить эту ситуацию (для тех, кто разрабатывает ПО или руководит такой разработкой), полезно ответить на следующий вопрос:

Сколько раз за последние шесть месяцев вы, или те, кто работает на вас, разрабатывали некоторый вариант табличного поиска?

Табличный поиск понимается здесь как выяснение того, содержится ли заданный элемент x в таблице t . Эта задача имеет много вариантов в зависимости от типа элементов, структуры данных, представляющей t , а также выбранного алгоритма поиска.

Вполне возможно, что вы или ваши коллеги многократно искали и находили собственное решение этой задачи. Наблюдатель со стороны посчитает табличный поиск легкодоступным и очевидным объектом применения повторно используемых компонентов. Ведь это одна из наиболее широко исследованных областей в компьютерных науках, которой посвящены сотни статей и многие книги, начиная с тома 3 знаменитого трактата Кнута. Базовый университетский курс по информатике на всех соответствующих факультетах включает в себя наиболее важные алгоритмы и структуры данных. Несомненно, в этой тематике нет ничего непостижимого.

Кроме того:

- Как уже отмечалось, вряд ли возможно создать полезную систему ПО, в которой не будут содержаться некоторые виды табличного поиска.
- Как будет подробнее показано ниже, большинство алгоритмов поиска следуют общему образцу, что, по-видимому, обеспечивает идеальную основу для повторно используемого решения.(См. библиографические ссылки в конце этой лекции.)

Нетехнические препятствия

Почему же повторное использование еще не является общепринятым?

Наиболее серьезные препятствия к этому являются техническими; пути их преодоления будут обсуждаться в последующих разделах этой лекции (да и в остальных лекциях курса). Но, конечно, имеются также некоторые организационные, экономические и политические препятствия.

Синдром NIH

Психологическим препятствием повторного использования является известный синдром: "Придумано Не Нами" (Not Invented Here или "NIH"). Говорят, что разработчики ПО являются индивидуалистами, предпочитающими все выполнять сами, не полагаясь на чужую работу.

Но на практике это не подтверждается. Разработчики ПО склонны к бесполезной работе не более других специалистов. Если имеется хорошее, широко известное и легкодоступное повторно используемое решение, то оно будет использовано.

Рассмотрим типичный случай лексического и синтаксического анализа. Намного проще создать программу грамматического анализа для командного языка или простого языка программирования, используя программные генераторы грамматического разбора (parser generators), например комбинацию известных программ Lex-Yacc, а не создавая все с нуля. Вывод очевиден: там, где инструментальные средства имеются, квалифицированные разработчики ПО повсеместно их используют.

В некоторых случаях имеет смысл создание собственного нестандартного анализатора, поскольку у упомянутых инструментальных средств имеются свои ограничения. Но обычно разработчики предпочитают обращаться к одному из этих средств. Это может привести к новому синдрому, **противоположному** синдрому NIH, который можно назвать синдромом "Привычки Препятствовать Нововведениям" (Habit Inhibiting Novelty или "HIN"). Повторно используемое решение, пусть даже полезное, но имеющее такие ограничения, которые сужают возможности разработчиков и подавляют внедрение новых идей, становится бесполезным. Попробуйте

убедить кого-нибудь из разработчиков Unix'a использовать генератор грамматического разбора, отличающийся от Yacc, и вы можете на собственном опыте столкнуться с синдромом NIH.

Конечно, существует кое-что, напоминающее NIH, но часто это просто вполне понятная реакция осмотрительных разработчиков на новые и неизвестные компоненты. Они могут опасаться, что с ошибками или другими проблемами в новой для них программе труднее будет справиться, чем с решением, над которым они имеют полный контроль. Часто такие опасения оправдываются неудачными прежними попытками повторного использования компонентов. Но если новые компоненты являются высококачественными и обеспечивают нормальное функционирование программы, то опасения быстро исчезают.

Таким образом, обеспечить высокое качество при создании повторно используемых компонентов существенно важнее, чем для других видов ПО.

Обозначим через N стоимость уникального решения, R - решения, основанного на повторно используемых компонентах. Значение R никогда не будет равно нулю: сюда войдут затраты на обучение, затраты на включение компонентов в систему, понадобиться создать интерфейс вызова. Так что даже если экономия на повторном использовании и другие выгоды

$$r = (N - R) / N$$

от повторного использования потенциально невелики, то придется все же убедить возможных "повторных пользователей" в том, что ради высокого качества повторно используемого решения стоит отказаться от желания полного контроля над всеми элементами системы.

Этим объясняется, почему ошибочной целью является политика фирмы, направленная на работу с потенциальными повторными пользователями (**потребителями**, как их называют разработчики). Вместо этого следует ужесточить требования к **производителям** внешних компонентов, требуя гарантий качества и пригодности предлагаемой ими продукции. Разработчики прикладных систем будут использовать ваши компоненты не в связи с вашими рекомендациями, а потому, что вы хорошо потрудились над тем, чтобы повторно используемые компоненты было выгодно применять в прикладных программах.

Фирмы по разработке ПО и их стратегии

У фирмы по разработке ПО всегда существует искушение создавать решения, преднамеренно **не** удовлетворяющие критериям повторного использования, из опасения не получить следующий заказ, - поскольку если возможности уже приобретенного решения окажутся излишне широкими, то покупателю следующий заказ не потребуется!

Мне довелось слышать в высшей степени откровенное высказывание по этому вопросу после моей лекции о повторном использовании и ОО-технологии.

Высокопоставленный администратор из крупной фирмы по поставкам ПО сказал мне, что хотя он сознает высокую ценность этих идей, но никогда не будет внедрять их в своей фирме, поскольку не хочет резать курицу, несущую золотые яйца. Более 90% доходов его фирма получает от "сдачи напрокат" личного состава, предоставляя заказчикам услуги своих аналитиков и программистов, и руководство фирмы стремится довести эту цифру до 100%. При таком отношении к разработке ПО навряд ли будет встречена с энтузиазмом перспектива появления общедоступных библиотек повторно используемых компонентов.

Это высказывание было примечательно своей откровенностью, но оно вызвало очевидное возражение: если вообще возможно создать повторно используемые компоненты, которые заменят некоторые дорогостоящие услуги консультантов из фирмы, поставляющей ПО, то рано или поздно кто-либо их создаст. А тогда фирма, отказывавшаяся пойти таким путем, и у которой не осталось ничего, кроме торговли услугами своих консультантов, может пожалеть о том, что, подобно испуганному страусу, зарыла голову в песок.

Технологическая составляющая (engineering part) в разработке ПО не идентична такой же составляющей в индустрии массового производства; человеческий фактор будет, вероятно, по-прежнему играть ключевую роль в процессе конструирования ПО.

Цель повторного использования состоит не в том, чтобы заменить людей инструментальными средствами (а это часто, несмотря на всяческие утверждения, происходит с другими отраслями производства), а в изменении соотношения между тем, что следует поручить людям, а что - инструментальным средствам. Так что для фирмы, приобретшей известность за счет своих консультантов, эти нововведения не так уж плохи. В частности:

- Во многих случаях разработчики, применяющие повторно используемые компоненты, могут по-прежнему успешно пользоваться помощью специалистов, которые посоветуют, как наилучшим образом применять эти компоненты. Тем самым сохраняется существенная роль фирм по поставкам ПО и их консультантов.
- Как будет показано ниже, возможность повторного использования неотделима от расширяемости: хорошие повторно используемые компоненты будут оставаться открытыми для адаптации к конкретным обстоятельствам. Консультанты фирмы, разработавшей соответствующую библиотеку программ, имеют

идеальную возможность выполнять настройку компонентов для отдельных заказчиков. Так что продажа компонентов и продажа услуг не обязательно являются взаимно исключающими видами деятельности; торговля компонентами может служить основой для торговли услугами.

- Хорошая повторно используемая библиотека может играть стратегическую роль в политике преуспевающей фирмы по производству ПО, даже если фирма продает решения, а не библиотеку, используя ее лишь для внутренних целей. Такая библиотека может дать фирме конкурентное преимущество в более быстрой и дешевой разработке нестандартных решений, удовлетворяющих требованиям заказчиков, чем могли бы сделать конкуренты, не опирающиеся на такую заранее заготовленную основу.

Организация доступа к компонентам

Вот что говорят скептики: прогресс в производстве повторно используемых ПО приведет к тому, что разработчики окажутся "заваленными" настолько большим количеством компонентов и это так усложнит их жизнь, что лучше бы этих компонентов не было.

Это высказывание следует рассматривать как предупреждение разработчикам повторно используемых ПО о том, что лучшие в мире повторно используемые компоненты бесполезны, если никто не знает об их существовании, или если для их получения придется затратить слишком много времени и усилий. Для практического успеха методов повторного использования требуется создание соответствующих баз данных, содержащих компоненты, запрос к которым позволял бы быстро выяснить, удовлетворяет ли нужным потребностям какой-либо из существующих компонентов.

Должны быть доступны и сетевые услуги, позволяющие осуществить заказ и немедленную доставку по сети выбранных компонентов.

Достижение этих целей требует решения технических и организационных проблем. Индексирование, поиск и доставка повторно используемых компонентов - это технические проблемы, решаемые известными средствами, в частности методами, основанными на использовании баз данных. Очевидно, справляясь с программными компонентами ничуть не сложнее, чем с данными о заказчиках, информацией об авиарейсах или с библиотечными книгами.

С созданием Всемирной паутины WWW появились мощные средства поиска, позволяющие намного проще размещать и отыскивать полезную информацию либо в Интернете, либо в корпоративной сети (Intranet). Несомненно, появятся и более совершенные решения (полученные, возможно, с помощью ОО-технологии). Из всего этого становится очевидным, что основной трудностью реализации повторного использования является не организация использования повторно используемых компонентов, а в первую очередь - создание этих чертовых штуковин.

Несколько слов об индексировании компонентов

На стыке технических и организационных проблем возникает вопрос: как следует связывать индексирующую информацию, например ключевые слова с программными компонентами?

Принцип Самодокументирования говорит о том, что вся информация о модуле, включая индексирующую информацию и другие виды документации, - должна содержаться в самом модуле. Это важное требование учтено при разработке нотации классов, развиваемой в лекциях 7-18 этого курса. Механизм предусматривает возможность подключения данных индексирования к каждому компоненту.

"Самодокументирование", [лекция 3](#).

Описание соответствующей синтаксической структуры не вызывает затруднений. В начале текста модуля предлагается написать **предложение индексирования (indexing clause)** в виде

```
Indexing
index_word1: value, value, value ...
index_word2: value, value, value ...
...
... Стандартное описание модуля (см. лекции 7-18) ...
```

Здесь каждое index_word (то есть - индексное слово) это идентификатор; каждое value (то есть - значение) это константа (целая, вещественная и т. д.), идентификатор, или какой либо другой стандартный лексический элемент. (Более подробно см. "Операторы индексирования", [лекция 8](#) курса "Основы объектно-ориентированного проектирования")

Конкретные ограничения на выбор индексных слов и соответствующих значений отсутствуют, но какая либо отрасль промышленности, ассоциация по вопросам стандартизации (standards group), организация или проектная группа может, при необходимости, определить свои правила. Средства индексирования и поиска могут затем извлекать эту информацию, чтобы помочь разработчикам ПО в отыскании компонентов,

удовлетворяющих определенным критериям.

Как показало обсуждение проблемы Самодокументирования, сохранение такой информации в самом модуле - а не во внешнем документе или базе данных - уменьшает вероятность ввода ложной информации и, в частности, не позволит забыть об обновлении информации при корректировке модуля (или наоборот). Операторы индексирования, довольно простые на первый взгляд, существенно помогают разработчикам приводить в порядок свои программные средства и регистрировать их свойства с тем, чтобы и другие разработчики могли о них узнать.

Форматы для распространения повторно используемых компонентов

Еще одной задачей, охватывающей как технические, так и организационные проблемы, является выбор представления для распространения: исходный текст или двоичный формат? Это спорный вопрос, и мы ограничимся рассмотрением только нескольких доводов с обеих сторон.

Разработчики коммерческого ПО часто распространяют лишь описание интерфейса (соответствующая **краткая форма (short form)** рассматривается в одной из последующих лекций) и исполняемый код. Тем самым разработчики защищают секреты производства и свои инвестиции. ("Использование утверждений (assertions) для документирования: сокращенная форма класса", [лекция 11](#))

Двоичный код и в самом деле является предпочтительной формой распространения коммерческих прикладных программ, операционных систем и других инструментальных средств, в том числе компиляторов, интерпретаторов и сред разработки для ОО-языков. Несмотря на непрекращающиеся нападки на такую концепцию, исходящие, в частности, от группы, называющейся Лигой Сторонников Свободного Программирования (League for Programming Freedom), маловероятно, что от такого способа распространения коммерческого ПО откажутся в ближайшем будущем. Но наше обсуждение относится не к обычным инструментальным средствам или прикладным программам: здесь рассматриваются библиотеки повторно используемых компонентов. В этом случае также могут быть найдены некоторые доводы в пользу распространения исходных текстов.

Для изготовителя программного компонента польза от распространения исходного текста состоит в том, что это облегчает перенос программ (porting efforts). Можно избежать утомительной и малорентабельной деятельности по адаптации ПО к множеству несовместимых платформ, существующих в современном компьютерном мире, рассчитывая на то, что разработчики ОО-компиляторов и программных сред выполнят эту работу за вас. (Для **потребителя** это, конечно, контраргумент, поскольку инсталляция исходного текста более трудоемка и может привести к непредвиденным ошибкам.)

Некоторые компиляторы создают вначале промежуточный код, который уже затем транслируется или интерпретируется на конкретной платформе. Это позволяет обеспечить мобильность ПО без полного доступа к исходному тексту. Если, как это часто бывает сейчас, в компиляторе формируется промежуточный код с использованием языка C, то вместо двоичного кода обычно можно поставлять переносимый код на языке C.²⁾

Этот прием использовался на разных этапах истории создания ПО, начиная с языка UNCOL (UNiversal COmputing Language - универсальный язык программирования), предложенного в конце пятидесятых годов 20-го века. Для него был определен формат команд, интерпретируемых на любой платформе, и представляющих промежуточный код, создаваемый компилятором. В связи с этой проблематикой в 1988 г. был сформирован консорциум компаний по компьютерным техническим средствам и программному обеспечению (Advanced computer environment (ACE) consortium). Вместе с языком Java появилась понятие байт-кода Java-компилятора, для которого были разработаны интерпретаторы на многих plataформах. Но для производителей компонентов вначале это привело к дополнительным трудозатратам. Необходима двойная гарантия того, что новый формат пригоден на любой платформе, представляющей практический интерес, и что не происходит потери эффективности выполнения промежуточного кода в сравнении с платформо-специфическим решением. Без этих гарантий нельзя будет отказаться от старой технологии и придется просто добавить новый формат кода к тем форматам, которые поддерживались ранее. Таким образом, решение, которое рекламируется как завершающее все проблемы переносимости, фактически на некоторое время порождает дополнительные проблемы переносимости.

Возможно, более важным доводом в пользу распространения текста исходного кода является то, что попытки защитить свои изобретения и секреты производства путем удаления исходного текста из реализации программного продукта могут не приносить никакой существенной пользы. Самая трудоемкая работа при составлении хорошей повторно используемой библиотеки связана с проектированием интерфейсов компонентов, а не с реализацией; и именно это вы вынуждены опубликовать. Это особенно очевидно в мире структур данных и алгоритмов, для которых почти все необходимые методы описаны в литературе по компьютерным наукам. Чтобы успешно создать библиотеку, требуется встроить эти методы в модули, интерфейс которых сделает их полезными для разработчиков многих других приложений. Такое проектирование интерфейса является частью того, что вы должны выпустить в свет.

Важно отметить, что в случае ОО-модулей имеются две формы повторного использования компонентов:

клиентами класса и наследниками класса. Вторая из этих форм объединяет повторное использование с расширяемостью. Описание интерфейсов (краткая форма) достаточно для клиентов, но не всегда достаточно для повторного использования на основе наследования.

Наконец, о педагогической стороне проблемы. Распространение исходных текстов библиотечных модулей является средством представления лучших образцов разработки ПО, способствующее разработке потребителями ПО в соответствующем стиле. Возникающая при этом стандартизация является одним из достоинств повторного использования. В определенной степени это будет иметь место даже в случае, когда доступны лишь интерфейсы, но лучше всего иметь полный текст.(Этот вопрос обсуждается в лекции, посвященной обучению ОО-технологии, в [лекции 11](#) курса "Основы объектно-ориентированного проектирования".)

Заметьте, что даже если доступен исходный код, то он не должен служить в качестве основного средства документации: для этого по-прежнему будет использоваться интерфейс модуля.

Это обсуждение затронуло некоторые спорные экономические вопросы, обусловленные отчасти появлением промышленного производства компонентов ПО и, в более общем плане, прогрессом в области ПО. Как же справедливо вознаградить разработчиков за их достижения и обеспечить приемлемую степень защиты их изобретений, не нарушая законных интересов пользователей? Существуют две противоположные точки зрения:

- С одной стороны, это принципы Лиги Сторонников Свободного Программирования (League for Programming Freedom): все ПО должно быть бесплатным и доступным в форме исходных текстов.(См. библиографические замечания.)
- С другой стороны, имеется идея **суперпоставки (superdistribution)**, предложенная Брэдом Коксом (Brad Cox) в нескольких статьях и книге. Суперпоставка должна дать возможность пользователям свободно копировать программы, оплачивая не их приобретение, а каждое использование. Представьте себе небольшой счетчик, присоединенный к каждому программному компоненту, который "выбивает" сумму в несколько пенсов всякий раз, когда вы пользуетесь этим компонентом, и в конце каждого месяца предъявляет вам соответствующий счет. Это, по-видимому, исключает возможность распространения исходных текстов, так как тогда было бы очень просто удалить из программы команды счетчика. Японская ассоциация по развитию электронной промышленности JEIDA (Japanese Electronic Industry Development Association) работает над механизмами создания технических и программных компьютерных средств поддержки такой концепции. Сам Кокс недавно подчеркнул особую роль не только технологических методов, а механизмов принуждения, основанных на соответствующих правовых нормах (наподобие авторского права). Пока идея суперпоставки вызывает множество технических, экономических и психологических вопросов.

Оценка

При любом всестороннем подходе к проблемам повторного использования следует наряду с техническими аспектами рассмотреть организационные и экономические вопросы: как сделать повторное использование частью культуры разработки ПО, как найти правильную структуру стоимости и правильную форму распространения компонентов, создать соответствующие средства для индексирования и поиска компонентов. Неудивительно, что эти вопросы легли в основу основных инициатив по повторному использованию, исходивших от правительства и больших корпораций, таких как программа STARS (**Software Technology for Adaptable, Reliable Systems - Технология создания ПО для адаптивных, надежных систем**) Министерства обороны США и "фабрики ПО" ("software factories"), введенные в действие некоторыми большими японскими фирмами.

Являясь важными в долгосрочной перспективе, эти вопросы не должны отвлекать внимания от главных проблем, являющихся все еще техническими. Для успешной реализации возможностей повторного использования требуется создание правильных модульных структур и высококачественных библиотек, содержащих десятки тысяч компонентов, необходимых индустрии.

Оставшаяся часть данной лекции посвящена первому из этих вопросов. В ней выясняется, почему общепринятые понятия модуля непригодны для широкомасштабного повторного использования, и определяются требования, которым должно удовлетворять лучшее решение, предлагаемое в последующих лекциях.

Техническая проблема

Как же должен выглядеть повторно используемый модуль?

Изменения и постоянство

Разработка ПО, как уже упоминалось, во многом связана с повторяемостью. Для понимания технической трудности повторного использования, следует понять природу повторяемости.

Несмотря на то, что программисты обычно время от времени повторяют одни и те же действия, но эти действия являются не совсем одинаковыми. Ведь если бы они были одинаковыми, то решение оказалось бы простым, по крайней мере, на бумаге. Однако на практике может измениться настолько много деталей задачи, что любая бесхитростная попытка обеспечить ее унификацию потерпит неудачу.

Наглядной иллюстрацией являются работы норвежского художника Эдварда Мунка, многие из которых можно видеть в посвященном ему музее в Осло, на родине языка программирования Simula. Творчеством Мунка владели несколько жизненно-важных, глубоких тем: любовь, страдание, ревность, танец, смерть. Он без конца воспроизводил их в своих рисунках и живописи, пользуясь всякий раз одними и теми же образцами, но меняя технические приемы, цвета, резкость контуров, размер, освещение, настроение.

В таком же положении находится и разработчик ПО, создавая новые варианты, развивающие одни и те же основные темы.

Возьмем пример, упоминавшийся в начале этой лекции: табличный поиск. Несомненно, алгоритм табличного поиска в общем виде всегда выглядит одинаково: начать с некоторой позиции в таблице t , затем приступить к последовательному просмотру таблицы, всякий раз проверяя, является ли искомым элемент в текущей позиции и, если это не так, то переходить к следующей позиции. Процесс завершается, если найден нужный элемент, либо проверка всех элементов оказалась безуспешной. Такая общая схема применима к многим возможным случаям представления данных и алгоритмам для табличного поиска, в том числе в массивах (отсортированных или не отсортированных), связных списках (отсортированных или не отсортированных), последовательных файлах, двоичных деревьях, Б-деревьях и различных хеш-таблицах.

Нетрудно превратить это неформальное описание в частично детализированную подпрограмму:

```
has (t: TABLE, x: ELEMENT): BOOLEAN is
    -- Присутствует ли x в t?
    local
        pos: POSITION
    do
        from
            pos := INITIAL_POSITION (x, t)
        until
            EXHAUSTED (pos, t) or else FOUND (pos, x, t)
        loop
            pos := NEXT (pos, x, t)
        end
        Result := not EXHAUSTED (pos, t)
    end
```

Некоторые пояснения к принятой здесь нотации: `from ... until ... loop ... end` описывает цикл, с начальным условием в предложении `from`, ни разу или повторно выполняющий действия предложения `loop`, и завершающийся при выполнении условия предложения `until`. Переменная `Result` содержит значение, возвращаемое функцией `has`. Если вы незнакомы с оператором `or else` (Оператор `or else` объясняется в [лекции 13](#)), то считайте, что здесь содержится просто логическое `or`.

Хотя приведенный выше текст описывает общую схему работы алгоритма, он не является непосредственно выполняемым, поскольку содержит некоторые не вполне определенные фрагменты (написанные заглавными буквами). Они соответствуют аспектам задачи табличного поиска, зависящим от выбранной реализации: тип элементов таблицы (`ELEMENT`), с какой позиции начинать поиск (`INITIAL_POSITION`), как переходить от текущей позиции к следующей (`NEXT`), как проверить наличие искомого элемента на некоторой позиции (`FOUND`), как определить, что все интересующие нас позиции уже проверены (`EXHAUSTED`).

Поэтому вышеприведенный текст является не столько подпрограммой, а шаблоном подпрограммы, который можно превратить в действующую подпрограмму, лишь после уточнения фрагментов, написанных заглавными буквами.

Повторно использовать или переделать? (The reuse-redo dilemma)

Наличие всех этих вариантов выдвигает на первый план проблемы, возникающие при любой попытке размышлять над созданием модулей общего назначения в заданной прикладной области: как же воспользоваться наличием единого шаблона для согласования с таким большим числом различных вариантов? Это не только проблема реализации: почти так же трудно **специфицировать** модуль таким образом, чтобы модули-клиенты могли рассчитывать на взаимодействие с ним, не располагая его реализацией.

По этим соображениям обречены на неуспех простые решения проблемы повторного использования. Ввиду многосторонности и изменчивости ПО - не зря оно называется "soft - модули, не обладающие "гибкостью", не могут претендовать на возможность повторного использования.

"Замороженность" модуля приводит к дилемме - **повторно использовать или переделать**: повторно использовать модуль таким, какой он есть, или заново все переделать. Оба подхода слишком ограничительные. Типичная ситуация, когда существует модуль, обеспечивающий лишь частичное решение текущей задачи, и требуется адаптация модуля к конкретным потребностям. В этом случае желательно и повторно использовать и переделать: кое что повторно использовать, а кое что переделать - или, лучше всего, многое повторно использовать, а совсем немного переделать. Без способности объединения возможностей повторного использования и адаптации, методы повторного использования не могут удовлетворять практическим потребностям разработки ПО.

Поэтому не случайно почти любое обсуждение проблем повторного использования в этой книге затрагивает и проблему расширяемости (что приводит к охватывающему оба эти понятия термину "модульность", являющемуся предметом обсуждения в предыдущей лекции). Всякий раз, когда вы начнете искать ответы на одно из этих требований, вы тут же столкнетесь и с другим требованием.

Такая взаимозависимость между повторным использованием и расширяемостью отмечалась ранее при обсуждении принципа Открыт-Закрыт. (См. "Принцип Открыт-Закрыт", [лекция 3](#))

Поиску подходящего представления модуля посвящена оставшаяся часть этой лекции и несколько следующих лекций. Нам предстоит согласовать между собой возможность повторного использования и расширяемость, закрытость и открытость, постоянство и изменчивость. Нам следует удовлетворить сегодняшние потребности и попытаться отгадать, что же понадобится завтра.

Пять требований к модульным структурам

Как же найти такие модульные структуры, которые позволят создать компоненты, непосредственно готовые к повторному использованию, и, в то же время, допускающие возможность их адаптации?

Задача табличного поиска и шаблон подпрограммы has иллюстрируют жесткие требования, предъявляемые к любому решению. Можно воспользоваться этим примером для выяснения, что же следует предпринять для перехода от обнаружения относительно нечеткой общности вариантов к реальному набору повторно используемых модулей. Такой анализ выявляет пять важных проблем:

- Изменчивость Типов (Type Variation).
- Группирование Подпрограмм (Routine Grouping).
- Изменчивость Реализаций (Implementation Variation).
- Независимость Представлений (Representation Independence).
- Факторизация Общего Поведения (Factoring Out Common Behaviors).

Изменчивость Типов (Type Variation)

Шаблон подпрограммы has предполагает, что таблица содержит объекты типа ELEMENT. При уточнении этой подпрограммы в применении к частному случаю можно использовать конкретный тип, например INTEGER или BANK_ACCOUNT, для таблицы целых чисел или банковских счетов.

Но это не совсем то, что требуется. Повторно используемый модуль поиска должен быть применим ко многим различным типам элементов без того чтобы пользователи вынуждены были производить "вручную" изменения в тексте программы. Другими словами, необходимо средство для описания модулей, в которых типы выступают в роли параметров (type-parameterized), или короче - **родовых** (полиморфных) модулей. Универсальность или полиморфность (genericity) (способность модулей быть родовыми) окажется важной частью ОО-метода; обзор этой концепции дается далее в этой лекции. (См. "Универсальность" ("Genericity"), [лекция 4](#))

Группирование Подпрограмм (Routine Grouping)

Шаблон подпрограммы has, даже если его полностью детализировать и ввести параметризацию типа, все еще не будет пригоден в качестве повторно используемого компонента. Поиск в таблице зависит от того, как таблица создавалась, как в нее включаются элементы, как они удаляются. Отдельно взятая программа поиска - это еще не модуль повторного использования. Самодостаточный, повторно используемый модуль должен включать множество подпрограмм, обеспечивающих каждую из упомянутых операций - создание, включение, удаление, поиск.

Эта идея лежит в основе формирования модуля как "пакета", что имеет место в языках с инкапсуляцией таких как: Ada, Modula-2 и родственных им языках. Более подробно об этом будет сказано ниже.

Изменчивость Реализаций (Implementation Variation)

Шаблон has является весьма общим; и, как мы уже убедились, на практике имеется широкий выбор соответствующих структур данных и алгоритмов. Нельзя ожидать, что один модуль сможет обеспечить работу

в столь разнообразных условиях, - он оказался бы просто огромным. Для охвата всех возможных реализаций требуется семейство модулей.

Общая методика создания и применения повторно используемых модулей должна поддерживать идею семейства модулей.

Независимость Представлений

Общая структура повторно используемого модуля должна позволять модулям-клиентам определять свои действия при отсутствии сведений о реализации модуля. Это требование называется Независимостью Представлений.

Предположим, что модулю-клиенту С некоторой прикладной системы (управления ресурсами банка, компилятора, системы географической информации) необходимо определить, содержит ли некоторый элемент х в некоторой таблице t (вкладов, слов языка, городов). Независимость Представлений для С означает возможность получить такую информацию с помощью обращения к подпрограмме

```
present := has (t, x)
```

не зная, какой вид имеет таблица t во время этого обращения. Автору модуля С нужно лишь знать, что t это таблица из элементов определенного типа, и что x означает объект того же типа. Ему безразлично, является ли t деревом двоичного поиска, хеш-таблицей или связным списком. Он должен иметь возможность сосредоточиться на своей задаче управления активами, компиляции или географии.

Выбор подходящего алгоритма поиска, основанного на реализации таблицы t, является делом лишь того модуля, который организует эту таблицу.

Модуль-клиент С, содержащий упомянутое обращение к подпрограмме, мог бы получить t от одного из своих собственных клиентов (в виде аргумента вызова подпрограммы). Тогда для С имя t является лишь абстрактным идентификатором структуры данных, к детальному описанию которой он и не может иметь доступа.

Можно рассматривать Независимость Представлений как расширение правила Скрытия Информации (инкапсуляции), существенное для беспрепятственной разработки больших систем: решения по реализации могут часто изменяться, и клиенты должны быть защищены от этого (См. "Скрытие информации", [лекция 3](#)). Но требование Независимости Представлений идет еще дальше. Если обратиться к его полномасштабным последствиям, то оно означает защиту клиентов модуля от изменений не только во время **жизненного цикла проекта, но и во время выполнения** - а это намного меньший временной интервал! В рассматриваемом примере, желательно, чтобы подпрограмма has адаптировалась автоматически к виду таблицы t во время выполнения программы, даже если этот вид изменился со времени последнего обращения к подпрограмме.

Выполнение требования Независимости Представлений поможет также реализовать связанный с ним принцип Единственного Выбора, сформулированный при обсуждении модульности, который предписывает избегать ситуаций, связанных с разбором вариантов, например

```
if "t это массив, управляемый хешированием" then
    "Применить поиск с хешированием"
elseif "t это дерево двоичного поиска" then
    "Применить обход дерева двоичного поиска"
elseif
    (и т.д.)
end
```

Было бы в равной степени неудобно иметь такую структуру в самом модуле (нельзя же ожидать, что модуль, организующий таблицу, знает обо всех текущих и будущих вариантах), так и воспроизводить ее в каждом модуле-клиенте. (См. "Единственный выбор", [лекция 3](#)) Решение состоит в том, чтобы обеспечить автоматический выбор, осуществляемый системой исполнения. Такова будет роль **динамического связывания (dynamic binding)**, ключевой составляющей ОО-подхода, которая подробно будет рассматриваться при обсуждении наследования. (См. "Динамическое связывание" ("Dynamic binding"), [лекция 14](#))

Факторизация Общего Поведения

Если требование Независимости Представлений отражает позицию клиента - игнорирование внутренних деталей и вариантов реализации - то последнее требование отражает позицию разработчиков повторно используемых классов. Их цель в получении преимуществ от любой общности (commonality), которая может существовать в семействе или подсемействе реализаций.

Многообразие реализаций, имеющее место в некоторых проблемных областях, требует, как уже отмечалось,

решения, основанного на семействе модулей. Часто это семейство настолько велико, что естественно поискать соответствующие подсемейства. В случае табличного поиска первая попытка классификации может привести к трем обширным подсемействам:

- Таблицы, организуемые по некоторой схеме хеширования.
- Таблицы, организуемые как некоторая разновидность деревьев.
- Таблицы, организуемые последовательно.

Каждая из этих категорий охватывает много вариантов, но в большинстве случаев можно найти существенную общность между этими вариантами. Рассмотрим, например, семейство последовательных реализаций - таких, в которых элементы сохраняются и отыскиваются в порядке их первоначального включения в таблицу.

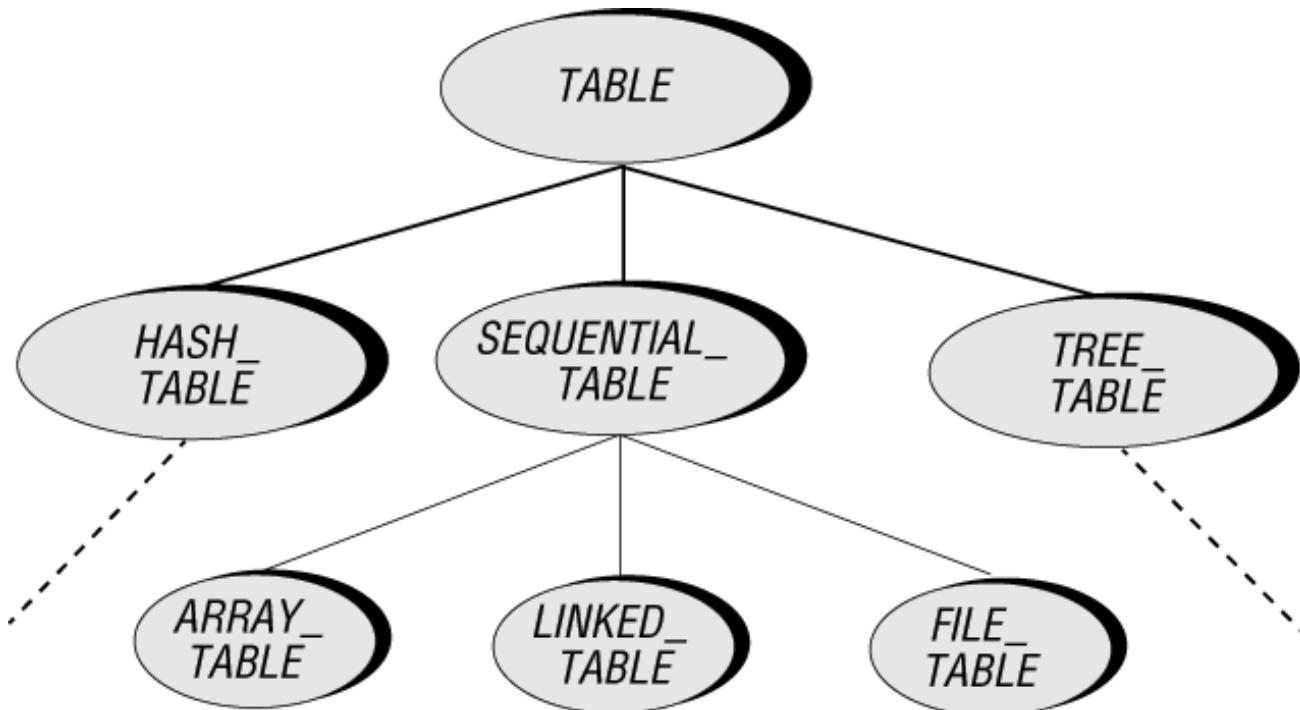


Рис. 4.1. Некоторые возможные реализации таблицы

Возможными представлениями последовательной таблицы являются массив, связный список и файл. Но независимо от варианта такой реализации, клиенты должны иметь возможность для любой последовательно организованной таблицы рассматривать ее элементы один за другим, перемещая (воображаемый) **курсор**, указывающий позицию элемента, рассматриваемого в настоящий момент. При таком подходе можно переписать подпрограмму поиска для последовательных таблиц в виде:

```
has (t: SEQUENTIAL_TABLE; x: ELEMENT): BOOLEAN is
    -- Содержится ли x в последовательной таблице t?
    do
        from start until
            after or else found (x)
        loop
            forth
        end
        Result := not after
    end
```

Это представление основано на использовании четырех подпрограмм, которые должны иметься в любой последовательной реализации таблицы (Подробно методика работы с курсором будет рассмотрена в [лекции 5](#) курса "Основы объектно-ориентированного проектирования""Активные структуры данных" ("Active data structures").):

- **start** (начать) , переместить курсор к первому элементу, если он имеется.
- **forth** (следующий) , переместить курсор к следующей позиции.
- **after** (после) , булев запрос, переместился ли курсор за последний элемент.
- **found (x)** , булев запрос, возвращающий true, когда курсор указывает на элемент, имеющий значение x.

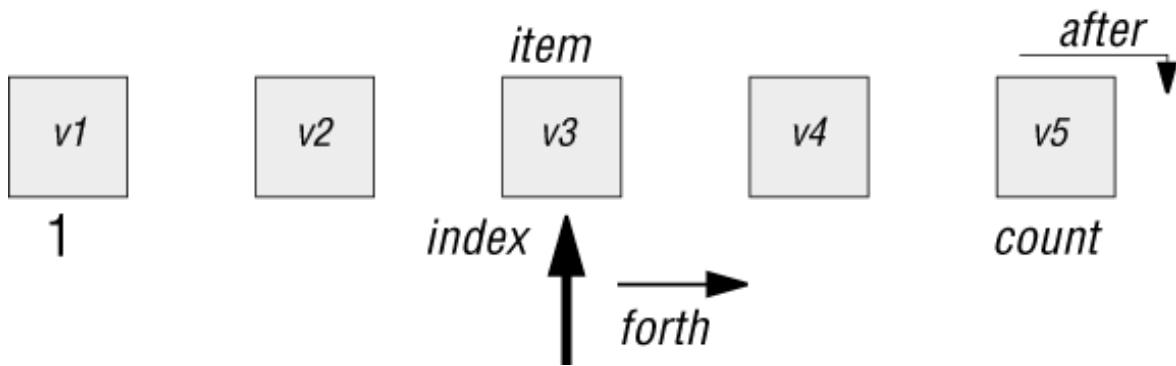


Рис. 4.2. Последовательная структура с курсором

Несмотря на сходство с шаблоном подпрограммы, использованным в начале этого обсуждения, новый текст - это уже не шаблон, это настоящая подпрограмма, написанная в непосредственно исполняемой нотации (такая нотация используется в лекциях 7-18 этого курса). Если задать реализации для четырех операций start, forth, after и found, то можно откомпилировать и выполнить последнюю версию has.

Каждое представление последовательной таблицы требует соответствующего представления курсора. Три примера таких представлений основаны на работе с массивом, связным списком и файлом.

В первом из них используется массив из `capacity` элементов, и таблица занимает позиции от 1 до `count + 1`. (Последнее значение необходимо в случае, когда курсор переместился на позицию после ("after") последнего элемента.)

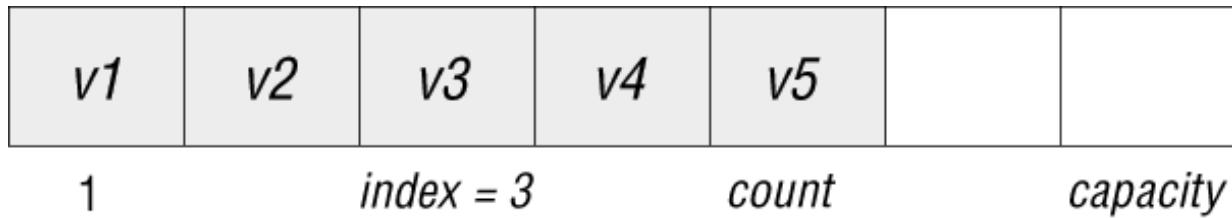


Рис. 4.3. Представление последовательной таблицы с курсором на основе массива

Во втором представлении используется связный список, в котором доступ к первому элементу обеспечивается по ссылке `first_cell` и каждый элемент связан со следующим по ссылке `right`. При этом курсор можно представить ссылкой `cursor`.

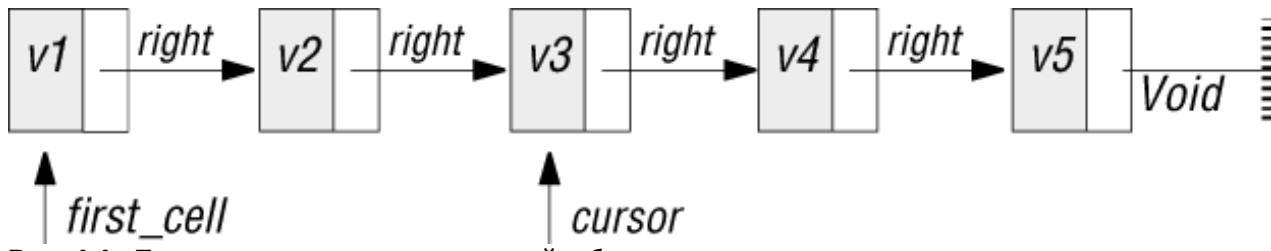


Рис. 4.4. Представление последовательной таблицы с курсором на основе связного списка

В третьем представлении используется последовательный файл, в котором курсор представляет просто текущую позицию чтения.

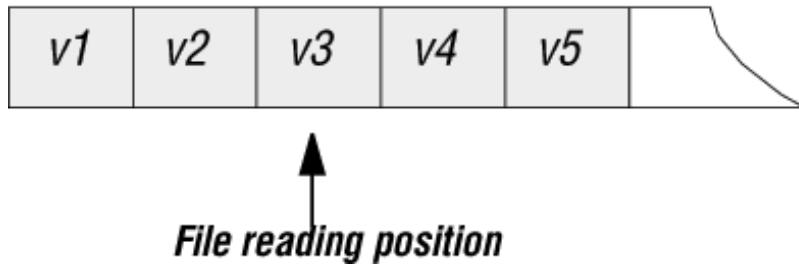


Рис. 4.5. Представление последовательной таблицы с курсором на основе последовательного файла

Реализация операций `start`, `forth`, `after` и `found` будет разной для каждого из вариантов. В следующей таблице³¹ показана реализация для каждого случая. Здесь `t @ i` означает *i*-й элемент массива `t`, который записывается как `t[i]` в языках Pascal или C; `Void` означает "пустую" ссылку; обозначение `f` - языка Pascal, для файла `f`, означает элемент в текущей позиции чтения из файла.

	start	forth	after	found (x)
Массив	i :=1	i :=i + 1	i >count	t @ i =x
Связный список	c := first_cell	c :=c. right	c =Void	c. item =x
Файл	rewind	read	end_of_file	f -=x

Повторное использование позволяет избежать ненужное дублирование, используя общность вариантов. Если в разных модулях появляются одинаковые или почти одинаковые фрагменты, то трудно обеспечить их целостность и гарантировать, что изменения или поправки достигли всех требуемых мест системы. Вновь могут возникнуть проблемы с управлением конфигурацией системы.

Все варианты последовательной таблицы совместно используют функцию `has`, и отличаются только реализацией операций. Хорошее решение проблемы повторного использования требует, чтобы в такой ситуации текст `has` находился бы лишь в одном месте, связанном с общим понятием последовательной таблицы. Для описания каждого нового варианта не нужно больше беспокоиться о подпрограмме `has`; требуется лишь подготовить подходящие версии `start`, `forth`, `after` и `found`.

Традиционные модульные структуры

Наряду с требованиями к модульности, изложенными в предыдущей лекции, пять требований Изменчивости Типов, Группирования Подпрограмм, Изменчивости Реализаций, Независимости Представлений и Факторизации Общего Поведения определяют, чего следует ожидать от наших повторно используемых компонентов - абстрактных модулей.

Рассмотрим решения, предшествовавшие ОО-подходу, чтобы понять, что нас не устраивает, и что следует взять с собой в ОО-мир.

Подпрограммы

Классический подход к повторному использованию состоит в том, чтобы создавать библиотеки подпрограмм. Здесь термин **подпрограмма (routine)** означает программный элемент, который может быть вызван другими элементами для выполнения некоторого алгоритма, используя некоторые входные данные, создавая некоторые выходные данные, и, возможно, модифицируя другие данные. Вызывающий элемент передает свои входные данные (а иногда - выходные данные и модифицируемые данные) в виде **фактических аргументов (actual arguments)**. Подпрограмма может также возвращать выходные данные в виде **результата**; в этом случае она называется **функцией**.

Библиотеки подпрограмм успешно использовались в различных прикладных областях, в частности, для численных расчетов, где применение отличных библиотек привело к первым сообщениям об успехах повторного использования. Декомпозицию систем на подпрограммы, функциональную декомпозицию, обеспечивает также метод нисходящего (сверху вниз) программирования. Подход, основанный на использовании библиотек подпрограмм, хорошо работает в случаях, когда можно определить множество (возможно - большое) отдельных задач, при наличии следующих ограничений:

- **R1** Каждая задача допускает простую спецификацию. Точнее, возможно охарактеризовать каждую отдельную задачу небольшим набором входных и выходных параметров.
- **R2** Задачи четко отличаются одна от другой, поскольку подход, основанный на подпрограммах, не позволяет воспользоваться возможной сколько-нибудь существенной их общностью - за исключением повторного использования некоторых конструкций.
- **R3** Отсутствуют сложные структуры данных, которые пришлось бы распределять между использующими их подпрограммами.

Поиск в таблице является хорошим примером ограниченных возможностей подпрограмм. Мы уже убедились, что подпрограмма поиска сама по себе не содержит достаточного контекста, чтобы служить в качестве функционально-завершенного модуля повторного использования. Даже если не обращать внимания на этот недостаток, мы столкнемся с двумя в равной степени неприятными решениями:

- Подпрограмма поиска существует в одном варианте. Но тогда, чтобы охватить все возможные ситуации, ей потребуется длинный список аргументов и она окажется очень сложной.
- Подпрограмм поиска много, каждая из которых относится к конкретному случаю и отличается от других лишь немногими деталями. Нарушается требование Факторизации Общего Поведения; возможные пользователи легко могут заблудиться в неразберихе подпрограмм.

В целом, подпрограммы являются недостаточно гибкими, чтобы удовлетворять потребностям повторного использования. Мы уже видели тесную связь между возможностью повторного использования и расширяемостью. Повторно используемый модуль должен быть открыт для расширения, но в случае подпрограммы единственным средством адаптации является передача аргументов. Это делает нас заложником дилеммы - Повторно использовать или Переделать: либо пользоваться этой подпрограммой в ее исходном

виде, либо написать собственную подпрограмму.

Пакеты

В семидесятые годы двадцатого века, в связи с развитием идей скрытия информации и абстракции данных, возникла необходимость в форме модуля, более совершенном, чем подпрограмма. Появились несколько языков проектирования и программирования, наиболее известные из них: CLU, Modula-2 и Ada. В них предлагается сходная форма модуля, называемого в языке Ada пакетом, CLU - кластером, Modula - модулем. В нашем обсуждении будет использоваться термин пакет.⁴⁾

Пакеты - это единицы программной декомпозиции, обладающие следующими свойствами:

- **P1** В соответствии с принципом Лингвистических Модульных Единиц, "пакет" это конструкция языка, так что каждый пакет имеет имя и синтаксически четко определенную область.
- **P2** Описание каждого пакета содержит ряд объявлений связанных с ним элементов, таких как подпрограммы и переменные, которые в дальнейшем будут называться компонентами (features) пакета.
- **P3** Каждый пакет может точно определять права доступа, ограничивающие использование его компонентов другими пакетами. Другими словами, механизм пакетов поддерживает скрытие информации.
- **P4** В компилируемом языке (таком, который может быть использован для реализации, а не только для спецификации и проектирования) поддерживается независимая компиляция пакетов.

Благодаря свойству **P3**, пакеты можно рассматривать как абстрактные модули. Их главным вкладом в программирование является свойство **P2**, удовлетворяющее требованию Группирования Подпрограмм. Пакет может содержать любое количество связанных с ним операций, таких как создание таблицы, включение, поиск и удаление элементов. И нетрудно увидеть, как решение, основанное на использовании пакета, будет работать в рассматриваемом здесь примере табличного поиска. Ниже - в системе обозначений, заимствованной из нотации, используемой в последующих лекциях этого курса для ОО-ПО - приводится набросок пакета INTEGER_TABLE_HANDLING, описывающий частную реализацию таблиц целых чисел, основанную на использовании двоичных деревьев:

```
package INTEGER_TABLE_HANDLING feature
    type INTBINTREE is
        record
            -- Описание представления двоичного дерева, например:
            info: INTEGER
            left, right: INTBINTREE
        end
    new: INTBINTREE is
        -- Возвращение нового инициализированного INTBINTREE.
        do ... end
    has (t: INTBINTREE; x: INTEGER): BOOLEAN is
        -- Содержится ли x в t?
        do ... Реализация операции поиска ... end
    put (t: INTBINTREE; x: INTEGER) is
        -- Включить x в t.
        do ... end
    remove (t: INTBINTREE; x: INTEGER) is
        -- Удалить x из t.
        do ... end
end -- пакета INTEGER_TABLE_HANDLING
```

Этот пакет содержит объявление типа (`INTBINTREE`), и ряда подпрограмм, представляющих операции над объектами этого типа. В данном примере не потребовалось описания переменных пакета (хотя в подпрограммах могут иметься локальные переменные).

Пакеты-клиенты теперь могут работать с таблицами, используя различные методы из `INTEGER_TABLE_HANDLING`. Введем синтаксическое соглашение, позволяющее клиенту пользоваться методом `f` из пакета, для чего позаимствуем нотацию из языка CLU: `P$f`. В нашем примере типичные фрагменты программного текста клиента могут иметь вид:

```
-- Вспомогательные описания:
x: INTEGER; b: BOOLEAN
-- Описание t типа, определенного в INTEGER_TABLE_HANDLING:
t: INTEGER_TABLE_HANDLING$INTBINTREE
-- Инициализация t новой таблицей, создаваемой функцией new пакета:
t := INTEGER_TABLE_HANDLING$new
-- Включение x в таблицу, используя процедуру put пакета:
```

```

INTEGER_TABLE_HANDLING$put (t, x)
    -- Присваивание True или False переменной b,
    -- для поиска используется функция has пакета:
b := INTEGER_TABLE_HANDLING$has (t, x)

```

Отметим необходимость введения двух связанных между собой имен: одного для модуля, здесь это INTEGER_TABLE_HANDLING, и одного для его основного типа данных, здесь это INTBINTREE. Одним из ключевых шагов к ОО-программированию явится объединение этих двух понятий. Но не будем опережать события.

Менее важной проблемой является утомительная необходимость неоднократно писать имя пакета (здесь это INTEGER_TABLE_HANDLING). В языках, поддерживающих работу с пакетами, эта проблема решается с помощью различных сокращенных синтаксических конструкций (shortcuts), таких как, например, в языке Ada:

```

with INTEGER_TABLE_HANDLING then
    ... Здесь has означает INTEGER_TABLE_HANDLING$has, и т.д. ...
end

```

Другим очевидным недостатком пакетов рассмотренного вида является их неспособность удовлетворять требованию Изменчивости Типов: приведенный выше модуль пригоден лишь для таблиц целых чисел. Однако, вскоре мы увидим, как устранить этот недостаток, делая пакеты универсальными (generic).

Механизм пакетов обеспечивает скрытие информации, ограничивая права клиентов на доступ к компонентам. Показанный выше клиент был в состоянии объявить одну из своих собственных переменных, используя тип INTBINTREE, взятый от своего поставщика, и вызывать подпрограммы, описанные этим поставщиком. Но он не имеет доступа ни к внутреннему описанию этого типа (к структуре record, определяющей реализацию таблиц), ни к телу подпрограмм (здесь это операторы do). Кроме того, можно скрыть от клиентов некоторые компоненты пакета (переменные, типы, подпрограммы), делая их используемыми только в тексте пакета.

Языки, поддерживающие работу с пакетами, несколько отличаются своими механизмами скрытия информации. Например, в языке Ada, внутренние свойства типа, такого как INTBINTREE, будут доступны клиентам, если не объявить тип как **private** (закрытый).

Часто для усиления скрытия информации в языках с инкапсуляцией предлагается объявлять пакет, состоящий из двух частей, интерфейса (interface) и реализации (implementation)(См. [лекция 11](#) и [лекция 5](#) курса "Основы объектно-ориентированного проектирования"). Закрытые элементы, такие как объявление типа или тело подпрограммы, включаются в раздел реализации. Однако такой подход приводит к добавочной работе для разработчиков модулей, заставляя их дублировать заголовки объявлений компонентов. При глубоком осмыслении правила Скрытия Информации все это не требуется. Подробнее эта проблема обсуждается в последующих лекциях.

Пакеты: оценка

По сравнению с подпрограммами, механизм пакетов приводит к существенному совершенствованию разбиения системы ПО на абстрактные модули. Собрать нужные компоненты "под одной крышей" крайне полезно как для поставщиков, так и для клиентов:

- Автор модуля-поставщика может хранить в одном месте и совместно компилировать все элементы, относящиеся к некоторому заданному понятию. Это облегчает отладку и изменения. В отличие от этого, при использовании отдельных самостоятельных подпрограмм всегда есть опасность забыть произвести обновление некоторых подпрограмм при изменениях проекта или реализации; например, можно обновить new, put и has, но забыть обновить remove.
- Для авторов модулей-клиентов несомненно легче найти и использовать множество взаимосвязанных компонентов, если все они собраны в одном месте.

Преимущество пакетов по сравнению с подпрограммами особенно очевидно в таких случаях, как рассмотренный здесь пример с таблицей, где в пакете собраны все операции, применимые к конкретной структуре данных.

Однако пакеты все же не обеспечивают полного решения проблем повторного использования. Как уже отмечалось, они отвечают требованию Группирования Подпрограмм, но не удовлетворяют всем остальным требованиям. В частности, они не обеспечивают возможности факторизации общего поведения - "вынесения за скобки" общих компонентов. Заметим, что INTEGER_TABLE_HANDLING в нашем наброске текста пакета основывается на одном частном выборе реализации, - двоичных деревьев поиска. Конечно, благодаря скрытию информации, клиентам незачем интересоваться этим выбором. Но библиотека повторно используемых компонентов должна будет содержать модули для многих различных реализаций. Возникающую при этом ситуацию нетрудно предвидеть: типичная библиотека пакетов будет предлагать массу похожих, но вовсе не идентичных, модулей для заданной прикладной области, например, для работы с таблицами, но без какого-либо учета их общности. Обеспечивая возможность повторного использования для клиентов, такая методика

приносит в жертву возможность повторного использования со стороны поставщиков.

Но даже со стороны клиентов ситуация остается не вполне приемлемой. Каждое использование таблицы клиентом требует упомянутого выше объявления вида:

```
t: INTEGER_TABLE_HANDLING$INTBINTREE
```

Клиент вынужден выбирать конкретную реализацию. Этим нарушается требование Независимости Представлений: авторы модулей-клиентов должны знать больше о реализациях представлений модуля-поставщика, чем это принципиально необходимо.

Перегрузка и универсальность

Два технических приема - перегрузка (overloading) и универсальность (genericity) предлагают свои решения, направленные на достижение большей гибкости описанных выше механизмов. Рассмотрим, что же они могут дать.

Синтаксическая перегрузка

Перегрузка - это связывание с одним именем более одного содержания. Наиболее часто перегружаются имена переменных: почти во всех языках программирования различные по смыслу переменные могут иметь одно и то же имя, если они принадлежат различным модулям (различным блокам - в языке Algol и подобных ему).

Для этого обсуждения более существенной является **перегрузка подпрограмм**, частным случаем которой является **перегрузка операторов**, которая позволяет использовать одинаковые имена для нескольких подпрограмм. Такая возможность почти всегда имеет место для арифметических операторов: одна и та же запись, $a + b$, означает различные виды сложения, в зависимости от типов a и b (целые, вещественные с обычной точностью, вещественные с удвоенной точностью). Начиная с языка Algol 68, в котором допускалась перегрузка основных операторов, некоторые языки программирования распространили возможность перегрузки на операции, определяемые пользователем, и на обычные подпрограммы.

Например, в языке Ada пакет может содержать несколько подпрограмм с одним и тем же именем, но с разной сигнатурой, определяемой здесь числом и типами аргументов. В общем случае сигнатура функций содержит также тип результата, но язык Ada разрешает перегрузку, учитывающую только аргументы. Например, пакет может содержать несколько функций square:⁵⁾

```
square (x: INTEGER): INTEGER is do ... end
square (x: REAL): REAL is do ... end
square (x: DOUBLE): DOUBLE is do ... end
square (x: COMPLEX): COMPLEX is do ... end
```

Тогда при вызове square (y) тип аргумента у определит, какой вариант подпрограммы имелся в виду.

Подобным же образом, пакет может описывать набор функций поиска одинакового вида:

```
has (t: "SOME_TABLE_TYPE"; x: ELEMENT) is do ... end
```

Каждая из них задает свою реализацию и отличается фактическим типом, используемым вместо "SOME_TABLE_TYPE". Тип первого фактического аргумента, в любом клиентском вызове has, позволяет определить, какая из подпрограмм имелась в виду.

Из этих соображений следует общая характеристика перегрузки, которая будет полезной, когда несколько позже это свойство будет сопоставляться с универсальностью:

Роль перегрузки

Перегрузка подпрограмм является средством, предназначенным для клиентов. Она позволяет писать один и тот же текст, используя разные реализации некоторого понятия.

Так что же дает перегрузка подпрограмм решению проблемы повторного использования? Не много. Это - синтаксическое средство, освобождающее разработчиков от необходимости придумывать различные имена для разных реализаций некоторой операции и, по существу, перекладывает эту ношу на компьютер. Но это не решает ни одной из ключевых задач повторного использования. В частности, перегрузка не дает ничего для выполнения требования Независимости Представлений. Когда записывается вызов

```
has (t, x)
```

то необходимо будет объявить t , а следовательно (даже если скрытие информации освобождает вас от заботы о деталях каждого варианта алгоритма поиска) нужно точно знать, каков вид таблицы t ! Единственной

достоинством перегрузки является то, что во всех случаях можно пользоваться одним и тем же именем. Без перегрузки в каждой реализации потребуется другое имя, например

```
has_binary_tree (t, x)
has_hash (t, x)
has_linked (t, x)
```

Но является ли таки достоинством возможность избежать использования различных имен? Наверное нет. Основным правилом создания ПО, объектно оно или нет, является **принцип честности (non-deception)**: различия в семантике должны отражаться в различиях текстов программ. Это позволяет существенно улучшить понятность ПО и минимизировать опасность возникновения ошибок. Если подпрограммы `has` являются различными, то использование для них одинакового имени может вводить в заблуждение - при чтении текста программы возникает предположение, что это одинаковые подпрограммы. Лучше предложить клиенту немного более многословный текст (как в случае введенных выше индивидуальных имен) и устраниТЬ какую-либо опасность путаницы.

Чем больше анализируешь перегрузку, тем более ограниченной она выглядит.

Критерий, используемый для устранения неоднозначности вызовов - сигнатуры списков аргументов - не обладает никакими конкретными достоинствами. Он работает в приведенных выше примерах, где все различные перегружаемые процедуры `square` и `has` имеют разные сигнатуры, но нетрудно представить себе множество случаев, когда у разных вариантов сигнатуры совпадают. Одним из простейших примеров перегрузки, по-видимому, является множество функций системы компьютерной графики, используемых для создания новых точек, например в виде:

```
p1 := new_point (u, v)
```

Точку можно задать: декартовыми координатами `x` и `y`; или полярными координатами `r` и `q` (расстоянием от начала координат и углом, отсчитываемым от горизонтальной оси). Но если перегрузить функцию `new_point`, то возникнет затруднение, связанное с тем, что оба варианта имеют одинаковую сигнатуру:

```
new_point (p, q: REAL): POINT
```

Этот пример, да и многие подобные ему, показывает, что сигнатура типов может не устранять неоднозначность перегружаемых вариантов. Но ничего лучшего не было предложено.

К сожалению, в относительно недавно появившемся языке Java используется описанная выше форма синтаксической перегрузки, в частности, для обеспечения альтернативных способов создания объектов.

Семантическая перегрузка (предварительное представление)

Описанную форму перегрузки подпрограмм можно назвать **синтаксической перегрузкой**. В ОО-подходе будет предложена намного более интересная методика, динамическое связывание, отвечающая целям Независимости Представлений. Динамическое связывание можно назвать **семантической перегрузкой**. При использовании этой методики и соответствующим образом подобранным синтаксисе можно записать некоторый эквивалент `has (t, x)` как запрос на выполнение.

Смысл такого запроса примерно таков:

Дорогой Компьютер (Hardware-Software Machine):

Разберитесь, пожалуйста, что такое `t`; я знаю, что это должна быть таблица, но не знаю, какую реализацию этой таблицы выбрал ее создатель - я, откровенно говоря, лучше, если я останусь в неведении об этом. Как-никак, я занимаюсь не организацией ведения таблиц, а банковскими инвестициями [или компиляцией, или автоматизированным проектированием и т.д.]. Начальник над таблицами здесь кто-то другой. Так что разберитесь в этом сами и, когда получите ответ, поищите подходящий алгоритм для '`has`', соответствующий этому конкретному виду таблицы. Затем используйте найденный алгоритм, чтобы установить, содержит ли '`x`' в '`t`', и сообщите мне результат. Я с нетерпением ожидаю вашего ответа.

С сожалением сообщаю вам что, кроме информации о том, что '`t`' это некоторого рода таблица, а '`x`' это ее возможный элемент, вы не получите от меня больше никакой помощи.

Примите мои дружеские пожелания,

Искренне Ваш разработчик приложений.

В отличие от синтаксической перегрузки, такая семантическая перегрузка является прямым ответом на требование Независимости Представлений. Все еще остается подозрение о нарушении принципа честности (non-deception), и ответом будет использование **утверждений** (assertions), задающих общую семантику

подпрограммы, имеющей много различных вариантов (например, общие свойства, характеризующие has при всевозможных реализациях таблицы).

Поскольку для надлежащей работы механизма семантической перегрузки требуется использование всего ОО-аппарата, в частности - наследования, то понятно, что синтаксическая перегрузка является лишь полумерой. В ОО-языке, наличие синтаксической перегрузки наряду с динамическим связыванием может лишь приводить к путанице, как это происходит в языках C++ и Java, которые позволяют классу использовать несколько процедур с одним и тем же именем, возлагая разрешение неоднозначности вызовов на компилятор и человека, читающего текст программы.

Универсальность (genericity)

Универсальность - это механизм определения параметризованных шаблонов модулей (*module patterns*), параметры которых представляют собой типы. Это средство является прямым ответом на требование Изменчивости Типов. Оно устраняет необходимость использования многих модулей, таких как:

```
INTEGER_TABLE_HANDLING  
ELECTRON_TABLE_HANDLING  
ACCOUNT_TABLE_HANDLING
```

Вместо этого разрешается написать единственный шаблон модуля в виде:

```
TABLE_HANDLING [G]
```

Имя G, представляющее произвольный тип, и называется **формальным родовым параметром (formal generic parameter)**. (Позже мы можем встретиться с необходимостью иметь два или более родовых параметров, но сейчас ограничимся одним.)

Такой параметризованный шаблон называется **универсальным модулем (generic module)**, хотя это еще не настоящий модуль, а лишь общая схема - шаблон многих возможных модулей. Для получения фактического модуля из шаблона, следует задать некоторый тип, называемый **фактическим родовым параметром**. Модули, получаемые из шаблона заменой формального параметра G на фактический, записываются, например, в виде:

```
TABLE_HANDLING [INTEGER]  
TABLE_HANDLING [ELECTRON]  
TABLE_HANDLING [ACCOUNT]
```

Типы INTEGER, ELECTRON и ACCOUNT использованы, соответственно, в качестве фактических родовых параметров. Такой процесс получения фактического модуля из универсального модуля (шаблона модуля) называется **родовым порождением (generic derivation)**, а сам модуль будет называться "универсально порожденным" (*generically derived*).

Два небольших замечания относительно терминологии. Во-первых, родовое порождение иногда называют родовой конкретизацией (*generic instantiation*), а порожденный модуль называют тогда родовым экземпляром (*generic instance*). Эта терминология может привести к недоразумениям в ОО-контексте, поскольку термин "экземпляр" применяется к объектам, созданные во время выполнения из соответствующих типов (или классов). Так что мы будем придерживаться термина "порождение".

Другим возможным источником недоразумений является слово "параметр". Подпрограмма может иметь **формальные аргументы**, представляющие значения, которые клиенты подпрограммы будут задавать при каждом обращении к ней. В литературе обычно используется термин "параметр" (формальный, фактический) как синоним аргумента (формального, фактического). Применение любого из этих терминов не является ошибкой, но мы, как правило, будем использовать термин "аргумент" для подпрограмм, а "параметр" для универсальных модулей.

Внутренне, описание унифицированного модуля TABLE_HANDLING будет напоминать приведенное выше описание INTEGER_TABLE_HANDLING, за исключением того, что для ссылки на тип элементов таблицы используется G вместо INTEGER. Например:

```
package TABLE_HANDLING [G] feature  
  type BINARY_TREE is  
    record  
      info: G  
      left, right: BINARY_TREE  
    end  
  has (t: BINARY_TREE; x: G): BOOLEAN  
    -- Содержится ли x в t?  
    do ... end
```

```

put (t: BINARY_TREE; x: G) is
    -- Включить x в t.
    do ... end
(и т.д.)
end -- пакета TABLE_HANDLING

```

В этом подходе некоторое замешательство вызывает то обстоятельство, что тип, объявленный BINARY_TREE, хотелось бы сделать универсальным и объявить его как BINARY_TREE [G]. Нет очевидного способа достижения этой возможности при "пакетном" подходе. Однако объектная технология объединит понятия модуля и типа, так что проблема будет решена автоматически. Мы убедимся в этом, когда узнаем, как интегрировать универсальность (genericity) в ОО-мир.

Интересно сопоставить определение универсальности с приведенным ранее определением перегрузки:

Роль универсальности

Универсальность является средством, предназначенным для поставщиков. Она позволяет писать один и тот же текст, используя одну и ту же реализацию некоторого понятия, применяемую к различным видам объектов.

Как же универсальность способствует реализации целей этой лекции? В отличие от синтаксической перегрузки, универсальность дает реальный вклад в решение наших проблем, поскольку, как было отмечено выше, она обеспечивает выполнение одного из основных требований, Изменчивости Типов. И при изложении объектной технологии в лекциях 7-18 этого курса значительное внимание будет уделено универсальности.

Основные методы модульности: оценка

Мы получили два основных результата. Одним из них является идея создания единого синтаксического "жилища", такого как пакетная конструкция (package construct), для множества подпрограмм, все из которых работают с однородными объектами. Вторым результатом является универсальность, приводящая к более гибкой форме модуля.

Все это, однако, охватывает лишь две проблемы повторного использования, Группирование Подпрограмм и Изменчивость Типов, и оказывает некоторое содействие в решении оставшихся трех проблем - Изменчивости Реализаций, Независимости Представлений и Факторизации Общего Поведения. Универсальность, в частности, недостаточна для решения проблемы Факторизации, поскольку определяет лишь два уровня. У нас появляется универсальный модуль, параметризованный и, следовательно, открытый для изменений, но непосредственно не применимый. На другом уровне у нас есть отдельные родовые порождения, пригодные для непосредственного применения, но закрытые для дальнейших изменений. Это не позволяет уловить тонкие различия, которые могут существовать между конкурирующими представлениями заданной общей идеи.

Что касается Независимости Представлений, то здесь мы почти не продвинулись. Ни один из рассмотренных методов - не считая беглого знакомства с семантической перегрузкой - не позволяет клиенту пользоваться различными реализациями некоторого общего понятия, не имея сведений о том, какая реализация будет выбрана в каждом случае.

Для решения этих проблем нам понадобится вся мощь ОО-концепций.

Ключевые концепции

- Для разработки ПО характерна повторяющаяся деятельность, включающая частое использование общих образцов (common patterns). Но имеются существенные вариации того, как используются и комбинируются эти образцы, так примитивные попытки работать с компонентами, имеющимися в наличии, терпят неудачу.
- При практическом внедрении повторного использования возникают экономические, психологические и организационные проблемы. Последние связаны, в частности, с необходимостью создания механизмов индексации, хранения и поиска большого числа повторно используемых компонентов. Более важными являются технические проблемы: общепринятые представления о модулях недостаточны для серьезной поддержки повторного использования.
- Основным затруднением при осуществлении повторного использования является необходимость сочетать повторное использование с расширяемостью. Дilemma - "повторно использовать или переделать" неприемлема. Хорошее решение должно обеспечить возможность сохранить одни свойства повторно используемого модуля и адаптировать другие.
- Простые подходы к решению проблем: повторное использование персонала, повторное использование проектов, повторное использование исходного кода, библиотеки подпрограмм привели к некоторому успеху, но не позволили полностью реализовать потенциальные достоинства повторного использования.
- Компонентом программы, пригодным для повторного использования, является абстрактный модуль, обеспечивающий инкапсуляцию функциональных возможностей с помощью хорошо определенного интерфейса.

- Пакеты обеспечивают лучшую реализацию метода инкапсуляции, чем подпрограммы, поскольку в них объединяются структура данных и связанные с ней операции.
- Два метода позволяют повысить гибкость пакетов: перегрузка подпрограмм и универсальность.
- Перегрузка подпрограмм является синтаксическим средством, которое не решает важных проблем повторного использования, но затрудняет читабельность текстов программ.
- Универсальность способствует повторному использованию, но решает лишь проблему изменчивости типов.
- Что же нам требуется: техника, помогающая поставщику учесть общность в группах взаимосвязанных реализаций структур данных; и техника, избавляющая клиентов от необходимости знать о том, какой вариант реализации выбран поставщиком.

Библиографические замечания

Первая публикация, обсуждающая проблемы повторного использования, упомянутая в начале этой лекции, принадлежит, по-видимому, Мак-Илрою (McIlroy's 1968 Mass-Produced Software Components). Его статья [McIlroy 1976] была представлена в 1968 г. на первой конференции по разработке ПО, созданной Комитетом НАТО по науке (NATO Science Affairs Committee). 1976 г. это дата издания трудов конференции, [Buxton 1976], публикация которых была задержана на несколько лет. Мак-Илрой пропагандировал развитие промышленного производства компонентов ПО.

Вот фрагмент его статьи:

"Производство ПО в наши дни оказывается по уровню индустриализации ниже наиболее отсталых отраслей строительной промышленности. Я думаю, что его надлежащее место значительно выше, и хотел бы выяснить перспективы реализации методов массового производства ПО ..."

Когда мы беремся за написание компилятора, то начинаем с вопроса: "Какой механизм работы с таблицами будем создавать?". А следует задавать вопрос: "Какой механизм будем использовать?" ...

Я выдвигаю тезис о том, что у индустрии программ слабая основа отчасти в связи с отсутствием подотрасли производства программных компонентов... Такое производство компонентов могло бы быть весьма успешным."

Одним из важных вопросов, рассмотренных в статье, был вопрос о необходимости иметь семейства модулей, обсуждавшийся выше как одно из требований к любому комплексному решению проблем повторного использования.

Наиболее важной характеристикой индустрии компонентов ПО является то, что она должна предлагать семейства [модулей] для выполнения заданной работы.

В тексте Мак-Илроя использовалось слово "подпрограмма" (routine), а не "модуль"; в свете обсуждения, проведенного в этой лекции, этот термин является - с ретроспективным учетом тридцати лет последующей эволюции методов разработки ПО - слишком ограничительным.

Специальный выпуск **Transactions on Software Engineering**, изданный Биггерстафом и Перлисом (Biggerstaff and Perlis) [Biggerstaff 1984], сыграл важную роль в привлечении внимания сообщества разработчиков ПО к вопросам повторного использования; смотрите в частности, в этом выпуске, статьи [Jones 1984], [Horowitz 1984], [Curry 1984], [Standish 1984] и [Goguen 1984]. Те же издатели включили все эти статьи (кроме первой из вышеупомянутых) в расширенный двухтомный сборник [Biggerstaff 1989]. Еще одним сборником статей по повторному использованию является [Tracz 1988]. Позже Трач (Tracz) собрал ряд своих материалов из **IEEE Computer** в полезную книгу [Tracz 1995], в которой особое значение придается организационным вопросам.

Один из подходов к повторному использованию, основанный на идеях искусственного интеллекта, воплощен в проекте Массачусетского технологического института по подготовке программистов (MIT Programmer's Apprentice project); смотрите статьи [Waters 1984] and [Rich 1989], воспроизведенные в первом и втором сборниках Биггерстафа-Перлиса, соответственно. Эта система использует не реальные повторно используемые модули, а шаблоны (называемые **cliches** and **plans**), представляющие общие стратегии разработки программы.

При обсуждении вопроса о пакетах упоминались три "языка с инкапсуляцией": Ada, Modula-2 и CLU. Язык Ada обсуждается в одной из последующих лекций, библиографический раздел которой содержит ссылки на языки Modula-2, CLU, а также Mesa and Alphard, причем два последних языка с инкапсуляцией принадлежат "модульному поколению" семидесятых и начала восьмидесятых годов прошлого века. Эквивалент пакета в языке Alphard был назван формой (form).

Важный проект STARS Министерства обороны США восьмидесятых годов прошлого века был акцентирован на проблеме повторного использования, особенно на организационных аспектах этой проблемы, причем в качестве языка для компонентов ПО использовался язык Ada. Ряд статей по этим вопросам можно найти в трудах конференции STARS DoD-Industry 1985 г. [NSIA 1985].

Двумя наиболее известными книгами по "образцам (шаблонам) проектов" являются [Gamma 1995] и [Pree 1994].

Работа [Weiser 1987] является призывом к распространению ПО в виде исходных текстов. Однако в этой статье недооценивается необходимость абстракции; как было показано в этой лекции, при необходимости можно сохранить возможность доступа к исходному тексту, но применить его высокоуровневую форму в качестве документации по умолчанию для пользователей модуля. Из других соображений Ричард Стallман (Richard Stallman), создатель Лиги Сторонников Свободы Программирования (League for Programming Freedom), утверждал, что представление в виде исходного текста всегда должно быть доступно; смотрите [Stallman 1992].

В работе [Cox 1992] описывается идея суперпоставки (superdistribution) Некоторая разновидность перегрузки имелась в языке Algol 68 [van Wijngaarden 1975]; в языках Ada (в котором это распространено на подпрограммы), C++ и Java, которые будут рассмотрены в последующих лекциях, этот механизм широко используется.

Универсальность или полиморфизм (genericity) появляется в языках Ada и CLU, и в ранней версии языка спецификаций Z [Abrial 1980]; в этой версии синтаксис Z близок к используемому для представления универсальности в этой книге. Язык LPG [Bert 1983], был явно предназначен для исследования универсальности. (Название этого языка является аббревиатурой из начальных букв "Language for Programming Generically".)

Работа, цитированная в начале этой лекции в качестве основной ссылки на табличный поиск, это [Knuth 1973]. Среди многих пособий по алгоритмам и структурам данных, которые освещают этот вопрос, стоит обратить внимание на [Aho 1974], [Aho 1983] или [M 1978].

Две книги автора данной книги содержат дальнейший анализ вопроса повторного использования. Книга **Reusable Software** [M 1994a], полностью посвященная этой теме, представляет принципы разработки и реализации для создания высококачественных библиотек, и полную спецификацию множества базисных библиотек. В книге **Object Success** [M 1995] обсуждаются организационные аспекты проблемы повторного использования, особенно те сферы деятельность, в которых должна прилагать усилия фирма, заинтересованная в повторном использовании, и области, в которых такие усилия будут, по-видимому, бесполезными (например, рекомендации повторного использования разработчикам приложений, или поощрение осуществления ими повторного использования). Смотрите также короткую статью на эту тему, [M 1996].

1) [Gamma 1995]; см также [Pree 1994].

2) Компиляторы ISE формируют как код C, так и байт-код.

3) В этой таблице индекс сокращенно обозначен как i, а курсор - как c.

4) Этот подход будет рассмотрен подробно в [лекции 15](#) курса "Основы объектно-ориентированного проектирования", с использованием понятия пакета из языка Ada. Напомним, что под "Ada" имеется в виду язык Ada 83. (В версии Ada 95 сохранены пакеты , но с некоторыми дополнениями.)

5) Эта нотация, совместимая с нотацией, используемой в остальных лекциях этого курса, является скорее Ada-подобной, чем точно соответствующей языку Ada. Тип REAL в языке Ada называется FLOAT; точки с запятой здесь были удалены.

Основы объектно-ориентированного программирования

5. Лекция: К объектной технологии

Расширяемость, возможность повторного использования и надежность - наши главные цели - требуют выполнения ряда условий, определенных в предыдущих лекциях. Для их достижения требуется систематический метод декомпозиции системы на модули. В этой лекции представлены основные элементы такого метода, основанного на простой, но далеко идущей идеи: строить каждый модуль на базе некоторого типа объектов. Здесь эта идея объясняется, логически обосновывается и из нее выводятся некоторые следствия. Предупреждение. Видя, что сегодня объектная технология широко известна и достаточно распространена, некоторые читатели могут подумать, что битва уже выиграна и нет необходимости в ее дальнейшем логическом обосновании. Это было бы ошибкой: если мы хотим избежать распространенных ошибок и ловушек, то нам нужно понимать основы метода. На самом деле, часто можно увидеть, что прилагательное "объектно-ориентированный" (подобно прилагательному "структурный" в предшествующую эпоху) используется просто как новая наклейка для самых традиционных методов разработки ПО. Только аккуратно построив здание объектной технологии можно научиться определять случаи неверного использования этого модного слова и избегать ошибок, рассматриваемых далее в этой лекции.

Ингредиенты вычисления

При поиске правильной архитектуры ПО критическим является вопрос о **модуляризации**: какие критерии нужно использовать при выделении модулей наших программ?

Чтобы верно ответить на него, нужно сравнить соперничающих кандидатов.

Базисный треугольник

Три силы вступают в игру, когда мы используем программу для выполнения каких-либо вычислений

Выполнить программную систему - значит использовать некоторые процессоры для применения некоторых действий к некоторым объектам.

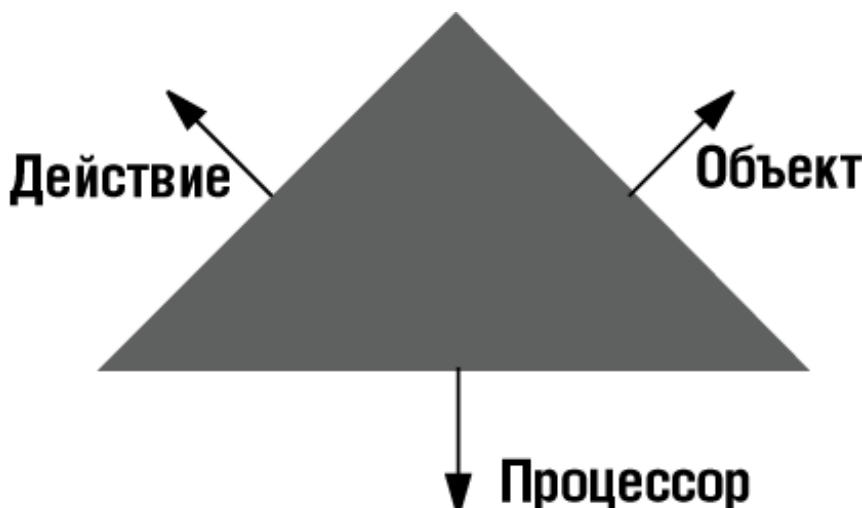


Рис. 5.1. Три силы вычисления

Процессоры - это вычислительные устройства (физические или виртуальные), выполняющие команды. Процессор может быть фактической единицей обработки (например, ЦПУ компьютера), процессом обычной операционной системы или одним ее "потоком" для многопоточной ОС.

Действия - это операции, производящие вычисления. Точная форма рассматриваемых нами действий будет зависеть от уровня детальности анализа. Например, на уровне оборудования действия являются операциями машинного языка, на аппаратно-программном уровне - операторами языка программирования, а на уровне программной системы можно рассматривать в качестве действия каждый большой шаг сложного алгоритма.

Объекты - это структуры данных, к которым применяются действия. Некоторые из этих объектов - структуры данных, построенные вычислением для своих собственных целей, - являются внутренними и существуют только во время вычисления, другие (содержащиеся в файлах, базах данных и других постоянных хранилищах) являются внешними и могут пережить вычисления, в которых используются.

Процессоры будут важны при обсуждении параллельных вычислений, в которых одновременно могут выполняться несколько подвычислений. В этой лекции мы ограничиваемся непараллельными или последовательными вычислениями, проводимыми одним (остающимся за рамками рассмотрения) процессором.

Таким образом, остаются действия и объекты. Дуализм между действиями и объектами - тем, что система делает, и тем, с чем она это делает - это популярная тема в разработке ПО.

- Замечание о терминологии. Для обозначения каждого из этих двух аспектов имеются соответствующие синонимы: слово **данные** будет использоваться как синоним слова **объекты**, а вместо слова **действие** мы, следуя обычной практике, будем говорить о **функциях** системы.
- Термин "функция" также не лишен недостатков, поскольку при обсуждении ПО он используется по крайней мере в двух смыслах: математическом и в смысле ПО как подпрограмма, возвращающая некоторый результат. Но, не боясь неоднозначности, мы будем использовать фразу "**функции системы**", требуемую здесь.
- Причина, по которой мы используем это слово, а не "действие" - чисто грамматическое удобство от использования соответствующего прилагательного, например во фразе "**функциональная декомпозиция**". Слово "действие" не имеет соответствующего производного прилагательного. Другим термином, чье значение в нашем обсуждении эквивалентно значению слова "действие", является слово "**операция**".

Всякое обсуждение, связанное с программированием, должно учитывать оба аспекта: объект и функцию, это относится и к проектированию программной системы. Но есть один вопрос, при ответе на который между ними нужно выбирать - это вопрос данной лекции: что является критерием выделения модулей системы? Здесь нужно решить, будут ли модули строиться как единицы функциональной декомпозиции или они будут создаваться вокруг главных типов объектов.

Ответ на этот вопрос демонстрирует различие между ОО-подходом и другими методами. При традиционных подходах каждый модуль строится вокруг некоторой единицы функциональной декомпозиции - некоторой части действия. В отличие от них, ОО-метод строит каждый модуль вокруг некоторого типа объектов.

Нетрудно догадаться, что в этой книге развивается именно этот подход. Но нам не следует принимать ОО-декомпозицию на веру лишь потому, что она подразумевается названием этой книги, или потому, что она является "вещью-в-себе", которую просто необходимо делать.

В последующих разделах мы проанализируем аргументы, обосновывающие использование типов объектов в качестве основы модуляризации, а начнем с исследования достоинств и ограничений традиционных не ОО-методов. Затем мы попытаемся получить ясное представление о том, что на самом деле означает слово "объект" для проектирования ПО, хотя полный ответ, требующий некоторых дополнительных теоретических рассуждений, появится только в следующей лекции.

Мы должны также отложить до следующей лекции урегулирование старинной и грозной битвы, ставшей темой нашего обсуждения - Войны Объектов и Функций. Пока же мы подготовим кампанию опорочивания функций в качестве базиса декомпозиции, и, соответственно, восхваления объектов для достижения этих целей. И все-таки нам не следует забывать сделанное выше наблюдение, в конечном счете, в нашем решении проблем должно найтись место, как объектам, так и функциям, хотя и не на равных основаниях. Для установления нового мирового порядка необходимо определить роли граждан первого и второго сорта.

Функциональная декомпозиция

Вначале мы рассмотрим достоинства и ограничения традиционного подхода, использующего функции в качестве основы архитектуры программных систем. Это не только приведет нас к пониманию того, почему требуется еще кое-что - объектная технология, но и поможет избежать некоторых методологических ловушек, таких как преждевременное упорядочение операций, которым, как известно, грешат даже опытные разработчики ОО-ПО.

Непрерывность

Ключевой проблемой при ответе на вопрос: "вокруг чего следует структурировать системы: вокруг функций или вокруг данных?" является проблема расширяемости, более точно - цель, названная **непрерывностью** в предшествующем обсуждении. Как вы помните, метод проектирования удовлетворяет этому критерию, если он приводит к устойчивой архитектуре, обеспечивающей объем изменений в проекте, соразмерный объему изменений в спецификации.

Обеспечение непрерывности - это главная забота при рассмотрении реального жизненного цикла программных систем, включающего не только производство приемлемой первоначальной версии, но и эволюцию системы на протяжении долгого времени. Большинство систем подвергаются многочисленным изменениям после их первоначальной поставки. Поэтому всякая модель разработки ПО, которая рассматривает только период, предшествующий этой поставке, и игнорирующую последующую эру изменений и пересмотров, весьма далека от реальной жизни, как те романы, которые заканчиваются женитьбой героя на героине в тот момент, когда, как каждый знает, только и начинаться самое интересное.

Чтобы оценить качество архитектуры (и породившего ее метода), нужно понять не только то, насколько просто было изначально получить эту архитектуру, не менее важно выяснить, насколько легко ее можно изменить.

Традиционным ответом на этот вопрос была функциональная декомпозиция "сверху вниз", кратко

определенная в одной из предыдущих лекций. Насколько хорошо разработка сверху вниз отвечает требованиям модульности?

Проектирование сверху вниз

Там был также весьма изобретательный архитектор, придумавший новый способ постройки домов. Постройка должна была начинаться с крыши и кончаться фундаментом. Он оправдывал мне этот способ ссылкой на приемы двух мудрых насекомых - пчелы и паука.

Джонатан Свифт, "Путешествия Гулливера"

При подходе сверху вниз система строится с помощью последовательных уточнений. Этот процесс начинается с самого общего утверждения об ее абстрактной функции, такого как

[C0]
"Оттранслировать СИ-программу в машинный код"

или

[P0]
"Обработать команду пользователя"

и продолжается путем последовательных шагов уточнения. На каждом шаге уровень абстракции получаемых элементов должен уменьшаться, каждая операция на нем разлагается на композицию одной или нескольких более простых операций. Например, следующий шаг в первом примере (транслятор с СИ) может привести к декомпозиции

[C1]
"Прочесть программу и породить последовательность лексем"
"Разобрать последовательность лексем и построить абстрактное синтаксическое дерево"
"Снабдить дерево семантической информацией"
"Сгенерировать по полученному дереву код"

или, используя другую структуру (и сделав упрощающее предположение, что СИ-программа - это последовательность определений функций):

```
[C'1]
from
    "Инициализировать структуры данных"
until
    "Определения всех функций обработаны"
loop
    "Прочесть определение следующей функции"
    "Сгенерировать частичный код"
end
"Заполнить перекрестные ссылки"
```

В любом случае разработчик должен на каждом шаге проверять оставшиеся не полностью уточненными элементы (такие как **"Читать программу..."** и **"Определения всех функций обработаны"**) и раскрывать их, используя тот же процесс уточнения до тех пор, пока все не окажется на достаточном низком уровне абстракции, допускающем непосредственную реализацию.

Процесс уточнения сверху вниз можно представить как построение дерева. Вершины представляют элементы декомпозиции, ветви показывают отношение "B есть уточнение A".

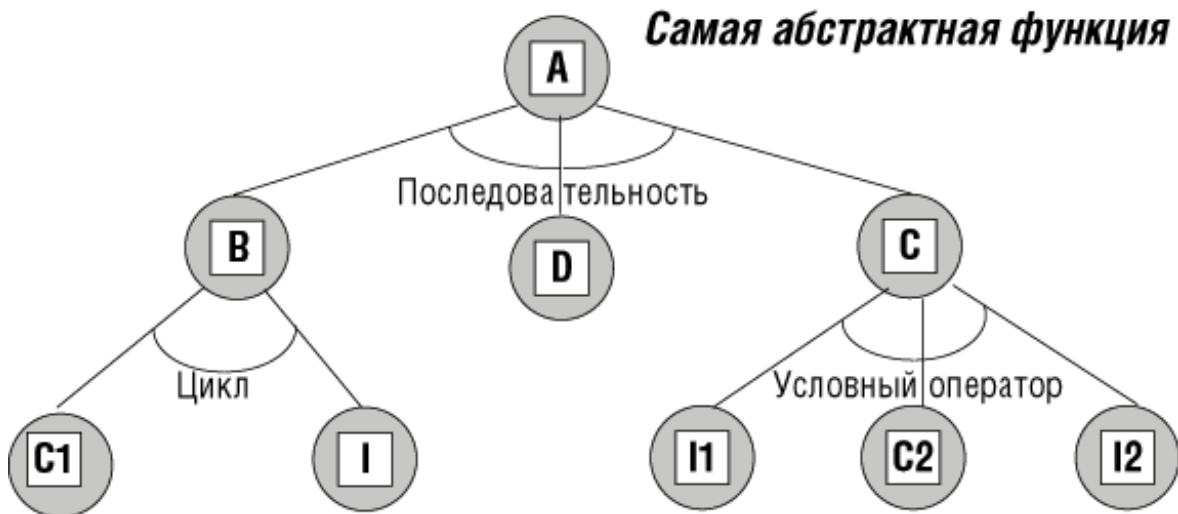


Рис. 5.2. Разработка сверху вниз: структура дерева

У метода проектирования сверху вниз имеется ряд достоинств. Он логичен, хорошо организует дисциплину мышления, поддается эффективному изучению, поощряет систематическое проектирование систем, помогает разработчику найти пути преодоления больших сложностей, возникающих обычно на начальной стадии разработки систем.

Несходящий подход может быть весьма полезен при разработке отдельных алгоритмов. Однако у него есть ряд ограничений, которые делают сомнительным использование этого подхода при проектировании целых систем:

- Сомнительной является сама идея охарактеризовать всю систему посредством только одной функции.
- Используя в качестве основы декомпозиции системы на модули свойства, которые склонны подвергаться наибольшим изменениям, этот метод не способен учесть эволюционную природу программных систем.

Не только одна главная функция

При эволюции системы то, что вначале воспринималось как ее главная функция, с течением времени может стать менее важным.

Рассмотрим типичную систему расчета зарплаты. При формулировке начальных требований заказчик мог представить лишь то, что следует из ее названия: систему для генерации чеков на зарплату по соответствующим данным. Его представление системы, явное или неявное, могло оказаться версией следующей схемы, возможно, чуть более амбициозное:



Рис. 5.3. Структура простой системы расчета зарплаты

Эта система получает некоторые входные данные (такие как часы работы служащего и некоторую информацию о нем) и производит некоторые выходные данные (чеки и т. п.). Это простая функциональная спецификация, в строгом смысле слова "функциональный". Она определяет программу как механизм для выполнения одной функции - платить зарплату служащим. Функциональный метод проектирования сверху вниз предназначен как раз для таких строго очерченных проблем, когда задание состоит в вычислении одной функции - "вершины" конструируемой системы.

Предположим, однако, что разработка нашей платежной системы благополучно завершена и программа выполняет всю необходимую работу. Скорее всего, на этом разработка не прекратится. Хорошие системы имеют противную привычку возбуждать в своих пользователях множество идей о других вещах, которые они могут делать. Как разработчику системы вам было сказано вначале, что все, что вы должны сделать - это сгенерировать чеки и пару вспомогательных выходных данных. Но затем просьбы о расширениях начинают попадать на ваш стол одна за другой: "Может ли программа собирать некоторую дополнительную статистику?" "Я говорил вам, что в следующем квартале мы собираемся начать платить некоторым служащим ежемесячно, а некоторым - дважды в месяц, не так ли?" "И, между прочим, мне нужен ежемесячный

суммарный отчет для администрации и еще один ежеквартальный для акционеров". "Бухгалтерам требуется отдельный отчет для начисления налогов". "Кстати, правильно ли вы храните информацию о зарплате? Очень хотелось бы предоставить персоналу интерактивный доступ к ней. Не понимаю, почему трудно добавить такую функцию?"

Этот феномен - желание добавить непредусмотренные заранее функции к успешным системам - встречается во всех прикладных областях. Программа для ядерной физики, которая вначале просто применяла некоторый алгоритм для выдачи таблицы чисел по пакетному входу, со временем непременно будет расширена. Она должна будет обрабатывать графический вход, выдавать графический выход и сохранять в базе данных полученные результаты. Компилятор, предназначенный только для трансляции корректных исходных текстов в объектные коды, будет через некоторое время существенно расширен, чтобы красиво распечатывать программы, а также служить верификатором синтаксиса, статическим анализатором и даже - программным окружением.

Процесс изменений происходит непрерывно. Новая система все еще является во многих отношениях "той же", что и старая: все еще платежной системой, программой для ядерной физики, компилятором. Но исходная "главная функция", которая вначале выглядела самой важной, часто становится просто одной из функций системы, а иногда и совсем исчезает, становясь ненужной.

Если при анализе и проектировании используется метод декомпозиции, основанный на функции, то структура системы будет вытекать из исходного понимания разработчиками главной функции системы. При этом добавление всякой новой функции, даже если оно кажется заказчику простым, может разрушить всю структуру системы. Поэтому очень важно найти в качестве критерия декомпозиции свойства менее изменчивые, чем главная функция системы.

Обнаружение вершины

Ниходящий метод проектирования предполагает, что каждая система характеризуется на самом абстрактом уровне своей главной функцией. Хотя многие учебные примеры алгоритмических проблем - "Ханойские башни", "Задача о 8 ферзях" и т. п. - действительно легко задать с помощью их "верхних" функций, более полезно описывать практические системы в терминах предоставляемых ими услуг.

Рассмотрим какую-либо операционную систему. Наиболее разумно представлять ее как систему, предоставляющую такие услуги, как распределение времени процессора, управление памятью, обращение с устройствами ввода-вывода, декодирование и исполнение команд пользователя. Модули хорошо структурированной ОС стремятся сгруппироваться вокруг этих групп функций. Но это не та структура, которую можно получить при ниходящей функциональной декомпозиции. Этот метод заставляет проектировщика отвечать на искусственный вопрос: "что является "верхней" функцией?", а затем использовать последовательные уточнения полученного ответа в качестве основы для структуры системы. При определенных усилиях можно прийти к следующему ответу на исходный вопрос

"Обработать все запросы пользователя",

который далее можно уточнять примерно так:

```
from
    начальная загрузка системы
until
    остановка или аварийный отказ
loop
    "Прочесть запрос пользователя и поместить во входную очередь"
    "Взять запрос r из входной очереди"
    "Обработать r"
    "Поместить результат в выходную очередь"
    "Взять результат q из выходной очереди"
    "Выдать результат q получателю"
end
```

Уточнения могут продолжаться. Однако маловероятно, что после такого начала кому-либо удастся спроектировать разумно структурированную операционную систему.

Вернемся к примеру с компилятором. Оставил в нем самую суть или представив точку зрения старых учебников, можно сказать, что компилятор - это реализация функции типа вход-выход, трансформирующей текст исходной программы на некотором языке программирования в машинный код некоторой платформы. Но для современных компиляторов этот взгляд недостаточен. Среди многих услуг, предоставляемых компилятором, обнаружение ошибок, форматирование программы, возможность управления конфигурацией системы, вход в систему, генерация отчетов.

По-видимому, очевидная отправная точка проектирования сверху вниз - взгляд, согласно которому для

каждой новой разработки требуется запросить некоторую специальную функцию - является весьма сомнительной:

У реальной системы нет "вершины"!

Функции и эволюция

Главная функция часто не только не является наилучшим критерием для начального определения системы, но она может также в процессе эволюции системы почти сразу оказаться среди изменяемых свойств.

Рассмотрим в качестве примера программу, имеющую две версии: одну "пакетную", которая выполняет во время сессии одно большое непрерывное вычисление, и другую - интерактивную, которая в каждой сессии реализует последовательность транзакций с разбиением взаимодействия пользователя с системой на более мелкие шаги. Большие научные программы очень часто имеют две версии: одну, которая "путь работает всю ночь, выполняя большую порцию вычислений", и другую, которая "позволяет мне сначала проверить некоторые вещи, посмотреть на результаты, а затем вычислить еще что-нибудь".

Уточнение сверху вниз пакетной версии могло начаться следующим образом.

```
[B0] - Абстракция верхнего уровня  
"Решить полный экземпляр проблемы"  
[B1] - Первое уточнение  
"Прочесть входные данные"  
"Вычислить результаты"  
"Вывести результаты"
```

и т. д. Проектирование интерактивной версии сверху вниз может происходить в следующем стиле.

```
[I1]  
"Обработать одну транзакцию"  
[I2]  
if "Пользователь предоставил новую информацию" then  
    "Ввести информацию"  
    "Запомнить ее"  
elseif "Запрошена ранее данная информация" then  
    "Извлечь запрошенную информацию"  
    "Вывести ее"  
elseif "Запрошен результат" then  
    if "Необходимая информация доступна" then  
        "Получить запрошенный результат"  
        "Вывести его"  
    else  
        "Запросить подтверждение запроса"  
    if Да then  
        "Получить требуемую информацию"  
        "Вычислить запрошенный результат"  
        "Вывести результат"  
    end  
end  
end
```

(и т. д.)

Начавшаяся таким образом разработка приведет к совершенно неверному результату. Подход сверху вниз не способен учесть то обстоятельство, что результирующие программы должны быть ничем иным как двумя версиями одной и той же программной системы, независимо от того, как они проектируются - одновременно или одна выводится из другой.

Этот пример выясняет два самых неприятных последствия подхода сверху вниз: во-первых, он сосредотачивается на внешнем интерфейсе (здесь это проявилось в раннем выборе между пакетной и интерактивной версиями), во-вторых, он преждевременно устанавливает временные отношения (т.е. порядок выполнения действий).

Интерфейсы и проектирование ПО

Архитектура системы должна основываться на содержании, а не на форме. Но проектирование сверху вниз стремится использовать в качестве основы для структуры самый поверхностный аспект системы - ее внешний интерфейс.

Такой упор на внешний интерфейс неизбежен для метода, ключевой вопрос которого: "Что система будет делать для конечного пользователя?" Ответ на него обязательно будет акцентироваться на самых внешних аспектах.

Интерфейс пользователя, как правило, оказывается одним из наиболее изменчивых компонентов, поскольку трудно получить правильный интерфейс с первой попытки. Довольно часто удается построить интерфейс отдельно от других компонент системы, используя один из множества доступных сегодня инструментов реализации элегантных и дружественных интерфейсов, основанных на ОО-методах. В таких случаях интерфейс пользователя почти не оказывает влияния на проектирование всей системы.

Преждевременное упорядочение

Предыдущие примеры иллюстрируют также и другой недостаток функциональной декомпозиции сверху вниз: преждевременную фиксацию временных ограничений. Каждое уточнение развертывает часть абстрактной структуры в более подробную архитектуру управления, задающую порядок выполнения различных функций (различных частей соответствующего действия). Такие уточнения и ограничения порядка становятся существенными свойствами архитектуры системы, но они также подвержены изменениям.

Напомним две альтернативные структуры для первого уточнения компилятора.

```
[C1]
"Прочесть программу и породить последовательность лексем"
"Разобрать последовательность лексем и построить абстрактное синтаксическое дерево"
"Снабдить дерево семантической информацией"
"Сгенерировать по полученному дереву код"
[C'1]
from
    "Инициализировать структуры данных"
until
    "Определения всех функций обработаны"
loop
    "Прочесть определение следующей функции"
    "Сгенерировать частичный код"
end
"Заполнить перекрестные ссылки"
```

Как и в предыдущем примере, мы начинаем с двух совершенно разных архитектур. Каждая из них задается некоторой структурой управления (последовательностью команд в первом случае и циклом, за которым идет команда - во втором), накладывающей строгие ограничения на порядок элементов в этой структуре. Но было бы неразумно зафиксировать такие отношения порядка на самых ранних стадиях проектирования. Такие вопросы как число проходов компилятора, установление последовательности различных этапов (лексического анализа, синтаксического разбора, семантической обработки, оптимизации) имеют много различных решений, к которым должны прийти разработчики, учитывая соотношения между памятью и временем и другие критерии, которыми они, возможно, руководствовались в начале проекта. Они могут успешно выполнять работу по проектированию и реализации отдельных компонентов задолго до фиксации временного порядка между ними, и захотят подольше сохранять свободу в выборе этого порядка. Функциональное проектирование сверху вниз не обеспечивает такой гибкости: требуется определять порядок выполнения операций до появления глубокого понимания того, что эти операции будут делать.

ОО-проектирование избегает преждевременного упорядочения. Разработчик изучает различные операции, применимые к определенным данным, и задает результат каждой из них, но при этом откладывает, насколько это возможно, определение порядка выполнения операций. Это можно назвать подходом **списка необходимых покупок**: здесь его роль играет список необходимых операций, т.е. всех операций, которые вам могут понадобиться. При этом ограничения на их порядок в процессе создания ПО не налагаются так долго, пока это возможно. В результате получаются намного более гибкие архитектуры.

Упорядочивание и ОО-разработка

Риск преждевременного упорядочивания заслуживает более глубокого рассмотрения, поскольку даже ОО-проектировщики не имеют к нему иммунитета. Подход списка покупок - это один из наименее понятных компонентов метода. Довольно часто можно встретить ОО-проекты, попавшие в старую ловушку, что немедленно отражается на их качестве. В частности, это может быть результатом неправильного использования идеи разбора случаев - case технологии, с которой мы встретимся при изучении ОО-методологии.

Проблема в том, что порядок операций, кажущийся очевидным свойством системы и ничему не обязывающий на ранних этапах проектирования, приводит к ужасным последствиям, если после всех уточнений его придется изменить. Альтернативный метод - подход списка покупок - кажется с первого взгляда менее

естественным, но значительно более гибок, поскольку использует логические, а не временные ограничения. Он основан на концепции утверждений, разрабатываемой позже в этой книге (см. [лекцию 11](#)). Продемонстрируем теперь основные идеи на не программистском примере.

Рассмотрим проблему покупки дома, сведя ее к трем операциям: нахождение подходящего дома, получение ссуды, подписание контракта. Используя метод, основанный на упорядочивании, опишем наш проект, как простую последовательность шагов:

```
[H]
найти_дом
получить_ссуду
подписать_контракт
```

В подходе списка покупок при ОО-разработке мы бы на данном этапе отказались бы придавать так много значения порядку операций. Но, конечно, ограничения между операциями существуют, - нельзя подписать контракт, если у вас нет подходящего дома и нет денег на его покупку. Мы можем выразить эти ограничения в логической форме:

```
[H1]
найти_дом
ensure
    дом_найден
получить_ссуду
ensure
    ссуда_получена
подписать_контракт
require
    дом_найден and ссуда_получена
```

Нотация будет введена только в [лекции 11](#), но и здесь все должно быть достаточно ясно. Предложение **require** задает предусловие, логическое свойство, требуемое операцией перед ее выполнением; **ensure** - задает постусловие, свойство, выполняемое после завершения операции. Тем самым нам удалось описать результат двух операций, и то, что последняя операция требует для своего выполнения достижения результата этих операций.

Почему логическая форма H1, устанавливающая ограничения, лучше, чем временная форма H? Ответ ясен: H1 выражает минимум требований, избегая чрезмерной спецификации, характерной для H. В самом деле, почему не получить ранее ссуду, а потом уже думать о покупке дома, располагая определенными деньгами, это тактика может быть вполне оправдана для покупателя, у которого главная проблема - финансовая. Насколько возможно, следует поддерживать оба возможных порядка действий, соблюдая логические ограничения.

Подход, основанный не на порядке операций, а на логических ограничениях, более уравновешенный. Каждая операция просто устанавливает, что ей необходимо и что она гарантирует, -все это в терминах абстрактных свойств.

Эти замечания важны, в частности, и для объектных проектировщиков, кто все еще может находиться в плену функциональных идей, и будет пытаться применить раннюю идентификацию системы, используя сценарии (case технологию) как основу анализа. Это несовместимо с ОО-принципами и часто приводит в чистом виде к функциональной декомпозиции сверху вниз, даже если члены команды уверены, что они используют ОО-метод.

Возможность повторного использования

После этого краткого вторжения в зону объектной территории вернемся к анализу метода сверху вниз и рассмотрим его на сей раз по отношению к одной из наших основных целей - возможности повторного использования ПО.

При разработке сверху вниз элементы программы создаются в ответ на отдельные уточненные спецификации, встретившиеся в древообразном проектировании системы. В текущей точке разработки, соответствующей уточнению некоторой вершины дерева, разработчиком будет осознана необходимость введения некоторой функции, например анализа входной командной строки. Затем будет задана ее спецификация, а реализовывать функцию, возможно, будет другой исполнитель.

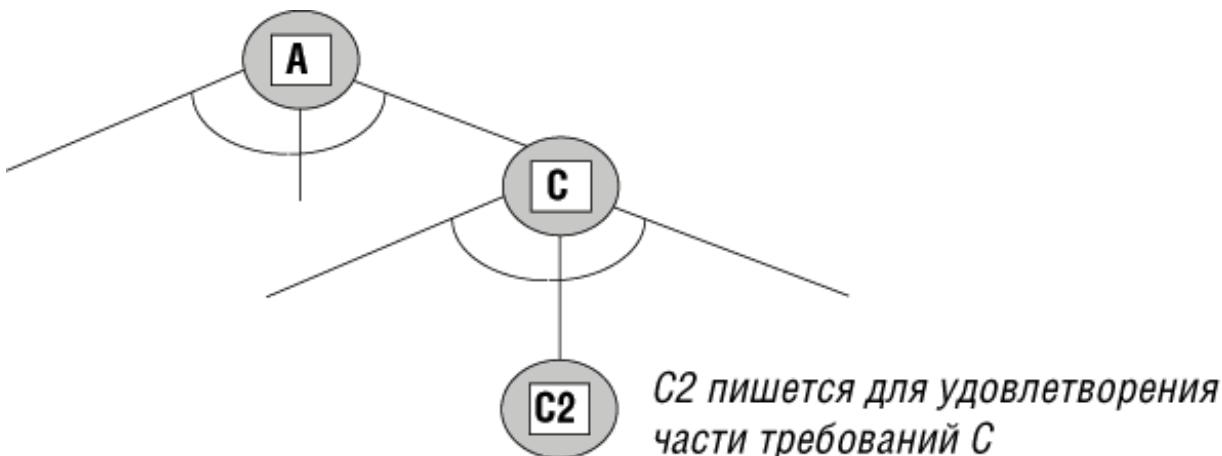


Рис. 5.4. Контекст модуля при разработке сверху вниз

Рисунок, показывающий часть дерева уточнений сверху вниз, иллюстрирует это свойство: С2 пишется, чтобы удовлетворить некоторой части требований С. Характеристики С2 полностью определяются его непосредственным контекстом, т.е. нуждами С. Например, С может быть модулем, отвечающим за анализ некоторых входных данных, а С2 может быть модулем, отвечающим за анализ одной строки (части всего длинного входа).

Такой подход обеспечивает хорошее соответствие проекта его начальной спецификации, но не способствует повторному его использованию. Модули разрабатываются в ответ на отдельные возникающие подзадачи и, как правило, являются не более общими, чем к этому их вынуждает непосредственный контекст. В нашем примере, если С пред назначен для входных текстов специального вида, то маловероятно, что С2, анализирующий одну строку таких текстов, будет применим к какому-либо другому виду входа.

Проектирование, имеющее в виду возможность повторного использования, подразумевает построение наиболее общих, по возможности, компонент, из которых затем составляются системы. Этот процесс идет снизу вверх и противоположен идее проектирования сверху вниз, требующей начинать с определения "задачи" и выводить ее решение путем последовательных уточнений.

Это обсуждение показывает, что проектирование сверху вниз является побочным продуктом того, что можно назвать культом проекта в разработке ПО, считающего, что единицей рассмотрения должен служить индивидуальный проект, никак не связанный с предыдущими или последующими проектами. Реальность не столь проста: n-ый проект компании обычно является вариацией (n-1)-го проекта и предшественником (n+1)-го. Сфокусировавшись лишь на одном проекте, разработка сверху вниз пренебрегает этой особенностью практического создания ПО.

Производство и описание

Одна из причин первоначальной привлекательности идей проектирования сверху вниз заключается в том, что этот стиль может быть удобен для объяснения каждого шага разработки. Но то, что хорошо для документации существующей разработки, не обязательно является наилучшим способом для ее проведения. Эта точка зрения была ярко представлена Майклом Джексоном в "Разработке систем" ([Jackson 1983], стр. 370-371):

Сверху вниз - это разумный способ описания уже полностью понятых вещей. Но это неподходящий способ для проектирования, разработки или открытия чего-либо нового. Здесь имеется близкая параллель с математикой. В учебниках по математике ее отдельные дисциплины описываются в логическом порядке: каждая сформулированная и доказанная теорема используется при доказательстве последующих теорем. Но на самом деле эти теоремы не создавались или открывались указанными способами или в указанном порядке... Если у разработчика системы или программы в голове уже имеется ясное представление об окончательном результате, то он может применить метод сверху вниз, чтобы описать на бумаге то, что имеется у него в голове. Именно поэтому люди могут считать, что они проектируют и разрабатывают сверху вниз и делают это весьма успешно: они смешивают способ описания с методом разработки. Когда начинается этап сверху вниз, задача уже решена и остались уточнить лишь некоторые детали.

Проектирование сверху вниз: общая оценка

Проведенное обсуждение функционального проектирования сверху вниз показывает, что этот метод плохо приспособлен для разработки важных систем. Он остается полезной парадигмой для небольших программ и отдельных алгоритмов, он также полезен для **описания** хорошо понятных алгоритмов, особенно в учебниках по программированию. Но он не масштабируем и не годится для больших практических программных систем.

Декомпозиция, основанная на объектах

Использование объектов (или, более точно, как будет видно далее, - типов объектов) как ключа для разбиения системы на модули основано на содержательных целях, определенных в [лекции 1](#), в частности, на расширяемости, возможности повторного использования и совместимости.

Доводы в пользу применения объектов будут довольно краткими, так как этот вопрос был уже ранее рассмотрен: многие из аргументов против основанного на функциях проектирования сверху вниз естественно превращаются в свидетельства в пользу основанного на объектах проектирования снизу вверх.

Эти свидетельства, тем не менее, не должны привести к полному отказу от функций. Как было отмечено в начале лекции, никакой подход к созданию ПО не может быть полным, если он не учитывает обе стороны - функции и объекты. Поэтому нам нужно и в ОО-методе сохранить надлежащее место для функций, даже если они в результирующей архитектуре системы будут подчинены объектам. Понятие абстрактного типа данных предоставит нам определение объектов, в котором для функций зарезервировано подходящее место.

Расширяемость

Так как функции системы имеют тенденцию изменяться в течение ее жизни, то возникает вопрос о поиске более стабильной характеристики ее существенных свойств, которая могла бы руководить нашим выбором модулей и соответствовала бы цели непрерывности.

Типы объектов, с которыми работает система, являются более перспективными кандидатами. Что бы ни случилось с использованной в примере выше системой расчета зарплаты, она все равно будет манипулировать объектами, представляющими служащих, штатные расписания с зарплатами, инструкции компании, табель учета рабочего времени, чеки. Что бы ни случилось с компилятором или другим средством обработки языка, он все еще будет манипулировать исходными текстами, последовательностями лексем, деревьями разбора, абстрактными синтаксическими деревьями, целевым кодом. Что бы ни случилось с системой, реализующей метод конечных элементов, она по-прежнему будет манипулировать матрицами, конечными элементами и сетками.

Этот аргумент справедлив только при рассмотрении объектов достаточно высокого уровня. Если рассматривать объекты на уровне их физических представлений, то расширяемость будет не намного лучше, чем у функций, на самом деле, даже хуже, так как функциональная декомпозиция сверху вниз, по крайней мере, поддерживает абстракцию. Поэтому вопрос поиска подходящего уровня абстракции для описания объектов является ключевым и ему будет посвящена остальная часть этой лекции.

Возможность повторного использования

Обсуждение возможности повторного использования показало, что процедура (элемент функциональной декомпозиции) обычно недостаточна как единица для повторного использования.

Мы рассмотрели ранее ([лекция 4](#)) типичный пример: поиск в таблице. Начав с,казалось бы, естественного кандидата на повторное использование - процедуры поиска, мы поняли, что ее нельзя повторно использовать отдельно от других операций, применяемых к таблице, таких как вставка и удаление.

Отсюда появилась идея, что в этой задаче модулем, достаточно хорошо допускающим повторное использование, должна быть совокупность таких операций. Но если попытаться понять, какая концепция все эти операции объединяет, то мы обнаружим тип объектов, к которым они применяются - таблицы.

Такие примеры подсказывают, что типы объектов, полностью снабженные связанными с ними операциями, и будут стабильными единицами для повторного использования.

Совместимость

Другой показатель качества ПО, совместимость, был определен как легкость, с которой программные продукты (в данном обсуждении - модули) можно комбинировать между собой.

Если структуры данных не проектировались с этой целью, то имеющие к ним доступ действия комбинировать очень сложно. Почему бы тогда не попробовать комбинировать целиком структуры данных?

Объектно-ориентированное конструирование ПО

У нас уже накоплено достаточно оснований, чтобы попытаться определить ОО-конструирование ПО. Это будет лишь первый набросок, более конкретное определение последует в следующей лекции.

ОО-конструирование ПО (определение 1)

ОО-конструирование ПО - это метод разработки ПО, который строит архитектуру всякой программной системы на модулях, выведенных из типов объектов, с которыми система работает (а

не на одной или нескольких функциях, которые она должна предоставлять).

Содержательная характеристика этого подхода может служить лозунгом ОО-проектировщика:

Объектный девиз

Не спрашивай вначале, что система делает.

Спроси, кто в системе это делает!

Чтобы получить работающую реализацию, вам придется рано или поздно узнать, что она делает. Отсюда слово **вначале**. ОО-мудрость говорит, что узнать что делается лучше позже, чем раньше. При этом подходе выбор главной функции является одним из последних шагов в процессе конструирования системы.

Вместо поиска самой верхней функции системы будут анализироваться типы входящих в нее объектов. Проектирование системы будет продвигаться вперед путем последовательного улучшения понимания классов этих объектов. Это процесс построения снизу вверх устойчивых и расширяемых решений для отдельных частей задачи и сборки из них все более и более мощных блоков до тех пор, пока не будет получен окончательный блок, доставляющий решение первоначальной задачи. При этом можно надеяться, что оно не является единственным возможным: если правильно применять метод, то те же компоненты, собранные по-другому и, возможно, объединенные с другими, окажутся достаточно общими, чтобы получить в качестве побочного продукта также и решения каких-то новых задач.

Для многих разработчиков программ такое изменение точки зрения является настолько же шокирующим, насколько шокирующей в далекие времена могла быть для многих мысль о том, что земля вращается вокруг солнца, а не наоборот. Это также противоречит многому в сложившейся практике разработки программного обеспечения, которая стремится представить построение системы как выполнение системной функции, представленной в подробном, привязанном к требованиям документе. Тем не менее, эта простая идея - вначале рассматривать данные, забыв о непосредственной цели системы, - может послужить ключом к повторному использованию и расширяемости.

Вопросы

Приведенное выше определение послужит отправной точкой для обсуждения ОО-метода. Оно не только дает ответ на некоторые относящиеся к ОО-проектированию вопросы, но и побуждает задать много новых вопросов таких, как:

- Как находить релевантные типы объектов?
- Как описывать типы объектов?
- Как описывать взаимоотношения типов объектов и их близость?
- Как использовать типы объектов для структурирования ПО?

Оставшаяся часть этой книги будет посвящена ответам на эти вопросы. Давайте рассмотрим предварительно некоторые ответы.

Выявление типов объектов

Вопрос "как мы будем находить объекты?" вначале может выглядеть пугающим. В [лекции 4](#) курса "Основы объектно-ориентированного проектирования" мы рассмотрим его более подробно, но здесь полезно рассеять некоторые из возникающих страхов. Этот вопрос может не отнять много времени у опытных ОО-разработчиков, в частности, благодаря доступности трех источников для ответа:

- Многие объекты лежат на поверхности. Они непосредственно моделируют объекты физической реальности, к которой применяется ПО. ОО-технология является мощным средством моделирования, использующим типы программных объектов (классы) для моделирования типов физических объектов и отношения между типами объектов (клиент, наследование) для моделирования отношений между типами физических объектов таких, как агрегирование и специализация. Разработчику ПО не требуется изучать трактаты по ОО-анализу, чтобы в системе мониторинга для телекоммуникаций использовать класс CALL (ВЫЗОВ) и класс LINE (ЛИНИЯ), а в системе обработки документов - класс DOCUMENT (ДОКУМЕНТ), класс PARAGRAPH (АБЗАЦ) и класс FONT (ШРИФТ).
- Одним из источников типов объектов является повторное использование: классы, ранее определенные другими. Этот метод, не всегда бросающийся в глаза в литературе по ОО-анализу, на практике часто оказывается наиболее полезным. Мы должны противостоять соблазну что-либо изобретать, если задача уже была удовлетворительно решена другими.
- Наконец, опыт и копирование тоже играют важную роль. Ознакомившись с успешными ОО-разработками и образцами проектов, проектировщик может вдохновиться этими более ранними усилиями.

Мы лучше поймем эти и другие методы выделения объектов, когда приобретем более глубокое понимание

сугубо понятие "объект" в программировании - не надо смешивать его с обыденным значением этого слова.

Описания типов и объектов

Предположим, что известно, как получить надлежащие типы объектов, служащие основой для структуры модулей нашей системы. Тогда немедленно возникнет вопрос, как описать эти типы и их объекты.

При ответе на него следует руководствоваться двумя требованиями:

- Нужно добиваться независимости описаний от представлений, чтобы не потерять главное преимущество проектирования сверху вниз: абстрактность.
- Нужно найти для функций подходящее место в архитектуре программ, чья декомпозиция основана на анализе типов объектов, так как оба двойственных аспекта - объекты и функции - должны получить в ней соответствующее место.

В следующей лекции развивается методика описания объектов, позволяющая достичь обе эти цели.

Описание отношений и структурирование ПО

Другой вопрос связан с тем, какие отношения допустимы между типами объектов. В рафинированной объектной технологии имеются только два отношения: "быть клиентом" и наследование. Они соответствуют различным видам возможных зависимостей между двумя типами объектов А и В :

В является клиентом А , если каждый объект типа В содержит информацию об одном или нескольких объектах типа А .

В является наследником А, если В представляет специализированную версию А .

В некоторых подходах к анализу, в частности, в таком подходе к информационному моделированию как моделирование сущность-связь, для описания возможных связей между элементами системы используются более богатые множества отношений. Для людей, привыкших к таким подходам, вначале кажется, что работать только с двумя видами отношений весьма неудобно. Но это опасение может и не подтвердиться:

- Отношение "быть клиентом" достаточно широкое и покрывает многие виды зависимостей. Примерами таких зависимостей является отношение, часто называемое агрегацией (присутствие в каждом объекте типа В подобъекта типа А), а также зависимость по ссылке и родовая зависимость.
- Отношение наследования покрывает многочисленные формы специализации.
- Многие зависимости можно выразить в общем виде другими способами. Например, для описания зависимости "от 1-го до n" (каждый объект типа В связан с не менее чем одним и не более чем с n объектами типа А) укажем, что В является клиентом А, и присоединим **инвариант класса**, точно определяющий природу отношения "быть клиентом". Так как инварианты классов выражаются с помощью логического языка, они покрывают намного больше различных отношений, чем может предложить подход сущность-связь или другие аналогичные подходы.

Ключевые концепции

- Вычисление включает три вида ингредиентов: процессоры (или потоки управления), действия (или функции) и данные (или объекты).
- Архитектуру системы можно получить исходя из функций или из типов объектов.
- Описание, основанное на типах объектов, с течением времени обеспечивает лучшую устойчивость и лучшие возможности для повторного использования, чем описание, основанное на анализе функций системы.
- Как правило, неестественно считать, что задача системы состоит в реализации только одной функции. У реальной системы обычно имеется не одна "вершина" и ее лучше описывать как систему, предоставляющую множество услуг.
- На ранних стадиях проектирования и разработки системы не нужно уделять много внимания ограничениям на порядок действий. Многие временные соотношения могут быть описаны более абстрактно в виде логических ограничений.
- Функциональное проектирование сверху вниз не подходит для программных систем с долгим жизненным циклом, включающим их изменения и повторное использование.
- При ОО-конструировании ПО структура системы основывается на типах объектов, с которыми она работает.
- При ОО-разработке первоначальный вопрос не в том, что система делает, а в том, с какими типами объектов она это делает. Решение о том, какая функция является самой верхней функцией системы (и имеется ли таковая), откладывается на последние этапы процесса проектирования.
- Чтобы проектируемое ПО было расширяемым и допускало повторное использование, ОО-конструирование должно выводить архитектуру из достаточно абстрактных описаний объектов.

- Между типами объектов могут существовать два вида отношений: "быть клиентом" и наследование.

Библиографические замечания

Вопрос об ОО-декомпозиции рассматривается с использованием различных аргументов в [Cox 1990] (первоначально в 1986), [Goldberg 1981], [Goldberg 1985], [Page-Jones 1995] и [M 1978], [M 1979], [M 1983], [M 1987], [M 1988].

Метод проектирования сверху вниз отстаивается во многих книгах и статьях. Вирт [Wirth 1971] развил понятие пошагового уточнения.

Что касается других методов, то, по-видимому, наиболее близким является метод структурного проектирования Джексона JSB [Jackson 1983] и его расширение высокого уровня в [Jackson 1975]. См. также предложенный Варнье метод проектирования от данных [Orr 1977]. Для знакомства с методами, которые ОО-технология призвана заменить, смотрите книги по методу структурного проектирования Константина и Йордана [Yourdon 1979], по структурному анализу [DeMarco 1978], [Page-Jones 1980],[McMenamin 1984], [Yourdon 1989]; по методу Merise [Tardieu 1984], [Tabourier 1986].

Метод моделирования сущность-связь был введен Ченом [Chen 1976].

Основы объектно-ориентированного программирования

6. Лекция: Абстрактные типы данных (АТД)

Чтобы объекты играли лидирующую роль в архитектуре ПО, нужно их адекватно описывать. В этой лекции показывается, как это делать. Если вам не терпится окунуться в глубины объектной технологии и подробно изучить множественное наследование, динамическое связывание и другие игрушки, то, на первый взгляд, эта лекция может показаться лишней задержкой на этом пути, поскольку она в основном посвящена изучению некоторых математических понятий (хотя вся используемая в ней математика элементарна). Но так же, как самый талантливый музыкант извлечет пользу из изучения основ музыкальной теории, знания об абстрактных типах данных помогут вам понять и получить удовольствие от практики ОО-анализа, проектирования и программирования, хотя привлекательность этих понятий, возможно, уже проявилась и без помощи теории. Поскольку абстрактные типы данных являются теоретическим базисом для всего метода, следствия идея, вводимых в этой лекции, будут ощущаться во всей оставшейся части книги. Более того, как будет видно в конце лекции, эти идеи выходят за рамки собственно ПО и приводят к принципам интеллектуальных исследований, которые, возможно, применимы и в других дисциплинах.

Это открыло мне глаза, я начал понимать, что значит использовать инструмент, называемый алгеброй. Черт возьми, никто никогда не говорил мне ничего подобного раньше. Мсье Дюлюи [Учитель математики] произносил напыщенные фразы об этом предмете, но ни разу не сказал этих простых слов: это разделение труда, которое, как и всякое другое разделение труда производит чудеса и позволяет уму сконцентрировать все свои силы только на одной стороне объектов, только на одном из их качеств.

Насколько другим это предстало бы перед нами, если бы мсье Дюлюи сказал нам: "Этот сыр мягкий или твердый, он белый, он синий, он старый, он молодой, он твой, он мой, он легкий или он тяжелый. Из всех его многочисленных качеств давайте рассматривать только вес. Каким ни был этот вес, давайте назовем его A. А теперь, не думая больше о весе, давайте применять к A все, что мы знаем о количестве."

Такая простая вещь, но до сих пор никто не говорил нам о ней в этой отдаленной провинции...

Стендалль, "Жизнь Анри Брюлара"

Что касается абстракции, то она состоит в отделении ощущимых свойств тел либо от других их свойств, либо от самих тел, которые ими обладают. Когда это отделение делается неудачно или неверно применяется, возникают ошибки, что возможно как в философских вопросах, так и в физических и математических вопросах. Прямой путь к ошибке в философии - недостаточно упростить изучаемые объекты, и верный путь к получению ошибочных результатов в физике и математике - это считать объекты менее сложными, чем они есть на самом деле.

Дени Дидро, "Письмо слепого на благо тех, кто может видеть"

Критерии

Чтобы получить надлежащие описания объектов, наш метод должен удовлетворять трем условиям:

- Описания должны быть точными и недвусмысличными.
- Они должны быть полными - или, по крайней мере, иметь в каждом конкретном случае нужную нам полноту (некоторые детали можно намеренно опускать).
- Они не должны быть **излишне специфицированы**.

Последний пункт делает ответ нетривиальным. В конце концов, легко сделать описание точным, недвусмысличным и полным, если мы готовы "выдать все секреты", указав все детали объектного представления. Но такое описание, как правило, будет включать чересчур много информации для авторов программ, которым требуется доступ к таким объектам.

Это замечания похожи на комментарии, которые привели к понятию скрытия информации. Там дело было в том, что, предоставляя в качестве первичного источника информации исходный код модуля (элементы, связанные с реализацией) авторам клиентских программ, зависящих от этого модуля, мы можем окунуть их в поток деталей, который помешает им сосредоточиться на своей собственной работе и затруднит перспективу развития проекта. Здесь нас ожидает та же опасность, что и в случае, когда мы позволяем модулям использовать некоторую структуру данных на основании информации, которая относится к представлению этой структуры, а не к ее существенным свойствам.

Различные реализации

Чтобы лучше понять всю важность описаний абстрактных типов данных, исследуем глубже потенциальные последствия использования физической реализации в качестве основы описания объектов.

Удобным и хорошо изученным примером является описание объектов типа стек. Объект стек служит для того, чтобы накапливать и доставать другие объекты в режиме "последним пришел - первым ушел" ("LIFO"),

элемент, вставленный в стек последним, будет извлечен из него первым. Стек повсеместно используется в информатике и во многих программных системах, в частности, компиляторы и интерпретаторы усыпаны разными видами стеков.

Надо сказать, что стеки присутствуют в дидактических представлениях абстрактных типов данных в таком большом количестве, что Э. Дейкстра как-то остроумно заметил, что "абстрактные типы данных являются прекрасной теорией, целью которой является описание стеков". Совершенно справедливо. Но в следующих лекциях курса понятие абстрактных типов данных так часто применяется в гораздо более сложных случаях, что я не чувствую стыда, начиная рассмотрение с этого ключевого примера. Он является простейшим из известных мне примеров, содержащих в себе почти все важные идеи абстрактных типов данных.

Представления стеков

Существует несколько физических представлений стеков:

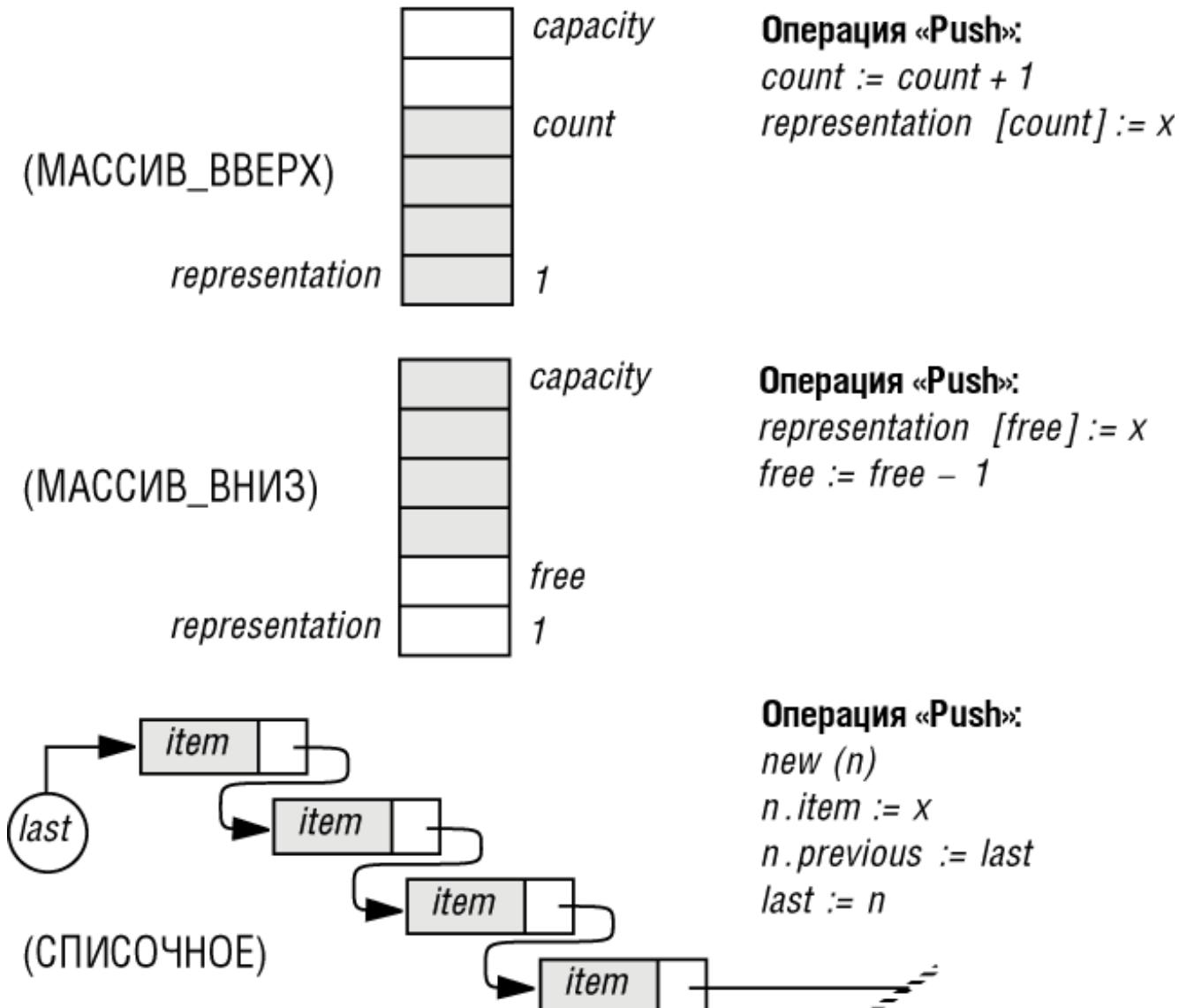


Рис. 6.1. Три возможных представления стеков

Этот рисунок иллюстрирует три наиболее популярных представления стеков. Для удобства ссылок дадим каждому из них свое имя:

- МАССИВ_ВВЕРХ: представляет стек посредством массива *representation* и целого числа *count*, с диапазоном значений от 0 (для пустого стека) до *capacity* - размера массива *representation*, элементы стека хранятся в массиве и индексируются от 1 до *count*.
- МАССИВ_ВНИЗ: похож на МАССИВ_ВВЕРХ, но элементы помещаются в конец стека, а не в начало. Здесь число, называемое *free*, является индексом верхней свободной позиции в стеке или 0, если все позиции в массиве заняты и изменяется в диапазоне от *capacity* для пустого стека до 0 для заполненного. Элементы стека хранятся в массиве и индексируются от *capacity* до *free*+1.
- СПИСОЧНОЕ: при списочном представлении каждый элемент стека хранится в ячейке с двумя полями: *item*, содержащем сам элемент, и *previous*, содержащем указатель на ячейку с предыдущим элементом. Для этого представления нужен также указатель *last* на ячейку, содержащую вершину

стека.

Рядом с каждым представлением на рисунке приведен фрагмент программы (в духе Паскаля), с соответствующей реализацией основной стековой операции: втолкнуть элемент x на вершину стека (push).

Для представлений с помощью массивов МАССИВ_ВВЕРХ и МАССИВ_ВНИЗ команды увеличивают или уменьшают указатель на вершину (count или free) и присваивают x соответствующему элементу массива. Так как эти представления поддерживают стеки с не более чем capacity элементами, то корректные реализации должны содержать защищающие от переполнения тесты соответствующего вида:

```
if count < capacity then ...  
if free = 0 then ...,
```

(на рисунке они для простоты опущены).

Для представления СПИСОЧНОЕ вталкивание элемента требует четырех действий:

- создания новой ячейки p (здесь оно выполняется с помощью процедуры Паскаля new, которая выделяет память для нового объекта);
- присваивания x полю item новой ячейки;
- присоединения новой ячейки к вершине стека путем присвоения ее полю previous текущего значения указателя last;
- изменения last так, чтобы он ссылался на только что созданную ячейку.

Хотя эти представления встречаются чаще всего, существует и много других представлений стеков. Например, если вам нужны два стека с однотипными элементами и память для их представления ограничена, то можно использовать один массив с двумя метками вершин count как в представлении МАССИВ_ВВЕРХ и free как в МАССИВ_ВНИЗ. При этом один стек будет расти вверх, а другой - вниз. Условием полного заполнения этого представления является равенство $count = free$.

Преимущество такого представления состоит в уменьшении риска переполнить память: при двух массивах размера n , представляющих стеки способом МАССИВ_ВВЕРХ или МАССИВ_ВНИЗ, память исчерпается, как только любой из стеков достигнет n элементов. А в случае одного массива размера $2n$, содержащего два стека лицом к лицу, работа продолжается до тех пор, пока их общая длина не превысит $2n$, что менее вероятно, если стеки растут независимо друг от друга. (Для любых переменных p и q , $\max(p+q) = \max(p) + \max(q)$).

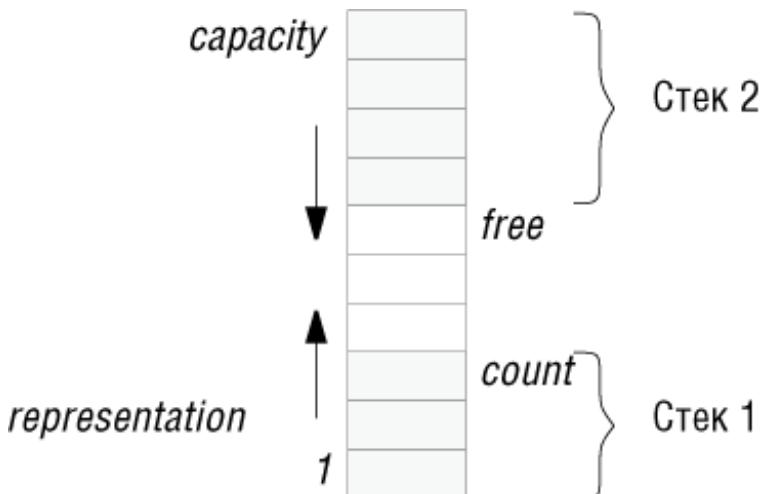


Рис. 6.2. Представление двух стеков лицом к лицу

Каждое из этих и другие возможные представления полезны в разных ситуациях. Выбор одного из них в качестве эталона для определения стека был бы типичным примером излишней спецификации. Почему мы должны, например, предпочесть МАССИВ_ВВЕРХ представлению СПИСОЧНОЕ? Большинство видимых свойств представления МАССИВ_ВВЕРХ - массив, число count, верхняя граница - несущественны для понимания представляющей ими структуры.

Опасность излишней спецификации

Почему так плохо использовать конкретное представление в качестве спецификации?

Можно напомнить результаты изучения Линцем (Lientz) и Свенсоном (Swanson) стоимости сопровождения. Было установлено, что более 17% стоимости ПО приходится на изменения в форматах данных. Ясно, что метод, который ставит анализ и проектирование в зависимость от физического представления структур

данных, не обеспечит разработку достаточно гибкого ПО.

Поэтому при использовании объектов или типов объектов в качестве основы для архитектуры системы требуется найти лучший способ описания, чем конкретное представление.

Какова длина второго имени?

Как бы стеки не заставили нас забыть, что кроме излюбленных специалистами по информатике примеров имеются структуры данных, тесно связанные с объектами реальной жизни. Вот забавный пример, взятый из почты форума Риски (Risks) (группа новостей Usenet comp.risks), который иллюстрирует опасности взгляда на данные, чересчур сильно зависящего от их конкретных свойств. Некто Даррелл Д. Е. Лонг, которого родители наградили двумя инициалами второго имени, получил кредитную карточку, в которой был указан лишь первый из них "Д". После обращения к менеджеру фирмы TRW ему была прислана другая карточка, в которой был лишь второй инициал "Е". Он пишет:

Я позвонил в бюро выдачи кредитов, и оказалось, что, по-видимому, программист, который проектировал базу данных TRW, решил, что каждому хорошему американцу пожаловано второе имя лишь с одним инициалом. Как вежливо объяснила мне по телефону дама: "Они выделили в системе достаточно мегабайт (sic) только для одного инициала второго имени и это чрезвычайно трудно изменить".

Кроме типичного примера технократического оправдания ("мегабайты"), урок в этом случае заключается в том, что нужно избегать ориентации программы на физические свойства данных.

Автор приведенного выше письма, в основном, беспокоился из-за ненужной почты, что неприятно, но не смертельно, архивы форума Риски (Risks) полны случаями вызванной компьютерами неразберихи с гораздо более серьезными последствиями. Уже отмечавшаяся выше "проблема миллениума" является другим примером опасности, возникающей при организации доступа к данным на основе их физического представления, ее последствия обошлись в сотни миллионов долларов.

К абстрактному взгляду на объекты

Как нам сохранить полноту, точность и однозначность, не заплатив за это излишней спецификацией?

Использование операций

Представления стека при всех их различиях объединяет то, что они описывают структуру "хранения" (т.е. структуру, используемую для хранения других объектов), к которой применяются определенные операции, обладающие определенными свойствами. Сосредоточившись не на выборе конкретного представления структуры, а на этих операциях и свойствах, можно получить достаточно абстрактное, но, тем не менее, полезное, описание понятия стек.

Обычно для стеков рассматриваются следующие операции:

- Команда вталкивания некоторого элемента на вершину стека. Назовем эту операцию `put`.
- Команда удаления верхнего элемента стека. Назовем ее `remove`.
- Запрос элемента, находящегося на вершине стека (если стек не пуст). Назовем его `item`.
- Запрос на проверку пустоты стека. (Он позволит клиентам заранее проверить возможность операций `remove` и `item`.)

Кроме того, нам понадобится операция-конструктор для создания пустого стека. Назовем ее `make`.

Две вещи заслуживают более подробных объяснений далее в этой лекции. Во-первых, могут показаться необычными имена операций, Давайте пока считать, что `put` означает `push`, `remove` означает `pop`, а `item` означает `top`. Во-вторых, операции разбиты на три категории: конструкторы, создающие объекты, запросы, возвращающие информацию об объектах, и команды, которые могут изменять объекты. Эта классификация также требует дополнительных объяснений.

При традиционном взгляде на структуры данных мы рассматривали бы понятие стека, заданное с помощью некоторого объявления данных, соответствующего одному из вышеуказанных представлений, например для представления МАССИВ_ВВЕРХ. В стиле Паскаля это выглядит как

```
count: INTEGER  
representation: array [1 .. capacity] of STACK_ELEMENT_TYPE
```

где константа `capacity` - это максимальное число элементов в стеке. Тогда `put`, `remove`, `item`, `empty` и `make` будут подпрограммами, которые работают на структурах, определенных этим объявлением объектов.

Чтобы сделать главный шаг в направлении абстракции данных, нужно стать на противоположную точку зрения: забыть на некоторое время о конкретном представлении и взять в качестве определения структуры данных операции сами по себе. Иначе говоря, стек - это любая структура, к которой клиенты могут применять перечисленные выше операции.

Политика невмешательства в обществе модулей

Только что намеченный метод описания структур данных выглядит довольно эгоистичным подходом в мире структур данных. Нас не столько интересует то, что они собой представляют внутренне, как то, что они могут друг другу предложить. В этом мы похожи на экономиста - пылкого приверженца теорий приоритета производства и невидимой руки, воспитанного в духе школы "пусть-все-решит-свободный-рынок". Мир объектов (а, следовательно, и архитектуры ПО) будет миром взаимодействующих объектов, общающихся на основе точно определенных протоколов.

Аналогия с экономикой будет сопровождать наше изложение и дальше, агенты - программные модули - называются **поставщиками и клиентами**, протоколы будут называться контрактами, и большая часть ОО-разработки, на самом деле, может рассматриваться как "Проектирование по Контракту" - это заголовок одной из следующих лекций.

Не следует чересчур увлекаться этой аналогией (как и всякой другой): эта работа не учебник по экономике и она не содержит даже намеков на точку зрения автора в этой области. Сейчас нам достаточно отметить поразительные аналогии подхода абстрактных типов данных с некоторыми теориями о взаимодействии агентов-людей.

Согласованность имен

Давайте убедимся в том что, приведенная выше спецификация и ее детали являются достаточно удобными. Для того, кто раньше сталкивался со стеками, выбранные при обсуждении стека имена операций могут показаться странными или даже шокирующими. Каждому уважающему себя специалисту по информатике операции со стеком известны под другими именами:

Таблица 6.1. Имена операций над стеком

Стандартное имя операции над стеком	Имя, используемое здесь
Push (втолкнуть)	Put (поместить)
Pop (вытолкнуть)	Remove (удалить)
Top (вершина)	Item (элемент)
New (новый)	Make (создать)

Зачем использовать терминологию, отличающуюся от общепринятой? Причина - в желании достичь более высокого уровня понимания структур данных - особенно "контейнеров", которые используются для хранения объектов.

Стеки это просто один из видов контейнеров, точнее они относятся к категории контейнеров, которые можно назвать **распределителями**. Распределитель предоставляет своим клиентам механизм для хранения (`put`), извлечения (`item`) и удаления (`remove`) объектов, но не дает им возможности управлять тем, какой объект будет извлекаться или удаляться. Например, метод доступа LIFO, используемый в стеках, позволяет извлекать или удалять только тот элемент, который был сохранен последним. Другой вид распределителей - очередь, которая использует метод доступа "первым в, первым из" (`FIFO`): элементы добавляются в один конец очереди, а извлекаются и удаляются - с другого конца. Пример контейнера, не являющегося распределителем, - это массив, в нем вы сами выбираете целочисленные номера позиций, в которые вставляются или из которых извлекаются объекты.

Поскольку схожесть разных видов контейнеров (распределителей, массивов и т.п.) более важна, чем различия между тем, как они хранят, извлекают или удаляют объекты, эта книга твердо придерживается стандартизованной терминологии, которая сглаживает различия между вариантами структур данных и, наоборот, подчеркивает их общность. Поэтому базисная операция извлечения элемента будет всегда называться `item`, базисная операция удаления элемента будет всегда называться `remove`, и т.д.

Вопросы именования могут вначале показаться поверхностными - "косметическими", как иногда говорят программисты. Но не забывайте, что одна из наших конечных целей - создать основу для мощных, профессиональных библиотек программных компонент, допускающих повторное использование. Такие библиотеки будут содержать десятки тысяч доступных операций. Без их систематической и ясной номенклатуры и разработчики, и пользователи этих библиотек быстро потонут в потоке специальных и несравнимых имен, что создаст сильное (и не имеющее оправдания) препятствие к масштабному повторному использованию.

Таким образом, вопросы именования - это не косметика. Хорошее, допускающее повторное использование ПО - это ПО, которое предоставляет пользователям соответствующий набор функций и предоставляет их под правильными именами.

Имена, использованные здесь для операций стеков, являются частью соглашений об именовании, которых мы придерживаемся во всей книге.

Можно ли обойтись без абстракций?

В разработке программного обеспечения, как и в других научных и технических дисциплинах, плодотворная идея после того, как ее раскрыли, может показаться очевидной даже, если потребовалось много времени, чтобы она возникла. Сначала зачастую появляются плохие и запутанные (что часто одно и то же) идеи, и требуется время, чтобы более простые и элегантные заняли их место.

Это замечание справедливо и для абстрактных типов данных. Хотя хорошие разработчики ПО всегда с пользой применяли абстракцию (вследствие хорошего образования или просто интуитивно), многие из существующих ныне систем были разработаны без учета этой цели.

Однажды я невольно провел один небольшой эксперимент, который хорошо иллюстрирует такое состояние дел. Как-то, когда в курсе, который я читал, пришло время готовить проекты, я решил предоставить студентам нечто вроде анонимного рынка, куда бы они могли помещать шутливые объявления о продаже программных модулей, не раскрывая их источников. (Идея, хорошая или не очень, состояла в том, чтобы процесс выбора модулей происходил только на основе точных спецификаций их возможностей.) Почтовые средства знаменитой операционной системы, предпочитаемой американскими университетами, казалось бы, предоставляли соответствующий базовый механизм, но, естественно, эта почтовая система показывала имя отправителя при доставке сообщения получателям. У меня был доступ к исходному коду - огромной программе на Си, и я решил, наверное, по глупости, взять этот код, убрать в нем все ссылки на имя отправителя в доставляемых сообщениях и перекомпилировать.

С помощью своего ассистента я взялся за работу, казавшуюся достаточно очевидной, хотя ей и не учат в курсах по разработке ПО, - систематическую разборку программы. Будучи уверенными в себе, мы быстро нашли первое место, в котором программа обращалась к имени отправителя, и удалили соответствующий код. После чего наивно решили, что работа сделана, и перекомпилировали код. Но когда мы послали тестовое сообщение, то обнаружили, что имя отправителя все еще сохранилось! После чего начался долгий и сюрреалистический процесс: снова и снова, веря, что мы, наконец, обнаружили последнее обращение к имени отправителя, мы удаляли его, перекомпилировали программу и посыпали тестовое сообщение, чтобы в очередной раз исправно обнаружить имя отправителя на обычном месте. Как знаменитая стоглавая гидра, почтовая программа отращивала новую голову всякий раз, когда мы считали, что отрубили ей последнюю.

Наконец, повторив в нашу эру древний подвиг Геракла, мы полностью уничтожили этого зверя, удалив более двадцати участков кода, каждый из которых, так или иначе, задавал информацию об отправителе.

В предыдущих разделах нам удалось сделать первые шаги по дороге к АТД. Их достаточно для понимания того, что программа, написанная в соответствии с самыми элементарными представлениями об абстракции данных, должна была бы рассматривать MAIL_MESSAGE (ПОЧТОВОЕ_СООБЩЕНИЕ) как точно определенное абстрактное понятие. Одной из операций сообщения мог быть запрос, называемый, например, sender (отправитель), возвращающий информацию об отправителе сообщения. Любой элемент почтовой программы, которому была бы нужна эта информация, получал бы ее только через этот запрос sender. Если бы почтовая программа была разработана в соответствии с этим, кажущимся очевидным, принципом, то для моего небольшого упражнения достаточно было бы изменить только код запроса sender. Более того, весьма вероятно, что в этом случае программа предоставляла бы также и операцию set_sender (установить_отправителя), которая позволила бы выполнить требуемую работу еще проще.

Отметим, что рассматриваемая почтовая программа использовалась весьма успешно. Но она является типичным представителем нынешнего стандарта в индустрии ПО. До тех пор, пока мы не выйдем далеко за пределы этого стандарта фраза "проектирование программного обеспечения" останется примером принятия желаемого за действительное.

Формализация спецификаций

Представленный выше беглый набросок абстракции данных слишком неформален, чтобы его можно было постоянно использовать. Вернемся к нашему главному примеру. Стек, как мы это поняли, должен определяться в терминах применимых к нему операций, но тогда нам нужно определить эти операции!

Приведенные содержательные описания явно недостаточны - *push* вталкивает элемент на "вершину" стека, *pop* выталкивает элемент, находящийся на вершине. Нам нужно точно знать, как клиенты могут использовать эти операции и что они для этого должны делать.

Спецификация АТД предоставит эту информацию. Она состоит из четырех разделов, разъясняемых в следующих разделах:

- ТИПЫ
- ФУНКЦИИ
- АКСИОМЫ
- ПРЕДУСЛОВИЯ

Для спецификации АТД в этих разделах будут использоваться простая математическая нотация.

Эту нотацию - математический формализм - не надо путать с программной нотацией в остальной части книги, даже если для согласования она использует тот же стиль синтаксиса. У нее нет специального имени, и она не является нотацией языка программирования. Она могла бы послужить отправной точкой для формального языка спецификаций, но мы удовлетворимся использованием не требующих объяснения соглашений для однозначной спецификации АТД.

Спецификация типов

В разделе ТИПЫ указываются специфицируемые типы. В общем случае, может оказаться удобным определять одновременно несколько АТД, хотя в нашем примере имеется лишь один тип STACK (СТЕК). Между прочим, что такое тип? Ответ на этот вопрос объединит все положения, развиваемые далее в этой лекции: тип - это совокупность объектов, характеризуемая функциями, аксиомами и предусловиями. Не будет большой ошибкой рассматривать пока тип как множество объектов в математическом смысле слова "множество" - тип STACK как множество всех возможных стеков, тип INTEGER как множество всех целых чисел и т.д.

Однако при этом не должно быть никакой путаницы: АТД, такой как STACK, - это не объект (один конкретный стек), а совокупность объектов (множество всех стеков). Напомним, в чем состоит наша главная цель: найти подходящую основу для модулей наших программных систем. Очевидно, не имеет смысла делать основой для модуля один конкретный объект - один стек, один самолет, один счет в банке. ОО-проектирование даст нам возможность строить модули, отражающие свойства всех стеков, всех самолетов, всех банковских счетов, или, по крайней мере, значительной их части.

Объект, принадлежащий множеству объектов, описываемых спецификацией АТД, называется **экземпляром** этого АТД. Например, конкретный стек, обладающий свойствами абстрактного типа данных STACK, будет экземпляром АТД STACK. Понятие экземпляра проходит через все ОО-проектирование и программирование, и будет играть важную роль в объяснении поведения программ во время исполнения.

В разделе ТИПЫ просто перечисляются типы, вводимые в данной спецификации. Здесь:

Типы

- STACK[G]

Таким образом, наша спецификация относится к одному абстрактному типу данных - STACK, задающему стеки объектов произвольного типа G.

Универсализация (Genericity)

В описании STACK[G] именем G обозначен произвольный, не определяемый тип. G называется формальным родовым параметром для типов элементов АТД STACK, а сам STACK называется родовым или универсальным АТД. Механизм, допускающий такие параметризованные спецификации, известен как универсализация, мы уже сталкивались с аналогичным понятием в обзоре конструкций пакетов.

Можно писать спецификации АТД без параметризации, но ценой будут неоправданные повторения. Кроме того, возможность повторного использования желательна не только для программ, но и для спецификаций! Благодаря механизму универсализации, можно выполнять параметризацию типов в явном виде, выбрав для параметра некоторое произвольное имя (здесь - G), представляющее переменную для типа элементов стека.

В результате такой АТД как STACK - это не просто тип, а скорее образец типа. Для получения непосредственно используемого типа стека нужно определить тип элементов стека, например ACCOUNT, и передать его в качестве **фактического родового параметра**, соответствующего формальному параметру G. Поэтому, хотя сам по себе STACK это образец типа, обозначение STACK[ACCOUNT] задает полностью определенный тип. Про такой тип, полученный с помощью передачи фактических параметров типов в родовой тип, говорят, что он **порожден из общего по образцу**.

Эти понятия можно применять рекурсивно: каждый тип должен, по крайней мере, в принципе, иметь

спецификацию АТД, поэтому можно и тип ACCOUNT считать абстрактным типом данных. Кроме того, тип, подставляемый в качестве фактического параметра типа в STACK (для получения типа, порожденного по образцу) может и сам быть порожденным по образцу. Например, можно вполне корректно использовать обозначение STACK[STACK [ACCOUNT]] для определения соответствующего абстрактного типа данных: элементами этого типа являются стеки, элементами которых, в свою очередь, являются банковские счета.

Как показывает этот пример, предыдущее определение "экземпляра" нуждается в некоторой модификации. Строго говоря, конкретный стек является экземпляром не типа STACK (который, как мы заметили, является скорее образцом типа, а не типом), а некоторого типа, порожденного типом STACK, например, образцом типа STACK[ACCOUNT]. Тем не менее, нам удобно и далее говорить об экземплярах типа S и других образцов типов, понимая при этом, что речь идет об экземплярах порожденных ими типов.

Аналогично, не очень правильно говорить о типе STACK как об АТД: правильный термин в этом случае - "образец АТД". Но для простоты в данном обсуждении мы будем и далее, если это не приведет к путанице, опускать слово "образец".

Это отличие перенесется и на ОО-проектирование и программирование, но там нам не потребуется два разных термина:

- Основным понятием будет класс, который может иметь родовые параметры.
- Описание реальных данных требует типов. Класс без параметров является также и типом, но класс с параметрами - только образец типа. Чтобы получить конкретный тип из такого класса, нужно передать ему фактические параметры типов, точно так, как мы это делали при получении АТД STACK[ACCOUNT], исходя из образца АТД STACK[G].

Перечисление функций

Вслед за разделом ТИПЫ идет раздел ФУНКЦИИ, в котором перечисляются операции, применяемые к экземплярам данного АТД. Как уже говорилось, эти операции будут главными компонентами определения типа, с их помощью описывается, что могут предложить его экземпляры, а не то, чем они являются.

Ниже приведен раздел ФУНКЦИИ для абстрактного типа данных STACK. Если вы разработчик ПО, то этот стиль описания вам знаком: строки этого раздела напоминают декларации типизированных языков программирования таких, как Pascal или Ada. Стока для операции new похожа на объявление переменной, остальные - на заголовки процедур.

Функции

- put: STACK [G] × G → STACK [G]
- remove: STACK [G] ↛ STACK [G]
- item: STACK [G] ↛ G
- empty: STACK [G] → BOOLEAN
- new: STACK [G]

В каждой строке вводится определенная математическая функция, моделирующая соответствующую операцию над стеком. Например, функция put представляет операцию, которая вталкивает элемент на вершину стека.

Почему функции? Большая часть программистов не посчитает такую операцию как put функцией. Когда во время работы программной системы операция put применяется к стеку, она, как правило, изменяет этот стек, добавляя к нему элемент. Вследствие этого в приведенной выше классификации операций put была "командой" - операцией, которая может модифицировать объекты. (Две другие категории операций - это конструкторы и запросы).

Однако спецификация АТД - это математическая модель и в ее основании должны быть корректные математические методы. В математике понятие команды или, более общо, изменение чего-либо как таковое отсутствует: вычисление квадратного корня из числа 2 не изменяет само это число. Математические выражения просто определяют одни математические объекты в терминах некоторых других математических объектов. В отличие от вычисления программы на компьютере, они никогда не изменяют никакие математические объекты. Но поскольку мы нуждаемся в некотором математическом объекте для моделирования операций компьютера, то понятие функции представляется наиболее близким приближением. Функция - это механизм для получения некоторого результата, принадлежащего некоторому результирующему множеству по любому допустимому входу, принадлежащему некоторому исходному множеству. Например, если R обозначает множество вещественных чисел, то определение функции

`square_plus_one: R → R`
`square_plus_one(x) = x2 + 1` (для каждого x из R)

вводит функцию `square_plus_one`, для которой R является и исходным и результирующим множеством и которая выдает для любого входа в качестве результата квадрат этого входа, увеличенный на 1.

Спецификации абстрактных типов данных используют именно это понятие. Например, операция `put` определяется как

`put: STACK [G] × G → STACK [G]`

и означает, что `put` будет брать два аргумента: STACK экземпляров типа G и экземпляр типа G и возвращать в качестве результата новый STACK [G]. (Более формально, множеством определения функции `put` является множество STACK [G] × G, являющееся **декартовым произведением** множеств STACK [G] и G, т.е. множеством пар $\langle s, x \rangle$, в которых первый элемент s принадлежит STACK [G], а второй элемент x принадлежит G.) Вот рисунок, иллюстрирующий это:

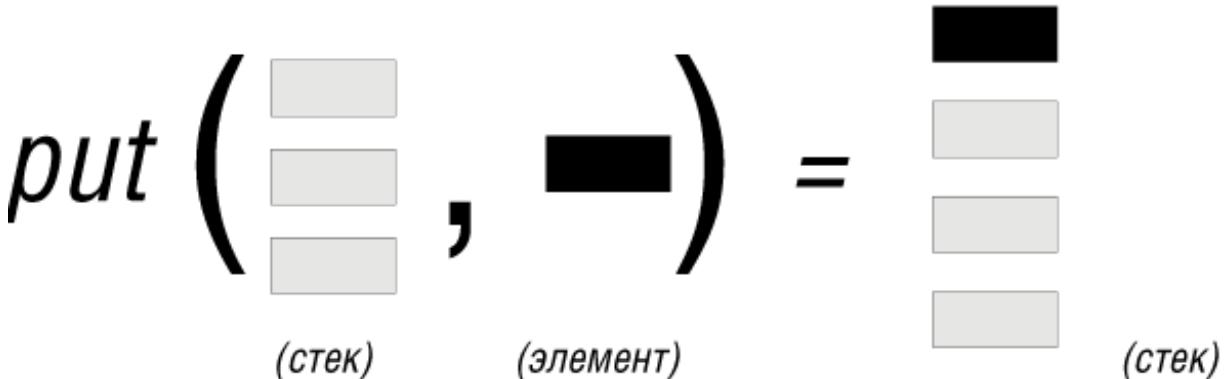


Рис. 6.3. Применение функции `put`

АТД имеют дело только с математическими функциями, у которых нет никаких побочных эффектов и которые, на самом деле, ничего не изменяют. Когда мы покинем утонченную сферу спецификации и попадем в неразбериху проектирования и реализации программ, нам придется восстановить понятие изменения, так как из-за накладных расходов мало кто одобрит программное окружение, в котором каждое выполнение операции "втолкнуть" в стек начинается с копирования этого стека. Мы рассмотрим позже переход от лишенного изменений мира АТД к полному изменений миру разработки ПО. Но поскольку сейчас мы хотим понять, как лучше всего определять типы, то математический взгляд на вещи нас вполне устраивает.

Из нашего обсуждения следуют роли операций, моделируемых каждой из функций спецификации STACK:

- Функция `put` возвращает новое состояние стека с одним новым элементом, помещенным на его вершину. Рисунок на предыдущей странице иллюстрирует операцию `put(s, x)`, выполняемую над стеком s и элементом x.
- Функция `remove` возвращает новое состояние стека с вытолкнутым верхним элементом, если таковой был. Как и `put`, эта функция при проектировании и реализации должна превращаться в команду (операцию, изменяющую объект, обычно реализуемую как процедура). Мы увидим далее, как учесть возможность пустого стека, с вершиной которого нечего удалять.
- Функция `item` возвращает верхний элемент стека, если таковой имеется.
- Функция `empty` выявляет пустоту стека, ее результатом является логическое значение (истина или ложь). Предполагается, что АТД BOOLEAN, задающий логические значения, определен отдельно.
- Функция `new` создает пустой стек.

В разделе ФУНКЦИИ эти функции определяются не полностью, вводятся только их **сигнатуры** - списки типов их аргументов и результата. Сигнтура функции `put`

`STACK [G] × G → STACK [G]`

показывает, что `put` берет в качестве аргумента пару вида $\langle s, x \rangle$, в которой s - экземпляр типа STACK [G], а x - экземпляр типа G, и возвращает в качестве результата экземпляр типа STACK [G]. Вообще говоря, множество значений функции (его тип указывается в сигнатуре правее стрелки, здесь это STACK [G]) может само быть декартовым произведением. Это можно использовать при описании операций, возвращающих два или более результатов.

В сигнатуре функций `remove` и `item` вместо обычной стрелки используется перечеркнутая стрелка $\cancel{\rightarrow}$. Это означает, что эти функции применимы не ко всем элементам множества входов. Описание функции `new`

выглядит просто как

new: STACK

без всякой стрелки в сигнатуре. Фактически, это сокращение для записи

new: \rightarrow STACK,

определенной функцию без аргументов. Здесь аргументы не нужны, поскольку new должна всегда возвращать один и тот же результат - пустой стек. Поэтому для простоты мы убрали здесь стрелку. Результат применения этой функции (т. е. пустой стек) будет записываться new, как сокращение для new(), обозначающего результат применения new к пустому списку аргументов.

Категории функций

В начале этой лекции операции над типами были разделены на конструкторы, запросы и команды. В спецификации АТД для нового типа T, например для STACK [G] в нашем примере можно определить эту классификацию более строго. Эта классификация просто проверяет, где по отношению к стрелке расположен в сигнатуре каждой функции тип T:

В альтернативной терминологии эти три категории называются "конструктор", "аксессор" и "модификатор". Здесь мы придерживаемся терминов, более непосредственно связанных с интерпретацией функций АТД как моделей операций над программными объектами.

- Функция, в сигнатуре которой T появляется лишь справа от стрелки, например new, является **функцией-конструктором**. Она моделирует операцию, создающую экземпляры T из экземпляров других типов или вообще не использующую аргументов, например как в случае константного конструктора new.
- Такие функции как item и empty, у которых T появляется только слева от стрелки, являются **функциями-запросами**. Они моделируют операции, которые устанавливают свойства T, выраженные в терминах экземпляров других типов (в наших примерах - это BOOLEAN и параметр типа G).
- Такие функции как put и remove, у которых T появляется с обеих сторон стрелки, являются **функциями-командами**. Они моделируют операции, которые по существующим экземплярам T и, возможно, экземплярам других типов выдают новые экземпляры типа T.

Раздел АКСИОМЫ

Мы уже видели, как типы данных (например, STACK) описываются посредством задания списка функций, применимых к их экземплярам. Все, что известно об этих функциях, - это их сигнатуры.

Чтобы указать, что речь идет о стеке, а не какой-либо другой структуре данных, имеющейся пока спецификации АТД совершенно недостаточно. Всякий распределитель, например очередь: "первым вошел - первым вышел", также будет удовлетворять этой спецификации.

Это, конечно, не должно удивлять, поскольку в разделе ФУНКЦИИ сами функции только объявляются (так же, как в программе объявляются переменные), но полностью не определяются. В ранее рассмотренном примере математического определения:

square_plus_one: R \rightarrow R
square_plus_one (x) = $x^2 + 1$ (для каждого x из R)

первая строка играет роль сигнатуры, но есть еще и вторая строка, в которой определяется значение функции. Как можно достичь того же для функций АТД?

Мы не будем использовать явные определения в духе второй строки определения функции square_plus_one, потому что это заставило бы нас выбрать интерпретацию, а все предшествующее обсуждение показало нам опасность раннего выбора представления.

Только чтобы убедиться в том, что мы понимаем, как может выглядеть явное определение, давайте напишем одно такое определение для приведенного ранее представления стека МАССИВ_ВВЕРХ. С точки зрения математики выбор этого представления означает, что экземпляр типа STACK - это пара $\langle \text{count}, \text{representation} \rangle$, где representation - это массив, а count - это число помещенных в стек элементов. Тогда явное определение функции put (для любого экземпляра x типа G) выглядит так:

put ($\langle \text{count}, \text{representation} \rangle, x$) = $\langle \text{count} + 1, \text{representation} [\text{count}+1: x] \rangle$

где $a[n:v]$ обозначает массив, полученный из a путем изменения значения элемента с индексом n на v (все остальные элементы не изменяются).

Это определение функции put является просто математической версией реализации операции put , набросок которой в стиле Паскаля приведен вслед за представлением МАССИВ_ВВЕРХ на рисунке с возможными представлениями стеков в начале этой лекции.

Но это не то определение, которое бы нас устроило. "Освободите нас от рабства представлений!" - этот лозунг Фронта Освобождения Объектов и его военного крыла (бригады АТД) является также и нашим. (Отметим, что его политическая ветвь, специализируется на тяжбах: класс - действие).

Поскольку всякое явное определение заставляет выбирать некоторое представление, обратимся к **неявным определениям**. При этом воздержимся от определения значений функций в спецификации АТД и вместо этого опишем свойства этих значений - все их существенные свойства, но только эти свойства.

Они формулируются в разделе АКСИОМЫ (AXIOMS). Для типа STACK он выглядит следующим образом.

Аксиомы

Для всех $x: G, s: \text{STACK}[G]$,

- (A1) $\text{item}(\text{put}(s, x)) = x$
- (A2) $\text{remove}(\text{put}(s, x)) = s$
- (A3) $\text{empty}(\text{new})$
- (A4) $\text{not empty}(\text{put}(s, x))$

Первые две аксиомы выражают основные свойства стеков (последним пришел - первым ушел) LIFO. Чтобы понять их, предположим, что у нас есть стек s и экземпляр x , и определим s' как результат $\text{put}(s, x)$, т. е. как результат втальивания x в s . Приспособим один из предыдущих рисунков:

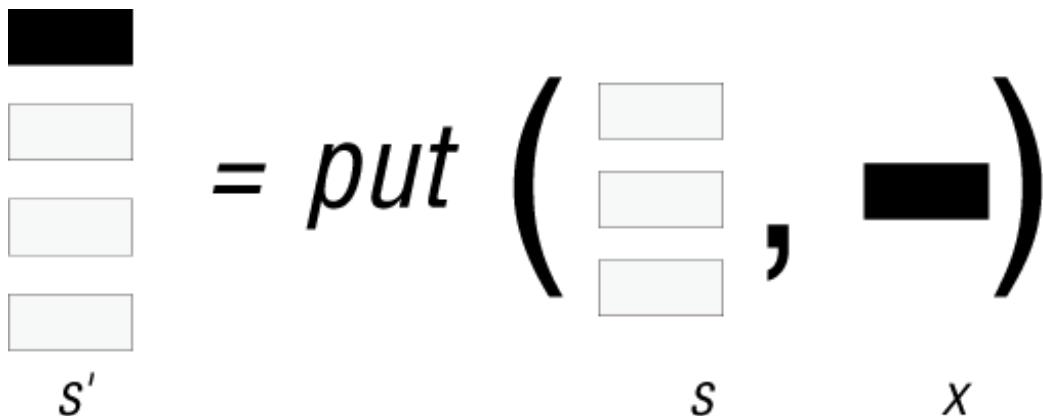


Рис. 6.4. Применение функции put

Здесь аксиома A1, говорит о том, что вершиной s' является x - последний элемент, который мы втолкнули, а аксиома A2 объясняет, что при удалении верхнего элемента s' мы снова получаем тот же стек s , который был до втальивания x . Эти две аксиомы дают лаконичное описание главного свойства стеков в чисто математических терминах без всякой помощи императивных рассуждений или ссылок на свойства представлений.

Аксиомы A3 и A4 говорят о том, когда стек пуст, а когда - нет: стек, полученный в результате работы конструктора new пустой, а всякий стек, полученный после втальивания элемента в уже существующий стек (пустой или непустой) не является пустым.

Эти аксиомы, как и остальные, являются предикатами (в смысле логики), выражающими истинность некоторых свойств для всех возможных значений s и x . Некоторые предпочитают рассматривать A3 и A4 в другой эквивалентной форме **как определение функции empty индукцией по размеру стеков**:

Для всех $x: G, s: \text{STACK}[G]$
A3' · $\text{empty}(\text{new}) = \text{true}$
A4' · $\text{empty}(\text{put}(s, x)) = \text{false}$

Две или три вещи, которые мы знаем о стеках

Спецификации АТД являются **неявными**. Имеются два вида "неявности":

- Метод АТД определяет неявно некоторое множество объектов, задавая применимые к ним функции. Из

этого определения никогда не следует, что в нем перечислены все операции; часто, на пути к представлению, будут добавлены и другие.

- Сами функции также определяются неявно. Вместо явных определений используются аксиомы, задающие свойства этих функций. Здесь тоже ничего не утверждается о полноте: когда вы, в конце концов, дойдете до реализации этих функций, они приобретут дополнительные свойства.

Эта неявность является ключевым аспектом абстрактных типов данных и, как следствие, - их будущих аналогов в построении ОО-ПО - классов. Когда мы определяем абстрактный тип данных или класс, мы всегда сообщаем кое-что об этом типе или классе, просто перечисляя те их свойства, которые знаем, и берем их в качестве определения. При этом никогда не предполагается, что других применимых свойств нет.

Неявность также предполагает открытость определений: всегда можно добавить новые свойства АТД или класса. Основным механизмом для выполнения таких расширений без разрушения уже существующего первоначального определения является наследование.

Этот "неявный" подход имеет далеко идущие последствия. В пункте "дополнительные темы" в конце этой лекции помещены еще некоторые комментарии о неявности.

Частичные функции

Спецификация всякого реалистичного примера, даже такого простого как стеки, необходимо сталкивается с проблемами не всюду определенных операций: некоторые операции применимы не ко всем возможным элементам исходных множеств. Например, это имеет место для функций `remove` и `item`: нельзя удалить элемент из пустого стека, и у пустого стека нет верхнего элемента.

Решение этой проблемы, использованное в приведенной выше спецификации, состоит в том, чтобы определить эти функции как частичные. Функция из исходного множества X в результирующее множество Y является **частичной**, если она определена не для всех элементов X . Функция, не являющаяся частичной, называется **полной**. Простым примером частичной функции в обычной математике является функция обращения действительных чисел inv , значение которой на действительном числе x равно

$$\text{inv}(x) = 1/x.$$

Поскольку inv не определена при $x = 0$, мы можем определить ее как частичную функцию на множестве R всех действительных чисел:

$$\text{Inv}: R \not\rightarrow R$$

Чтобы указать, что функция частичная, используется перечеркнутая стрелка $\not\rightarrow$, а обычная стрелка \rightarrow будет означать, что функция заведомо полная.

Областью (определения) частичной функции типа $X \not\rightarrow Y$ является подмножество тех элементов X , для которых эта функция имеет некоторое значение. В нашем примере областью функции inv является $R - \{0\}$, т.е. множество действительных чисел, отличных от 0.

В спецификации АТД STACK эти идеи использованы для стеков при объявлении `remove` и `item` как частичных функций в разделе ФУНКЦИИ - это указано с помощью перечеркнутых стрелок в их сигнатуре. При этом возникает новая проблема, обсуждаемая в следующем пункте: как задавать области таких функций?

В некоторых случаях функцию `put` тоже желательно описывать как частичную, например, это требуется в таких реализациях как `МАССИВ_ВВЕРХ` и `МАССИВ_ВНИЗ`, которые поддерживают выполнение лишь конечного числа подряд идущих операций `put` для каждого заданного стека. Это на самом деле полезное упражнение - приспособить спецификацию STACK к тому, чтобы она описывала ограниченные стеки конечного объема, поскольку в приведенном выше виде она не содержит никаких ограничений на размеры стеков.

Это будет новым применением частичных функций, отражающим ограничения реализации. В отличие от этого, объявление функций `remove` и `item` как частичных отражает абстрактное свойство этих операций, относящееся ко всем реализациям.

Предусловия

Частичные функции являются неустранимым фактом процесса проектирования ПО, отражающим очевидное наблюдение: не каждая операция применима ко всем объектам. Но они также являются и потенциальным источником ошибок: если функция f из X в Y является частичной, то нельзя быть уверенным в том, что выражение $f(e)$ имеет смысл, даже если e принадлежит X - требуется гарантировать, что это значение принадлежит области f .

Для этого всякая спецификация АТД, содержащая частичные функции, должна задавать их области. В этом и состоит роль раздела ПРЕДУСЛОВИЯ (PRECONDITIONS). Для АТД STACK этот раздел выглядит так:

Предусловия (preconditions)

- `remove (s: STACK [G]) require not empty (s)`
- `item (s: STACK [G]) require not empty (s)`

В нем у каждой из функций в пункте "требует" перечисляются условия, которым должны удовлетворять аргументы функции, чтобы входить в ее область.

Булевское выражение, которое определяет область функции, называется **предусловием** соответствующей частичной функции. В нашем случае предусловия обеих функций `remove` и `item` утверждают, что стек должен быть непустым. Перед "требует" помещается имя функции с именами ее аргументов (в примере для аргумента-стека использовано `s`), так что предусловие может ссылаться на эти аргументы.

С точки зрения математики предусловие функции `f` - это характеристическая функция области `f`.

Характеристической функцией подмножества Амножества X называется полная функция $ch: X \rightarrow \text{BOOLEAN}$ такая, что $ch(x)$ истинна, если x принадлежит A , и ложна в противном случае.

Полная спецификация

Раздел ПРЕДУСЛОВИЯ (PRECONDITIONS) завершает простую спецификацию абстрактного типа данных STACK. Для удобства ссылок полезно собрать вместе разные компоненты спецификации, приведенные выше. Вот полная спецификация.

Спецификация стеков как АТД

ТИПЫ (TYPES)

- `STACK [G]`

ФУНКЦИИ (FUNCTIONS)

- `put: STACK [G] × G → STACK [G]`
- `remove: STACK [G] ↛ STACK [G]`
- `item: STACK [G] ↛ G`
- `empty: STACK [G] → BOOLEAN`
- `new: STACK [G]`

АКСИОМЫ (AXIOMS)

Для всех $x: G$, $s: \text{STACK } [G]$

- (A1) $\text{item} (\text{put} (s, x)) = x$
- (A2) $\text{remove} (\text{put} (s, x)) = s$
- (A3) $\text{empty} (\text{new})$
- (A4) $\text{not empty} (\text{put} (s, x))$

ПРЕДУСЛОВИЯ (PRECONDITIONS)

- `remove (s: STACK [G]) require not empty (s)`
- `item (s: STACK [G]) require not empty (s)`

Ничего кроме правды

Сила спецификаций АТД проистекает из их способности отражать только существенные свойства структур данных без лишних деталей. Приведенная выше спецификация стеков выражает все, что нужно по существу знать о понятии стека, и не включает ничего, что относилось бы к каким-либо конкретным реализациям стеков. Это вся правда о стеках, и ничего кроме правды.

Такие спецификации задают общую модель вычислений на соответствующих структурах данных. Определенные в спецификации абстрактного типа данных функции позволяют строить сложные выражения, а аксиомы АТД позволяют упрощать такие выражения и получать более простые результаты. Сложное стековое выражение является математическим эквивалентом программы, а процесс упрощения является математическим эквивалентом вычисления или выполнения этой программы.

Вот пример. Рассмотрим для приведенной выше спецификации АТД STACK следующее выражение stackexp:

```
item (remove (put (remove (put (put (put (new, x1), x2), x3),
remove (put (put (new, x4), x5))), x6)), x7))
```

По-видимому, выражение stackexp будет проще понять, если мы представим его как последовательность вспомогательных выражений:

```
s1 = new
s2 = put (put (put (s1, x1), x2), x3)
s3 = remove (s2)
s4 = new
s5 = put (put (s4, x4), x5)
s6 = remove (s5)
y1 = item (s6)
s7 = put (s3, y1)
s8 = put (s7, x6)
s9 = remove (s8)
s10 = put (s9, x7)
s11 = remove (s10)
stackexp = item (s11)
```

Какой бы вариант определения вы ни выбрали, по нему несложно восстановить вычисление, математической моделью которого является stackexp: создать новый стек; втолкнуть в него элементы x_1, x_2, x_3 (в указанном порядке); удалить верхний элемент (x_3), назвав получившийся стек s_3 ; создать другой пустой стек и т. д. Этот процесс графически представлен на [рис. 6.5](#).

Можно легко найти значение такого АТД выражения, нарисовав последовательно несколько таких рисунков. (Здесь найдено x_4). Но теория позволяет нам получить этот результат формально, не обращаясь к рисункам, а только последовательно применяя аксиомы для упрощения выражения, до тех пор, пока дальнейшее упрощение станет невозможным. Например:

- Применить А2 для упрощения s_3 - т. е. заменить $\text{remove}(\text{put}(\text{put}(s_1, x_1), x_2), x_3)$ на выражение $\text{put}(\text{put}(s_1, x_1), x_2)$. (Согласно А2 всякую пару remove-put можно выбросить).

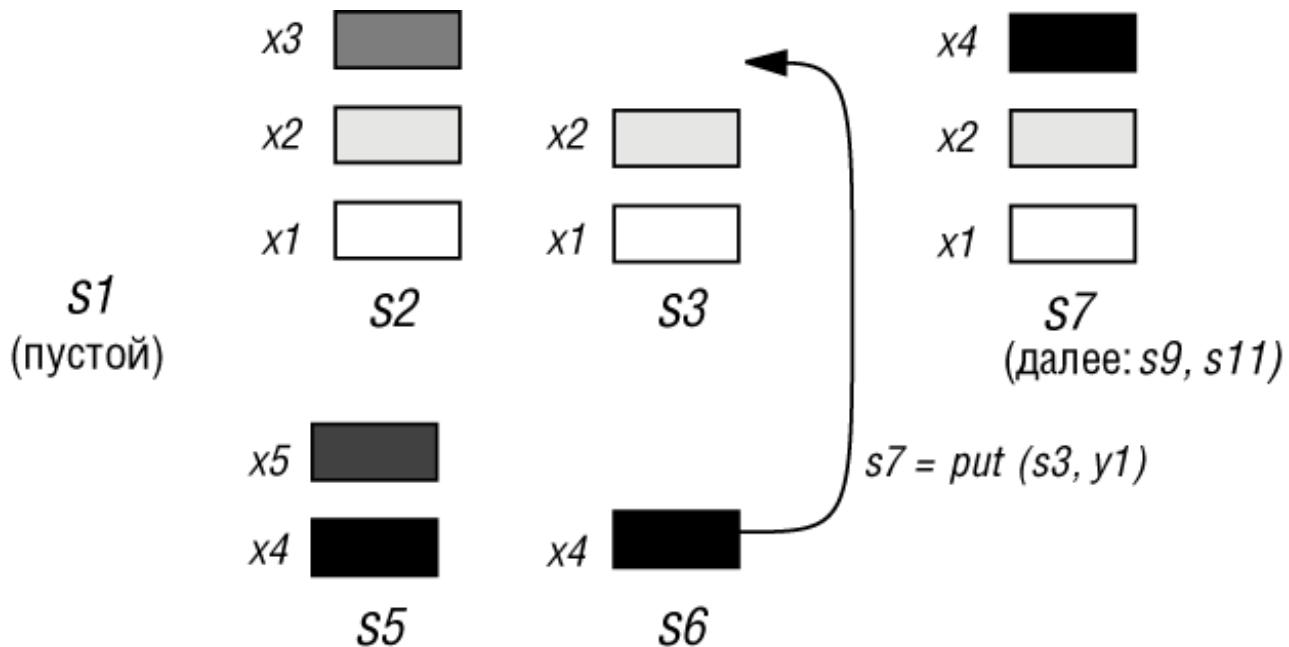


Рис. 6.5. Манипуляции со стеком

- По той же аксиоме s_6 равно $\text{put}(s_4, x_4)$. Затем можно применить аксиому А1 и вывести, что y_1 , т. е. $\text{item}(\text{put}(s_4, x_4))$ на самом деле равно x_4 , установив тем самым (как указано стрелкой на рисунке), что s_7 получается в результате вталкивания x_4 на вершину стека s_3 .

И так далее. Последовательность таких упрощений, выполненная механически так же легко и как последовательность упрощений в элементарной арифметике, приведет к значению выражения stackexp, которое действительно равно x_4 (попробуйте проверить это сами, аккуратно проведя весь процесс

упрощения).

Этот пример позволяет отметить одну из важнейших теоретических ролей абстрактных типов данных: они предоставляют формальную модель для понятий программы и выполнения программы. Эта модель чисто математическая: в ней нет императивных понятий состояния программы, переменных с изменяемыми во времени значениями, последовательности выполняемых действий. Она основана на обычных математических методах преобразования выражений.

От абстрактных типов данных к классам

Итак, у нас имеется отправная точка - элегантная математическая теория для моделирования структур данных и, как мы только что видели, в целом - программ. Но наша цель - это архитектура ПО, а не математическая или даже теоретическая информатика! Не сбились ли мы с нашего пути? Отнюдь. При поиске подходящей модульной структуры, основанной на типах объектов, АТД предоставляют механизм описания высокого уровня, не связанный с особенностями реализации. Это приведет нас к фундаментальным структурам ОО-технологии.

Классы

В поиске, начатом в [лекции 3](#), АТД будут служить непосредственной основой модулей. Точнее, ОО-система будет строиться (на уровне анализа, проектирования и реализации) как совокупность взаимодействующих, частично или полностью реализованных АТД. Основное понятие здесь - **класс**:

Определение: класс

Класс - это абстрактный тип данных, снабженный некоторой (возможно частичной) реализацией

Таким образом, чтобы получить класс, мы должны построить АТД и решить, как его реализовывать. АТД - это математическое понятие, а реализация - это его версия, ориентированная на компьютер. Приведенное определение, однако, утверждает, что реализация может быть частичной. Введенные ниже термины позволяют отделить этот случай от полностью реализованного класса:

Определение: отложенный и эффективный классы

Полностью реализованный класс называется эффективным (effective). Класс, который реализован лишь частично или совсем не реализован, называется отложенным (deferred). Всякий класс является либо отложенным, либо эффективным.

Чтобы получить эффективный класс, требуется предусмотреть все детали реализации. Для отложенного класса можно выбрать определенный уровень реализации, но при этом оставить некоторые аспекты реализации незавершенными. В самом крайнем случае при частичной реализации можно вообще отказаться от принятия каких-либо решений о ее уточнении. В этом случае получившийся класс будет полностью отложенным и будет эквивалентен АТД.

Как создавать эффективный класс

Рассмотрим вначале эффективные классы. Что нужно сделать для реализации АТД? Результирующий эффективный класс будет формироваться из элементов трех видов:

- (E1) Спецификации АТД (множество функций с соответствующими аксиомами и предусловиями, описывающими их свойства).
- (E2) Выбора представления.
- (E3) Отображения из множества функций (E1) в представление (E2) в виде множества механизмов (или компонентов (features)), каждый из которых реализует одну из функций в терминах представления и при этом удовлетворяет аксиомам и предусловиям. Многие из этих компонентов будут методами - обычными процедурами, но некоторые могут появляться в качестве полей данных или "атрибутов" (это будет показано в следующих лекциях).

Например, для АТД STACK можно выбрать в качестве представления (шаг E2) решение, названное выше МАССИВ_ВВЕРХ, при котором каждый стек реализуется парой

```
<representation, count>,
```

где representation - это массив, а count - это целое число. При реализации функций (E3) у нас будут процедуры для функций put, remove, item, empty и new, выполняющие соответствующие действия.

Например, функцию put можно реализовать программой вида

```
put (x: G)
```

```

is -- Втолкнуть x в стек.
-- (без проверки стека на возможное переполнение.)
do
  count := count + 1
  representation [count]:= x
end

```

Объединение элементов, полученных в пунктах (E1), (E2) и (E3), приведет к классу - модульной структуре объектной технологии.

Роль отложенных классов

В определении эффективного класса должна присутствовать полная информация о реализации (пункты E2 и E3). Если она хоть в чем-то неполна, то класс является отложенным.

Чем более "отложенным" является класс, тем он ближе к АТД, одетому в некоторую синтаксическую одежду, которая скорее поможет завоевать признание разработчиков ПО, чем математиков. Отложенные классы особенно полезны при анализе и проектировании:

- При ОО-проектировании многие аспекты реализации будут опущены, проектирование должно сосредотачиваться на архитектурных свойствах высокого уровня - на том, какую функциональность обеспечивает каждый модуль системы, а не на том, как он это делает.
- При постепенном продвижении к полной реализации будут добавляться все новые и новые ее свойства до тех пор, пока не будет получен эффективный класс.

Но на этом роль отложенных классов не завершается, даже в полностью реализованной системе можно часто обнаружить много таких классов. Кое-что следует из только что перечисленных применений: когда из отложенных классов получаются эффективные, то появляется желание сохранить их в качестве предков (в смысле наследования) эффективных классов как живую память о процессе анализа и проектирования.

Очень часто при разработке ПО с помощью не ОО-подходов система в окончательном виде не содержит никаких записей о тех значительных усилиях, которые были затрачены на ее получение. Для тех, кто вынужден будет обслуживать такую систему - расширять, переносить, отлаживать - понять ее без этих записей будет так же трудно, как трудно геологу понять видимый ландшафт, не имея доступа к осадочным слоям. Один из лучших способов обеспечить необходимую для сопровождения системы информацию - это сохранить отложенные классы в ее окончательной форме.

У отложенных классов имеется также применение, полностью связанное с реализацией. Они служат для классификации групп связанных типов объектов, предоставляют некоторые наиболее важные многократно используемые модули высокого уровня, фиксируют общие свойства поведения многих вариантов и играют ключевую роль (вместе с полиморфизмом и динамическим связыванием) в обеспечении децентрализации и расширяемости программной архитектуры.

Несколько следующих лекций, в которых вводятся основные ОО-методы, будут сосредоточены на эффективных классах. Но при этом следует помнить о понятии отложенного класса, чья важность будет расти по мере овладения всей мощью ОО-метода.

Абстрактные типы данных и скрытие информации

Особенно интересным следствием ОО-политики, в которой модули основаны на реализациях АТД (классах), является то, что она дает ясный ответ на вопрос, который остался нерешенным при обсуждении скрытия информации: как нам следует разделять общедоступные и скрытые свойства модуля - видимую и невидимую части айсберга?

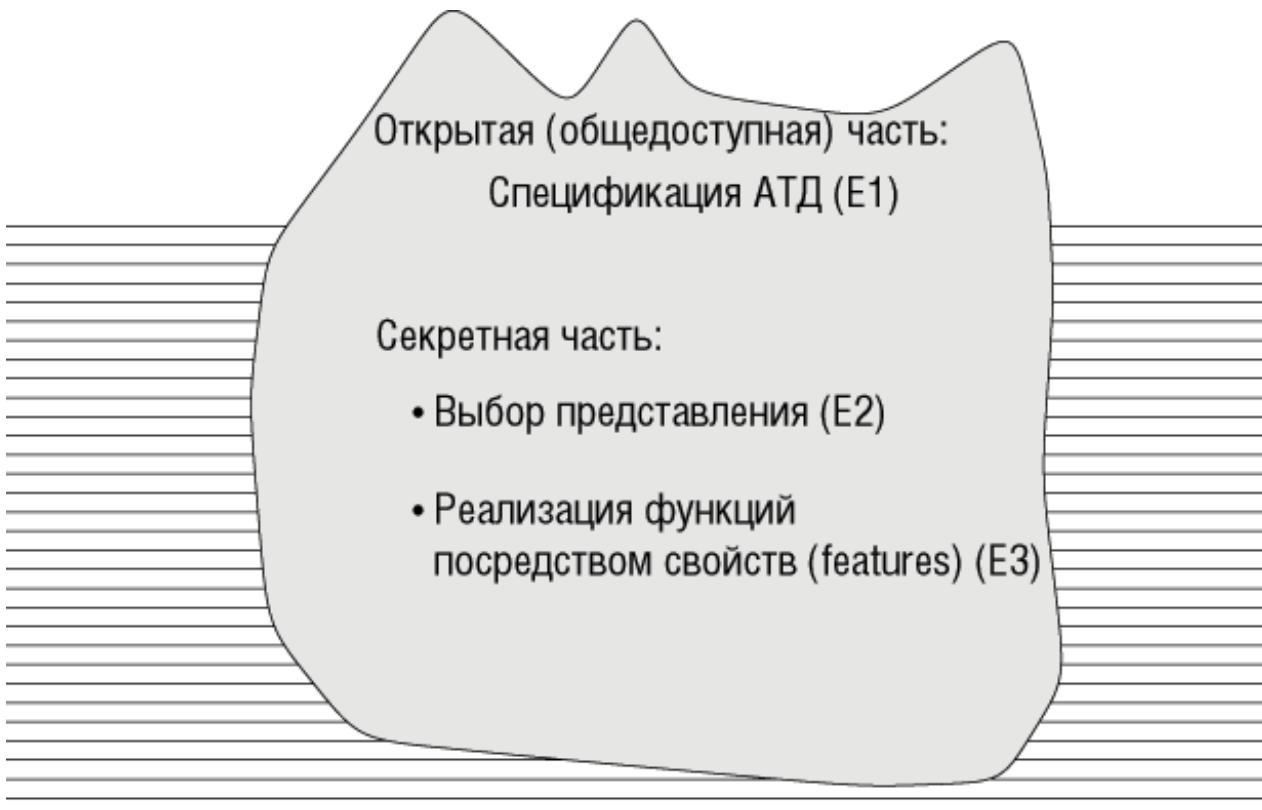


Рис. 6.6. АТД вид модуля при скрытии информации

Если модуль является классом, полученным из АТД, то ответ ясен. Из трех частей, вовлеченных в эту эволюцию, Е1- спецификация АТД, является открытой, а Е2 и Е3 - выбор представления и реализация функций АТД в терминах этого представления - должны быть закрытыми (секретными). Когда мы начнем строить классы, то столкнемся еще с четвертой частью, также секретной, - вспомогательными свойствами, необходимыми только для внутренних нужд этих программ.

Таким образом, использование абстрактных типов данных в качестве источника модулей дает нам практическое, однозначное указание для применения скрытия информации в наших проектах.

Переход к более императивной точке зрения

Переход от АТД к классам включает существенное изменение стилистики: введение изменений и императивных аргументов.

Как вы помните, спецификация абстрактных типов данных не описывает явно изменений, т. е., используя термин из теоретической информатики, является **аппликативной**. Все свойства АТД моделируются как математические функции, это относится к конструкторам, запросам и командам. Например, операция вталкивания для стеков моделируется функцией-командой:

`put: STACK [G] × G → STACK [G],`

задающей операцию, которая возвращает новый стек, а не изменяет существующий.

Классы отказываются от чисто аппликативной точки зрения на функции и переопределяют команды как операции, которые могут изменять объекты. Например, операция `put` будет определена как процедура, которая получает некоторый элемент типа `G` (формальный параметр) и модифицирует стек, вталкивая новый элемент на его вершину, не создавая нового стека.

Такое изменение стиля отражает императивные настроения, преобладающие при разработке ПО. (В качестве синонима слова "императивный" иногда используется термин "операционный"). При этом потребуется изменять аксиомы АТД. Аксиомы стеков А1 и А4, которые имели вид

- (A1) `item (put (s, x)) = x`
- (A4) `not empty (put (s, x))`

превратятся в императивной форме в предложение, называемое постусловием программы (*routine postcondition*), вводимое ключевым словом `ensure` (обеспечивает):

```
put (x: G) is
-- Втолкнуть x на вершину стека
```

```

require
    ... Предусловие (если таковое имеется) ...
do
    ... Соответствующая реализация (если известна) ...
ensure
    item = x
    not empty
end

```

Здесь постусловие объясняет, что результатом вызова программы риt значение item будет равно x (втолкнутому элементу), а значение empty будет ложно.

Другие аксиомы спецификации АТД приводят к утверждению, известному как **инвариант класса**. Постусловия, инварианты класса и другие перевоплощения предусловий и аксиом АТД мы рассмотрим во время обсуждения утверждений и проектирования по контракту (п. 11.10 "Связь с АТД").

Назад к тому, с чего начали?

Если вы внимательно следили, начиная с лекции о модульности, за главной линией рассуждений, которая привела нас к абстрактным типам данных, а затем и к классам, то сейчас, быть может, вы будете удивлены. Поставив целью получить по возможности наилучшую модульную структуру, мы пришли к тому, что объекты, точнее - типы объектов, будут лучшей основой для модулей, чем их традиционные соперники - функции. Это привело к следующему вопросу: как описать эти типы объектов. Но, когда мы на него ответили: описывать нужно в виде абстрактных типов данных (и их заменителей на практике - классов), то оказалось, что нужно основывать описание данных на ... применяемых к ним функциях! Не получился ли у нас порочный круг?

Нет. Типы объектов, представляемые АТД и классами, остаются неизменной основой модуляризации.

Неудивительно, что и объектный, и функциональный аспект должен проявиться в окончательной архитектуре системы: никакое описание вопросов ПО не может считаться полным, если в нем опущена одна из этих компонент. Фундаментальное различие ОО-методов и старых подходов состоит в распределении ролей: типы объектов - безусловные победители при выборе критерии для построения модулей. Функциям достается только роль их слуг.

При ОО-декомпозиции никакая функция не существует сама по себе - каждая функция прикреплена к некоторому типу объектов. Это относится и к уровню проектирования, и к уровню разработки: никакое свойство не существует само по себе, каждое из них прикреплено к некоторому классу.

Конструирование объектно-ориентированного ПО

Мы уже давали определение конструирования ОО-ПО: будучи весьма общим, оно представляет метод следующим образом: "основывать архитектуру всякой программной системы на модулях, полученных из типов объектов, с которыми оперирует система". Придерживаясь рамок этого определения, мы можем дополнить его теперь более техническим определением:

Конструирование объектно-ориентированного ПО (определение 2)

Конструирование ОО-ПО - это построение программной системы как структурированной совокупности реализаций (возможно частичных) абстрактных типов данных.

Это определение будет нашим рабочим определением. Все его компоненты являются важными:

- В основе лежит понятие абстрактного типа данных.
- Для конструирования программ нам нужны не сами по себе АТД (как математическое понятие), а **реализации** АТД - программистское понятие.
- При этом эти реализации не обязаны быть полными, оговорка "возможно частичные" позволяет использовать и отложенные классы, включая, как крайний случай, полностью отложенный класс без какой-либо реализации.
- Система представляет собой **совокупность** классов без выделения какого-либо главного или ответственного класса или головной программы.
- Эта совокупность является структурированной благодаря двум отношениям между классами: "быть клиентом" и наследованию.

За пределами программ

Подчеркнем теперь важность понятия АТД для областей, лежащих вне непосредственной области его предполагаемого применения.

Подход, основанный на АТД, говорит нам, что серьезное интеллектуальное исследование должно отвергать всякую попытку понять суть вещей изнутри как бесполезную, и вместо этого должно сосредотачиваться на понимании используемых свойств этих вещей. Не объясняйте мне, что вы собой представляете, скажите мне, что у вас есть - что я могу от вас получить. Если потребуется дать имя этой эпистемологической дисциплине, мы скажем, что это **принцип разумного эгоизма**.

Если я испытываю жажду, то апельсин - это то, из чего я могу выдавить сок, если художник, то цвет - это то, что может воодушевить мою палитру, если фермер, то это - продукт, который я могу продать на рынке, если архитектор, то это - чертежи, показывающие мне, как спроектировать новый оперный театр, но если я - ни один из них, и никак не использую апельсин, то я не должен говорить о нем, поскольку понятие "апельсин" для меня даже не существует.

Принцип эгоизма, утверждающий, что вы - это то, что у вас есть, является крайним выражением идеи, играющей центральную роль в развитии науки: идеи абстракции или важности разделения понятий. Две цитаты, приведенные в начале этой лекции, каждая из которых по-своему замечательна, выражают важность этой идеи. Их авторы Дидро и Стендаль были писателями, а не учеными, хотя очевидно, что у обоих имелось хорошее понимание сути научного метода. (Дидро был пылким вдохновителем Большой энциклопедии, а Стендаль готовился к поступлению в Политехническую школу, хотя затем решил, что может найти более подходящие занятия). Просто поразительно, насколько обе цитаты применимы к использованию абстракции при конструировании программ.

Но в принципе эгоизма есть и кое-что помимо абстракции - это, кажущаяся с первого взгляда шокирующей, идея о том, что ни о каком свойстве не стоит говорить, если от него нет никакой прямой пользы говорящему.

Это приводит к мысли рассмотреть общее интеллектуальное значение нашей области.

На протяжении ряда лет во многих статьях и выступлениях предлагалось проверить, как разработчики ПО могут извлечь выгоду от изучения философии, общей теории систем, "когнитивных наук", психологии. Но для практикующих разработчиков программ результаты оказываются разочаровывающими. Если исключить из рассмотрения универсально применимые законы рационального (разумного) исследования, известные просвещенным умам уже в течение многих веков (по крайней мере, с Декарта), которые, разумеется, применимы к информатике, как и ко всему прочему, то иногда кажется, что специалисты в вышеуказанных дисциплинах могут получить больше, обучаясь у специалистов по программному обеспечению, чем наоборот.

Конструкторы программ брались - с различной степенью успеха - за решение ряда самых сложных из когда-либо рассматриваемых интеллектуальных задач. Немногие из инженерных проектов могут сравниться по сложности с программными проектами, содержащими много миллионов строк, которые регулярно производятся в наши дни. Приложив немало амбициозных усилий, программистское сообщество достигло точного понимания таких предметов и понятий, как размер, сложность, структура, абстракция, таксономия, параллельность, рекурсивный вывод, различие между описанием и предписанием, язык, изменение и инварианты. Все это произошло так недавно и настолько интуитивно, что сама эта профессиональная среда еще не осознала эпистемологических последствий собственной деятельности.

В конце концов, появится кто-нибудь, кто объяснит, какие уроки весь интеллектуальный мир может извлечь из опыта конструирования ПО. Нет сомнений в том, что абстрактные типы данных будут играть в них выдающуюся роль.

Дополнительные темы

Представленное выше описание абстрактных типов данных вполне достаточно для использования АТД в рамках данной книги. (Чтобы дополнить его, выполните упражнения, которые помогут уточнить ваше понимание этого понятия).

Если же, как я надеюсь, АТД уже завоевали вас своей элегантностью, простотой и мощью, то не исключено, что вам захочется узнать побольше об их свойствах, даже о таких, которые не будут использоваться в обсуждении ОО-методов. Далее на нескольких страницах рассмотрены следующие дополнительные темы, которые можно опустить при первом чтении:

- неявность и ее связь с процессом конструирования ПО;
- различие между спецификацией и проектированием;
- различие между классами и записями;
- возможные альтернативы использованию частичных функций;
- решение о полноте или неполноте спецификации.

Библиографические ссылки к этой лекции указывают на более специальную литературу по АТД.

Еще раз о неявности

Неявная природа абстрактных типов данных и классов, рассмотренная выше, отражает одну из важных

проблем конструирования программ.

Вполне законен вопрос о различии между упрощенной спецификацией АТД, использующей объявление функций

```
x: POINT → REAL  
y: POINT → REAL
```

и объявлением типа в таком традиционном языке программирования, как Pascal:

```
type  
POINT =  
record  
    x, y: real  
end
```

На первый взгляд эти два объявления представляются эквивалентными: оба утверждают, что с типом POINT связаны два значения x и y типа REAL. Но между ними имеется существенная, хотя и тонкая разница:

- Запись в языке Pascal является законченной и явной: она показывает, что объект POINT включает два данных поля и ничего кроме них.
- Объявление функций АТД не несут такого смысла. Они показывают, что объект типа POINT можно запрашивать о значениях его x и y, но не исключают других запросов, например, о массе и скорости точки в кинематическом приложении.

С упрощенной математической точки зрения можно считать, что приведенное выше объявление в Паскале является определением математического множества POINT как декартова произведения:

```
POINT ≡ REAL × REAL,
```

где знак \equiv означает "определяется как" ("равно по определению"), и оно полностью задает POINT. В отличие от этого спецификация АТД не определяет явно POINT посредством такой математической модели как декартово произведение, она просто неявно характеризует POINT, перечисляя два запроса, применимых к объектам этого типа.

Если имеется спецификации некоторого понятия, то может появиться желание переместить ее из неявного мира в явный, идентифицируя понятие с декартовым произведением применимых к нему простых запросов, например захочется идентифицировать точки с парами $\langle x, y \rangle$. Такой процесс идентификации можно рассматривать как определение перехода от анализа и спецификации к проектированию и реализации.

Соотношение спецификации и проектирования

Предыдущее наблюдение помогает уточнить один из центральных вопросов, возникающих при изучении ПО: различие между начальным этапом разработки ПО - его спецификацией, называемым также анализом, - и более поздними стадиями такими, как проектирование и реализация.

В литературе по разработке программ обычно объясняется, что это различие между "определением задачи" и "построением ее решения". Будучи в принципе правильным, такое объяснение не всегда применимо на практике и иногда бывает трудно понять, где заканчивается спецификация и начинается проектирование. Даже в среде исследователей люди запросто критикуют друг друга в связи с этой темой: "вы рекламируете язык x как язык спецификаций, но на самом деле он предназначен для проектирования". Наивысшим оскорблением считается обвинение некоторой системы обозначений в обслуживании **реализации** (подробнее об этом в одной из следующих лекций).

Приведенное выше определение дает более точный критерий: пересечь Рубикон между спецификацией и проектированием - это перейти от неявного к явному, другими словами:

Определение: переход от анализа (спецификации) к проектированию

Перейти от спецификации к проектированию - это идентифицировать каждую абстракцию с декартовым произведением ее простых запросов.

Последующий переход - от проектирования к реализации - это просто движение от одного явного вида к другому: форма при проектировании более абстрактна и ближе к математическим понятиям, а при реализации более конкретна и ближе к компьютеру, но обе они являются явными. Этот переход менее драматичен, чем предыдущий - действительно, при дальнейшем чтении станет понятно, что объектная технология почти стирает различие между проектированием и реализацией. При хорошей системе ОО-нотации нашими

компьютерами непосредственно выполняется (с помощью компиляторов) то, что в не ОО-мире часто рассматривалось бы как проекты.

Соотношение классов и записей

Другим замечательным свойством объектной технологии является то, что при ней можно сохранять неявные описания гораздо дольше, чем при других подходах. В последующих лекциях будет введена система обозначений, позволяющая определять класс в виде:

```
class POINT feature
    x, y: REAL
end
```

Это выглядит подозрительно похожим на приведенное выше определение записи в Паскале. Но, несмотря на внешнее сходство, определение класса другое - оно неявное! Эта неявность проявляется при наследовании: автор класса или (что еще более интересно) кто-либо другой может в любой момент определить новый класс, например:

```
class MOVING_POINT inherit
    POINT
feature
    mass: REAL
    velocity: VECTOR [REAL]
end
```

который расширяет исходный класс совершенно незапланированным способом. Тогда переменная (или сущность, если использовать вводимую далее терминологию) типа POINT, объявленная как

```
p1: POINT
```

может быть связана с объектом не только типа POINT, но и с каждым потомком этого типа, например с объектом типа MOVING_POINT. Это может получиться, в частности, с помощью "полиморфных присваиваний" вида:

```
p1 := mp1
где mp1 имеет тип MOVING_POINT.
```

Эти возможности иллюстрируют неявность и открытость определения класса: соответствующие экземпляры представляют не только точки в узком смысле, т. е. непосредственно экземпляры класса POINT, но и экземпляры всякого класса, описывающего понятия, выводимые из исходного класса.

Способность определять элементы программ (классы), которые немедленно используются (посредством наследования), оставаясь неявными, является одним из главных нововведений объектной технологии, непосредственно отвечающему принципу Открыт-Закрыт. В последующих лекциях будут раскрыты все вытекающие из нее следствия.

Альтернативы частичным функциям

Один из технических приемов, используемый в этой лекции, мог вызвать удивление, - применение частичных функций. Он связан с неустранимой проблемой применения в некоторой спецификации не всюду определенных операций. Но являются ли частичные функции лучшим решением этой проблемы?

Конечно, это не единственное возможное решение. Другим способом, который приходит на ум и действительно используется в некоторых работах по АТД, является превращение частичной функции во всюду определенную за счет введения специального значения "ошибка" для случаев применения функции к неподходящим аргументам.

Каждый тип T дополняется значением "ошибка". Обозначим его через w_T . Тогда для всякой функции f сигнатура

```
f: ... Типы входов ... → T
```

определяет, что всякое применение f к объекту, для которого соответствующее вычисление не может быть выполнено, выдаст значение w_T .

Хотя этот метод и используется, он приводит к математическим и практическим неудобствам. Проблема в том, что такие специальные значения являются весьма эксцентричными существами, которые могут чрезвычайно осложнить жизнь невинных математических существ.

Предположим, например, что рассматриваются стеки целых чисел - экземпляры типа STACK [INTEGER], где INTEGER - это АТД, экземпляры которого - целые числа. Хотя для нашего примера не требуется полностью выписывать спецификацию INTEGER, этот АТД должен моделировать основные операции (сложение, вычитание, "меньше чем" и т. п.), определенные на математическом множестве целых чисел. Аксиомы этого АТД должны выражать обычные свойства целых чисел. Вот одно из таких типичных свойств: для всякого целого n :

[Z1]

$$n + 1 \neq n$$

Пусть теперь n будет результатом запроса верхнего элемента пустого стека, т. е. значением выражения `item(new)`, где `new` - это пустой стек целых чисел. При этом запросе n должно получить специальное значение `WINTEGER`. Что же тогда должно быть значением выражения $n+1$? Если у нас в распоряжении имеются в качестве значений только обычные целые числа и `WINTEGER`, то в качестве ответа мы вынуждены выбрать `WINTEGER`:

$$WINTEGER + 1 = WINTEGER.$$

Это единственный допустимый выбор. Если присвоить `WINTEGER+1` любое другое значение, "нормальное" число q , то это означает, что после попытки доступа к вершине пустого стека и получения в качестве результата ошибочного значения мы можем волшебным образом устраниТЬ всякую память об этой ошибке, просто прибавив к результату единицу!

Но, при выборе `WINTEGER` в качестве значения $n + 1$ при n равном `WINTEGER`, нарушается указанное выше свойство Z1. В общем случае, выражение `WINTEGER+r` будет равно `WINTEGER` для любого r . Это означает, что для измененного типа данных (INTEGER, дополненные ошибочным элементом) требуется новая система аксиом, объясняющая, что всякая операция над целыми числами возвращает значение `WINTEGER`, если хоть один из ее аргументов равен `WINTEGER`. Аналогичные изменения потребуются для каждого типа.

Получившееся усложнение не кажется обоснованным. Мы не можем изменять спецификацию целых чисел только для того, чтобы промоделировать каждую отдельную структуру данных (в нашем случае - стеки). При использовании частичных функций ситуация более простая. Конечно, для всякого выражения, содержащего частичные функции, приходится проверять, что их аргументы удовлетворяют соответствующим предусловиям. После завершения такой проверки, можно беспрепятственно применять аксиомы. При этом не требуется изменять существующие системы аксиом.

Полна ли моя спецификация?

Другой вопрос, который может вас тревожить: есть ли какой-нибудь способ убедиться в том, что спецификация описывает все нужные свойства объектов, для которых она предназначена? Студенты, которым требуется написать их первые спецификации (например, проделать упражнения в конце этой лекции), часто приходят с аналогичным вопросом: "Как узнать, что я уже специфицировал достаточно свойств и могу остановиться?"

В более общей форме вопрос звучит так: существует ли метод, позволяющий определять полноту спецификации АТД?

Если непосредственно задать вопрос в такой форме, то ответ будет простой - нет. Понятно, что для формальной спецификации сказать, что она полна - это утверждать, что она покрывает все необходимые свойства, но это имеет смысл только по отношению к некоторому эталонному документу, в котором все эти свойства перечислены. Тогда мы сталкиваемся с двумя равно неутешительными ситуациями:

- Если эталонный документ является неформальным (например, документом с требованиями на естественном языке или просто текстом упражнения), то отсутствие формальности предотвращает всякую попытку систематической проверки соответствия спецификации всем требованиям, описанным в этом документе.
- Если же эталонный документ является формальным, и мы можем, используя его, проверить полноту нашей спецификации, то это просто отодвигает проблему дальше: как можно убедиться в полноте самого эталонного документа?

Таким образом, в этой тривиальной форме вопрос о полноте неинтересен. Но имеется и более полезное понятие полноты, соответствующее значению этого слова в математической логике. Для математика некоторая теория является полной, если ее аксиомы и правила вывода являются достаточно мощными, чтобы доказать истинность или ложность любой формулы, выразимой в языке данной теории. Хотя такое понятие полноты является более ограниченным, но оно интеллектуально вполне удовлетворительно, поскольку показывает, что если теория позволяет нам выражать некоторое свойство, то она также дает возможность

определить имеет ли это свойство место.

Как можно перенести эту идею на спецификации АТД? Здесь "язык теории" - это множество правильно построенных выражений, т.е. тех выражений, которые можно построить, используя функции АТД, применяемые к аргументам соответствующих типов. Например, используя спецификацию АТД STACK и считая, что x является правильно построенным выражением типа G, можно указать следующие правильно построенные выражения:

new

put (new, x)

item (new) - если это кажется странным, то см. комментарии ниже.

empty (put (new, x))

stackexp - ранее определенное сложное выражение.

Однако выражения **put (x)** и **put (x, new)** не являются правильно построенными, так как они не соответствуют правилу: **put** всегда должно иметь два аргумента - первый типа STACK [G] и второй типа G.

Третий пример в рамке **item (new)** не задает никакого осмысленного вычисления, поскольку аргумент **new** не удовлетворяет предусловию для **item**. Хотя это выражение и правильно построено, оно не является корректным. Вот точное определение этого понятия.

Определение: корректное выражение АТД

Пусть $f(x_1, \dots, x_n)$ - правильно построенное выражение, содержащее одну или более функций некоторого АТД. Это выражение является корректным тогда и только тогда, когда все его аргументы x_i являются (по рекурсии) корректными и их значения удовлетворяют предусловию f , если оно имеется.

Не следует путать "корректное" и "правильно построенное". "Правильно построенное" - это структурное свойство, указывающее на то, что функции, входящие в выражение, имеют правильное число аргументов соответствующих типов, а корректность, которой могут обладать лишь **правильно построенные** выражения, означает, что данное выражение задает осмысленное вычисление. Как мы видели, выражение **put (x)** не является правильно построенным (и поэтому бессмысленно спрашивать, корректно ли оно), а выражение **item (new)** правильно построено, но некорректно.

Правильно построенное, но некорректное выражение похоже на программу, которая компилируется (поскольку построена в соответствии с требованиями синтаксиса языка программирования и удовлетворяет ограничениям, накладываемым в нем на типы), но аварийно завершается во время выполнения из-за выполнения некоторой недопустимой операции, например, деления на 0 или выталкивания элемента из пустого стека.

Особый интерес с точки зрения полноты представляют **выражения-запросы**, у которых самая внешняя функция является запросом. Вот примеры таких выражений:

```
empty (put (put (new, x1), x2))
item (put (put (new, x1), x2))
stackexp
```

Выражение-запрос задает значение, которое (если оно определено) принадлежит не определяемому АТД, а некоторому другому ранее определенному типу. Так, первое приведенное выше выражение имеет значение типа BOOLEAN, а второе и третье - тип G формального параметра для элементов стека, например если мы рассматриваем АТД STACK [INTEGER], то это будет тип INTEGER.

Выражения-запросы представляют внешние наблюдения, которые можно сделать о результатах некоторого вычисления, использующего экземпляры нового АТД. Если спецификация этого АТД хорошая, то она должна позволить нам установить определены ли эти результаты, и если да, то каковы они. Представляется, что спецификация стека обладает этим свойством, по крайней мере, для трех представленных в примере выражений, поскольку она позволяет установить, что все эти выражения определены, и с помощью аксиом можно получить их значения:

```
empty (put (put (new, x1), x2)) = False
item (put (put (new, x1), x2)) = x2
stackexp = x4
```

Эти наблюдения, перенесенные на произвольные спецификации АТД, приводят к pragматическому понятию

полноты, известному как **достаточная** полнота, она означает, что спецификация содержит достаточно сильные аксиомы, которые позволяют находить для любого выражения-запроса его результат в виде некоторого простого значения.

Приведем точное определение достаточной полноты. (Не расположенные к математике читатели могут пропустить остаток этого раздела).

Определение: достаточная полнота

Спецификация АТД Т является достаточно полной тогда и только тогда, когда аксиомы ее теории позволяют для каждого выражения expr решить следующие задачи:

- (S1) Определить, является ли expr корректным.
- (S2) Если expr - выражение-запрос и в пункте S1 установлена его корректность, то представить значение expr в виде, не включающем никаких значений типа Т.

В S2 выражение expr имеет вид $f(x_1, \dots, x_n)$, где f - функция вида запрос такая, как empty и item для стеков. S1 говорит о том, что у expr есть значение, но этого недостаточно, нам хотелось бы знать, каково это значение, представленное в терминах значений других типов (в примере со стеком это значения типов BOOLEAN и G). Если аксиомы настолько сильны, что всегда позволяют ответить на этот вопрос, то спецификация является достаточно полной.

Достаточная полнота свидетельствует о том, что никакое важное свойство не осталось вне нашей спецификации. Поэтому ее можно считать ответом на поставленный выше вопрос: как понять, что можно прекратить поиски новых свойств при построении спецификации? На практике хорошо бы проводить такую проверку, по крайней мере неформально, для любой спецификации АТД, которую вы пишите - начните с решений упражнений, приведенных в этой лекции. Часто, можно получить формальное доказательство достаточной полноты; приведенное ниже доказательство для спецификации STACK является образцом, которому во многих случаях можно следовать.

Пункт S2 оптимистически говорит об одном значении expr, а что, если аксиомы приводят к двум или более значениям? Это сделало бы спецификацию бесполезной. Чтобы устранить такую ситуацию нам нужно еще одно свойство, называемое в математической логике непротиворечивостью:

Определение: непротиворечивость АТД

Спецификация АТД является непротиворечивой тогда и только тогда, когда для всякого правильно построенного выражения expr ее аксиомы позволяют вывести не более одного значения.

Эти два свойства являются взаимно дополнительными. Нам хотелось бы для каждого выражения-запроса выводить ровно одно значение: хотя бы одно (достаточная полнота), но не более одного (непротиворечивость).

Доказательство достаточной полноты

(Этот раздел и остаток этой лекции содержат дополнительный материал и их результаты не нужны для остальной части книги).

Достаточная полнота спецификаций АТД является, в общем случае, алгоритмически неразрешимой проблемой. Иными словами, не существует общего метода доказательства, который мог бы по заданной спецификации АТД выяснить за конечное время ее достаточную полноту. Непротиворечивость также в общем случае неразрешима.

Несмотря на это, часто удается доказать достаточную полноту и непротиворечивость конкретной спецификации АТД. Чтобы удовлетворить любопытство читателей-любителей математики, в заключение этой лекции мы приведем доказательство того, что спецификация STACK на самом деле является достаточно полной. Доказательство ее непротиворечивости будет оставлено в качестве упражнения.

Для доказательства достаточной полноты спецификации стека нужно придумать эффективное правило для решения указанных выше задач S1 и S2, другими словами, такое правило, которое позволит нам для любого стекового выражения e:

- (S1) Определить, является ли e корректным.
- (S2) Если в пункте S1 установлена корректность e и его внешними функциями являются item или empty (т.е. функции-запросы), то представить значение e с помощью значений типов BOOLEAN и G без ссылок на значения типа STACK [G] или на функции из спецификации STACK.

Для начала мы рассмотрим только правильно построенные выражения, не включающие ни одну из двух функций-запросов item и empty, т. е. выражения, построенные только из функций new, put и remove. Таким

образом, на этом этапе нас будет интересовать лишь задача S1 (установить определено ли выражение). Функции-запросы и S2 будут рассмотрены далее.

Правило для решения задачи S1 задается следующим свойством:

Правило корректного веса

Правильно построенное стековое выражение e , не содержащее ни $item$, ни $empty$, является корректным тогда и только тогда, когда его вес неотрицателен и каждое его подвыражение является (по индукции) корректным.

Здесь "вес" выражения представляет число элементов в соответствующем стеке, это значение также совпадает с разностью между числом вложенных вхождений функций put и $remove$. Приведем точное определение этого понятия:

Определение: вес

Вес правильно построенного стекового выражения, не содержащего ни $item$, ни $empty$, определяется по индукции следующим образом:

- (W1) Вес выражения new равен 0.
- (W2) Вес выражения $put (s, x)$ равен $ws + 1$, где ws - это вес s .
- (W3) Вес выражения $remove (s)$ равен $ws - 1$, где ws - это вес s .

Содержательно, правило корректного веса утверждает, что стековое выражение корректно тогда и только тогда, когда в нем самое и в каждом из его подвыражений имеется не меньше операций put (вставляющих элементы в стек), чем операций $remove$ (выталкивающих элементы с вершины стека). Если рассмотреть такое выражение как представление некоторого вычисления над стеком, то это означает, что мы никогда не будем пытаться вытолкнуть больше элементов, чем втолкнули. Напомним, что на этом этапе мы сосредоточились на функциях put и $remove$, оставив в стороне запросы $item$ и $empty$.

Интуитивно сформулированное правило выглядит верным, но нам следует все же доказать, что оно имеет место. Удобно ввести еще одно вспомогательное правило и одновременно доказывать справедливость обоих этих правил:

Правило нулевого веса

Пусть e - это правильно построенное и корректное стековое выражение, не содержащее $item$ или $empty$. Тогда $empty (e)$ истинно тогда и только тогда, когда вес e равен 0.

Доказательство использует индукцию по уровню вложенности (максимальному числу вложенных пар скобок) выражения. Для удобства ссылок напомним аксиомы, относящиеся к функции $empty$:

Аксиомы стека

Для всех $x: G, s: \text{STACK}[G]$

- (A3) $\text{empty}(\text{new})$
- (A4) $\text{not empty}(\text{put}(s, x))$

При уровне вложенности 0 (без скобок) выражение e должно совпадать с new , поэтому его вес равен 0 и оно корректно, так как у new нет никаких предусловий. Аксиома A3 утверждает, что $\text{empty}(\text{new})$ истинно. Это обеспечивает базис индукции как для правила корректного веса, так и для правила нулевого веса.

Индукционный шаг: предположим, что оба правила выполняются для всех выражений с уровнем вложенности не более n . Нужно доказать, что тогда они выполняются и для любого выражения e с уровнем вложенности $n+1$. Поскольку наши выражения сейчас не содержат функций-запросов, то e должно иметь один из следующих двух видов:

$$\begin{aligned} E1 \cdot e &= \text{put}(s, x) \\ E2 \cdot e &= \text{remove}(s) \end{aligned}$$

где x имеет тип G , а уровень вложенности у s равен n . Пусть ws - это вес s .

В случае E1, поскольку put - всюду определенная функция, e корректно тогда и только тогда, когда s корректно, т. е. (по предположению индукции) тогда и только тогда, когда s и все его подвыражения имеют неотрицательные веса. Но это эквивалентно тому, что e и все его подвыражения имеют неотрицательные веса, что и доказывает правило корректного веса в этом случае. Кроме того, e имеет положительный вес $ws+1$, и (по аксиоме A4) является непустым, что доказывает правило нулевого веса.

В случае E2 выражение е корректно тогда и только тогда, когда выполняются два следующих условия:

- EB1 _ s и все его подвыражения являются корректными.
- EB2 _ not empty (s) (это предусловие для функции remove).

По предположению индукции условие EB2 означает, что вес s ws положителен или, что эквивалентно, вес e, равный ws - 1, является неотрицательным. Следовательно, e удовлетворяет Правилу корректного веса. Чтобы доказать, что оно также удовлетворяет правилу нулевого веса, нужно показать, что e пусто тогда и только тогда, когда его вес равен 0. Так как вес s положителен, то s должно содержать по крайней мере одно вхождение put, которое также входит в e. Рассмотрим самое внешнее вхождение put в e, это вхождение находится непосредственно внутри remove (так как remove находится на самом внешнем уровне у e). Это означает, что у e имеется подвыражение (быть может, совпадающее с самим e) вида

```
remove (put (stack_expression, g_expression)),
```

которое по аксиоме A2 можно сократить просто до stack_expression. После выполнения этой замены вес e уменьшится на 2, и получившееся выражение, имеющее то же значение, что и e, удовлетворяет по предположению индукции правилу нулевого веса. Это доказывает утверждение индукции в случае E2.

Это доказательство попутно показывает, что во всяком правильно построенном выражении, не содержащем функций-запросов item и empty, можно устраниć все вхождения remove, т.е. получить, применяя всюду, где это возможно, аксиому A2, некоторую каноническую форму, в которую будут входить только put и new. Например, выражение:

```
put (remove (remove (put (put (remove (put (put (new, x1), x2)), x3), x4))), x5)
```

имеет то же значение, что и каноническая форма:

```
put (put (new, x1), x5).
```

Давайте дадим этому механизму имя и приведем его определение:

Правило канонического сокращения

Всякое правильно построенное и корректное стековое выражение, не содержащее функций-запросов item и empty, имеет эквивалентную каноническую форму, которая не содержит функции remove (т.е. состоит только из функций put и new). Эта каноническая форма получается путем применения аксиомы стека A2 всегда, пока это возможно.

Таким образом, мы завершили доказательство достаточной полноты, но только для выражений, не содержащих функции-запросы, и, следовательно, только свойства S1 (проверка корректности выражения). Для завершения доказательства нужно рассмотреть выражения, включающие функции-запросы, и обсудить задачу S2 (нахождение значений для выражений-запросов). Это означает, что нам нужно некоторое правило для определения корректности и значения всякого правильно построенного выражения вида f(s), где s - это правильно построенное выражение, а f - это либо item, либо empty.

Это правило и доказательство его корректности также используют индукцию по уровню вложенности. Пусть n - это уровень вложенности s. Если n=0, то s может быть только new, поскольку остальные функции требуют аргументов и, следовательно, содержат хоть одну пару скобок. Тогда для обеих функций-запросов ситуация ясна:

- empty (new) корректно и имеет значение истина (**true**) (по аксиоме A3);
- item (new) некорректно, так как предусловие item требует выполнения not empty (s) .

Индукционный шаг: предположим, что s имеет уровень вложенности n не менее 1. Если у какого-либо подвыражения i выражения s внешняя функция есть item или empty, то уровень вложенности i не превосходит n-1, что по предположению индукции позволяет определить корректность i и, если i корректно, получить его значение, применяя аксиомы. Выполнив замены всех таких подвыражений, получим для s эквивалентную форму, в которую входят только функции put, remove и new.

Далее используем идею введенной выше канонической формы, чтобы избавиться от всех вложений remove, так что результирующая форма для s будет включать только функции put и new. Случай, когда s это просто new уже был рассмотрен, остался случай, когда s имеет вид put(s', x) . В этом случае для двух рассматриваемых выражений имеем:

- empty (s) корректно и по аксиоме A3 значение этого выражения есть ложь (**false**);
- item (s) корректно, так как предусловие not empty (s) для item выполнено; из аксиомы A1 следует, что значение этого выражения равно x.

Это завершает доказательство достаточной полноты, так как мы показали справедливость множества правил - правила корректного веса и правила канонического сокращения, позволяющего нам выяснить корректность заданного стекового выражения, а для корректного выражения-запроса - определять его значение в терминах значений типов BOOLEAN и G.

Ключевые концепции

- Теория абстрактных типов данных (АТД) примиряет необходимость в точности и полноте спецификаций с желанием избежать лишних деталей в спецификации.
- Спецификация абстрактного типа данных является формальным математическим описанием, а не текстом программы. Она **аппликативна**, т.е. не включает в явном виде изменений.
- АТД может быть родовым, и он задается функциями, аксиомами и предусловиями. Аксиомы и предусловия выражают семантику данного типа и важны для полного и однозначного его описания.
- Частичные функции образуют удобную математическую модель для описания не всюду определенных операций. У каждой частичной функции имеется предусловие, задающее условие, при котором она будет выдавать результат для заданного конкретного аргумента.
- ОО-система - это совокупность классов. Каждый класс основан на некотором абстрактном типе данных и задает частичную или полную реализацию этого АТД.
- Класс является эффективным, если он полностью реализован, в противном случае он называется отложенным.
- Классы должны разрабатываться в наиболее общем виде, допускающем повторное использование; процесс их объединения в систему часто идет снизу-вверх.
- Абстрактные типы данных являются скорее неявными, чем явными описаниями. Эта неявность, которая также означает открытость, переносится на весь ОО-метод.
- Не существует формального определения интуитивно ясного понятия "полноты" спецификации абстрактного типа данных. Строго определяемое понятие **достаточной** полноты как правило обеспечивает удовлетворительный ответ. Хотя не существует метода, устанавливающего достаточную полноту произвольной спецификации, часто удается ее доказать для конкретных спецификаций; приведенное в этой лекции доказательство достаточной полноты для спецификации стеков может служить образцом и для других случаев.

Библиографические замечания

Несколько работ, опубликованных в начале 1970-х, сделали возможным появление абстрактных типов данных. Среди них наиболее известны статья Хоара о "доказательстве корректности представлений данных" [Hoare 1972a], в которой было введено понятие абстракции функций, и работа Парнasa по скрытию информации, отмеченная в библиографических заметках к [лекции 3](#).

Конечно, абстрактные типы данных не ограничиваются вопросами скрытия информации, хотя многие их элементарные изложения дальше этого не идут. Собственно АТД были введены Лисков и Зиллеса [Liskov 1974]; более алгебраические представления были приведены в [M1976] и [Guttag 1977]. Так называемая группа ADJ (Гоген, Тэтчер, Вагнер) исследовали алгебраические основания абстрактных типов данных, используя теорию категорий. В частности, см. их важную статью [Goguen 1978], опубликованную в коллективной монографии.

На основе абстрактных типов данных основано несколько языков спецификаций. Двумя результатами группы ADJ являются CLEAR [Burstall 1977] [Burstall 1981] и OBJ-2 [Futatsugi 1985]. См. также Larch, предложенный Гуттагом, Хорнингом и Вингом [Guttag 1985].

Идеи АТД повлияли на такие языки формальных спецификаций как Z в ряде его воплощений [Abrial 1980] [Abrial 1980a] [Spivey 1988] [Spivey 1992] и VDM [Jones 1986]. Недавние расширения Z обнаружили тесную связь с ОО-идеями, см. например, Object Z [Duke 1991] и дальнейшие ссылки в гл. 11.

Фраза "разделение интересов" является центральной в работе Дейкстры, см. в частности, его "Дисциплину программирования" [Dijkstra 1976].

Понятие достаточной полноты было впервые опубликовано Гуттагом и Хорнингом [Guttag 1978] (оно основано на диссертации Гуттага 1975г.).

Идея о том, что переход от спецификации к проектированию означает переключение с неявного на явное путем отождествления АТД с декартовым произведением его простых запросов, была предложена в [M 1982] как часть теории описания структур данных в терминах трех разных уровней (физического, структурного, неявного).

Упражнения

У6.1 Точки

Написать спецификацию, задающую абстрактный тип данных ТОЧКА (POINT), моделирующий точки на плоскости в планиметрии. Эта спецификация должна отражать следующие аспекты: декартовы и полярные координаты, повороты, параллельные переносы, расстояние от начала координат, расстояние до другой точки.

У6.2 Боксеры

Члены Ассоциации Боевых Петухов - боксерской лиги - регулярно встречаются в поединках, чтобы установить их относительную силу. В поединке встречаются два боксера, и его результатом является победа одного и поражение другого боксера или ничья. Если выявлен победитель, то результат поединка используется для изменения рангов боксеров лиги: объявляется, что победитель превосходит побежденного и каждого боксера b, которого до поединка превосходил проигравший. Остальные соотношения остаются без изменений.

Опишите эту проблему как набор абстрактных типов данных: АТД_ЛИГА, БОКСЕР, ПОЕДИНОК. (Указание: не вводите явно понятие "ранг", а промоделируйте его с помощью функции "**превосходит**", выражающей отношение превосходства на множестве боксеров лиги.)

У6.3 Банковские счета

Написать спецификацию АТД "счет в банке" с такими операциями как " положить на счет", "снять со счета", "текущий баланс", "владелец", "смена владельца".

Каким образом добавить функции, представляющие операции открытия и закрытия счета? (Указание: эти функции являются функциями другого АТД).

У6.4 Сообщения

Рассмотрите знакомую вам систему электронной почты. Определите в духе этой лекции абстрактный тип данных ПОЧТОВОЕ_СООБЩЕНИЕ. Включите в него не только функции-запросы, но и команды и конструкторы.

У6.5 Имена

Разработайте абстрактный тип данных ИМЯ, в котором учитывались бы различные компоненты полного имени человека.

У6.6 Текст

Рассмотрите понятие текста, обрабатываемого текстовым редактором. Задайте это понятие в виде АТД. (Это задание оставляет достаточно много свободы спецификатору, не забудьте включить содержательное описание тех свойств текста, которые вы избрали для моделирования в АТД).

У6.7 Покупка дома

Напишите спецификацию абстрактного типа данных для задачи покупки дома, описанной в предыдущей лекции. Уделите особое внимание определению логических ограничений, выраженных в виде предусловий и аксиом спецификации АТД.

У6.8 Дополнительные операции для стеков

Модифицируйте спецификацию АТД для стеков, включив в нее операции count (возвращает число элементов стека), change_top (заменяет верхний элемент стека заданным элементом) и wipe_out (удаляет все элементы). Не забудьте включить необходимые аксиомы и предусловия.

У6.9 Ограниченные стеки

Измените приведенную в этой лекции спецификацию стеков так, чтобы она описывала стеки ограниченной емкости. (Указание: введите емкость как явную функцию-запрос и сделайте функцию put частичной).

У6.10 Очереди

Описать в виде АТД очереди (первым пришел - первым ушел) в том же стиле, что и стеки. Обратите внимание на общие и отличительные черты этих АТД. (Указание: аксиомы для item и remove должны отличаться, при описании put (s, x) рассмотрите случаи, когда очередь s пуста и непустая).

У6.11 Распределители

(Это упражнение предполагает, что вы выполнили предыдущее).

Определите общий АТД РАСПРЕДЕЛИТЕЛЬ, покрывающий и стеки и очереди.

Рассмотрите механизм для задания более специальных спецификаций АТД (таких как стеки и очереди) с помощью ссылок на общие спецификации такие, как спецификация распределителей. (**Указание:** посмотрите на механизм наследования, изучаемый в следующих лекциях).

У6.12 Булевский -- BOOLEAN

Определите абстрактный тип данных BOOLEAN так, чтобы его можно было использовать в определениях других АТД из этой лекции. Можно считать, что операции равенства и неравенства (= и \neq) автоматически определены для каждого АТД.

У6.13 Достаточная полнота

(Это упражнение предполагает, что вы выполнили одно или несколько предыдущих упражнений).

Изучите спецификацию АТД, написанную вами в качестве решения одного из предыдущих упражнений, и попытайтесь доказать, что она является достаточно полной. Если она не достаточно полная, то объясните, почему и покажите, как ее можно исправить или расширить, чтобы сделать достаточно полной.

У6.14 Непротиворечивость

Докажите, что приведенная в этой лекции спецификация стеков является непротиворечивой.

Основы объектно-ориентированного программирования

7. Лекция: Статические структуры: классы

Анализируя основы программной инженерии, мы поняли причины, требующие совершенствования модульного подхода - повторное использование и расширяемость кода. Мы осознали, что традиционные методы исчерпали себя, - централизованная архитектура ограничивает гибкость. Мы выявили хорошую теоретическую основу ОО-подхода - абстрактные типы данных. Теперь, когда проблемам уделено достаточно внимания, вперед к их решению! Раздел С содержит введение в фундаментальные методы ОО-анализа, проектирования и программирования. Необходимые обозначения (элементы описания) будут вводиться по мере необходимости. Сначала необходимо рассмотреть базовые строительные блоки - классы.

Классы, а не объекты - предмет обсуждения

Какова центральная концепция объектной технологии? Необходимо дважды подумать, прежде чем ответить "объект". Объекты полезны, но в них нет ничего нового.

С тех пор, как структуры используются в Cobol, с тех пор, как в Pascal существуют записи, с тех пор как программист написал на С первое определение структуры, человечество располагает объектами.

Объекты важны при описании выполнения ОО-систем. Но базовым понятием объектной технологии является класс. Обратимся вновь к его определению. (Детальное обсуждение объектов содержится в следующей лекции.)

Определение класса

Класс - это абстрактный тип данных, поставляемый с возможно частичной реализацией.

Абстрактные типы данных (АТД) являются математическим понятием, пригодным на этапе подготовки спецификации - в процессе анализа. Понятие класса, предусматривая частичную или полную реализацию, обеспечивает необходимую связь с разработкой ПО на этапах проектирования и программирования. Напомним, класс называется эффективным, если его реализация полна, и отложенным - при частичной реализации.

Аналогично АТД, класс это тип, описывающий множество возможных структур данных, называемых **экземплярами (instances)** класса. Экземпляры АТД являются абстракциями - элементами математического множества. Экземпляр класса конкретен - это структура данных, размещаемая в памяти компьютера и обрабатываемая программой.

Например, если определить класс STACK, взяв за основу спецификацию АТД из предыдущей лекции и добавив информацию, необходимую для адекватного представления, то экземплярами класса будут структуры данных - конкретные стеки. Другим примером является класс POINT, моделирующий точку на плоскости. Если для представления точки выбрана декартова система координат, то каждый экземпляр POINT представляет собой запись с полями x, y - абсциссой точки и ее ординатой.

Термин "объект" появляется как побочный продукт определения "класса". Объект это просто экземпляр некоторого класса.

Программные тексты, описывающие создаваемую систему, содержат определения классов. Объекты создаются только в процессе выполнения программ.

Настоящая лекция посвящена основным приемам создания программных элементов и объединения их в системы, именно поэтому в центре внимания - классы. В следующей лекции будут рассмотрены структуры периода выполнения, порождаемые ОО-системой, что потребует изучения некоторых особенностей реализации и более детального рассмотрения природы объектов.

Устранение традиционной путаницы

Класс это модель, а объект экземпляр такой модели. Эта особенность настолько очевидна, что обычно не требует дополнительных комментариев. Тем не менее, в определенной категории специальной литературы имеет место весьма небрежное обращение с этими понятиями, - смешиваются понятие отдельного объекта и концепция объектов в целом, которую характеризует класс. У этой путаницы два источника. Один - возникает из-за широкого толкования термина "объект" в естественном языке. Другой источник недоразумений связан с метаклассами, - с ситуациями, когда классы сами выступают в роли объектов. Классическим примером может служить транслятор объектного языка, для которого классы языка являются объектами трансляции.

Некоторые ОО-языки, особенно Smalltalk, для выхода из рассмотренной ситуации используют понятие **метакласс (metaclass)**. Метакласс - это класс, экземпляры которого сами являются классами. В романе "Имя Розы", отрывок из которого приведен в эпиграфе к данной лекции, встречается понятие "знаки знаков". По сути, это и есть неформальное определение метаклассов.

Мы будем избегать введения метаклассов, поскольку создаваемых ими проблем больше, чем тех, которые они решают. В частности, введение метаклассов создает трудности при проведении статической проверки типов, что является необходимым условием разработки надежного ПО. Основные функции метаклассов могут быть гораздо лучше реализованы с помощью других средств:

- Метаклассы можно использовать для задания свойств, доступных многим или всем классам. Тот же результат

можно достичь, создавая семейство классов, наследников общего предка - класса ANY, содержащего объявления универсальных свойств.

- Некоторые операции характерны, скорее, для класса в целом, а не для отдельных его экземпляров, так что их можно рассматривать как методы метакласса. Но этих операций обычно немного и они хорошо известны. Опять-таки, их можно ввести при определении класса ANY, или реализовать введением специальных языковых конструкций. Наиболее очевидным примером является конструктор класса, выполняющий операцию создания объектов.
- Метакласс может использоваться для получения дополнительной информации о классе - имени, списке свойств, списке родителей и т.д. Но и здесь нет необходимости в метаклассе. Достаточно разработать специальный библиотечный класс E_CLASS, экземпляры которого представляют классы и их свойства. При создании такого экземпляра необходимо передать в качестве параметра соответствующий класс С и далее использовать этот экземпляр для получения информации о классе С, обращаясь к соответствующим компонентам E_CLASS.

В данной книге не используется самостоятельная концепция метакласса. Присутствие метаклассов в том или ином языке или среде разработки не оправдывает смешение понятий моделей и их экземпляров - классов и объектов.

Роль классов

Затратив немного времени на устранение абсурдных, но распространенных и вредных заблуждений, можно вернуться к рассмотрению центральных свойств классов и выяснить, в частности, почему они столь важны в объектной технологии.

Для понимания ОО-подхода необходимо ясно представлять, что классы выполняют две функции, которые до появления ОО-технологий всегда были разделены. Класс одновременно является модулем и типом.

Модули и типы

Средства, используемые при разработке ПО, - языки программирования, проектирования, спецификаций, графические системы обозначений для анализа, - всегда включали в себя как возможность применения модулей, так и систему типов.

Модули - это структурные единицы, из которых состоит программа. Различные виды модулей, такие как подпрограммы и пакеты, рассматривались в одной из предыдущих лекций (см. [лекция 3](#)). Независимо от конкретного выбора той или иной модульной структуры, модуль всегда рассматривается как **синтаксическая концепция**. Отсюда следует, что разбиение на модули влияет лишь на форму записи исходных текстов программ, но не определяет их функциональность. В самом деле, принципиально можно написать программу Ada в виде единственного пакета, или программу Pascal как единую основную программу. Безусловно, такой подход не рекомендуется, и любой компетентный программист будет использовать модульные возможности языка для деления программы на обозримые и управляемые части. Но если взять существующую программу, например на Паскале, то всегда можно собрать воедино все модули и получить работоспособную программу с эквивалентной семантикой. (Присутствие рекурсивных подпрограмм делает этот процесс менее тривиальным, но не оказывает принципиального влияния на данную дискуссию.) Таким образом, деление на модули диктуется принципами управления проектами, а не внутренней необходимости.

Концепция **типов** на первый взгляд совершенно иная. Тип является статическим описанием вполне определенных динамических объектов - элементов данных, которые обрабатываются во время выполнения программной системы. Набор типов обычно содержит предопределенные типы, такие как INTEGER или CHARACTER, а также пользовательские типы: записи (структуры), указатели, множества (в Pascal), массивы и другие. Понятие типа является **семантической** концепцией, и каждый тип непосредственно влияет на выполнение программной системы, так как описывает форму объектов, которые система создает и которыми она манипулирует.

Класс как модуль и как тип

В не ОО-подходах концепции модуля и типа существуют независимо друг от друга. Наиболее замечательным свойством класса является одновременное использование обеих концепций в рамках единой лингвистической конструкции. Класс является модулем или единицей программной декомпозиции, но одновременно класс это тип (или шаблон типа в тех случаях, когда поддерживается параметризация).

Мощь ОО-метода, во многом, следствие этого отождествления. Наследование, в частности, может быть полностью понято, только при рассмотрении его как модульного расширения, так и, одновременно, уточнения специализации типа.

Как практически соединить две столь различные на первый взгляд концепции? Последующая дискуссия и примеры позволят ответить на этот вопрос.

Унифицированная система типов

Важным аспектом ОО-подхода является простота и универсальность системы типов, которая строится на основе фундаментального принципа.

Объектный принцип

Каждый объект является экземпляром некоторого класса

Объектный принцип будет распространяться не только на составные объекты, определяемые разработчиками (такие как структуры данных, содержащие несколько полей), но и на базовые объекты - целые и действительные числа, булевые значения и символы, которые будут рассматриваться как экземпляры предопределенных библиотечных классов (INTEGER, REAL, DOUBLE, BOOLEAN, CHARACTER).

На первый взгляд подобное стремление превратить любое сколь угодно простое значение в экземпляр некоторого класса может показаться преувеличенным и даже экстравагантным. В конце концов, математики и инженеры в течение многих лет успешно используют целые и действительные числа, не подозревая о том, что они работают с экземплярами классов. Однако настойчивое требование к унификации вполне окупается по ряду причин.

- Всегда желательно иметь простую и универсальную схему, нежели множество частных случаев. Предлагаемая система типов полностью опирается на понятие класса.
- Описание базовых типов как абстрактных структур данных и далее как классов является простым и естественным. Нетрудно представить, например, определение класса INTEGER с функциональностью включающей арифметические операции, такие как "+", операции сравнения, такие как ""=" и ассоциированные свойства, следующие из соответствующих математических аксиом.
- Определение базовых типов как классов позволяет использовать все возможности ОО, главным образом наследование и родовые средства. Если базовые типы не будут классами, то придется вводить ряд ограничений и рассматривать частные случаи.

Пример наследования - классы INTEGER, REAL и DOUBLE могут быть наследниками двух более общих классов; NUMERIC, в котором определены основные арифметические операции ("+", "-", "*"); COMPARABLE, представляющий операции сравнения ("<" и другие). В качестве примера использования универсализации можно рассмотреть родовой класс MATRIX, родовой параметр которого определяет тип элементов матрицы. Экземпляры класса MATRIX [INTEGER] будут целочисленными матрицами, а экземпляры MATRIX [REAL] будут в качестве элементов содержать действительные числа. В качестве комплексного примера одновременного использования наследования и родовых классов можно использовать класс MATRIX [NUMERIC], экземпляры которого могут содержать элементы типа INTEGER или REAL или любого нового типа T, определенного разработчиком как наследника класса NUMERIC.

При условии хорошей реализации нет необходимости опасаться каких-либо негативных последствий решения определять все типы как классы. Ничто не мешает предоставить компилятору специальную информацию о базовых классах. В этом случае порождаемый код для операций со значениями классов INTEGER и BOOLEAN может быть столь же эффективным, как если бы они были встроенными типами данного языка.

Построение непротиворечивой и универсальной системы типов требует комплексного применения ряда важных ОО-методик, которые будут рассмотрены позже. К их числу относятся расширяемые классы, гарантирующие корректное представление простых значений; инфиксные и префиксные операции, обеспечивающие возможность использования привычного синтаксиса ($a < b$ или $-a$ вместо неуклюжих конструкций $a.less_than(b)$ или $a.negated$); ограниченная универсализация, необходимая для описания классов, адаптируемых к типам со специфическими операциями. Например, класс MATRIX может представлять целочисленные матрицы, а также матрицы, элементами которых являются числа других типов.

Простой класс

Что представляет собой класс можно выяснить, изучая простой, но типичный пример, который демонстрирует фундаментальные свойства, применимые практически ко всем классам.

Компоненты

Пример использует представление точки в двумерной графической системе:

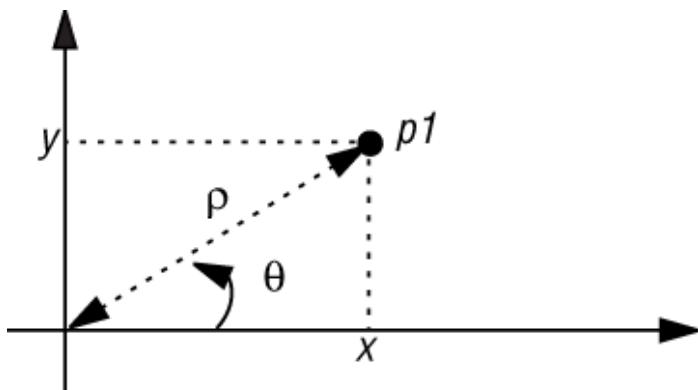


Рис. 7.1. Точка и ее координаты

Для определения типа POINT как абстрактного типа данных потребуется четыре функции-запроса: x, y, rho, theta. (В текстах подпрограмм для двух последних функций будут использоваться имена rho и theta). Функция x возвращает

абсциссу точки (горизонтальную координату), у - ординату (вертикальную координату), ρ - расстояние от начала координат, θ - полярный угол, отсчитываемый от горизонтальной оси. Значения x и y являются декартовыми, а ρ и θ - полярными координатами точки. Другой полезной функцией является `distance`, возвращающая расстояние между двумя точками.

Далее спецификация АТД будет содержать такие команды, как `translate` (перемещение точки на заданное расстояние по горизонтали и вертикали), `rotate` (поворот на определенный угол вокруг начала координат) и `scale` (уменьшение или увеличение расстояния до начала координат в заданное число раз).

Нетрудно написать полную спецификацию АТД, включающую указанные функции и некоторые ассоциированные аксиомы. Далее в качестве примера приведены две из перечисленных функций:

```
x: POINT → REAL  
translate: POINT × REAL × REAL → POINT
```

и одна из аксиом:

$$x(\text{translate}(p1, a, b)) = x(p1) + a$$

утверждающая, что для произвольной точки $p1$ и действительных значений a и b при трансляции точки на a, b абсцисса увеличивается на a .

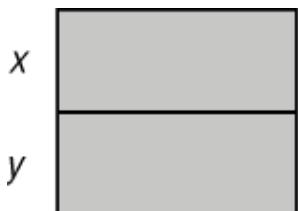
Читатель, если пожелает, может самостоятельно завершить спецификацию АТД. В дальнейшей дискуссии подразумевается, что вы понимаете, как устроен данный АТД, вне зависимости от того, написали ли вы его полную формализацию или нет. Сосредоточим внимание на реализации АТД - классе.

Атрибуты и подпрограммы

Любой абстрактный тип данных и `POINT` в частности характеризуется набором функций, описывающих операции применимые к экземплярам АТД. В классе, реализующем АТД, функции становятся **компонентами** (features) - операциями, применимыми к экземплярам класса.

В [лекции 6](#) было показано, что в АТД существуют функции трех видов: запросы (queries), команды (commands) и конструкторы (creators). Для компонентов классов необходима дополнительная классификация, основанная на том, каким образом реализован данный компонент - в пространстве или во времени (by space or by time). (См. "Категории функций", [лекция 6](#))

Пример координат точки отчетливо демонстрирует эту разницу. Для точек доступны два общепринятых представления - в декартовых или полярных координатах. Если для представления выбрана декартова система координат, то каждый экземпляр класса содержит два поля представляющих координаты x и y соответствующей точки:



(точка в декартовой системе координат)

Рис. 7.2. Представление точки в декартовых координатах

Если $p1$ является такой точкой, то получение значений x и y сводится просто к просмотру соответствующих полей данной структуры. Однако определение значений ρ и θ требует вычисления выражения $\sqrt{x^2 + y^2}$ для ρ и $\arctg(y/x)$ для θ (при условии ненулевого x).

Использование полярной системы координат ([рис. 7.3](#)) приводит к противоположной ситуации. Теперь ρ и θ доступны просто как значения полей, а определение x и y возможно после простых вычислений ($\rho \cos\theta, \rho \sin\theta$, соответственно).



(точка в полярной
системе координат)

Рис. 7.3. Представление точки в полярных координатах

Приведенный пример указывает на необходимость рассмотрения компонентов двух видов:

- Некоторые компоненты представлены в пространстве и, можно сказать, ассоциируются с некоторой частью информации каждого экземпляра класса. Они называются **атрибутами (attributes)**. Для точки, представленной в декартовых координатах, атрибутами являются x и y , а в полярных координатах в роли атрибутов выступают ρ и θ .
- Другие компоненты представлены во времени, и для доступа к ним требуется описать некоторые вычисления (алгоритмы), применимые далее ко всем экземплярам данного класса. В дальнейшем они называются **подпрограммами** или методами класса (**routines**). В декартовом представлении точек - ρ и θ это подпрограммы, а x и y выступают в качестве подпрограмм при использования полярных координат.

Вторая категория - подпрограммы - нуждается в дальнейшей дополнительной классификации. Часть подпрограмм возвращает результат, и их называют функциями (functions). В приведенном примере функциями являются x и y в представлении в полярных координатах, в то время как ρ и θ - функции в декартовых координатах, все они возвращают результат типа REAL. Подпрограммы, не возвращающие результат, соответствуют командам в спецификации АТД и называются процедурами (procedures). Например, класс POINT содержит процедуры translate, rotate и scale.

Не следует путать понятие "функция", обозначающее в классах программу, возвращающую результат, с использованным ранее толкованием функции как математического описания операций АТД. Эта досадная путаница понятий обусловлена устоявшейся терминологией в математике и программировании.

На [рис. 7.4](#) дана рассмотренная выше классификация, представленная в виде дерева:



Рис. 7.4. Классификация компонентов класса по их роли

Эта классификация является внешней, основанной на том, каким образом данный компонент выглядит для использующего его клиента.

Можно предложить другую, внутреннюю классификацию, использующую в качестве основного критерия способ реализации компонента в классе:

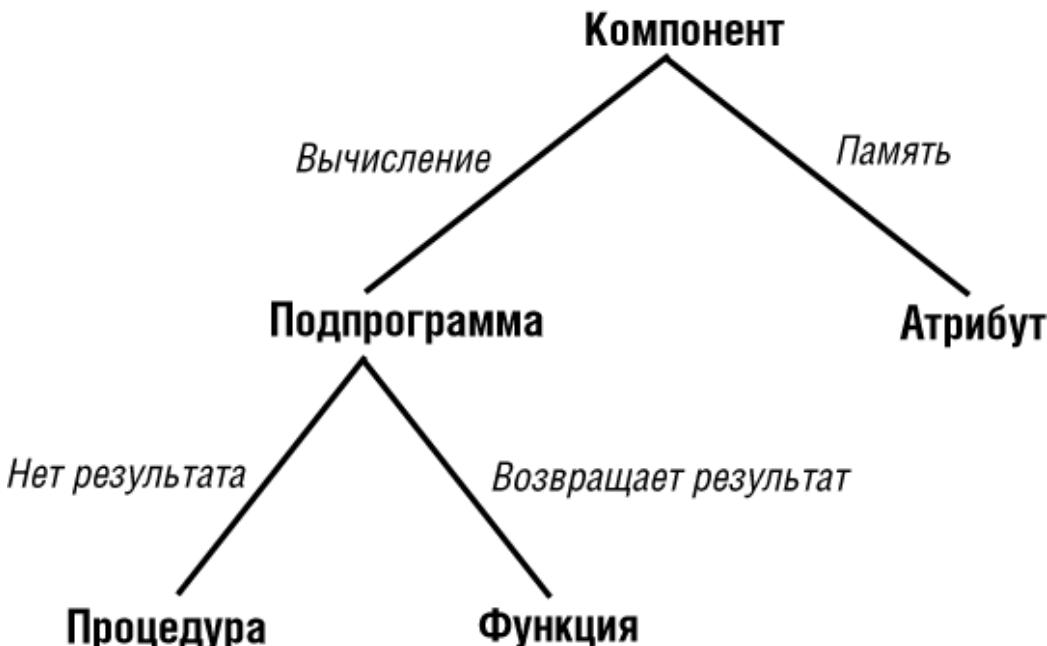


Рис. 7.5. Классификация компонентов класса по способу реализации

Унифицированный доступ

На первый взгляд один из аспектов приведенной выше классификации может вызывать беспокойство. Во многих случаях необходимо иметь возможность работать с объектом, например с точкой p_1 , не заботясь о том, какое внутреннее представление используется для p_1 - декартово, полярное или иное. Необходимо ли для этого отличать атрибуты от функций?

Ответ зависит от того, с какой точки зрения рассматривать данную проблему - разработчика, автора данного класса POINT или клиента, создавшего класс, использующий POINT. Для разработчика разница между атрибутами и функциями принципиально важна и имеет смысл. Ему необходимо принимать решения о том, какие компоненты будут реализованы как данные в памяти и какие будут доступны в результате вычислений. Но, заставлять клиента осознавать эту разницу, было бы серьезной ошибкой. Клиент должен обращаться к значениям x или r для точки p_1 , не заботясь и не имея информации о том, как реализованы соответствующие запросы.

Решение проблемы дает **принцип унифицированного доступа (Uniform Access principle)**, введенный в дискуссии о модульности ([лекция 3](#)). Принцип декларирует, что клиент должен иметь возможность доступа к свойствам объекта, используя одинаковую нотацию, вне зависимости от того, как это свойство реализовано - в памяти или как результат вычислений (в пространстве или во времени, в виде атрибута или подпрограммы). Этому важному принципу необходимо следовать при разработке нотации для обращения к компонентам класса. Так выражение, обозначающее значение компонента x объекта p_1 будет всегда записываться в виде:

$p_1.x$

вне зависимости от того, осуществляется ли доступ к полю данных объекта или выполняется подпрограмма.

При использовании такой нотации неопределенность может возникать только для запросов без аргументов, которые могут быть реализованы и как функции и как атрибуты. Команда должна быть процедурой, запрос с аргументами должен быть функцией, так как атрибуты не могут иметь аргументов.

Принцип унифицированного доступа необходим для гарантирования автономности компонентов ПО. Он защищает право создателя класса свободно экспериментировать с различными способами реализации, не создавая помех клиентам. (См. "Использование утверждений для документирования: краткая форма класса", [лекция 11](#))

Pascal, C и Ada нарушают этот принцип, предоставляя различную нотацию для вызова функций и для доступа к атрибутам. Это объяснимо для таких не ОО-языков (хотя еще в 1966 г. синтаксис Algol W предшественника Pascal удовлетворял этому принципу). Более новые языки, такие как C++ и Java, также не следуют этому принципу. Отход от этого принципа может служить причиной того, что изменения внесенные во внутренние представления (например переход от полярной системы координат к декартовой или иные) повлекут за собой неработоспособность многих клиентских классов. Это является одной из причин нестабильности программных разработок.

Принцип унифицированного доступа является источником определенных требований и к подготовке документации. Последовательное применение этого принципа должно гарантировать, например, что в официальной документации не содержится сведений о том, является ли данный запрос без аргументов функцией или атрибутом. Это одно из требований к стандартной методике документирования классов, известной как краткая форма класса.

Класс POINT

Ниже приведена версия исходного текста класса POINT. Фрагменты, начинающиеся с двух тире "--", представляют собой комментарии, продолжающиеся до конца строки. Комментарии содержат пояснения, облегчающие понимание текста, и не влияют на семантику класса.

```
indexing
  description: "Точка на плоскости"
class POINT feature
  x, y: REAL
    -- Абсцисса и ордината
  rho: REAL is
    -- Расстояние до начала координат (0, 0)
  do
    Result := sqrt (x^2 + y^2)
  end
  theta: REAL is
    -- Полярный угол
  do
    -- Предлагается реализовать в качестве упражнения (упр. У7.3)
  end
  distance (p: POINT): REAL is
    -- Расстояние до точки p
  do
    Result := sqrt ((x - p.x)^2 + (y - p.y)^2)
  end
  translate (a, b: REAL) is
    -- Перемещение на a по горизонтали, b по вертикали
  do
    x := x + a
    y := y + b
  end
  scale (factor: REAL) is
    -- Изменение расстояния до начала координат в factor раз
  do
    x := factor * x
    y := factor * y
  end
  rotate (p: POINT; angle: REAL) is
    -- Поворот вокруг p на угол angle
  do
    -- Предлагается реализовать в качестве упражнения (упр. У7.3)
  end
end
```

Некоторые аспекты приведенного текста неочевидны и требуют дополнительных разъяснений.

Класс в основном состоит из предложения, перечисляющего различные компоненты, вводимого ключевым словом **feature**. Кроме того, присутствует предложение **indexing** дающее общее описание (**description**), полезное для понимания функциональности класса, но никак не влияющее на семантику исполнения. Позднее будут рассмотрены три дополнительных предложения: **inherit** - для наследования; **creation** - при необходимости использования специального конструктора; **invariant** - для объявления инвариантов класса. Будет рассмотрена также возможность включения в класс двух или более предложений **feature**.

Основные соглашения

Класс POINT демонстрирует ряд приемов, которые будут использованы в последующих примерах. Необходимо оговорить основные соглашения.

Распознавание вида компонент

Компоненты x и y объявлены как относящиеся к типу REAL без ассоциированного алгоритма, следовательно, они являются атрибутами. Все остальные компоненты содержат конструкции вида

```
is
  do
    ... Инструкции ...
  end
```

которые описывают алгоритм, что является признаком подпрограмм. Подпрограммы rho, theta и distance возвращают результат типа REAL во всех трех случаях, что отражено в объявлениях вида

```
rho: REAL is ...
```

Это определяет их как функции. Две другие подпрограммы, `translate` и `scale`, не возвращают результата (объявление не завершается конструкцией: `T`, где `T` некоторый тип) и, соответственно, являются процедурами.

Поскольку `x` и `y` являются атрибутами, а `rho` и `theta` функциями, данный конкретный класс использует для представления точки декартову систему координат.

Тело подпрограммы и комментарии к заголовку

Тело подпрограммы (предложение `do`) представляет собой последовательность инструкций. Можно разделять последовательные инструкции и объявления точкой с запятой в традициях Algol-Pascal, но это не обязательно. Далее с целью упрощения точка с запятой будет опускаться между элементами на отдельных строках, но всегда будет использоваться как разделитель нескольких инструкций или объявлений в одной строке. (См. "Война вокруг точек с запятой", [лекция 8](#) курса "Основы объектно-ориентированного проектирования")

В подпрограммах класса `POINT` все инструкции являются присваиваниями значений. В данной нотации для обозначения присваивания используется символ `:=` также следуя соглашениям, принятым в Algol и Pascal. Этот символ нельзя перепутать с символом равенства `=`, применяемым, как и в математике, в операциях сравнения.

Другое соглашение о нотации касается использования комментария к заголовку подпрограммы. Уже отмечалось, что комментарии начинаются с двух последовательных тире `--`. Они могут размещаться в любом месте, где, по мнению автора, дополнительные разъяснения могут принести пользу. Особую роль играет **комментарий к заголовку (header comment)**. В соответствии с общим стилем правилом он должен помещаться в начале каждой подпрограммы после ключевого слова `is` с отступом как в примере класса `POINT`. Комментарий к заголовку должен кратко отражать назначение подпрограммы.

Атрибуты также сопровождаются комментариями, следующими непосредственно за их объявлением и имеющими тот же отступ, что и комментарии к заголовку подпрограмм. Иллюстрацией могут служить объявления `x` и `y`.

Предложение `indexing`

В начале нашего класса помещено предложение, начинающееся с ключевого слова `indexing`. Оно содержит единственный пункт, помеченный как `description`. Предложение `indexing` не оказывает влияния на выполнение программ и служит для размещения информации, ассоциированной с классом. В общем случае оно содержит лишь один пункт вида

```
index_word: index_value, index_value, ...
```

где `index_word` - произвольный идентификатор (элемент индексирования), а каждое значение `index_value` - произвольный элемент языка (идентификатор, целое число, строка и т.д.) (См. "Заметки об indexing", [лекция 4](#)).

Это дает два преимущества:

- Читатели исходного текста получают сводку свойств класса без необходимости рассмотрения деталей.
- В средах разработки с поддержкой повторного использования кода соответствующие инструментальные средства (часто называемые браузерами, навигаторами кода, инспекторами кода и т.д.) могут использовать информацию из данного раздела, помогая потенциальным пользователям найти нужные им классы. Эти средства обычно позволяют вести поиск по заданному шаблону среди элементов индексирования и их значений `index_value`. (В [лекции 18](#) курса "Основы объектно-ориентированного проектирования" рассмотрен базовый механизм ОО-браузеров.)

Приведенный пример содержит единственный индексный элемент - `description`, значение которого - строка, описывающая назначение класса. Все примеры классов в данной книге будут также содержать элемент `description`. Настоятельно рекомендуется следовать этому примеру и начинать исходный текст любого класса с предложения `indexing`, дающего краткую характеристику класса по аналогии с тем, как каждая подпрограмма начинается с комментария к заголовку.

Предложения `indexing` и комментарии к заголовку являются иллюстрацией правильного применения принципа **самодокументирования (Self-Documentation principle)**: везде, где это возможно, документация модуля должна размещаться непосредственно в самом модуле. (См. "Самодокументирование", [лекция 3](#))

Обозначение результата функции

Для понимания текстов функций `rho`, `theta` и `distance` в классе `POINT` необходимо еще одно соглашение.

Любой язык программирования, поддерживающий функции (подпрограммы, возвращающие результат) должен предусматривать нотацию, позволяющую установить в теле функции значение, возвращаемое в результате ее вызова. В качестве значения, возвращаемого функцией, в данной книге будет использоваться предопределенная сущность (`entity`) `Result`. (Полное определение сущности будет дано в конце этой лекции.)

Например, тело функции `rho` содержит следующее присваивание

```
Result := sqrt (x^2 + y^2)
```

`Result` - зарезервированное слово, которое может присутствовать только в теле функций. В функции, возвращающей результат типа `T`, `Result` рассматривается наряду с другими сущностями и ему может быть присвоено значение с помощью инструкций присваивания, как это показано выше.

При любом вызове функции в качестве результата будет возвращаться последнее присвоенное `Result` значение. Оно всегда определено благодаря правилам языка (они будут детально рассмотрены позже), требующим обязательной инициализации `Result` в начале каждой подпрограммы путем присваивания значения, предопределенного типом `T`. Для типа данных `REAL` инициализирующее значение равно нулю и следующая функция:

```
non_negative_value (x: REAL): REAL is
    -- Возвращает значение аргумента при x>0; ноль при x<=0
    do
        if x > 0.0 then
            Result := x
        end
    end
```

будет всегда возвращать вполне определенное значение (как указано в комментарии к заголовку), несмотря на то, что условная инструкция не содержит части `else`.

Дискуссия в конце данной лекции обсуждает логику использования соглашения `Result` в сопоставлении с другими приемами, такими как инструкции возврата. Хотя это соглашение касается всех языков программирования, оно является особенно важным при ОО-подходе.

Правила стиля

Исходные тексты классов в данной книге строго подчиняются основным правилам стиля. Они регламентируют отступы, шрифты, выбор имен классов и их компонент, использование нижнего и верхнего регистров.

Далее этим правилам будет уделяться серьезное внимание, а их подробному обсуждению полностью посвящена [лекция 8](#) курса "Основы объектно-ориентированного проектирования". Правила стиля не следует рассматривать как "косметическое" средство. Разработка качественного ПО требует последовательности и внимания ко всем деталям, - к форме в той же степени, что и к содержанию. Задача повторного использования делает соблюдение этих правил еще более важным, поскольку предполагается, что исходные тексты ждет долгая жизнь, в течение которой многие люди будут в них разбираться и развивать их.

Следует правильно применять правила стиля с самого начала написания исходного текста класса. Так, никогда не следует начинать подпрограмму, не задав комментарий к заголовку. Это не займет много времени и это время нельзя считать потерянным. Фактически достигается существенная экономия времени при дальнейшей работе с этим классом его автором или другими программистами, возможно, через полчаса, скорее, через пять лет. Использование одинаковых отступов, грамотное написание комментариев и выбор идентификаторов, применение адекватных лексических соглашений (пробел перед каждой открывающей скобкой, но не после нее и т. д.) не слишком усложнят задачу, но сделают более совершенным результат многомесячного труда над громадой исходных текстов. Внимание к деталям, безусловно, не достаточное, но необходимое условие разработки качественного ПО.

Элементарные правила стиля совершенно понятны из приведенного примера класса. Поскольку целью настоящего раздела является изучение базовых механизмов объектной технологии, то детальному описанию правил стиля будет посвящена одна из последующих лекций ([лекция 8](#) курса "Основы объектно-ориентированного проектирования").

Наследование функциональных возможностей общего характера

Другим аспектом класса `POINT`, требующим разъяснений, является присутствие в функциях `rho` и `distance` вызовов функции `sqr t`. Понятно, что эта функция возвращает квадратный корень действительного числа, но откуда она появилась?

Поскольку загромождать универсальный язык специализированными арифметическими операциями нецелесообразно, наилучшим решением будет определение подобных операций как компонентов некоторого специализированного класса, который называется, например, `ARITHMETIC`. Далее любой класс, в котором необходимо использовать указанные возможности, нужно просто объявить потомком этого специализированного класса. Для этого достаточно переписать класс `POINT` следующим образом

```
class POINT inherit
    ARITHMETIC
feature
    ... Остальная часть кода без изменений ...
end
```

Эта методика наследования функциональных возможностей общего характера является до некоторой степени спорной. Кто-то может полагать, что принципы ОО-подразумевают включение функций типа `sqr t` в качестве компонентов класса, которому принадлежит объект, например, `REAL`. Однако существует ряд операций с действительными числами, не все из которых стоит включать в данный класс. В дискуссии о принципах дизайна мы

вернемся к вопросу о полезности "вспомогательных" классов, таких как ARITHMETIC. (См. "Наследование функциональных возможностей", [лекция 6](#) курса "Основы объектно-ориентированного проектирования".)

Объектно-ориентированный стиль вычислений

Обратимся теперь к фундаментальным свойствам класса POINT и попытаемся понять, как устроено типичное тело подпрограммы и составляющие его инструкции. Далее выясним, каким образом класс и его компоненты могут использоваться другими классами - клиентами данного.

Текущий экземпляр

Обратимся опять к тексту одной из подпрограмм, процедуре translate:

```
translate (a, b: REAL) is
    -- Перемещение на a по горизонтали, b по вертикали
    do
        x := x + a
        y := y + b
    end
```

На первый взгляд этот текст совершенно понятен - для перемещения точки на расстояние a по горизонтали и b по вертикали значение a прибавляется к x, а b к y. При более внимательном рассмотрении все становится не столь очевидным. Из приведенного текста непонятно, о какой точке идет речь. Какому объекту принадлежат x и y, к которым прибавляются a и b? Этот вопрос связан с одним из наиболее характерных аспектов ОО-стиля разработки. Прежде чем получить ответ, следует разобраться в некоторых промежуточных деталях.

Текст класса описывает свойства и поведение объектов определенного типа, в данном случае точек. Это достигается путем описания свойств и поведения типичного экземпляра такого типа. Можно было бы назвать этот экземпляр "точкой на улице" по примеру того, как газеты представляют мнение "человека с улицы". Мы будем использовать более формальное имя - **текущий экземпляр класса**.

Иногда возникает необходимость явного обращения к текущему экземпляру. Зарезервированное слово

Current

обеспечивает эту возможность. В тексте класса Current обозначает текущий экземпляр этого класса. Потребность в использовании Current может возникнуть, если попытаться переписать функцию distance таким образом, чтобы осуществлялась проверка, не совпадает ли аргумент p с текущей точкой; в этом случае результат равнялся бы нулю без последующих вычислений. Эта версия distance будет выглядеть следующим образом:

```
distance (p: POINT): REAL is
    -- Расстояние до точки p
    do
        if p /= Current then
            Result := sqrt ((x - p.x)^2 + (y - p.y)^2)
        end
    end
```

Здесь /= операция неравенства. В соответствии с сформулированным ранее правилом инициализации условная инструкция не нуждается в части else, поскольку результат равен нулю при p = Current.

Тем не менее, в большинстве случаев текущий экземпляр подразумевается, и нет необходимости обращаться к Current по имени. Так ссылка на x в теле translate и других подпрограмм обозначает "значение x текущего экземпляра" без дополнительного уточнения.

Конечно, по-прежнему остается загадкой, кто же он - "Current"? Ответ придет позже при изучении вызовов подпрограмм, пока же при рассмотрении текста достаточно полагать, что все операции можно рассматривать только относительно некоторого неявно определенного объекта - текущего экземпляра.

Клиенты и поставщики

Игнорируя ряд моментов, связанных с загадкой идентификации Current, можно считать выясненным, как определять простые классы. Теперь необходимо обсудить применение этих определений, - как они используются в других классах. При последовательном ОО-подходе каждый программный элемент является частью некоторого класса, поэтому использовать эти определения будут другие классы.

Существуют лишь две возможности использования класса, например, POINT. Первый способ - наследование будет детально рассмотрен позднее. Для реализации второй возможности необходимо создать класс, являющийся **клиентом (client)** класса POINT. (Наследованию посвящены лекции 14-16.)

Чтобы стать клиентом класса S, простейший и наиболее общий путь - объявить сущность типа S.

Определение: клиент, поставщик

Пусть **S** некоторый класс. Класс **C** называется **клиентом (client)** **S**, если содержит объявление сущности **a**: **S**. Класс **S** называется **поставщиком (supplier)** **C**.

В этом определении **a** может быть атрибутом или функцией класса **C**, или локальной сущностью, или аргументом подпрограммы в классе **C**.

Например, наличие в классе **POINT** объявлений **x**, **y**, **rho**, **theta** и **distance** делает этот класс клиентом класса **REAL**. Напротив, другие классы могут стать клиентами **POINT**. Например:

```
class GRAPHICS feature
    p1: POINT
    ...
    some_routine is
        -- Выполнение некоторых действий с p1.
        do
            ... Создание экземпляра POINT и присоединение его к p1 ...
            p1.translate (4.0, -1.5)      --**
        ...
    end
    ...
end
```

Перед выполнением инструкции помеченной "**--****" атрибут **p1** принимает значение, соответствующее конкретному экземпляру класса **POINT**. Предположим, что этот объект представляет точку, совпадающую с началом координат **x = 0**, **y = 0**:

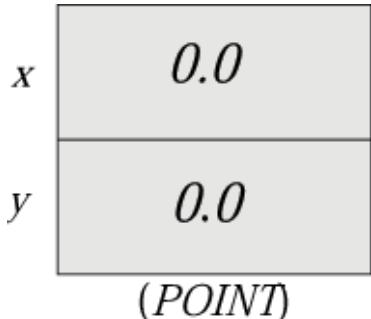


Рис. 7.6. Начало координат

В таких случаях говорят, что сущность **p1** **присоединена (attached)** к данному объекту (объект связан с сущностью). На данном этапе можно не беспокоиться о том, как был создан и инициализирован объект (строка "... Создание экземпляра POINT ..." до конца не раскрыта). В следующей лекции эти вопросы будут подробно обсуждаться как часть объектной модели. Пока достаточно знать, что объект существует и связан с сущностью **p1** (она присоединена к объекту).

Вызов компонента

Отмеченная звездочками инструкция

```
p1.translate (4.0, -1.5)
```

заслуживает внимательного изучения, поскольку представляет собой первый пример использования **базового механизма ОО-вычислений (basic mechanism of object-oriented computation)**. Это обращение к компоненту или вызов компонента (feature call). В процессе выполнения кода ОО-системы все вычисления реализуются путем вызова соответствующих компонентов определенных объектов.

Приведенный конкретный пример означает вызов компонента **translate** класса **POINT** применительно к объекту **p1** с аргументами **4.0** и **-1.5**, соответствующими **a** и **b** в объявлении **translate** в указанном классе. В общем случае допустимы две основные формы записи вызова компонента.

```
x.f
x.f (u, v, ...)
```

Здесь **x** называется **целью (target)** вызова и может быть сущностью или выражением, которые во время выполнения присоединены к конкретному объекту. Цель **x**, как любая сущность или выражение, имеет определенный тип, заданный классом **C**, следовательно, **f** должен быть одним из компонентов класса **C**. Точнее говоря, в первом случае **f** должен быть атрибутом или подпрограммой без аргументов, а во втором - подпрограммой с аргументами. Значения **u**, **v**, ... называются **фактическими аргументами (actual arguments)** вызова и они должны быть выражениями, число и тип которых должны в точности соответствовать числу и типу **формальных аргументов (formal arguments)** объявленных для **f** в классе **C**.

Кроме того, компонент *f* должен быть доступен (экспортирован) клиенту, содержащему данный вызов. Ограничению прав доступа посвящен следующий раздел (см. [лекция 7](#)), пока по умолчанию все компоненты доступны всем клиентам.

Результат рассмотренного выше вызова во время выполнения определяется следующим образом:

Эффект вызова компонента *f* для цели *x*

Применить компонент *f* к объекту, присоединенному к *x*, после инициализации всех формальных аргументов *f* (если таковые предусмотрены) значениями соответствующих фактических аргументов.

Принцип единственности цели

Чем так замечателен вызов компонента? В конце концов, каждый программист знает, как написать процедуру *translate*, которая перемещает точку на заданное расстояние. Традиционная форма вызова, доступная с незначительными вариациями во всех языках программирования, будет выглядеть следующим образом:

```
translate (p1, 4.0, -1.5)
```

В отличие от ОО-стиля в данном вызове все аргументы равноправны. Объектно-ориентированная форма не столь симметрична, определенный объект (в данном случае точка *p1*) выбирается в качестве цели, другим аргументам (действительные числа 4.0 и -1.5) отводится вспомогательная роль. Выбор единственного объекта в качестве цели для каждого вызова занимает центральное место в ОО-методе вычислений.

Принцип единственности цели

Каждая операция при ОО-вычислениях связана с определенным объектом - текущим экземпляром на момент выполнения операции

Этот аспект метода часто вызывает наибольшие затруднения у новичков. При разработке объектно-ориентированного ПО никогда не говорят: "Применение данной операции к этим объектам", но "Применение данной операции к **данному** объекту в **данный момент**". Если предусмотрены аргументы, то возможно такое дополнение: "Между прочим, я едва не забыл, вам необходимы здесь эти значения в качестве аргументов".

Слияние понятий модуль и тип

Принцип единственности цели является прямым следствием слияния понятий модуля и типа, рассмотренного ранее в качестве отправной точки ОО-декомпозиции. Поскольку каждый модуль является типом, каждая операция в данном модуле рассматривается относительно конкретного экземпляра данного типа (текущего экземпляра). Однако до сих пор детали этого слияния оставались немного загадочными. Как уже было сказано, класс одновременно представляет собой модуль и тип, но как согласовать синтаксическое понятие модуля (объединение родственных функциональных возможностей, формирование части программной системы) с семантическим понятием типа (статическое описание некоторых возможных объектов времени выполнения). Пример класса *POINT* дает определенный ответ:

Как функционирует слияние модуль-тип

Функциональные возможности класса *POINT*, рассматриваемого как модуль, в точности соответствуют операциям доступным для экземпляров класса *POINT*, рассматриваемого как тип

Эта идентификация операций экземпляров типа и служб (*services*), предоставляемых модулем, лежит в основе структурной дисциплины, называемой ОО-методом.

Роль объекта *Current*

Теперь настало время с помощью того же примера раскрыть тайну текущего экземпляра и выяснить, что он собой представляет в действительности.

Сама форма вызова показывает, почему текст подпрограммы (*translate* в классе *POINT*) не нуждается в дополнительной идентификации объекта *Current*. Поскольку любой вызов подпрограммы связан с определенной целью, которая явно обозначена при вызове, то при выполнении вызова имя каждого компонента в тексте подпрограммы (например, *x* в тексте *translate*) будет присоединено к той же цели. Таким образом, при выполнении вызова

```
p1.translate (4.0, -1.5)
```

каждое вхождение *x* в тело *translate*, как в следующей инструкции

```
x := x + a
```

означает: "x объекта *p1*".

Из этих соображений следует точный смысл понятия *Current*, как цели текущего вызова. Так в течение всего

времени выполнения приведенного выше вызова `Current` будет обозначать объект, присоединенный к `p1`. При другом вызове `Current` будет обозначать цель нового вызова. Можно сформулировать следующий **принцип вызова компонент (Feature Call principle)**:

Принцип вызова компонента

- (F1) Любой элемент программы может выполняться только как часть вызова подпрограммы.
- (F2) Каждый вызов имеет цель.

Квалифицированные и неквалифицированные вызовы

Выше было отмечено, что ОО-вычисления основаны на вызове компонентов. Как следствие этого положения исходные тексты в действительности содержат гораздо больше вызовов, чем может показаться на первый взгляд. До сих пор рассматривались две формы вызовов:

```
x.f  
x.f (u, v, ...)
```

Подобные вызовы используют так называемую точечную нотацию и их называют **квалифицированными (qualified)**, так как точно указана цель вызова, идентификатор которой расположен перед точкой.

Однако другие вызовы могут быть неквалифицированы, поскольку их цель не указана. В качестве примера предположим, что необходимо в класс `POINT` добавить процедуру `transform`, которая будет комбинацией процедур `translate` и `scale` точки. Текст такой процедуры может обращаться к процедурам `translate` и `scale`:

```
transform (a, b, factor: REAL) is  
-- Сместиться на a по горизонтали, на b по вертикали,  
-- затем изменить расстояние до начала координат в factor раз.  
do  
    translate (a, b)  
    scale (factor)  
end
```

Тело процедуры содержит вызовы `translate` и `scale`. В отличие от предыдущих примеров здесь не указана точная цель и не применяется точечная нотация. Такие вызовы называют **неквалифицированными (unqualified)**.

Неквалифицированные вызовы не нарушают пункта F2 принципа вызова компонент, так как тоже имеют цель. В данном случае целью является текущий экземпляр. Когда процедура `transform` вызывается по отношению к определенной цели, вызовы `translate` и `scale` имеют ту же цель. Фактически приведенный выше код эквивалентен следующему

```
do  
    Current.translate (a, b)  
    Current.scale (factor)
```

Можно переписать любой вызов как квалифицированный, указав `Current` в качестве цели (строго говоря, это справедливо только для экспорттированных компонент). Форма неквалифицированного вызова конечно проще и вполне понятна.

Приведенные неквалифицированные вызовы являются вызовами процедур. Аналогичные соображения можно распространить и на атрибуты, хотя наличие вызовов в этом случае возможно менее очевидно. Ранее было отмечено, что в теле процедуры `translate` присутствие `x` в выражении `x + a` означает поле `x` текущего экземпляра. Можно истолковать это иначе - как вызов компонента `x` и выражение в полной форме примет вид `Current.x+a`.

В общем случае любые инструкции или выражения вида:

```
f
```

или:

```
f (u, v, ...)
```

фактически являются неквалифицированными вызовами и могут быть переписаны в форме квалифицированных вызовов:

```
Current.f  
Current.f (u, v, ...)
```

хотя неквалифицированная форма является более удобной. Если подобная нотация используется как инструкция, то `f` представляет процедуру (без параметров в первом случае или с соответствующим числом параметров определенного типа - во втором). В выражениях `f` может быть функцией или атрибутом (в первом варианте записи).

Компоненты-операции

Рассмотрение выражения:

x + a

приводит к важному понятию **компоненты-операции (operator feature)**. Это понятие может восприниматься как чисто косметическое, имеющее только синтаксическую значимость, и реально не вносящее ничего нового в ОО-метод. Но именно такие синтаксические свойства способны существенно облегчить жизнь разработчика, если они существуют, и сделать ее убогой, если их нет. Компоненты-операции являются хорошим примером успешного использования ОО-парадигмы в давно известных областях.

Для реализации этой идеи нужно догадаться, что выражение x + a содержит не один вызов (компоненты x), а два. В вычислениях, не использующих объектный подход, + рассматривается как операция сложения двух значений x и a типа REAL. Как уже отмечалось, в чистой ОО-модели единственным механизмом вычислений является вызов компонентов. Следовательно, можно считать, по крайней мере теоретически, что и сложение является вызовом соответствующего компонента.

Для лучшего понимания необходимо обсудить определение типа REAL. Сформулированное ранее объектное правило ([лекция 7](#)) подразумевает, что каждый тип основан на каком-то классе. Это в равной мере относится к предопределенным классам, аналогичным REAL, и к классам, определенным разработчиком, таким как POINT. Предположим, что необходимо описать REAL как класс. Нетрудно определить набор существенных компонентов: арифметические операции (сложение, вычитание, изменение знака...), операции сравнения (меньше чем, больше чем...). Итак, первый набросок будет выглядеть так:

```
indexing
  description: "Действительные числа (не окончательная версия !)"
class REAL feature
  plus (other: REAL): REAL is
    -- Сумма текущего значения и other
    do
      ...
    end
  minus (other: REAL) REAL is
    -- Разность между текущим значением и other
    do
      ...
    end
  negated: REAL is
    -- Текущее значение, взятое с противоположным знаком
    do
      ...
    end
  less_than (other: REAL): BOOLEAN is
    -- Текущее значение меньше чем other?
    do
      ...
    end
  ... Другие компоненты ...
end
```

При использовании такого описания класса уже нельзя более записывать арифметическое выражение в виде: x + a. Вместо этого надо использовать следующий вызов:

x.plus (a)

По аналогии, вместо привычного -x следует теперь писать x.negated.

Можно попытаться оправдать такой отход от привычной математической нотации стремлением к последовательной реализации ОО-модели и призвать в качестве примера Lisp для обоснования возможности отхода от стандартной нотации в сообществе разработчиков ПО. Но такой аргумент нельзя считать убедительным: использование Lisp было всегда весьма ограниченным. Отход от нотации, существующей уже много столетий и знакомой всем с начальной школы, чрезвычайно опасен. Тем более что в этой нотации нет ничего неправильного.

Простой синтаксический прием позволяет сохранить последовательность подхода (требование унификации вычислительного механизма, основанного на вызове компонент) и обеспечивает совместимость с традиционной нотацией. Достаточно рассматривать выражение вида

x + a

как вызов дополнительного компонента класса REAL. Для реализации такого подхода необходимо переписать компоненту plus таким образом, чтобы для ее вызовов использовать знак операции, а не точечную нотацию. Вот описание класса, реализующее эту цель:

```
indexing
```

```

description: "Real numbers"
class REAL feature
  infix "+" (other: REAL): REAL is
    -- Сумма текущего значения и other
    do
      ...
    end
  infix "-" (other: REAL) REAL is
    -- Разность между текущим значением и other
    do
      ...
    end
  prefix "-": REAL is
    -- Текущее значение, взятое с противоположным знаком
    do
      ...
    end
  infix "<" (other: REAL): BOOLEAN is
    -- Текущее значение меньше чем other?
    do
      ...
    end
  ... Other features ...
end

```

Введены два новых ключевых слова - **infix** и **prefix**. Единственное синтаксическое новшество заключается в том, что имена компонент не являются идентификаторами (такими как `distance` или `plus`), а записываются в одной из двух форм (В следующей лекции будет показано, как определить "развернутый класс". См. "Роль развернутых типов".)

```

infix "§"
prefix "§"

```

где § заменяется конкретным знаком операции (+, -, *, <, <= и др.). Компонент может иметь имя в инфиксной форме только если является функцией с одним аргументом, примерами могут служить `plus`, `minus` и `less_than` в первоначальной версии класса `REAL`. Префиксная форма может использоваться только для функций без аргументов или атрибутов.

Инфиксные и префиксные компоненты, называемые далее **компоненты-операции (operator features)**, используются аналогично **именованным компонентам (identifier features)**. Существуют лишь два синтаксических различия. Для имен компонентов-операций при их объявлении используются формы **infix "§"** или **prefix "§"**, а не идентификаторы. Вызов компонентов-операций в случае инфиксных компонент имеет вид:

`u § v`

для префиксных:

`§ u`

Компоненты-операции поддерживают только квалифицированные вызовы. Неквалифицированный вызов `plus (y)` в подпрограмме первой версии класса `REAL` во второй версии должен быть записан в виде `Current + y`. Для именованных компонентов аналогичная нотация `Current.plus (y)` допустима, но обычно не используется.

Кроме указанных отличий во всем остальном компоненты-операции полностью синтаксически эквивалентны именованным компонентам, в частности могут наследоваться обычным образом. Не только базовые классы аналогичные `REAL`, но и любые другие, могут использовать компоненты-операции, например для функции сложения двух векторов в классе `VECTOR` вполне допустимо использовать инфиксную компоненту "+".

Операции, используемые в компонентах-операциях, должны подчиняться следующим правилам. Знак операции - последовательность из одного или более отображаемых символов, не содержащая пробелов и переводов строки, причем первым символом может быть только один из ниже перечисленных:

`+ - a / < > = \ ^ @ # | &`

Ограничения, налагаемые на первый символ, облегчают распознавание инфиксных и префиксных операций.

Кроме того, для совместимости с традиционной нотацией для булевых выражений следующие ключевые слова используются для обозначения операций:

`not and or xor and then or else implies`

Базовые классы (`INTEGER` и другие) используют так называемые стандартные операции:

- префиксные: + - not

- инфиксные: + - a / < > <= >= // \ \ ^ and or xor and then or else implies .

Здесь // обозначает целочисленное деление, \ \ - остаток при целочисленном делении, ^ - операцию возведения в степень, xor - исключающее "или". В классе BOOLEAN **and then** и **or else** являются вариантами **and** и **or** (отличия обсуждаются далее), implies обозначает импликацию: выражение a implies b эквивалентно (not a) or else b .

Операции, не входящие в число "стандартных", называют свободными операциями. Приведем два примера свободных операций.

- Далее в классе ARRAY будет использован инфиксный компонент-операция "@" для функции, возвращающей указанный элемент массива. Обращение к i-ому элементу массива будет выглядеть как a @ i .
- В класс POINT вместо функции distance можно ввести компонент-операцию "| - |" и расстояние между точками p1 and p2 будет записываться в виде p1 | - | p2 , а не как p1.distance(p2) .

Все операции имеют фиксированный приоритет, стандартные операции имеют свой обычный приоритет, а все свободные операции обладают более высоким приоритетом.

Использование компонентов-операций позволяет использовать общепринятую нотацию для выражений и одновременно отвечает требованиям полной унификации системы типов. Реализация арифметических и булевых операций как компонентов класса INTEGER вовсе не должна быть причиной снижения производительности. Концептуально a + x является вызовом компонента, но хороший компилятор может создать в результате обработки такого вызова код не менее эффективный, чем компиляторы C, Pascal, Ada или других языков, в которых "+" это жестко зафиксированная языковая конструкция.

В большинстве случаев мы можем забыть о том, что использование операций в выражениях фактически является вызовом процедур, поскольку конечный эффект будет таким же, как и при традиционном подходе. В то же время приятно сознавать, что и в этом случае не допущено отхода от принципов ОО-подхода.

Селективный экспорт и скрытие информации

До сих пор все компоненты класса были доступны всем потенциальным клиентам. Это, безусловно, не всегда приемлемо, поскольку скрытие информации является важным элементом построения последовательной и гибкой архитектуры.

Рассмотрим способы скрытия компонент от всех или некоторых клиентов. Данный раздел содержит лишь введение в нотацию - подробному рассмотрению интерфейсов классов посвящена одна из последующих лекций ([лекция 5](#) курса "Основы объектно-ориентированного проектирования"). В примерах для простоты будут рассматриваться только именованные компоненты, однако все изложенные ниже соображения справедливы и для компонент-операций.

Неограниченный доступ

По умолчанию все компоненты доступны для всех клиентов. Для класса

```
class S1 feature
  f ...
  g ...
  ...
end
```

компоненты f, g, ... доступны всем клиентам S1. Это означает, что если в классе С объявлена сущность x класса S1, то вызов

```
x.f ...
```

является допустимым, если выполнены все другие условия корректности вызова f.

Ограничение доступа клиентам

Для ограничения доступа клиентов к некоторой компоненте h, будет использована возможность включения в объявление класса двух или более разделов **feature**. Объявление будет выглядеть следующим образом

```
class S2 feature
  f ...
  g ...
feature {A, B}
  h ...
  ...
end
```

Компоненты f и g по-прежнему доступны всем клиентам. Компонент h доступен только для классов А и В, а также их потомков (прямых или косвенных). Это означает, что для некоторого x типа S2 следующий вызов

x.h

является допустимым только в исходных текстах классов A, B или одного из их потомков.

В особом случае, когда необходимо скрыть компонент i от всех клиентов, можно объявить его экспортируемым пустому списку клиентов (Не рекомендуемый стиль (см. ниже S5).):

```
class S3 feature { }
    i ...
end
```

В этом случае любой вызов x.i(...) недопустим. Единственная возможность обращения к i - неквалифицированный вызов

```
i (...)
```

в тексте подпрограммы класса S3 или его потомков. Такой механизм обеспечивает полное скрытие информации.

Возможность полного скрытия компонента от клиентов доступна во многих ОО-языках, а вот механизм селективного ограничения доступа, проиллюстрированный на примере h, к сожалению, практически не поддерживается. Подобный более тонкий контроль доступа необходим достаточно часто. Вопрос о важности селективного экспорта обсуждается в дискуссии в конце лекции.

В примерах последующих лекций мы столкнемся с различными примерами селективного экспорта и рассмотрим его методологическую роль при разработке интерфейсов.

Стиль объявления скрытых компонент

Использованный выше стиль объявления скрытой компоненты i не слишком удачен. Это хорошо видно в следующем примере (Не рекомендуемый стиль (см. ниже S5).)

```
class S4 feature
    exported ...
feature {}
    secret ...
end
```

где secret является скрытой компонентой, а exported - общедоступной. Разница в написании **feature {}** с пустым списком в скобках и **feature** без всяких скобок едва заметна. Гораздо разумнее вместо пустого использовать список, содержащий единственный класс NONE (Рекомендуемый стиль.)

```
class S5 feature
    ... exported ...
feature {NONE}
    ... secret ...
end
```

Класс NONE является базовым библиотечным классом и обсуждается далее в связи с наследованием. По определению он не может иметь потомков и нельзя создать его экземпляр. Таким образом, компонент, экспортенный классу NONE, фактически является скрытым. Между объявлениями S4 и S5 нет принципиальной разницы, однако во втором случае исходный текст становится более понятным и удобочитаемым. Именно такой стиль объявления скрытых компонент будет использоваться далее в этой книге.

"Внутренний" экспорт

Рассмотрим объявление класса

```
indexing
    замечание: "Ошибка в объявлении (объяснение см. ниже)"
class S6 feature
    x: S6
        my_routine is do ... print (x.secret) ... end
feature {NONE}
    secret: INTEGER
end -- class S6
```

Наличие в объявлении класса атрибута x типа S6 и вызова x.secret делает его собственным клиентом. Но такой вызов недопустим, так как компонент secret скрыт от всех клиентов! Тот факт, что неавторизованным клиентом является сам класс S6, нечего не меняет - объявленный статус secret делает недопустимым любой вызов вида x.secret. Всякие исключения нарушают простоту сформулированного правила.

Есть простое решение: написать вместо **feature {NONE}** предложение **feature {S6}**, экспортируя компоненту самому себе и своим потомкам.

Необходимо отметить, что подобный прием необходим, только если в тексте класса присутствует квалифицированный вызов аналогичный `print (x.secret)`. Очевидно, что неквалифицированный вызов `secret` в инструкции `print (secret)` допустим без дополнительных ухищрений. Все компоненты, объявленные в данном классе, могут использоваться в подпрограммах данного класса и его потомков. Только при наличии квалифицированных вызовов приходится экспортировать компонент самому себе.

Собираем все вместе

После введения в базовые механизмы ОО-вычислений настало время ответить на вопрос, каким образом можно построить исполняемую систему на основе отдельных классов.

Общая относительность

Удивительно, но все приведенные до сих пор описания того, что происходит во время выполнения, были относительными. Результат выполнения подпрограммы всегда связан с текущим экземпляром, который в исходном тексте класса неизвестен. Можно попытаться понять действие вызова, только принимая во внимание цель этого вызова, например `p1` в следующем примере:

```
p1.translate (u, v)
```

Однако возникает следующий вопрос: что в действительности обозначает `p1`? Ответ опять относителен. Предположим, приведенный вызов присутствует в тексте некоторого класса GRAPHICS, а `p1` это атрибут GRAPHICS. Тогда очевидно, что в этом случае `p1` фактически означает `Current.p1`. Но это не ответ на поставленный вопрос, так как неизвестно, что представляет собой объект `Current` в момент вызова! Другими словами, теперь необходимо установить клиента,зывающего подпрограмму класса GRAPHICS, в которой используется наш вызов.

Большой Взрыв

Рассмотрим произвольный вызов. Понимание смысла, происходящего в процессе произвольного вызова, позволит полностью разобраться в механизме ОО-вычислений. Используем сформулированный ранее принцип вызова компонентов:

- (F1) Любой элемент программы может выполняться только как часть вызова подпрограммы.
- (F2) Каждый вызов имеет цель.

Любой вызов может принимать одну из следующих форм:

- неквалифицированная: `f (a, b, ...)`;
- квалифицированная: `x.g (u, v, ...)`.

Аргументы в обоих случаях могут отсутствовать. Вызов размещен в теле подпрограммы `g` и может выполняться только как часть вызова `g`. Предположим, что известна цель этого вызова - некий объект ОВJ. Тогда можно легко установить цель этого вызова - `t`. Возможны четыре варианта, первый из которых относится к неквалифицированному вызову, а остальные - к квалифицированному:

- (T1) Для неквалифицированного вызова `t` это просто ОВJ.
- (T2) Если `x` это атрибут, то `x` - поле объекта ОВJ- имеет значение, которое, в свою очередь, присоединено к некоторому объекту - он и есть `t`.
- (T3) Если `x` - функция, то необходимо сначала осуществить ее вызов (неквалифицированный), результат которого и дает `t`.
- (T4) Если `x` - локальная сущность `r`, то к моменту вызова предыдущие инструкции вычислят значение `x`, присоединенное к определенному объекту, который и является объектом `t`.

Проблема в том, что все четыре ответа опять относительны и могут помочь только в том случае, если известно, чем является текущий экземпляр ОВJ. Очевидно, что ОВJ это цель текущего вызова! Ситуация как в песенке о том, как у попа была собака (в оригинале: котенок съел козленка, котенка укусил щенок, щенка ударила палка ...) - бесконечная цепь.

Для приведения относительных ответов к абсолютным необходимо выяснить, что происходит тогда, когда все только начинается - в момент Большого Взрыва. Итак, определение:

Определение: выполнение системы

Выполнение ОО-программной системы состоит из следующих двух шагов:

- Создание определенного объекта, называемого **корневым объектом выполнения**.
- Применение определенной процедуры, называемой **процедурой создания**, к данному объекту.

В момент Большого Взрыва создается объект и начинается выполнение процедуры создания. Корневой объект является экземпляром **корневого класса** системы, а процедура создания - одной из процедур этого класса. Выполнение системы в целом сводится к успешному развертыванию отдельных частей (прямо или косвенно

зажженных от начальной искры) в гигантский комплексный фейерверк.

Зная, где все началось, несложно проследить судьбу Current в процессе этой цепной реакции. Первым текущим объектом, созданным в момент Большого Взрыва, является корневой объект. Рассмотрим далее некоторый этап выполнения системы. Пусть r - последняя вызванная подпрограмма, а текущим на момент вызова r был объект ОВJ. Тогда во время выполнения r объект Current определяется следующим образом:

- (C1) Если в r выполняется инструкция, не являющаяся вызовом подпрограммы (например, присваивание), то текущим остается прежний объект.
- (C2) Неквалифицированный вызов также оставляет тот же объект текущим.
- (C3) Запуск квалифицированного вызова x.f . . . делает текущим объект, присоединенный к x. Зная объект ОВJ, можно идентифицировать x, используя сформулированные ранее правила T1-T4. После завершения вызова роль текущего возвращается к объекту ОВJ.

В случаях С2 и С3 вызов может в свою очередь содержать последующие квалифицированные или неквалифицированные вызовы, и данные правила нужно применять рекурсивно.

Итак, нет ничего загадочного и запутанного в определении цели любого вызова, несмотря на всю относительность и рекурсивность правил. Что действительно является удивительным, так это мощь компьютеров, которую мы используем, выступая в роли учеников чародея. Мы создаем относительно небольшой текст заклинания - ПО, и затем выполняем его, в результате чего создаются объекты и выполняются вычисления, и число этих объектов и вычислений столь велико, что кажется почти бесконечным по меркам человеческого сознания.

Системы

Эта лекция акцентирует внимание на классах - элементах конструкции ОО-ПО. Для получения исполняемого кода классы необходимо скомпоновать в систему.

Определение системы вытекает из предшествующего обсуждения. Для построения системы необходимы три вещи:

- Создать совокупность классов CS, называемую **множеством классов (class set)** системы.
- Указать класс из CS, являющийся **корневым (root class)**.
- Указать в корневом классе процедуру, играющую роль **корневой процедуры создания (root creation procedure)**.

Для получения системы эти элементы должны удовлетворять критерию целостности. Каждый класс, прямо или косвенно необходимый корневому, должен быть частью множества CS. Это условие **замыкания системы (system closure)**.

Понятие необходимости следует уточнить, как это обычно делается при построении замыкания:

- Класс D **непосредственно необходим** классу C , если текст C ссылается на D. Здесь можно выделить два варианта: C может быть либо клиентом D, либо потомком D.
- Класс E **необходим** классу C, либо, когда C совпадает с E, либо существует класс D непосредственно необходимый классу C, и классу D необходим (возможно, рекурсивно) класс E. Другими словами, существует цепочка классов, связанных отношением непосредственной необходимости, и началом этой цепочки является класс C, а концом - класс E.

Теперь можно дать определение замкнутой системы.

Определение: замкнутая система

Система является замкнутой если и только если множество ее классов содержит все классы, необходимые корневому классу.

Специализированная программа, например компилятор, может обработать все классы замкнутой системы, начиная с корневого. Рекурсивное обращение к необходимым классам будет происходить по мере того, как встретится упоминание о них. В результате будет сформирован исполняемый код, соответствующий системе в целом.

Этот процесс называется **компоновкой** или **сборкой (assembly)** системы и является завершающим этапом разработки.

Программа main отсутствует

Неоднократно подчеркивалось, что системы, разработанные с помощью ОО-подхода, не используют понятия основной программы. Не впускаем ли мы основную программу с черного хода, вводя определение корневого класса и корневой процедуры?

Не совсем. В традиционном понятии основной программы объединены две не связанные концепции:

- Место, с которого начинается выполнение.
- Вершина или фундаментальный компонент архитектуры системы.

Первое условие, безусловно, необходимо. Выполнение любой системы должно начинаться с вполне определенной позиции. В ОО-системах эта позиция определяется корневым классом и корневой процедурой. В случае параллельных, а не последовательных вычислений можно определить несколько начальных точек - по одной для каждой независимой нити или потока (Thread) вычислений.

Концепция вершины уже достаточно обсуждалась ранее и не требует дополнительных комментариев.

Нет никаких оснований для объединения столь разных понятий. Нельзя приписывать особую роль точке начала выполнения кода в архитектуре системы. Типичным примером может служить инициализация операционной системы, выполняемая процедурой загрузки. Этот небольшой и незначительный компонент безусловно нельзя считать центральным в архитектуре операционной системы. Объектная технология исходит из прямо противоположной предпосылки, считая, что важнейшими свойствами системы являются входящий в нее ансамбль классов, функциональные возможности этих классов и их взаимосвязь. В таком контексте выбор корневого класса играет второстепенную роль и при необходимости его можно легко изменить.

Ранее уже указывалось, что необходимо отказаться на раннем этапе разработки системы от вопроса, - "где основная программа?". Если строить архитектуру системы на основе ответа на этот вопрос, то нельзя обеспечить расширяемость и повторное использование кода. Другой подход - готовые к повторному использованию классы, реализации АТД. Программные системы в этом случае представляют собой перестраиваемые ансамбли таких компонент. (О критике функциональной декомпозиции см. "Функциональная декомпозиция", [лекция 5](#))

Не всегда конечной целью разработки является создание систем. Важным приложением метода является разработка библиотек классов для повторного использования. Библиотека это не система и она не имеет корневого класса. В процессе разработки библиотеки часто создают несколько систем, но такие системы используются только для отладки и не являются частью завершенной версии библиотеки. Окончательный продукт является набором классов, который другие разработчики будут использовать для разработки своих систем или своих библиотек.

Компоновка системы

Как практически реализовать процесс компоновки системы?

Допустим, что операционная система использует обычный способ хранения исходных текстов классов в файлах. Инструментальному средству компоновки (компилятор, интерпретатор) необходима следующая информация:

- (A1) Имя корневого класса.
- (A2) Генеральная совокупность (**universe**) файлов, содержащих тексты классов, необходимых корневому.

Эта информация не должна содержаться непосредственно в исходных текстах классов. Идентификация класса как корневого в его исходном тексте (A1) нарушает принцип отсутствия основной программы. Включение в исходные тексты классов информации о местонахождении соответствующих файлов означало бы жесткую привязку к файловой системе и, очевидно, является неприемлемым решением. Если размещение изменить, то использование таких классов становится невозможным.

Из этих рассуждений следует, что для сборки системы необходима информация, размещенная вне исходных текстов классов. Для обеспечения такой информацией будем использовать небольшой управляющий язык под названием Lace. Рассмотрим процесс сборки, но сразу отметим, что детали Lace совершенно несущественны в контексте ОО-подхода. Язык Lace просто конкретный пример управляющего языка, позволяющего сохранить автономность и возможность повторного использования классов, используя некий механизм для сборки файлов системы.

Рассмотрим типичный документ Lace, так называемый **файл Ace**:

```
system painting root
    GRAPHICS ("painting_application")
cluster
    base_library: "\ library\ base";
    graphical_library: "\ library\ graphics";
    painting_application: "\ user\ application"
end -- system painting
```

Предложение **cluster** определяет генеральную совокупность файлов, содержащих тексты классов. Оно содержит список кластеров. Кластер - это группа связанных классов, представляющих подсистему или библиотеку. (Модель кластеров обсуждается в [лекции 10](#) курса "Основы объектно-ориентированного проектирования")

Операционные системы, такие как Windows, VMS или Unix, содержат удобный механизм поддержки кластеров - подкаталоги. Их файловые системы имеют древовидную структуру. Конечные узлы дерева (листья), называемые "обычными файлами", содержат непосредственно информацию, а промежуточные узлы, подкаталоги, содержат наборы файлов, состоящие из обычных файлов и подкаталогов.

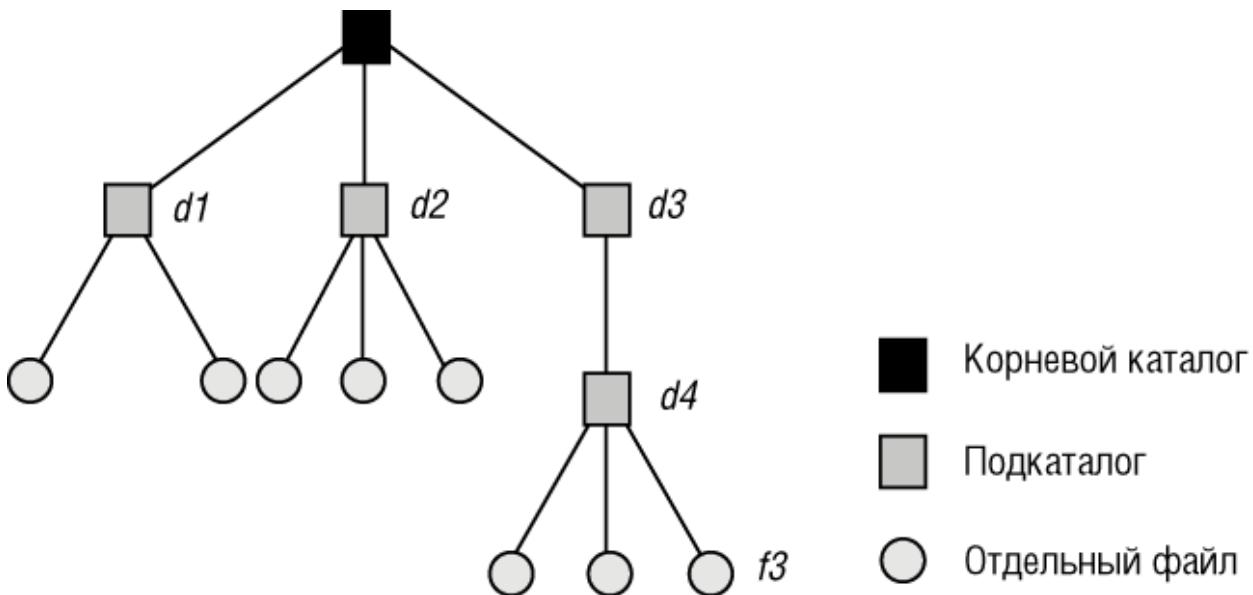


Рис. 7.7. Структура каталогов

Можно ассоциировать каждый кластер с подкаталогом. В Lace используется следующее соглашение: каждый кластер, например `base_library`, имеет связанный с ним подкаталог, имя которого дано в двойных апострофах - "`\library\ base`". Такое соглашение об именах файлов используется в Windows (`\dir1\dir2\ ...`) и здесь приведено только ради примера. Соответствующие имена Unix получаются заменой символов обратной косой черты на обычную.

Можно использовать иерархию подкаталогов для определения иерархии кластеров. Кроме того, Lace поддерживает понятие субкластера, что позволяет определить логическую структуру иерархии вложенных кластеров независимо от их физического положения в файловой системе.

Каталоги, перечисленные в предложении **cluster**, могут содержать файлы всех типов. Для работы с генеральной совокупностью процессу компоновки системы необходима информация о том, какие из файлов содержат тексты классов. Используем простое соглашение - текст некоторого класса с именем NAME размещается в файле `name.e` (нижний регистр). В этом случае, генеральная совокупность представляет собой набор файлов с именами вида `name.e` в каталогах, перечисленных в предложении **cluster**.

Предложение **root** Lace служит для задания корневого класса системы. В данном случае корневым является класс GRAPHICS и он находится в кластере painting_application. Если только один класс в генеральной совокупности называется GRAPHICS, то нет необходимости указывать кластер.

Предположим, что компилятор начинает создание системы, описанной в приведенном файле Ace. Далее предположим, что ни один из файлов системы еще не откомпилирован. Компилятор находит текст корневого класса GRAPHICS в файле `graphics.e` кластера `painting_application`, который размещается в каталоге `\user\application`. Анализируя текст класса GRAPHICS, компилятор находит имена классов, которые необходимы GRAPHICS и ведет поиск файлов с соответствующими именами в каталогах трех кластеров. Далее этот поиск повторяется до тех пор, пока не будут обнаружены все классы, прямо или косвенно необходимые корневому классу GRAPHICS.

Важнейшей особенностью этого процесса является возможность его автоматизации. Разработчику ПО не нужно составлять списки зависимых модулей, известных как "Make-файлы", или указывать в каждом файле имена файлов, необходимых для его компиляции ("директивы Include" в C и C++). Кроме своей утомительности процесс создания этой информации вручную является потенциальным источником ошибок. Единственное, что самостоятельно не сможет определить ни одна утилита - это имя корневого класса и размещение необходимых классов в файловой системе.

Для дальнейшего упрощения работы программиста хороший компилятор должен уметь создавать шаблоны файлов Ace, предложение **cluster** которых включает базовые библиотеки (ядро, фундаментальные структуры данных и алгоритмы, графика и т. д.) и указание на текущий каталог. В этом случае разработчику остается только указать имя системы и ее корневого класса без необходимости глубокого знания синтаксиса Lace.

Конечным продуктом процесса компиляции является исполняемый файл, имя которого совпадает с именем системы в файле Ace, в данном примере - `painting`.

Язык содержит ряд других простых конструкций, поддерживающих управление действиями инструментальных средств компоновки, в частности директив компилятора и уровней контроля утверждений. При дальнейшем изучении ОО-метода некоторые из них будут использованы. Уже отмечалось, что Lace поддерживает понятие логического субкластера и может использоваться для описания комплексных структур, включая подсистемы и многоуровневые библиотеки.

Использование независимого от языка разработки языка описания системы аналогичного Lace позволяют классам

оставаться системно независимыми. Классы являются компонентами ПО, аналогичными электронным микросхемам, и система собрана из конкретного набора классов подобно компьютеру, собранному из определенного набора микросхем.

Классическое "Hello"

Повторное использование замечательная вещь, но иногда надо решить очень простую задачу, например вывести строку. Интересно, как написать такую "программу". После введения понятия системы можно ответить и на этот животрепещущий вопрос.

Следующий маленький класс содержит процедуру, выводящую строку:

```
class SIMPLE creation
  make
feature
  make is
    -- Вывод строки.
    do
      print_line ("Hello Sarah!")
    end
end
```

Процедура `print_line` с параметром некоторого типа выводит значение соответствующего объекта, в данном случае строки. Другая процедура с именем `print` делает то же самое, но без перехода на новую строку. Обе процедуры доступны во всех классах и унаследованы от универсального предка `GENERAL`, обсуждаемого далее. (О классе `GENERAL` см. "Универсальные классы", [лекция 16](#))

Для получения системы, которая будет выводить данную строку необходимо сделать следующее:

- (E1) Поместить текст класса в файл `simple.e`.
- (E2) Запустить компилятор.
- (E3) Если файл Ace заранее не создан, то можно запросить автоматическое создание шаблона и в режиме его редактирования заполнить имя корневого класса - `SIMPLE`, системы - `my_first` и указать каталог кластера.
- (E4) После выхода из редактора компилятор осуществит компоновку системы и создаст исполняемый файл `my_first`.
- (E5) Выполнить `my_first`. В режиме командной строки необходимо просто ввести `my_first`. В системах с графическим интерфейсом появится новая пиктограмма с именем `my_first` и запуск программы производится двойным щелчком на ней.

В результате на консоли появится сообщение:

Hello Sarah!

Структура и порядок: программист в роли поджигателя

Общую картину процесса построения ПО ОО-методом мы уже знаем. Нам также известно, как восстановить цепочку событий, связанную с выполнением некоторой операции. Рассмотрим операцию:

```
[A]
x.g (u, v, ...)
```

присутствующую в тексте подпрограммы `g` класса `C` и предположим, что `x` это атрибут. Как и когда она будет выполняться? Класс `C` должен быть включен в систему, скомпонованную затем с помощью соответствующего файла Ace. Далее следует запустить выполнение системы, которое начнется с создания экземпляра корневого класса. Корневая процедура создания должна выполнить одну или более операций, которые прямо или косвенно создадут объект `C_OBJ` - экземпляр класса `C`, а затем выполнят вызов:

```
[B]
a.r (...)
```

где `a` присоединено к `C_OBJ`. Далее вызов [A] выполнит `g` с заданными аргументами, используя в качестве цели объект, присоединенный к полю `x` объекта `C_OBJ`.

Итак, теперь мы знаем, как восстановить точную последовательность событий, происходящих в процессе выполнения системы. Подразумевается, что мы видим систему целиком. Текст одного класса, естественно, не позволяет определить порядок, в котором клиенты будут вызывать его подпрограммы. В этом случае единственная доступная для обозрения последовательность событий это порядок, в котором выполняются инструкции в теле данной подпрограммы.

Даже на уровне системы структура настолько децентрализована, что задача точного определения порядка операций, безусловно разрешимая, практически оказывается очень сложной. Важно то, что это и не очень интересно. Необходимо помнить, что корневой класс является весьма поверхностным свойством системы. Это частный выбор,

сделанный уже после формирования набора классов. Всегда есть возможность достаточно просто изменить выбор корневого класса.

Этот уход от упорядочения является частью объектной технологии и стимулирует создание децентрализованной архитектуры систем. В центре внимания не "порядок выполнения программы", а функциональные возможности набора классов. "Порядок", в котором эти возможности будут реализованы в процессе выполнения конкретной системы, является вторичным свойством. (См. "Преждевременное упорядочение", [лекция 5](#))

Данные наблюдения позволяют рассматривать роль программиста как пиротехника или человека, разжигающего огромный костер. Он складывает дрова, следя за тем, чтобы все компоненты были готовы для компоновки и необходимые связи присутствовали. Далее он зажигает спичку и следит за огнем. Если структура правильно подготовлена, то нет необходимости стараться предсказать последовательность возгораний. Достаточно знать, что каждая часть, которая должна вспыхнуть, загорится и это произойдет не раньше положенного времени.

Обсуждение

В заключение данной лекции имеет смысл рассмотреть обоснования и альтернативы некоторых принятых решений, связанных с разработкой метода и нотации. Аналогичными разделами завершаются все лекции, в которых вводятся новые понятия.

Форма объявлений

Отточим наши критические навыки вначале на чем-либо не столь существенном. Поэтому начнем с синтаксиса. Рассмотрим нотацию, используемую при объявлении компонентов. В отличие от многих языков мы не использовали для подпрограмм ключевых слов **procedure** или **function**. Форма объявления компонента позволяет отличить, будет ли он атрибутом, процедурой или функцией. Любое объявление компонента всегда начинается с его имени:

```
f ...
```

Тем самым сохраняется возможность дальнейшего определения компонента любого типа. Если далее присутствует список параметров

```
g (a1: A; b1: B; ...) ...
```

то понятно, что g подпрограмма, которая может быть процедурой или функцией. Далее может следовать:

```
f: T ...
g (a1: A; b1: B; ...): T ...
```

В первом примере все еще есть выбор - f может быть либо атрибутом, либо функцией без аргументов. Во втором случае неопределенность заканчивается и g может быть только функцией. Для f неопределенность разрешается в зависимости от того, что следует за T. Если ничего, то f это атрибут, как в следующем примере:

```
my_file: FILE
```

Но если далее присутствует ключевое слово **is**, а за ним тело подпрограммы (**do** или варианты **once** и **external**, рассматриваемые позже), как в примере:

```
f: T is
  -- ...
  do ... end
```

то f - функция. Еще один вариант

```
f: T is some_value
```

определяет f как **атрибут-константу (constant attribute)**, значение которой равно some_value. (Атрибуты-константы обсуждаются в [лекции 18](#))

Такой синтаксис позволяет легко распознавать различные виды компонентов, подчеркивая в то же время их фундаментальные общности. Само понятие компонента, объединяющее подпрограммы и атрибуты лежит в русле принципа унифицированного доступа. Общность в объявлениях атрибутов основана на тех же принципах.

Атрибуты или функции?

Рассмотрим подробнее следствия принципа унифицированного доступа и объединения атрибутов и подпрограмм под общим заголовком - компоненты. (См. "Унифицированный доступ", [лекция 3](#). См. также данную лекцию.)

Принцип декларирует, что клиенты модуля обращаются ко всем его сервисам идентичным образом независимо от способа их реализации. В данном случае в роли сервисов выступают компоненты класса, и для клиентов имеет значение только доступность соответствующих компонентов, независимо от того, как они реализованы атрибутами или функциями.

Рассмотрим класс PERSON, содержащий компонент типа INTEGER без параметров. Если автор клиентского класса записывает выражение

Isabelle.age

то единственным важным будет то, что age возвращает целое число - значение возраста экземпляра PERSON, который во время выполнения присоединен к сущности Isabelle. Компонент age может быть как атрибутом, так и функцией, вычисляющей результат, используя значение атрибута birth_date и текущую дату. Автору клиентского класса нет необходимости знать, какое из этих решений выбрал автор PERSON.

Нотация для доступа к атрибуту идентична вызову подпрограммы, а нотации для объявления этих видов компонентов одинаковы настолько, насколько это концептуально возможно. Если в дальнейшем автор класса заменит реализацию функции на атрибут или наоборот, то это никак не отразится на клиентах данного класса.

Различие в точках зрения поставщика и клиента на атрибут представлено на [рис. 7.4](#) и [рис. 7.5](#), использованных для определения понятия компонента. [Рис. 7.5](#) иллюстрирует разницу между подпрограммами и атрибутами - это внутреннее представление с позиций реализации, используемое поставщиком. [Рис. 7.4](#) в качестве первичного критерия использует разницу между командами и запросами - это внешнее представление клиента.

Решение рассматривать атрибуты и функции без параметров как эквивалентные для клиентов имеет два важных следствия, рассматриваемые подробно в последующих лекциях:

- Первое следствие касается программной документации. Стандартная документация класса для клиента, известная как краткая форма класса, составляется так, чтобы отсутствовала разница в описаниях атрибутов и функций без параметров. (См. "Использование утверждений в документации: краткая форма класса", [лекция 11](#))
- Второе следствие связано с наследованием, как основным способом адаптации программных элементов к новым условиям без разрушения существующего ПО. Если некий класс содержит компонент, представляющий собой функцию без аргументов, то вполне допустимо в классах-потомках переопределить его как атрибут. (См. "Предопределение функции в качестве атрибута", [лекция 14](#))

Экспорт атрибутов

В завершение предшествующей дискуссии необходимо обсудить вопрос об экспорте атрибутов. Рассмотренный в этой лекции класс POINT имеет атрибуты x и y и экспортирует их клиентам, также как и функции rho и theta. Для получения значения атрибута некоторого объекта используется обычная нотация для вызова компонентов в виде my_point.x или my_point.theta.

Эта возможность экспорта атрибутов отличается от соглашений, принятых во многих ОО-языках. Типичным примером является Smalltalk, в котором только подпрограммы (методы) могут быть экспортаны классом, а прямой доступ к атрибутам (свойствам) запрещен.

Следуя подходу Smalltalk, доступ к атрибуту можно обеспечить только с помощью небольшой экспортанной функции, возвращающей значение атрибута. В примере класса POINT назовем атрибуты internal_x, internal_y и добавим функции abscissa и ordinate. Лаконичный синтаксис Smalltalk допускает присваивание одинаковых имен атрибуту и функции, избавляя от необходимости придумывать специальные имена для атрибутов.

```
class POINT feature
    -- Общедоступные компоненты:
    abscissa: REAL is
        -- Горизонтальная координата
        do Result := internal_x end
    ordinate: REAL is
        -- Вертикальная координата
        do Result := internal_y end
    ... Другие компоненты аналогичны предыдущей версии ...
feature {NONE}
    -- Компоненты недоступные клиентам:
    internal_x, internal_y: REAL
end
```

Этот подход имеет два недостатка:

- Он побуждает авторов классов писать много маленьких функций, аналогичных abscissa и ordinate. Несмотря на то, что такие функции будут очень короткими, автор класса будет тратить на их написание дополнительные усилия, а их присутствие затрудняет восприятие исходного текста.
- Существенное снижение производительности, так как каждое обращение к полю объекта требует вызова функции. Ничего удивительного в том, что объектная технология в некоторых кругах заработала репутацию неэффективной. Можно конечно разработать оптимизирующий компилятор, осуществляющий подстановку вместо вызова функций, но тогда какова роль таких функций?

Подход, обсуждаемый в данной лекции, представляется предпочтительным. Он избавляет от необходимости загромождать исходные тексты многочисленными крошечными функциями и предоставляет возможность экспорта, где это необходимо. Такая практика не мешает скрытию информации, а фактически является непосредственной

реализацией этого принципа, как и принципа унифицированного доступа.

Эта методика удовлетворяет требованиям унифицированного доступа (преимущество для клиентов), упрощает восприятие исходных текстов (преимущество для поставщиков) и повышает эффективность (преимущество для всех).

Доступ клиентов к атрибутам

Экспорт атрибута с использованием рассмотренной техники делает его доступным клиентам только для чтения в виде `my_point.x`. Модификация атрибута путем присваивания не разрешается. Следующая синтаксическая конструкция недопустима для атрибутов (**Внимание: недопустимая конструкция - только для иллюстрации.**):

```
my_point.x := 3.7
```

Действует простое правило. Если `attrib` является атрибутом, то `a.attrib` является выражением, а не сущностью. Следовательно, ему нельзя присвоить значение, как нельзя присвоить значение выражению `a + b`.

Возможность модификации `attrib` достигается добавлением экспортируемой процедуры вида:

```
set_attrib (v: G) is
    -- Установка значения attrib равным v.
    do
        attrib := v
    end
```

Вместо этого можно было бы представить следующий синтаксис для разграничения прав доступа пользователей (**Внимание: не поддерживаемая нотация. Только для обсуждения.**)

```
class C feature [AM]
    ...
feature [A]{D, E}
    ...
```

здесь А обозначает возможность чтения, а М - модификации. Это устранило бы потребность в частом написании процедур аналогичных `set_attrib`.

Помимо неоправданных дополнительных языковых сложностей такой подход не слишком гибок. Во многих случаях может потребоваться специфический способ модификации атрибута. Например, некоторый класс экспортирует счетчик, значения которого нельзя изменять произвольно, а только с шагом +1 или -1:

```
class COUNTING feature
    counter: INTEGER
    increment is
        -- Увеличение значения счетчика
        do
            counter := counter + 1
        end
    decrement is
        -- Уменьшение значения счетчика
        do
            counter := counter - 1
        end
    end
```

Аналогичным образом клиенты класса POINT не имеют возможности непосредственно изменять координаты точки `x` и `y`. Для этой цели служат экспортированные процедуры `translate` и `scale`.

При изучении утверждений мы рассмотрим еще одну принципиальную причину недопустимости непосредственных присваиваний `a.attrib := some_value`. Причина в том что не любые значения `some_value` могут быть допустимыми. Можно определить процедуру

```
set_polygon_size (new_size: INTEGER) is
    -- Установить новое значение числа вершин многоугольника
    require
        new_size >= 3
    do
        size := new_size
    end
```

параметр которой может равен 3 или больше. Прямое присваивание не позволяет учесть это условие и в результате получается некорректный объект.

Эти рассуждения показывают, что автор класса имеет в своем распоряжении пять возможных уровней предоставления прав доступа клиентов к атрибутам ([рис. 7.8](#)).



Рис. 7.8. Возможные варианты прав доступа клиентов к атрибутам

Уровень 0 соответствует полному отсутствию доступа к атрибуту. На уровне 1 открыт доступ только для чтения. На уровне 2 разрешена модификация с помощью специальных алгоритмов. На уровне 3 новое значение может быть присвоено, только если удовлетворяет определенным условиям, как в примере для многоугольника. На уровне 4 ограничения снимаются.

Решение, описанное в данной лекции, следует из приведенного анализа. Экспорт атрибута дает клиентам право доступа только для чтения (уровень 1). Разрешение на модификацию обеспечивается написанием и экспортом соответствующих процедур. Они предоставляют ограниченные права, как в примере для счетчика (уровень 2), право модификации при соблюдении определенных условий (3) и неограниченный доступ (4).

Это решение является развитием идей, существующих в различных ОО-языках:

- В Smalltalk для обеспечения доступа клиентов к атрибуту на уровне 1 приходится писать специальные функции подобные `abscissa` and `ordinate`. Это источник дополнительной работы для программиста и причина снижения производительности.
- C++ и Java представляют другую крайность. Если атрибут экспортирован, то он сразу становится доступным на уровне 4 для чтения и для записи путем прямого присваивания в стиле `my_point.x := 3.7`. Единственный путь реализации других уровней это полное скрытие атрибута и написание экспортированных процедур для поддержки уровней 2 и 4 и функций для уровня 1. Далее все аналогично Smalltalk. Поддержка уровня 3 невозможна в связи с отсутствием в этих языках механизма утверждений.

Данная дискуссия иллюстрирует два важных принципа построения языка: не создавать без необходимости дополнительных проблем программисту и не вводить лишних языковых конструкций.

Оптимизация вызовов

На уровнях 2 и 3 неизбежно использование явных вызовов процедуры подобных `my_polygon.set_size(5)` для изменения значения атрибута. Существует опасение, что использование такого стиля на уровне 4 негативно скажется на производительности. Тем не менее компилятор может создавать для вызова `my_point.set_x(3.7)` код столь же эффективный, как и для `my_point.x := 3.7`, если бы такое присваивание было разрешено.

Компилятор ISE добивается этого путем общего механизма непосредственного встраивания кода подпрограмм с подстановкой соответствующих параметров и необходимость вызовов устраняется.

Встраивание кода подпрограмм является одним из преобразований, которое должен обеспечивать оптимизирующими компилятор ОО-языка. Модульный стиль разработки, поощряемый объектной технологией, сопряжен с наличием большого числа небольших подпрограмм. Программисты не должны беспокоиться, что соответствующие вызовы приведут к снижению производительности. Они должны заботиться о последовательном соблюдении принципов объектной архитектуры, а не об особенностях выполнения.

В некоторых языках программирования, особенно в Ada и C++, разработчики могут отметить, какие подпрограммы они хотели бы встраивать. По ряду причин предпочтительно, чтобы эта работа выполнялась в режиме автоматической оптимизации.

- Встраивание кода далеко не всегда применимо, и компилятор гораздо корректнее может принять правильное решение.
- При внесении изменений в ПО, в частности с использованием наследования, встроенная подпрограмма может стать не встроенной. Компилятор выявит такие ситуации гораздо лучше, чем человек.
- В случае больших систем компилятор всегда более эффективен. На основе анализа размера подпрограмм и числа вызовов он может точнее определить, какие подпрограммы целесообразно встраивать. Это опять же существенно в случае изменений ПО, поскольку человек не в состоянии отследить эволюцию каждого фрагмента.
- Программисты могут занять время более полезной работой.

Современная концепция разработки ПО подразумевает, что утомительную, автоматизируемую и тонкую работу по оптимизации нужно возлагать на соответствующие утилиты, а не на человека. Это обстоятельство является одной из причин принципиальной критики C++ и Ada. Мы вернемся к этому вопросу при обсуждении двух других ключевых моментов объектной технологии - управления памятью и динамического связывания. (См. "Требования к сборщику мусора", [лекция 9](#), и "Подход C++ к связыванию", [лекция 14](#))

Архитектурная роль селективного экспорта

Селективный экспорт это не просто удобство, а неотъемлемая часть ОО-архитектуры. Он позволяет группе концептуально связанных классов обеспечить друг другу доступ ко всем своим компонентам, скрыв их от остального

мира в соответствии с принципом скрытия информации. Кроме того, это ключ к пониманию вопроса о том, нужны ли вообще модули более высокого уровня, чем классы.

Без селективного экспорта единственным решением будет введение нового типа модулей, представляющего собой группу классов. Такие супермодули - аналоги пакетов Ada и Java - будут осуществлять скрытие информации и экспорт по своим правилам. Добавление в элегантную структуру, основанную на классах, нового и частично несовместимого модульного уровня приведет к усложнению и увеличению объема языка.

Лучшим решением является использование в качестве супермодулей самих классов. Такой подход реализован в Simula, допускающем вложение классов. Однако он не дает ощутимых преимуществ.

Простота объектной технологии в значительной степени базируется на использовании простой концепции модулей. Поддержка классами повторного использования основана на возможности их извлечения из контекста, сохраняя лишь их логические зависимости. Существует риск потери этих преимуществ, если ввести супермодули. В частности, становится невозможным непосредственное повторное использование класса, являющегося частью пакета. Придется либо полностью импортировать весь пакет, либо делать копию класса. Явно непривлекательная форма повторного применения.

Необходимость объединения классов в структурированные коллекции сохраняется. В данной книге она реализована через понятие кластера ([лекция 10](#) курса "Основы объектно-ориентированного проектирования"). Однако понятие кластера относится к области управления и организации. Если включить его в качестве языковой конструкции, то это угроза потери простоты ОО-подхода и его поддержки модульности.

Если необходима группа классов, в которой каждый наделен специальными привилегиями, то нет нужды в супермодулях. Простое решение обеспечивается за счет селективного экспорта, что позволяет сохранить классам свой независимый статус.

Импорт листингов

В исходных текстах классов, в предложениях **feature**, перечислены компоненты, доступные другим классам. Почему бы, в свою очередь, не включать списки компонентов, полученных от других классов? Язык Modula-2 поддерживает, например, объявление **import**.

Тем не менее, при ОО-подходе это ничего не дает кроме документирования. Для использования компонента **f** из другого класса **C**, данный класс должен быть клиентом или потомком этого класса. В первом случае это означает, что **f** используется как

a . f

но тогда должно присутствовать объявление **a**:

a : C

недвусмысленно показывающее, что **f** компонента **C**. В случае классов потомков информация будет доступна из официальной документации класса, его плоской краткой формы.

Следовательно, нет необходимости в предложении **import**. ("Плоская краткая форма", [лекция 11](#))

Тем не менее, удобная графическая среда разработки должна обладать возможностью предоставления программисту информации о поставщиках и предках данного класса и их поставщиках и предках, следуя далее по цепочке.

Присваивание функции результата

Присваивание функции результата является интересной языковой проблемой, обсуждение которой было начато ранее в данной лекции. Стоит изучить ее подробнее ввиду ее важности и для языков, не использующих ОО-подход.

Рассмотрим функцию - подпрограмму, возвращающую результат. Целью любого вызова функции является вычисление некоторого результата и возвращение его в вызывающую подпрограмму. Вопрос в том, каким образом обозначить этот результат в тексте самой функции, в частности в инструкциях инициализирующих и изменяющих результат.

Введенное в данной лекции соглашение использует специальную сущность **Result**. Она рассматривается как локальная сущность, инициализируется соответствующим значением по умолчанию, а возвращаемое значение равно окончательному значению **Result**. В соответствии с правилами инициализации это значение всегда определено, даже если в теле функции нет присваивания **Result** значения. Так функция

```
f: INTEGER is
  do
    if some_condition then Result := 10 end
  end
```

возвратит 10 при выполнении условия **some_condition** на момент вызова и 0 (значение по умолчанию при инициализации **INTEGER**) в противном случае. Несколько известно автору, техника использования **Result** была

впервые предложена в данной книге. С момента выхода первого издания она была включена по крайней мере в один язык - Borland Delphi. Надо заметить, что она неприемлема для языков, допускающих объявление функций внутри других функций, поскольку имя Result становится двусмысленным. В различных языках наиболее часто используются следующие приемы:

- (A) Заключительные инструкции **return** (C, C++/Java, Ada, Modula-2).
- (B) Использование имени функции в качестве переменной (Fortran, Algol 60, Simula, Algol 68, Pascal).

Соглашение А основано на инструкции вида **return e**, выполнением которой завершается функция, возвращая **e** в качестве результата. Преимущество этого метода в его ясности, поскольку возвращаемое значение четко выделено в тексте функции. Однако он имеет и отрицательные стороны:

- (A1) На практике результат часто определяется в процессе вычислений, включающих инициализацию и ряд промежуточных изменений значения. Возникает необходимость во временной переменной для хранения промежуточных результатов.
- (A2) Методика имеет тенденцию к использованию модулей с несколькими точками завершения. Это противоречит принципам хорошего структурирования программ.
- (A3) В языке должна быть предусмотрена ситуация, когда последняя инструкция, выполненная при вызове функции, не является **return**. В программах Ada в этом случае возбуждается исключение времени выполнения.

Две последние проблемы разрешаются, если рассматривать **return** не как инструкцию, а как синтаксическое предложение, являющееся обязательной частью текста любой функции:

```
function name (arguments): TYPE is
  do
    ...
  return
    expression
  end
```

Это решение развивает идею инструкции **return** и устраняет ее наиболее серьезные недостатки. Тем не менее, ни один язык его не использует, оставляя проблему А1 открытой.

Методика В использует имя функции как переменную в тексте функции. Возвращаемое значение совпадает с окончательным значением этой переменной. Это избавляет от необходимости объявления временной переменной, упомянутой в А1.

При таком подходе указанные три проблемы не проявляются. Но возникают другие трудности, поскольку одно и то же имя обозначает одновременно и функцию, и переменную. Присутствие имени функции в ее теле может быть истолковано двояко: как имя переменной и как рекурсивный вызов. Поэтому язык должен точно регламентировать, в каких ситуациях речь идет о переменной, а в каких о рекурсивном вызове функции. Если в теле функции **f**, имя **f** присутствует как цель присваивания, то речь идет о переменной

```
f := x
```

а если **f** является частью выражения, то подразумевается рекурсивный вызов функции

```
x := f
```

который допустим только при отсутствии у **f** параметров. Однако присваивания вида

```
f := f + 1
```

будут отклонены компилятором в случае наличия у **f** параметров, а при отсутствии таковых будут поняты как рекурсивные вызовы, результат которых присваивается переменной **f**. Последняя интерпретация скорее всего не будет соответствовать замыслу разработчика, который просто хотел увеличить переменную **f** на единицу, а в результате получит бесконечный цикл. Для достижения требуемого эффекта придется все равно ввести временную переменную.

Соглашение, основанное на предопределенной сущности **Result**, устраниет проблемы приемов А и В. В языках, предусматривающих инициализацию по умолчанию всех сущностей, включая **Result**, достигается дополнительное преимущество. Упрощается написание функций, так как часто функция должна во всех случаях, кроме специально обусловленных, возвращать значение по умолчанию. Например, функция

```
do
  if some_condition then Result := "Some specific value" end
end
```

не нуждается в предложении **else**. Подразумевается, что язык должен строго определить значения по умолчанию. Такие соглашения будут введены в следующей лекции.

Последнее преимущество соглашения **Result** вытекает из принципа проектирования по контракту (см. гл. 11). Можно использовать **Result** для выражения абстрактного свойства результата функции, не зависящего от реализации в постусловии подпрограммы. Никакой другой подход не позволит написать следующее:

```

prefix "|_": INTEGER is
    -- Целая часть числа
do
    ... Реализация опущена ...
ensure
    no_greater: Result <= Current
    smallest_possible: Result + 1 > Current
end

```

В предложении **ensure** содержатся постусловия, утверждающие два свойства результата: результат не должен быть больше значения, к которому применяется операция, и это значение должно быть меньше чем результат плюс единица.

Дополнение: точное определение сущности

Будет полезно в процессе обсуждения проблем нотации уточнить понятие сущности, которое мы постоянно использовали. Это в значительной степени техническое понятие, обобщающее традиционное понятие переменной.

Сущности, в том смысле, в котором они используются в данной книге, обозначают имена некоторых величин времени выполнения, связанных с объектами. Можно выделить три возможных случая:

Определение: сущность (entity)

Сущность может представлять собой:

- (E1) Атрибут класса
- (E2) Локальную сущность подпрограммы, включая предопределенную сущность **Result** для функции
- (E3) Формальный аргумент подпрограммы

Случай E2 подчеркивает, что сущность **Result** всегда рассматривается как локальная. Другие локальные сущности введены в объявлении **local**. **Result** и другие локальные сущности заново инициализируются при каждом вызове подпрограммы.

Все сущности, за исключением формальных аргументов (E3), доступны для записи, то есть могут присутствовать как цель **x** в присваивании **x := some_value**.

Ключевые концепции

- Фундаментальная концепция объектной технологии основана на понятии класса. Класс это абстрактный тип данных, частично или полностью реализованный.
- Класс может иметь экземпляры, называемые объектами.
- Нельзя путать объекты (динамические элементы) с классами (статическим описанием свойств, общих для множества объектов времени выполнения).
- При последовательном подходе к объектной технологии каждый объект является экземпляром класса.
- Класс одновременно служит модулем и типом. Оригинальность и мощь ОО-модели следует частично из интеграции этих понятий.
- Класс характеризуется компонентами, включая атрибуты, представляющие поля в экземплярах класса, и подпрограммы, представляющие вычисления с участием данных экземпляров. Подпрограмма может быть функцией возвращающей результат или процедурой, если результат не возвращается.
- Базовым механизмом ОО-вычислений является вызов компонентов (обращение к компонентам) класса. Вызов компонента применяет компонент к экземпляру класса (возможно с аргументами).
- При вызове именованных компонентов используется точечная нотация, а при вызове компонент-операций - инфиксная или префиксная нотация.
- Каждая операция относительна к "текущему экземпляру" класса.
- Для клиентов класса (других классов, которые используют его компоненты) атрибут ничем не отличается от функции без аргументов, в соответствии с принципом унифицированного доступа.
- Исполняемый ансамбль классов называется системой. Система содержит корневой класс и все классы, которые необходимы корневому прямо или косвенно через клиентские отношения или наследование. Выполнение системы сводится к созданию экземпляра корневого класса и вызову процедуры создания для данного экземпляра.
- Системы имеют децентрализованную архитектуру. Порядок действий несуществен для разработки.
- Уточнение процесса сборки достигается с помощью простого языка описания систем **Lace**. В спецификации **Lace**, называемой файлом **Ace**, указывается корневой класс и набор каталогов, в которых размещены кластеры системы.
- Процесс компоновки может быть автоматизирован без использования **Make**-файлов и директив **Include**.
- Механизм скрытия информации требует гибкости. Наряду с неограниченным доступом и полным скрытием может потребоваться экспорт только для части клиентов. Атрибуты могут быть доступны только для чтения, для чтения и ограниченной модификации и в режиме полного доступа.
- Экспорт атрибута означает доступ к нему только для чтения. Модификация требует вызова соответствующей экспортированной процедуры.
- Селективный экспорт дает возможность группам родственных классов обеспечить специальный режим доступа

- для каждого компонента.
- Необходимость в надстройках над классами - супермодулей - отсутствует. Классы должны оставаться независимыми программными компонентами.
 - Модульный стиль ОО-разработок требует большого числа небольших подпрограмм. Потенциальная опасность снижения производительности может быть достигнута путем встраивания этих подпрограмм оптимизирующими компилятором. Ответственность за поиск таких фрагментов следует возложить на компилятор, а не на разработчиков.

Библиографические замечания

Понятие класса пришло из языка Simula 67 (см. библиографические ссылки к [лекции 17](#) курса "Основы объектно-ориентированного проектирования"). Класс в Simula является одновременно модулем и типом, однако эта особенность специально не подчеркивалась и была утрачена у преемников Simula.

Принцип единственности цели может рассматриваться как аналог приема, хорошо известного в математической логике и теоретической компьютерной науке: **редукция (currying)**. Редукция функции двух переменных f означает замену ее функцией g одной переменной, возвращающей в качестве результата функцию одной переменной. В результате редукции для любых допустимых значений x и y :

$$(g(x))(y) = f(x, y)$$

Редуцировать функцию это, другими словами, специализировать ее по первому аргументу. Этот прием аналогичен использованной в данной лекции замене традиционной процедуры `rotate`, имеющей два параметра:

```
rotate (some_point, some_angle)
```

на функцию с одним параметром, имеющую цель:

```
some_point.rotate (some_angle)
```

В [М 1990] описана редукция и некоторые из ее применений в информатике, в частности, при формальном изучении синтаксиса и семантики языков программирования. Редукция будет еще рассматриваться при обсуждении графического интерфейса пользователя ([лекция 14](#) курса "Основы объектно-ориентированного проектирования").

В отличие от положений данной лекции в некоторых языках объект рассматривается как языковая конструкция, а не как понятие времени выполнения. Такой подход предназначен для исследовательских целей и не нуждается в понятии класса. Наиболее известным представителем этой школы является язык Self [Chambers 1991], в котором вместо классов используются "прототипы".

Детали соглашения об инфиксных и префиксных операциях, в частности таблица приоритетов, приведены в [М 1992].

James McKim обратил мое внимание на последний аргумент в пользу соглашения `Result` (использование для постусловий).

Упражнения

У7.1 POINT как абстрактный тип данных

Напишите спецификацию абстрактного типа данных для описания точки на плоскости.

У7.2 Завершение реализации POINT

Завершите исходный текст класса `POINT`. Заполните недостающие фрагменты, добавьте процедуру `rotate` (вращение точки вокруг начала координат), а также другие компоненты, которые считаете необходимыми.

У7.3 Полярные координаты

Перепишите класс `POINT` таким образом, чтобы в качестве базового использовалось бы представление точки в полярных, а не декартовых координатах.

Основы объектно-ориентированного программирования

8. Лекция: Динамические структуры: объекты

В предыдущей лекции отмечалось, что экземпляры классов называют объектами. Настало время переключить внимание на эти объекты, и в общем смысле - на модель ОО-вычислений в времени выполнения. В предыдущих лекциях рассматривались в основном концептуальные вопросы. Теперь необходимо обратиться к аспектам реализации. В частности, рассмотреть вопросы использования памяти (обсуждение будет продолжено в следующей лекции в связи со сборкой мусора). Неоднократно отмечалось, что одним из преимуществ в объектной технологии разработки ПО является учет в полном объеме деталей реализации. Поэтому экскурсия в область реализации будет полезной, даже если сфера ваших интересов связана в основном с вопросами анализа и проектирования. Невозможно понять метод, не рассматривая его влияние на структуры времени выполнения. Изучение объектных структур в данной лекции может служить весьма хорошим примером того, насколько неправильно отделять вопросы реализации от проблем будто бы "высокого" уровня. В процессе рассмотрения новых технических приемов, связанных с вопросами реализации, приходит более глубокое понимание абстрактных понятий. Типичным примером может служить введение ссылочных и развернутых значений, представляющих, на первый взгляд, неприметное техническое решение. В действительности, это ответ на общий вопрос об отношении части и целого, постоянно обсуждаемый в дискуссиях по ОО-анализу. В некоторой части компьютерной литературы приписывается значение реализации и считается, что самое важное - это анализ. Но разработка ПО - это разработка моделей. Хорошая техника реализации часто является одновременно и хорошим средством моделирования. Помимо программных систем ее можно использовать и во многих других областях. Данная лекция в большей степени посвящена моделированию, нежели реализации в строгом смысле этого термина. В ней показано, как можно использовать объектные структуры для построения реалистичных и полезных операционных описаний различного вида систем.

Объекты

В процессе выполнения ОО-система создает некоторое число объектов. Организация этих объектов и отношения между ними определяют конструкцию времени выполнения. Рассмотрим свойства объектов.

Что такое объект?

Прежде всего, необходимо напомнить смысл термина "объект". Полная ясность была внесена в предыдущей лекции в виде строгого определения (Определение и объективное правило, см. [лекцию 7](#)):

Определение: объект

Объект - это экземпляр некоторого класса

Во время выполнения программная система, содержащая класс C, может в разных точках, используя процедуры создания или клонирования, создавать экземпляры C, - структуры данных, соответствующие образцу, заданному классом C. Например, экземпляр класса POINT представляет собой структуру данных, состоящую из двух полей, соответствующих атрибутам x и y класса. Экземпляры всех возможных классов составляют множество объектов системы.

Это официальное определение в мире ОО-ПО. Но в повседневном языке термин "объект" имеет гораздо более широкий смысл. Любая программная система связана с определенной внешней системой, которая может содержать "объекты": точки, линии, поверхности и тела в графической системе; сотрудников и их оклады в системе расчета заработной платы и т.д. В таких ситуациях, как правило, реальным объектам соответствуют программные объекты. Примером может служить класс EMPLOYEE в системе расчета зарплаты, экземпляры которого являются компьютерными моделями сотрудников.

Хорошим следствием дуализма слова "объект" является естественность и мощь ОО-метода, применяемого для целей моделирования реальных систем. Это уже отмечалось при рассмотрении принципа Прямого Отображения (*direct mapping*), который, как отмечалось, является принципиальным требованием модульного проектирования. Неудивительно, что некоторые классы являются моделями внешних типов объектов проблемной области, а экземпляры классов - моделями реальных объектов. (См. "Прямое отображение", [лекция 3](#))

Но не стоит переоценивать "реальность" слова "объект". В науке и технике существует большой риск в заимствовании слов естественного языка и придания им специального смысла. Термин "объект" настолько перегружен повседневным смыслом, что техническое его использование может стать источником недоразумений. В частности:

- Не все классы соответствуют типам проблемной области. Многие классы, введенные в интересах проектирования и реализации, не имеют двойников в моделируемой системе. Именно эти классы на практике могут иметь наибольшее значение и именно их最难нее всего спроектировать.
- Некоторые концепции проблемной области естественно приводят к классам, хотя в проблемной области не существует реальных объектов, которые можно было бы поставить в соответствие экземплярам этих классов. Примерами могут быть класс STATE, описывающий состояние системы, или класс COMMAND. (См. [лекцию 20](#) и [лекцию 21](#) курса "Основы объектно-ориентированного проектирования")

Когда слово "объект" используется в этой книге, то из контекста ясно, в общем или техническом смысле используется этот термин. В тех случаях, когда эту разницу необходимо подчеркнуть, используется уточнение - программный объект или внешний объект.

Базовая форма

Программный объект довольно простое существо, если известен класс, которому он принадлежит.

Пусть O - объект. По определению он является экземпляром некоторого класса. Точнее, он является **прямым экземпляром (*direct instance*)** только одного класса, например C.

С учетом наследования О будет тогда косвенным экземпляром других классов, - предков С. Это тема дальнейшего обсуждения; в данной дискуссии достаточно понятия прямого экземпляра. Везде, где не может возникнуть недоразумений, слово "прямой" будет опущено.

Класс С называется **порождающим** классом (**generating class**) или просто **генератором (generator)** объекта О. Заметьте, С- программный текст, а О - структура данных времени выполнения, появляющаяся в результате работы рассмотренных ниже механизмов создания объектов.

Часть компонентов Сявляется атрибутами. Эти атрибуты полностью определяют форму объекта, представляющего собой просто набор полей, по одному на каждый атрибут.

Рассмотрим класс POINT из предшествующей лекции (Текст класса POINT см. в [лекции 7](#)). Исходный текст имеет вид:

```
class POINT feature
    x, y: REAL
    ... Объявления подпрограмм ...
end
```

Подпрограммы опущены, так как форма объектов полностью определяется атрибутами соответствующих классов. Данный класс имеет два атрибута x и у типа REAL, следовательно, его экземпляр - это объект с двумя полями, содержащими значения этого типа:

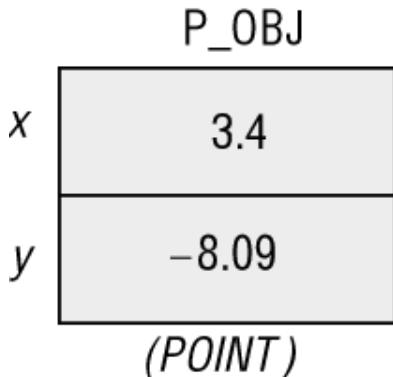


Рис. 8.1. Экземпляр класса POINT

В данной книге объекты - набор полей - изображаются в виде смежных прямоугольников, содержащих соответствующие значения. Внизу курсивом в скобках дается имя класса (в данном случае - POINT). Против каждого поля, тоже курсивом - имена соответствующих атрибутов (x и у). Иногда сверху указывается имя объекта (здесь P_OBJ), не имеющее двойника в программном тексте, но позволяющее ссылаться на объект при обсуждении.

На диаграммах, представляющих структуру ОО-системы или части такой системы, классы изображаются в виде эллипсов. Эти соглашения позволяют не путать классы и объекты.

Простые поля

Оба атрибута класса POINT относятся к типу REAL. Следовательно, соответствующие поля прямого экземпляра POINT содержат действительные числа.

Это пример полей, соответствующих атрибутам одного из "базовых" типов. Формально эти типы определены как классы, а их экземпляры принимают значения из предопределенных диапазонов. К базовым (предопределенным, встроенным) типам относятся:

- BOOLEAN, может иметь только два различных экземпляра, соответствующих булевым значениям true и false;
- CHARACTER, экземпляры которого представляют символы;
- INTEGER, экземпляры которого представляют целые числа;
- REAL и DOUBLE, экземпляры которых представляют действительные числа одинарной и двойной точности.

Тип STRING, представляющий конечную последовательность символов, на данном этапе рассматривается как базовый. Далее будет показано, что в действительности он относится к другой категории. ("Строки", см. [лекцию 13](#))

Для каждого базового типа необходимо определить правила записи их значений в исходных текстах. Соглашения просты:

- Для типа BOOLEAN два различных экземпляра обозначаются как True и False.
- Экземпляр CHARACTER будет записываться как символ в апострофах: 'A'.
- Экземпляр STRING обозначается как последовательность символов в двойных апострофах: "Это строка".
- Для обозначения экземпляра INTEGER используем обычную десятичную нотацию: 34, -675, +4.
- Для экземпляров REAL или DOUBLE будет применяться как обычная нотация: 3.5 или -0.05, так и экспоненциальное представление: -5.e-2.

Простое представление книги - класс BOOK

Рассмотрим класс с атрибутами базовых типов:

```
class BOOK1 feature
    title: STRING
    date, page_count: INTEGER
end
```

Типичный экземпляр класса выглядит так:

<i>title</i>	"The Red and the Black"
<i>date</i>	1830
<i>page_count</i>	341
<i>(BOOK1)</i>	

Рис. 8.2. Объект, представляющий книгу

Поскольку в настоящий момент нас в первую очередь интересует структура объектов, то в последующих примерах все компоненты классов будут атрибутами, а подпрограммы отсутствуют.

Это означает, что на данном этапе обсуждения объекты подобны записям или структурам в языках Pascal и С. Принципиальное отличие от этих языков выражается в том, что, благодаря наличию механизмов скрытия информации, клиенты классов не могут непосредственно присваивать значения полям таких объектов. В Pascal и в С с незначительными синтаксическими различиями допустимо объявление записи с последующим присваиванием (**Внимание: Недопустимая нотация! Только для обсуждения.:**):

```
b1: BOOK1
...
b1.page_count := 355
```

Здесь во время выполнения полю *page_count* объекта, присоединенного к *b1*, присваивается значение 355. Для классов такая возможность не допускается. Предоставлять клиентам классов разрешение менять поля объектов было бы насмешкой над правилом скрытия информации. В этом случае терял бы смысл выборочный экспорт, управляемый автором класса. В ОО-подходе модификация значений полей допустима только с помощью процедур класса, добавляемых в том случае, если автор класса решит предоставить такую возможность своим клиентам. Далее такая процедура будет добавлена в класс *BOOK1*.

Разрешение присваиваний вида *b1.page_count := 355* в C++ и Java отражает ограничения, возникающие при попытках внедрения объектной технологии в контекст языка С.

Сами разработчики Java отмечают, что программист может испортить объект при наличии общедоступного поля, так как значение такого поля можно изменить путем непосредственного присваивания. Многие авторы языков вводят такую возможность, а затем предупреждают: "не делайте этого". Логичнее определить метод и нотацию, поддерживающие такие ограничения.

В ОО-ПО классы без подпрограмм редко имеют практическое значение. Исключением являются ситуации, когда в родительских классах определяется набор атрибутов, а потомки содержат необходимые подпрограммы. Другим примером могут служить классы, представляющие внешние объекты, которые принципиально невозможно модифицировать, например данные от внешних датчиков в системе реального времени. Но на данном этапе такой подход полезен для понимания основных концепций, подпрограммы будут добавлены позже.

Писатели

Используя указанные выше типы, определим класс WRITER для описания автора книги:

```
class WRITER feature
    name, real_name: STRING
    birth_year, death_year: INTEGER
end
```

<i>name</i>	"Stendhal"
<i>real_name</i>	"Henri Beyle"
<i>birth_year</i>	1783
<i>death_year</i>	1842

(WRITER)

Рис. 8.3. Объект «писатель»

Ссылки

Чаще всего нам необходимы объекты с полями, представляющими другие объекты. Например, книга имеет автора, который представлен экземпляром класса WRITER.

Можно ввести понятие подобъекта. В новой версии класса BOOK2 его экземпляры содержат поле, являющееся объектом - экземпляром класса WRITER.

<i>title</i>	"The Red and the Black"								
<i>date</i>	1830								
<i>page_count</i>	341								
<table border="1"> <tr> <td><i>name</i></td> <td>'Stendhal'</td> </tr> <tr> <td><i>real_name</i></td> <td>'Henri Beyle'</td> </tr> <tr> <td><i>birth_year</i></td> <td>1783</td> </tr> <tr> <td><i>death_year</i></td> <td>1842</td> </tr> </table>		<i>name</i>	'Stendhal'	<i>real_name</i>	'Henri Beyle'	<i>birth_year</i>	1783	<i>death_year</i>	1842
<i>name</i>	'Stendhal'								
<i>real_name</i>	'Henri Beyle'								
<i>birth_year</i>	1783								
<i>death_year</i>	1842								
(WRITER)									

<i>title</i>	"Life of Rossini"								
<i>date</i>	1823								
<i>page_count</i>	307								
<table border="1"> <tr> <td><i>name</i></td> <td>'Stendhal'</td> </tr> <tr> <td><i>real_name</i></td> <td>'Henri Beyle'</td> </tr> <tr> <td><i>birth_year</i></td> <td>1783</td> </tr> <tr> <td><i>death_year</i></td> <td>1842</td> </tr> </table>		<i>name</i>	'Stendhal'	<i>real_name</i>	'Henri Beyle'	<i>birth_year</i>	1783	<i>death_year</i>	1842
<i>name</i>	'Stendhal'								
<i>real_name</i>	'Henri Beyle'								
<i>birth_year</i>	1783								
<i>death_year</i>	1842								
(WRITER)									

(BOOK2)

(BOOK2)

Рис. 8.4. Два объекта «книга» с подобъектами «писатель»

Такое понятие подобъекта, несомненно, полезно, и далее в этой лекции будет показано, как создавать соответствующие классы.

Но это не совсем то, что необходимо. В каждом экземпляре BOOK2 приходится дублировать информацию об одном и том же авторе в виде подобъекта. Причины неприемлемости такого решения:

- Расходуется дополнительная память. Можно привести в качестве более характерного примера совокупность объектов, представляющих людей. Каждый объект в качестве подобъекта содержит информацию о стране гражданства. Очевидно, что численность населения намного превышает число стран.
- Более важно, что такая техника не обеспечивает разделения информации. Вполне естественно желание, чтобы внесение изменений в объект WRITER повлекло за собой автоматическое обновление этой информации для всех объектов - книг данного автора.

Лучшим является решение, представленное на [рис.8.5](#). Оно основано на новой версии класса, BOOK3.

Каждый экземпляр BOOK3 в поле author содержит **ссылку (reference)** на объект типа WRITER. Нетрудно дать точное определение.

Определение: ссылка

Ссылка это значение времени выполнения. Она может быть пустой (**void**) или присоединенной (**attached**).

Присоединенная ссылка однозначно идентифицирует объект (присоединена к конкретному объекту).

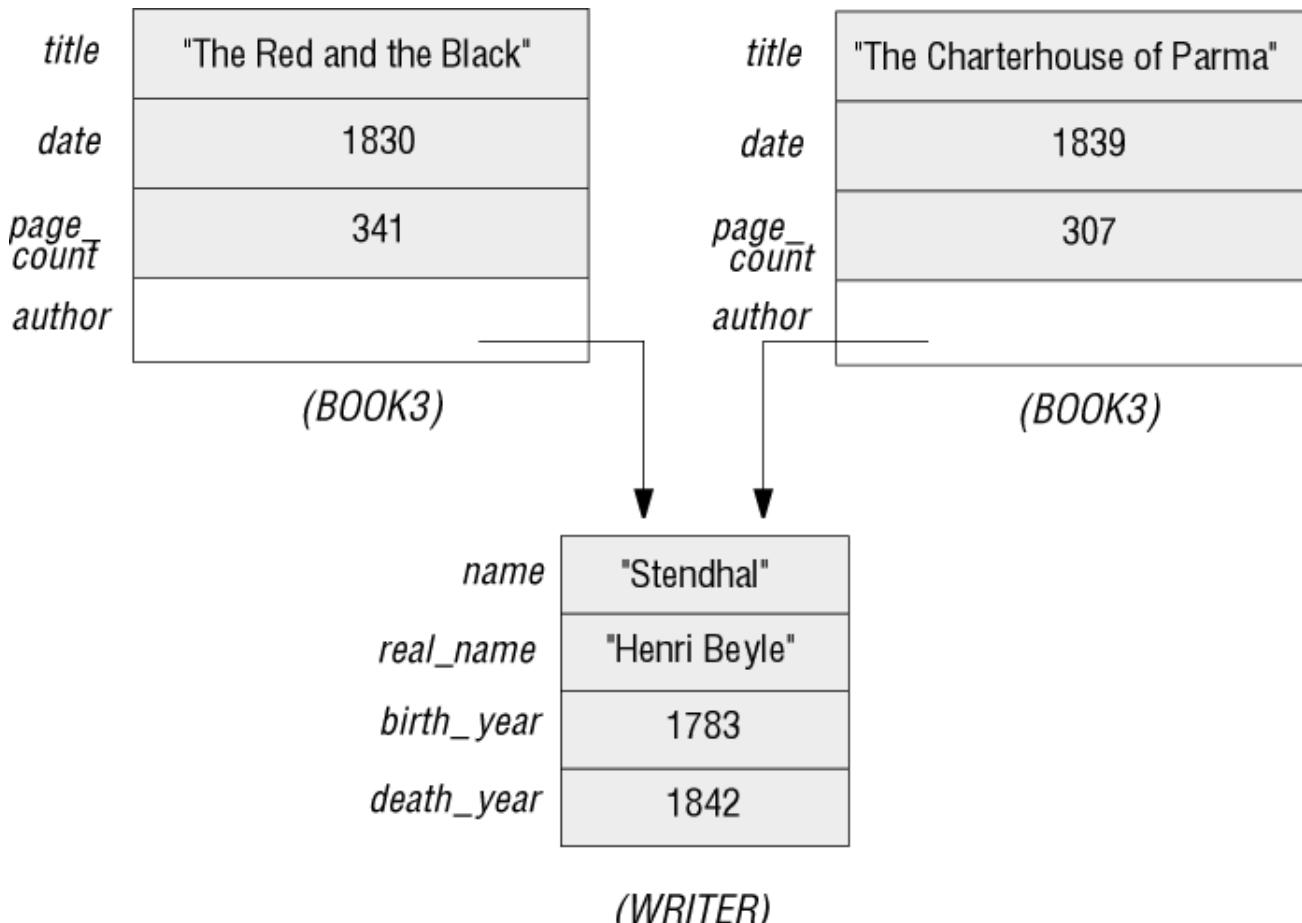


Рис. 8.5. Два объекта «книга» со ссылками на один и тот же объект «писатель»

На [рис.8.5](#) оба поля *author* экземпляров BOOK3 присоединены к одному экземпляру WRITER. Здесь и далее ссылки, присоединенные к объектам, обозначаются стрелками. На следующем рисунке используется графическое обозначение пустой ссылки, которая может обозначать неизвестного автора.

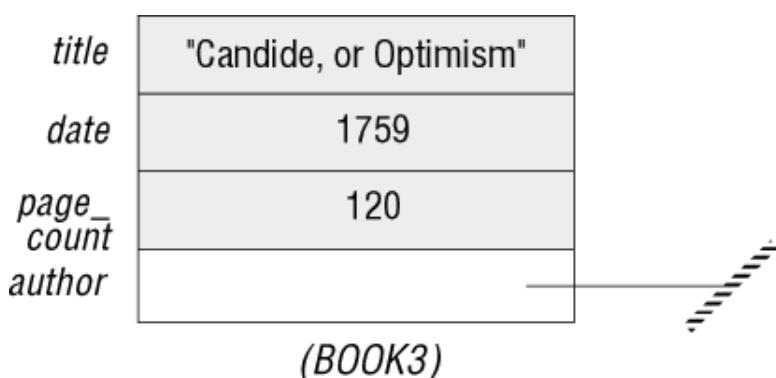


Рис. 8.6. Объект, содержащий пустую ссылку (Роман «Кандид» (Candide) опубликован анонимно)

Определение ссылки не подразумевает конкретной аппаратно-программной реализации. Если ссылка не пуста, то она идентифицирует объект и может рассматриваться как абстрактное имя объекта.

Концепция ссылки должна, безусловно, иметь аналог при реализации. Программирование на уровне машинного кода использует адресацию, многие языки программирования содержат понятие указателя. Понятие ссылки является более абстрактным. Хотя ссылка, в конечном итоге, может быть представлена адресом, не следует из этого исходить. Ссылка может содержать адрес наряду с другой информацией.

Отличие ссылок от указателей выражается в том, что они типизированы. Они напоминают типизированные указатели в Pascal и Ada (но не в C). Это означает, что данная ссылка может быть связана только с объектами определенных типов. По аналогии с обычной жизнью - код города имеет смысл только при наборе телефонных номеров. Он может выглядеть как обычное целое, но никому не придет в голову суммировать коды.

Идентичность объектов

Понятие ссылки приводит к концепции идентичности объектов. Каждый объект, созданный в процессе выполнения ОО-системы, уникален и идентифицируется независимо от значений его полей. Возможны две ситуации:

- (I1) Два различных объекта могут иметь абсолютно одинаковые поля.
- (I2) Напротив, поля данного объекта могут изменяться в процессе выполнения системы, но это не влияет на идентификацию объекта.

Эти наблюдения свидетельствуют о неоднозначности высказывания "а обозначает тот же объект, что и b". Можно подразумевать различные объекты с одинаковыми данными (I1) или состояния одного и того же объекта до и после изменения значений полей (I2). Мы будем использовать второе толкование и считать, что значения полей заданного объекта могут изменяться в процессе выполнения, а он остается "тем же самым объектом". В случае (I1) будем говорить о равных (но различных) объектах, точное определение понятия равенства будет дано позже.

В соответствии с определением I2 можно сказать, что поля объекта могут изменяться и это не будет ошибкой. Термин "поле" обозначает одно из значений, составляющих объект, а не соответствующий идентификатор поля - имя одного из атрибутов порождающего класса.

Каждому атрибуту класса соответствует поле объекта (1832 для атрибута date класса BOOK3 на [рис.8.6](#)). Атрибуты неизменны в процессе выполнения, как неизменно и деление объекта на поля, а значения полей меняются. Любой экземпляр BOOK3 будет всегда содержать четыре поля, соответствующие атрибутам title, date, page_count, author. Значения этих полей могут меняться у каждого экземпляра.

Изучение того, как сделать объекты сохранямыми (persistent), заставит нас продолжить изучение свойств идентичности объектов. (См. "Идентичность объектов", [лекция 13](#) курса "Основы объектно-ориентированного проектирования")

Объявление ссылок

Класс BOOK1 содержал атрибуты только базовых типов, его вариант BOOK3, содержит атрибут, представляющий ссылку на автора.

```
class BOOK3 feature
    title: STRING
    date, page_count: INTEGER
    author: WRITER -- Новый атрибут.
end
```

Объявленный тип дополнительного атрибута author это просто имя соответствующего класса: WRITER. Это будет общим правилом: если имеется стандартное объявление класса

```
class C feature ... end
```

то объявление некоторой сущности типа C

```
x: C
```

обозначает значения, являющиеся ссылками на потенциальные объекты типа C. Такое соглашение, использующее ссылки, обеспечивает большую гибкость и приемлемо в большинстве случаев. Подробное обсуждение этого правила и других возможных решений содержится в последнем разделе данной лекции.

Ссылка на себя

Ничто не препятствует объекту O1 в определенный момент выполнения системы содержать ссылку, присоединенную к самому O1. Такая ссылка на себя может быть косвенной. В ситуации на [рис.8.7](#) объект, имеющий значением поля name: "AlmaViva", сам является своим лендрордом (прямая циклическая ссылка). Фигаро любит Сюзанну, которая любит Фигаро (косвенная циклическая ссылка).

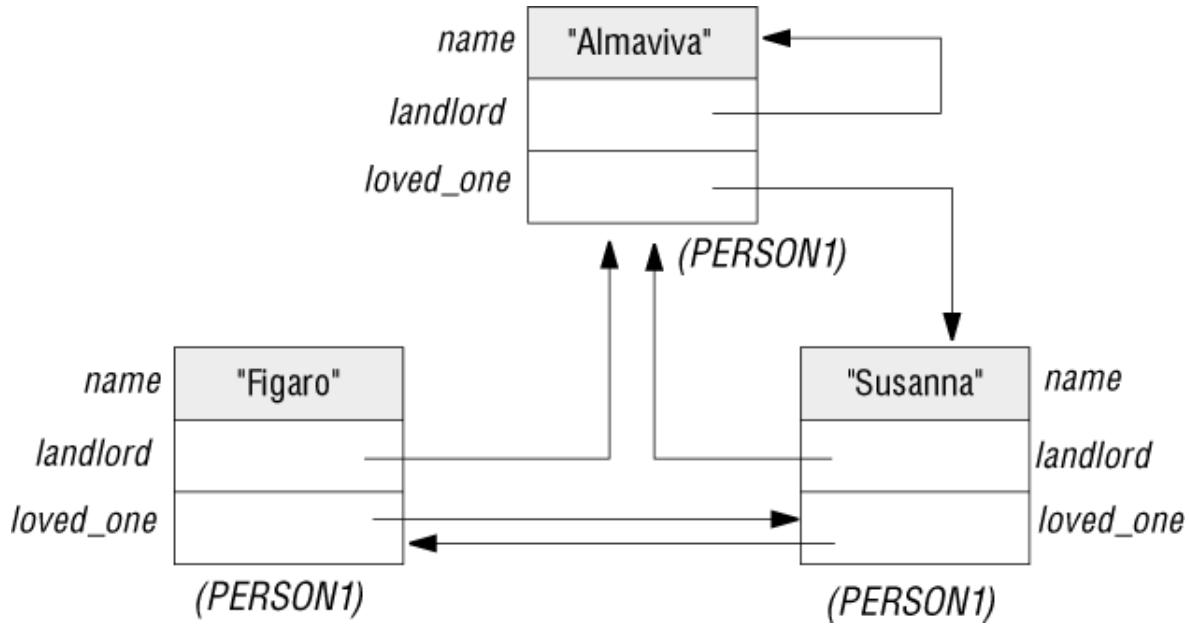


Рис. 8.7. Прямые и косвенные ссылки на себя

Такие циклы в динамических структурах возможны, только если клиентские отношения между соответствующими классами также содержат прямые или косвенные циклы. Объявление класса

```

class PERSON1 feature
    name: STRING
    loved_one, landlord: PERSON1
end

```

содержит прямой цикл (PERSON1 - клиент PERSON1).

Обратное утверждение неверно - присутствие цикла в объявлении класса не означает, что циклы обязательно появятся в структурах времени выполнения. Можно объявить класс

```

class PERSON2 feature
    mother, father: PERSON2
end

```

Класс является собственным клиентом. Однако если он моделирует соответствующие именам атрибутов отношения между людьми, то структуры времени выполнения никогда не будут содержать циклов, поскольку ни один человек не может быть собственным родителем или предком.

Взгляд на структуру объектов периода выполнения

На основе предшествующего рассмотрения выясняется в первом приближении структура ОО-системы в процессе выполнения.

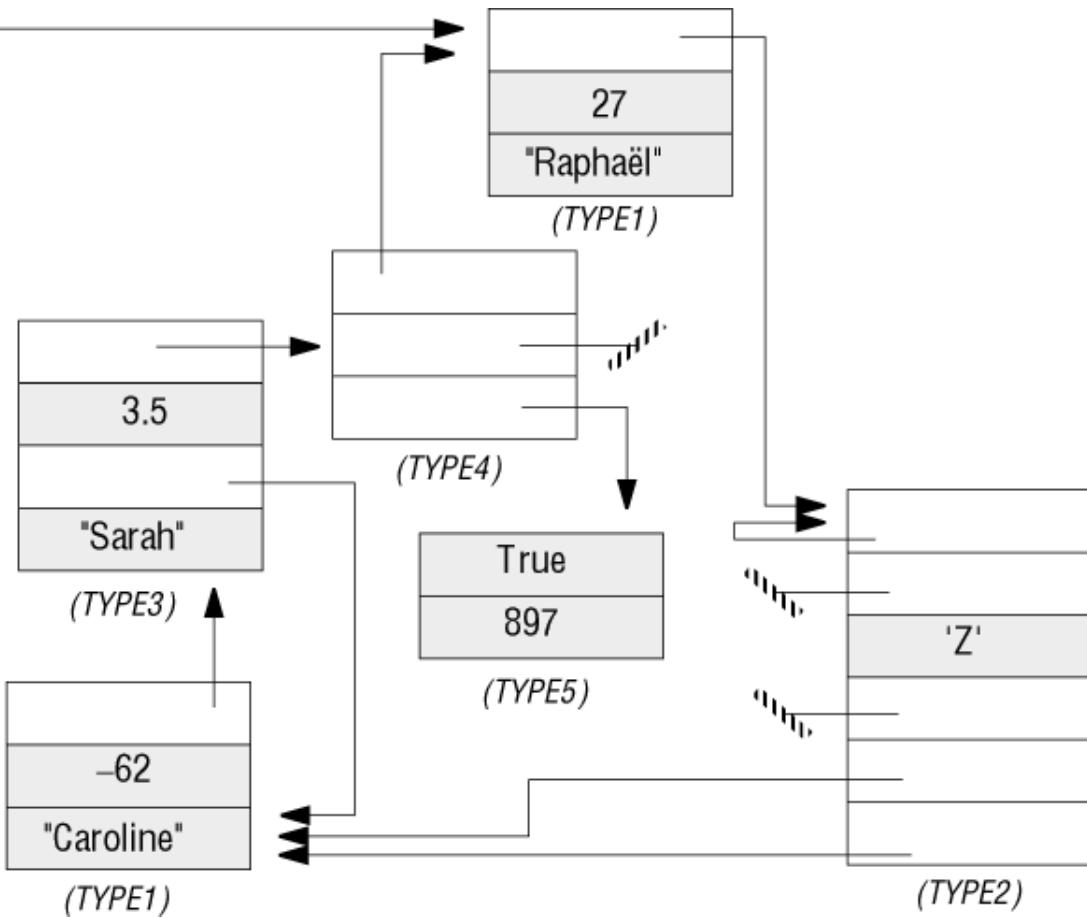


Рис. 8.8. Возможная структура объектов во время выполнения

Система состоит из нескольких объектов с различными полями. Некоторые поля содержат значения базовых типов, а другие являются пустыми или присоединенными ссылками на другие объекты. Каждый объект является экземпляром некоторого типа, основанного на классе (на рисунке тип указывается под объектом). Некоторые типы представлены единственным экземпляром, но гораздо чаще присутствует несколько экземпляров одного типа. На [рис.8.8](#) тип TYPE1 представлен двумя экземплярами, остальные - единственным. Некоторые объекты содержат поля только ссылочного типа (экземпляр TYPE4) или только базовых типов (экземпляр TYPE5). Могут присутствовать прямые или косвенные циклические ссылки (верхнее поле экземпляра TYPE2, по часовой стрелке от нижнего экземпляра TYPE1).

Подобная структура может показаться слишком запутанной. Впечатление от приведенной иллюстрации, демонстрирующей различные возможности, можно выразить выражением: "блюдо спагетти".

Это впечатление не совсем правильно. Впечатление простоты должен создавать программный текст, но не структура объектов периода выполнения. Текст отражает определенные отношения (такие как "любит", "имеет хозяина"). Конкретную структуру объектов периода выполнения можно назвать экземпляром таких отношений, она фиксирует связи между элементами данного набора объектов. Моделируемые отношения могут быть простыми, в то время как отношения индивидуумов конкретного множества объектов - достаточно сложными. Понятие "любит" очень просто, однако любовные отношения конкретных людей могут быть безнадежно запутаны.

Во время выполнения могут неизбежно возникать структуры, содержащие много объектов и имеющие запутанную структуру ссылок. Хорошая среда разработки должна предоставлять средства анализа объектных структур для тестирования и отладки.

Сложность динамических структур не должна влиять на статическую картину. Необходимо стараться сохранить набор классов и их отношения настолько простыми, насколько это возможно.

Тот факт, что простым моделям могут соответствовать сложные структуры данных, частично отражает мощь наших компьютеров. Короткий исходный текст может описывать огромные вычисления. Простая ОО-система может порождать в процессе выполнения миллионы объектов, связанных большим числом ссылок. Важнейшей целью программной инженерии является сохранение простоты ПО, даже когда экземпляры объектов такой простотой не обладают.

Объекты как средство моделирования

Рассмотренные приемы позволяют продвинуться в понимании возможностей ОО-подхода как средства моделирования. Важно, в частности, прояснить два аспекта: рассмотреть различные миры, связанные с разработкой ПО и отношения между ПО и внешней реальностью.

Четыре мира программной разработки

Из предшествующей дискуссии следует, что когда мы говорим об ОО-разработке, следует различать четыре отдельных мира:

- Моделируемую систему, - внешнюю по отношению к программной системе, описываемую типами объектов и их абстрактными отношениями.
- Частную конкретизацию внешней системы, состоящую из объектов с фиксированными отношениями.
- Программную систему, состоящую из классов, связанных ОО-отношениями ("быть клиентом", "быть наследником").
- Объектную структуру в том виде, в котором она существует в процессе выполнения программной системы, то есть множество программных объектов, связанных ссылками.

Соотношения между этими мирами представлены на [рис.8.9](#).

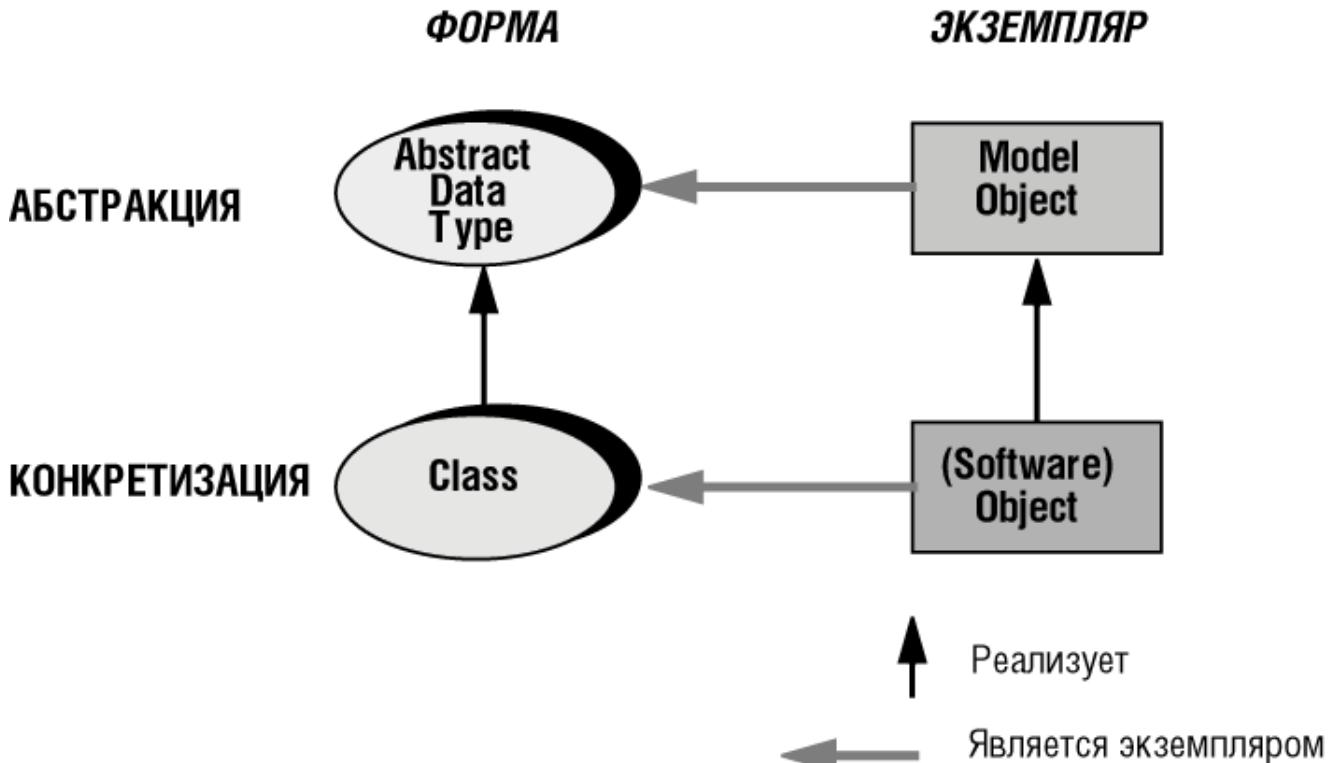


Рис. 8.9. Формы и их экземпляры

И на программном и на внешнем уровне (нижняя и верхняя части рисунка) важно разграничить общие понятия и их конкретные реализации (классы и абстрактные отношения слева, объекты и отношения экземпляров справа). Данный момент уже обсуждался в дискуссии о сравнительной роли классов и объектов в предыдущей лекции. Применительно к отношениям необходимо отличать абстрактные отношения `loved_one` от множества связей `loved_one`, существующих между элементами конкретного множества объектов.

Это различие невыразимо ни в стандартных математических определениях понятия "отношение", ни в программистской терминологии, например в теории реляционных баз данных. Если ограничиться бинарными отношениями, то и в математике и в теории баз данных отношение определяется как множество пар в форме $\langle x, y \rangle$, где x и y являются элементами заданных множеств TX и TY . В терминах программирования все x относятся к типу TX , а все y - к типу TY . Будучи пригодными для математиков, эти определения не подходят для целей моделирования, поскольку не позволяют различать абстрактные отношения и отношения конкретных экземпляров. При моделировании системы отношение "любит" имеет свои общие и абстрактные свойства, совершенно не зависящие от записи того, кто кого любит в конкретной группе людей в некоторый момент времени.

Это обсуждение будет продолжено в [лекции 11](#), когда будут рассматриваться преобразования между абстрактными и конкретными объектами, и будет дано имя вертикальным стрелкам предыдущего рисунка - функция абстракции. (См. "Функции абстракции", [лекция 11](#))

Реальность: "седьмая вода на киселе"

Предшествующее обсуждение не содержит ссылок на "реальный мир", - вместо этого используется термин "моделируемая система".

Такое разграничение проводится не всегда. Во многих дискуссиях используется выражение "моделирование реального мира"; аналогичные высказывания содержат и книги по ОО-анализу. Однако говорить о "реальности" применительно к программной системе ошибочно, по крайней мере, по четырем причинам.

Во-первых, реальность отражается в глазах очевидца. Не впадая в профессиональный шовинизм, программист всегда вправе спросить своих заказчиков, почему их системы более реальны, чем его. Возьмите программу, выполняющую математические вычисления - проверку гипотезы четырех красок в теории графов, интегрирование дифференциальных уравнений или решение геометрических проблем на четырехмерной римановой поверхности. Нужно ли нам, программистам, спорить с друзьями (математиками, заказчиками) о том, чьи искусственные объекты - артефакты более реальны - фрагменты программного кода или полное подпространство отрицательной кривизны? (См. также [лекцию 2](#))

Во-вторых, понятие реального мира рушится в нередких ситуациях, когда ПО предназначено для разрешения проблем

ПО. Рассмотрим компилятор С, написанный на Pascal. Для него "реальными" объектами являются программы на С. Насколько эти программы более реальны, чем сам компилятор? Это наблюдение применимо и к другим системам, работающим с объектами, существующими только в компьютере. (См. [лекцию 6](#))

Третье соображение обобщает второе. В сегодняшнем информационном мире компьютеры стали частью реальности. На заре появления компьютеров можно было говорить, что создаваемая программная система моделирует реальную систему. Предприятие приобретало компьютеры для автоматизации бизнес процессов. При описании процессов современного банка его ПО является фундаментальной частью банковской системы. Ситуация аналогична квантовой физике, где невозможно отделить измерение от измеряемого механизма. Термин "виртуальная реальность" в какой-то мере отражает данную ситуацию. Программные продукты не менее реальны, чем те, что приходят из внешнего мира. Во всех таких ситуациях программная система пересекается с реальностью, отчего возникает положительная обратная связь, когда работа существующей системы приводит к новым и важным изменениям самой модели, приводя к изменениям программной системы.

Последний довод наиболее фундаментален. Программная система не является моделью реальности. В лучшем случае это модель модели некоторой части некоторой реальности. Система мониторинга пациента больницы не является моделью больницы, но реализацией конкретной точки зрения на некоторые аспекты работы больницы. Это модель модели некоторой части реальности больницы. Астрономическая программа это не модель вселенной, а всего лишь программная модель чьей-то модели некоторых свойств некоторой части вселенной. Финансовая информационная система не является моделью фондового рынка. Это программная реализация модели, разработанной конкретной компанией для описания тех аспектов фондового рынка, которые соответствуют целям данной компании.

Абстрактные типы данных, лежащие в основе ОО-метода, помогают понять, почему не следует придерживаться широко распространенной, но иллюзорной точкой зрения, что мы имеем дело с "реальным миром". Первый шаг к объектной ориентации, выражаемый теорией АТД, состоит в отказе от реальности ради менее грандиозного, но более аппетитного яства, - представляющего множество абстракций, характеризующих операции, доступные клиентам, и их формальные свойства. (На этом построен девиз модельера АТД - не говорите мне, кто вы, скажите, чем вы обладаете.) Мы никогда не претендуем на то, что рассмотрели все возможные операции и свойства: мы выбрали некоторые из них, подходящие для наших целей и отбросили остальные. Моделирование означает отсечение лишнего.

В идеальном случае программная система приходится соответствующей реальности лишь "седьмой водице на киселе" (*cousin twice removed*).

Работа с объектами и ссылками

Вернемся к более приземленным проблемам и рассмотрим, как программные системы работают с объектами, как создают и используют гибкие структуры данных.

Динамическое создание и повторное связывание

Что не было показано при описании структуры объектов периода выполнения, так это в высшей степени динамичная природа настоящей ОО-модели. Статическая и ориентированная на стеки политика управления объектами характерна для языков уровня Fortran и Pascal соответственно. Противоположной является политика в настоящем ОО-окружении, позволяющая создавать объекты в период выполнения, когда в них возникает потребность. Какому образцу (типу) соответствуют создаваемые объекты, как правило, невозможно предсказать при статической проверке программного текста.

В начальном состоянии, как описано в предыдущей лекции, создается единственный корневой объект. Затем система повторно выполняет операции создания новых объектов, связывает изначально пустые ссылки с этими объектами, делает ранее присоединенные ссылки пустыми или присоединяет их к другим объектам. Динамическая и непредсказуемая природа этих операций обеспечивает гибкий подход и позволяет поддерживать динамические структуры данных, необходимые для реализации сложных алгоритмов и моделирования быстро меняющихся свойств внешних систем.

Следующий раздел посвящен механизмам, необходимым для создания объектов и манипулирования их полями, в частности, ссылками.

Инструкция создания

Рассмотрим создание экземпляра класса BOOK3. Это возможно только с помощью подпрограммы класса, являющегося клиентом BOOK3, как, например:

```
class QUOTATION feature
  source: BOOK3
  page: INTEGER
  make_book is
    -- Создание объекта BOOK3 и присоединение его к source.
    do
      ... См. ниже ...
    end
  end
```

Этот класс описывает цитирование книги в других публикациях. Он содержит два поля: ссылку на цитируемую книгу и число страниц, содержащих ссылки на нее.

Механизм создания экземпляра QUOTATION (скоро он будет рассмотрен) предусматривает инициализацию всех его

полей. Правило инициализации по умолчанию определяет, что любое ссылочное поле (в данном примере - поле, соответствующее атрибуту source) после инициализации должно содержать пустую ссылку. Другими словами, создание объекта типа QUOTATION не сопровождается созданием объекта типа BOOK3.

Ссылка остается пустой, пока над ней не будут выполнены некоторые действия, - таково общее правило. Изменить значение ссылки можно, создав, например, новый объект. В процедуре make_book это делается следующим образом:

```
make_book is
    -- Создание объекта BOOK3 и присоединение его к source.
    do
        create source
    end
```

Это иллюстрация простейшей формы инструкции создания: **create x**, где x - атрибут охватывающего (enclosing) класса или, как будет показано позже, локальная сущность охватывающей подпрограммы. Далее эта базовая нотация будет расширена.

Сущность x, именованная в инструкции (в данном примере source), называется **целью (target)** инструкции создания.

Данная форма известна как "базовая инструкция создания". Другая форма, включающая вызов процедуры класса, скоро появится. Вот точное определение действия базовой инструкции создания:

Результат базовой инструкции создания

Эффект инструкции создания вида **create x**, где тип цели x является ссылочным типом, основанном на классе C, состоит в выполнении трех следующих действий:

- (C1) Создание нового экземпляра C(набора полей, по одному на каждый атрибут C). Пусть ОС - это новый экземпляр.
- (C2) Инициализация каждого поля ОС соответствующими стандартными значениями по умолчанию.
- (C3) Присоединение значения x (ссылки) к ОС.

На этапе С1 создается экземпляр C. На этапе С2 устанавливаются предопределенные значения всех полей, зависящие от типа соответствующего атрибута:

Значения по умолчанию при инициализации

Для ссылок значение по умолчанию - пустая ссылка.

Для полей BOOLEAN значение по умолчанию - False.

Для полей CHARACTER значение по умолчанию - символ null.

Для чисел (типов INTEGER, REAL или DOUBLE) значение по умолчанию - ноль в соответствующем данному типу представлении.

Итак, для цели source типа BOOK3 в соответствии с объявлением класса

```
class BOOK3 feature
    title: STRING
    date, page_count: INTEGER
    author: WRITER
end
```

результатом инструкции создания **create source**, выполняемой при вызове процедуры make_book класса QUOTATION, будет объект изображенный на [рис.8.10](#).

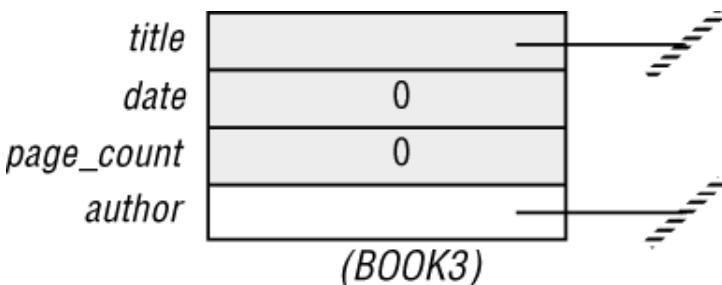


Рис. 8.10. Созданный и инициализированный объект

После инициализации значения целочисленных полей равны нулю. Ссылочное поле author и поле title типа STRING, содержат пустые ссылки. Тип STRING, о котором ничего не говорится в правилах инициализации, двойственен, - фактически являясь ссылочным типом, он рассматривается во многих ситуациях как базовый тип. (О строках см. [лекцию 13](#))

Общая картина

Важно проследить за последовательностью происходящих событий. Для рассмотренного выше экземпляра BOOK3 происходит следующее:

- (B1) Создан экземпляр QUOTATION. Пусть Q_OBJ - этот экземпляр и имеется сущность a, значение которой ссылка, присоединенная к Q_OBJ.
- (B2) Спустя некоторое время после B1 вызов вида a.make_book приводит к выполнению процедуры make_book с Q_OBJ в качестве цели.

Правомерен вопрос - как будет создан сам Q_OBJ (шаг B1)? Это, оставляя проблему, отодвигает ее вглубь. Но к этому моменту мы уже знаем ответ на этот вопрос: все возвращается к первопричине - Большому Взрыву. Для выполнения системы необходимо снабдить ее корневым классом и процедурой этого класса, названной процедурой создания. В начале выполнения автоматически создается один объект - корневой объект - экземпляр корневого класса. Корневой объект является единственным объектом, не создаваемым инструкциями программного текста; он приходит извне, как *objectus ex machine* (объект от машины). Начав с одного, провидением посланного объекта, далее уже программа может создавать объекты нормальным путем через подпрограммы, выполняющие инструкции создания. Первой выполняемой подпрограммой является процедура создания, автоматически применяемая к корневому объекту. Не всегда, но чаще всего она содержит по крайней мере одну инструкцию создания, что в предыдущей лекции называлось началом грандиозного фейерверка, процесса, создающего столько новых объектов, сколько нужно текущему выполнению.

Для чего необходимо явное создание объектов?

Объекты создаются явным образом. Объявление сущности

b: BOOK3

не влечет за собой создание объекта во время выполнения, это происходит, когда некий элемент системы выполнит операцию

create b

Это может показаться удивительным. Разве объявления b недостаточно для создания объекта? Что хорошего в объявлении, если объект не создается?

Достаточно минуты размышления для понимания того, что разделение объявления и создания объекта является единственно разумным решением.

Первый аргумент - **reductio ad absurdum** (доведение до абсурда). Предположим, что начата обработка объявления и немедленно создается соответствующий объект. Но это экземпляр класса BOOK3, имеющий атрибут author ссылочного типа WRITER, значит поле author - ссылка, для которой опять нужно создавать объект. Этот объект вновь содержит ссылочные поля, требуется опять делать то же самое и начинается длинный путь рекурсивного создания объектов.

Этот аргумент еще более убедителен для таких классов как PERSON1, содержащих ссылки на себя:

```
class PERSON1 feature
  name: STRING
  loved_one, landlord: PERSON1
end
```

Появление каждого экземпляра PERSON1 повлечет за собой создание двух других таких объектов (соответствующих loved_one и landlord) и начнется бесконечный цикл. Такие прямые или косвенные циклические ссылки не экзотика - они часто встречаются и необходимы.

Другой аргумент следует из обсуждения роли объектной технологии как мощного метода моделирования. Если для каждого ссылочного поля будет создаваться новый объект, то не было бы возможности выделить пустые ссылки и множественные ссылки на один и тот же объект. И то, и другое необходимо для реалистичного моделирования систем:

- В некоторых случаях требуется, чтобы ссылка не была связана ни с каким объектом. Примером может служить пустая ссылка author для обозначения неизвестного автора.
- В других случаях, в соответствии с моделью две ссылки должны быть присоединены к одному объекту. (См. [рис.8.7](#)) В примере с циклическими ссылками присутствовали поля loved_one двух персон PERSON1, присоединенные к одному и тому же объекту. Не имело бы смысла создание своего объекта для каждого из этих полей. Все, что требуется, - это операция присваивания (рассмотрена далее в этой лекции) для присоединения ссылки к уже существующему объекту. В еще большей степени это соображение применимо для ссылки на себя (поле landlord верхнего объекта в том же примере).

Механизм управления объектами никогда не присоединяет ссылку неявно. Он создает объекты через инструкции создания (или операции клонирования, тоже явные), инициализируя их ссылочные поля пустыми ссылками. Эти поля, в свою очередь, могут стать присоединенными к объектам, только в результате явных операций над этими полями.

В дискуссии о наследовании будет показано, что инструкция создания может использовать синтаксис **create {T}x** для создания объекта, чей тип T является наследником типа объявленного для x. (Полиморфное создание, см. [лекцию 14](#))

Процедуры создания

Все до сих пор рассмотренные инструкции создания основывались на инициализации по умолчанию. В некоторых случаях инициализация, определенная в языке, может нас не устраивать - хотелось бы обеспечить создаваемый объект специфической информацией. В этом предназначение процедур создания.

Перекрытие инициализации по умолчанию

Для использования инициализации, отличной от предопределенной умолчанием, необходимо класс снабдить одной или несколькими процедурами создания. Такие процедуры должны быть перечислены в предложении, начинающимся ключевым словом **creation** в начале класса перед первым предложением feature. Схема такова:

```
indexing
...
class C creation
  p1, p2, ...
feature
  ... Объявления компонент, включая реализацию процедур p1, p2, ...
end
```

Совет, отражающий стиль: в случае класса с единственной процедурой создания - для нее рекомендуется имя **make**. Для классов с двумя и более процедурами создания желателен префикс **make_**, за которым следует квалификатор, как в следующем примере POINT. (См. "Правильный выбор имен", [лекция 8](#) курса "Основы объектно-ориентированного проектирования")

Соответствующая инструкция создания в этих случаях имеет другую форму:

```
create x.p (...)
```

где **p** одна из процедур создания перечисленных в разделе **creation**, и в круглых скобках (...) перечисляются фактические аргументы **p**. Результатом является создание объекта с использованием значений по умолчанию, как и ранее, а затем вызов **p** с заданными аргументами. Такая инструкция является комбинацией инструкции создания и вызова процедуры и называется **порождающим вызовом (creation call)**. (Оригинальная версия класса POINT приведена в [лекции 7](#))

В качестве примера добавим две процедуры создания в класс POINT, что позволит клиентам при создании новой точки указывать ее начальные координаты - декартовы или полярные. Введем процедуры создания: **make_cartesian** и **make_polar**. Вот схема:

```
class POINT1 creation
  make_cartesian, make_polar
feature
  ... Компоненты из предыдущей версии класса:
  x, y, ro, theta, translate, scale, ...
feature {NONE} - Этот вариант экспорта рассмотрен ниже.
  make_cartesian (a, b: REAL) is
    -- Инициализация точки с декартовыми координатами a и b.
    do
      x := a; y := b
    end
  make_polar (r, t: REAL) is
    -- Инициализация точки с полярными координатами r и t.
    do
      x := r * cos (t); y := r * sin (t)
    end
end
```

Для такого класса клиент будет создавать точки инструкциями вида:

```
create my_point.make_cartesian (0, 1)
create my_point.make_polar (1, Pi/2)
```

В обоих случаях создается точка с одинаковыми координатами в предположении, что константа **Pi** имеет общепринятый смысл. Вот правило, определяющее эффект порождающего вызова. Первые три пункта правила такие же, как и для базисной формы, приведенной ранее:

Эффект порождающего вызова

Рассмотрим порождающий вызов в форме **create x.p(...)**.

Пусть тип цели **x** это ссылочный тип, основанный на классе **C**, **p(...)** - процедура создания класса **C**, с заданным списком фактических аргументов. Эффект вызова состоит в выполнении следующих четырех шагов:

- (C1) Создание нового экземпляра **C** (набора полей, по одному на каждый атрибут **C**). Пусть **OC** - это новый экземпляр.
- (C2) Инициализация каждого поля **OC** соответствующими стандартными значениями по умолчанию.
- (C3) Присоединение значения **x** (ссылки) к **OC**.

- (C4) Вызов процедуры `r` с заданными аргументами и с целевым объектом ОС.

Статус экспорта процедур создания

Для двух процедур создания, объявленных в классе `POINT1`, предложение `feature` имело вид `feature {NONE}`. Это означает, что эти процедуры закрыты для обычных вызовов, но остаются открытыми для порождающих вызовов. Только что представленные два примера порождающих вызовов являются корректными, но нормальные вызовы, например `my_point.make_cartesian (0, 1)` или `my_point.make_polar (1, Pi/2)` некорректны, так как процедуры недоступны клиентам со статусом обычных компонентов.

Решение о закрытости процедур означает, что мы не хотим после создания точки дать возможность клиентам прямого доступа к изменению их координат, хотя они могут делать это через другие процедуры класса, например такие, как `translate` и `scale`. Конечно, это лишь одна из возможных политик, вполне разумно экспортовать процедуры создания клиентам, придавая им дополнительно статус обычных процедур.

Для процедур создания можно установить выборочный статус порождающего вызова. Для этого достаточно в предложении `creation` перечислить классы, которым разрешается создавать объекты:

```
class C creation {A, B, ...}
    p1, p2,
    ...
    ...
```

Этот прием применяется значительно реже, чем задание статуса экспорта этих процедур как обычных компонентов класса в предложении `feature`. Важно помнить, что статус экспорта порождающего вызова и статус экспорта обычного вызова не зависят друг от друга, они устанавливаются независимо в разных предложениях.

Правила, применимые к процедурам создания

Две формы инструкций создания: `create x` и `create x.p (...)`, являются взаимно исключающими. Если в классе задано предложение `creation`, то допускается только порождающие вызовы, базовая форма создания считается в этом случае недопустимой и отвергается компилятором.

Это соглашение кажется на первый взгляд странным, но смысл его становится понятным при рассмотрении требований согласованности объекта. Объект - это не просто набор полей, это реализация АТД, так что поля его будут согласованы только если они удовлетворяют ограничениям, заданным спецификацией АТД. Вот типичный пример. Предположим, что объект задает некоторую личность с двумя полями - год рождения и возраст. Понятно, что согласованность этого объекта не допускает независимых значений этих полей, они связаны вполне определенным соотношением, которое может быть частью спецификации. Инструкция создания **обязана всегда** производить на свет согласованный объект. Базовая форма этой инструкции применима только в тех частных и довольно редких случаях, когда стратегия умолчания удовлетворяет требованиям согласованности. Во всех остальных случаях в классе требуется определять процедуры создания, что автоматически запрещает использование базовой инструкции создания.

В тех редких случаях, когда инициализация по умолчанию допустима, поскольку удовлетворяет инварианту класса, может появиться желание включить ее в состав процедур создания. Для этого необходимо в список процедур создания включить специальную процедуру, наследуемую от класса ANY, с именем `nothing`. Как следует из ее имени, эта процедура без аргументов ничего не делает, имея пустое тело. Вот пример подобного включения:

```
class C creation
    nothing, some_creation_procedure, some_other_creation_procedure...
feature
    ...
    ...
```

Хотя по-прежнему базовая инструкция создания является некорректной в этом случае, но теперь клиент имеет возможность создать объект порождающим вызовом `create x.nothing`

В заключение обратите внимание на специальное правило - теперь появилась возможность определить класс, клиенты которого не смогут создавать экземпляры класса. Вот пример того, как этого можно добиться:

```
class C creation
    -- Здесь ничего не указано!
feature
    ...
    Текст класса, как обычно ...
end
```

Класс имеет предложение `creation`, но пустое. Это означает согласно установленным правилам, что создавать объекты можно только с помощью процедур создания, которых нет, что означает невозможность создания объектов.

Если ограничиться ОО-механизмом, рассмотренным до сих пор, то такая возможность запрета на создание объектов класса кажется надуманной. Знакомство с наследованием придает ей смысл, - иногда желательно использовать класс только в интересах наследования. Эта цель и может быть достигнута таким способом. Заметьте, этого же можно добиться, сделав класс абстрактным (отложенным). Но в этом случае у класса должен быть, по крайней мере, один отложенный метод. Иногда разумно полностью определить методы, но не включить в класс процедуры создания.

Процедуры создания и перегрузка

В продолжение обсуждения полезно сравнить применяемый подход с некоторыми процедурами создания с подходом, используемым в языках C++/Java. В этих языках применяется техника, основанная на перегрузке. Суть ее такова: все процедуры создания, называемые конструкторами, перегружены - они имеют одно и то же имя, совпадающее с именем класса. Конструкторы должны иметь различную сигнатуру (отличаться числом и/или типами аргументов).

Как мы видели при обсуждении перегрузки, сигнтура не является подходящим критерием распознавания. Например: конструкторы `make_cartesian` и `make_polar` имеют одинаковую сигнтуру, так что придется вводить искусственный аргумент в один из конструкторов, чтобы стало возможным отличить вызов нужного конструктора.

Наша техника кажется предпочтительнее во всех отношениях. Минимум усилий (никаких процедур создания), если инициализация по умолчанию применима; при желании позволяет предотвратить создание объектов клиентами; вводит столько процедур создания, сколько необходимо, не создавая никаких коллизий; каждая процедура создания имеет собственное имя, облегчая понимание ее предназначения, например `make_polar`.

Еще о ссылках

Модель периода выполнения определяет важную роль ссылок. Рассмотрим некоторые их свойства, в частности, понятие пустой (`void`) ссылки и связанные с ней проблемы.

Состояния ссылок

Ссылка может находиться в одном из двух состояний - она может быть пустой или присоединенной. Мы уже видели, что изначально ссылка всегда находится в состоянии `void` и может стать присоединенной благодаря созданию объекта. Вот как выглядит более полная картина, показывающая все возможности перехода между состояниями:

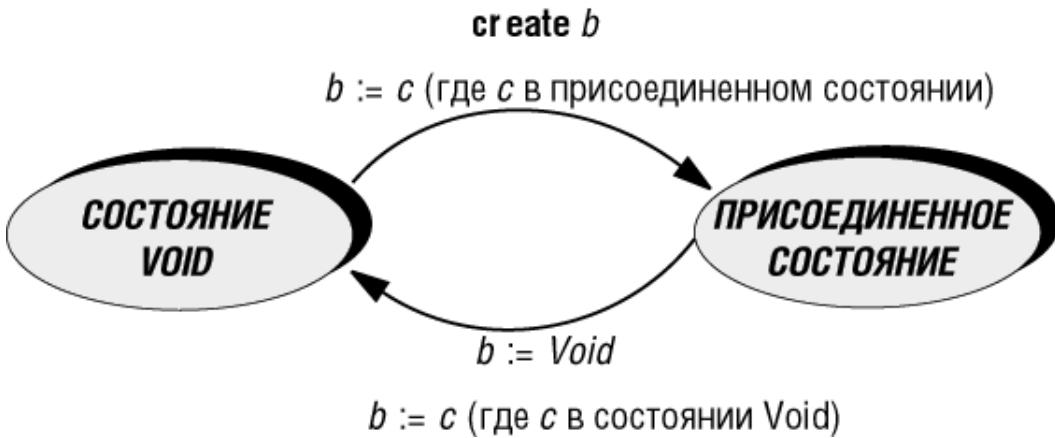


Рис. 8.11. Возможные состояния ссылки и переходы

Помимо создания, ссылка может изменять состояние в результате присваивания. Проверьте себя, понимаете ли вы разницу между тремя понятиями - объектом, ссылкой и сущностью:

- "Объект" - это понятие периода выполнения; любой объект является экземпляром класса, создавается во время выполнения системы и представляет собой набор полей.
- "Ссылка" - это понятие периода выполнения. Значение ссылки либо `void`, либо она присоединена к объекту. Точное определение "присоединения" уже появлялось. Присоединенная ссылка однозначно идентифицирует объект.
- "Сущность" - это статическое понятие, применимое к программному тексту, - это идентификатор в тексте класса, представляющий значение или множество значений в период выполнения. Сущностями являются обычные переменные, именованные константы, аргументы подпрограмм и результаты функций.

Если `b` - сущность ссылочного типа, то ее значением в период выполнения является ссылка, которая может быть присоединена к объекту 0. В этом случае говорим, что сущность `b` присоединена к 0.

Вызовы и пустые ссылки

В большинстве случаев мы ожидаем, что ссылка присоединена к объекту, хотя допустимо иметь и пустые ссылки. Ссылки `void` играют важную роль в ОО-модели вычислений. В предыдущей лекции подробно разбиралась фундаментальная операция ОО-модели - вызов компонента - применение к экземпляру класса компонента этого класса. Вот как это пишется:

```
some_entity.some_feature (arg1, ...)
```

Для корректного выполнения вызова сущность `some_entity` должна быть присоединена к нужному целевому объекту. Если случится, что `some_entity` ссылочного типа и имеет значение `void`, то вызов не может быть обработан, так как необходим целевой объект.

ОО-система никогда в момент выполнения вызывать компонент с целевым объектом `void`. Результатом подобного вызова будет исключение (exception). (Исключения и их обработка будут изучаться в [лекции 12](#))

Было бы прекрасно, если бы компилятор проверял программный текст и гарантировал, что подобные события не

встретятся в период выполнения, точно также как он проверяет отсутствие несовместимости типов, используя соответствующие правила типизации. К сожалению, такая цель недостижима для компиляторов, если только не накладывать жестких ограничений на язык. Так что ответственность за то, чтобы все вызовы имели присоединенный целевой объект, возлагается на разработчика. Конечно, есть простой способ - окружать все вызовы тестом:

```
if "x не void" then
    x.f (...)

else
...
end
```

Этот прием не применим в качестве универсального требования, хотя и может использоваться, когда из контекста не следует, что целевой объект не будет пуст.

Вопрос о не пустоте ссылок является частью вопроса корректности ПО. Для проверки корректности системы необходимо проверить, что нет вызовов, применимых к void ссылкам, и что все утверждения (изучаемые в последующих лекциях) удовлетворяются в соответствующий момент выполнения. Проверка не пустоты, также как и проверка утверждений, могла бы проводиться специальным автоматом - верификатором, встроенным в компилятор или являющимся независимым средством. В отсутствие такого механизма подобные нарушения приведут к ошибке периода выполнения - исключению. Разработчики могут защитить ПО двумя путями:

- В процессе разработки использовать все доступные приемы, позволяющие избежать ошибочных ситуаций, применяя, например, средства, позволяющие выполнять частичную проверку.
- Если остаются малейшие сомнения, то поставлять ПО с механизмом обработки исключений.

Операции над ссылками

Мы уже знакомы с одним из способов изменения значения ссылки x: использование инструкции создания в форме `create x`, позволяющей создать новый объект и присоединить его к ссылке. Имеются и другие интересные операции, доступные при работе со ссылками.

Присоединение ссылки к объекту

Классы, появляющиеся в этой лекции, не имели подпрограмм - у них были только атрибуты. Как отмечалось, такие классы почти бесполезны, так как у них нет способа изменить значение атрибутов. Необходимы способы модификации ссылок, не использующие при этом инструкций в духе языков Pascal-C-Java-C++, подобных присваиванию: `my_beloved.loved_one := me` (напрямую изменяющих у объекта поле `loved_one`), что нарушает принцип скрытия информации и синтаксически некорректно в нашей нотации.

Для модификации полей объекта клиент обязан вызвать подпрограмму, специально поставляемую разработчиком класса для этих целей. Давайте включим в класс PERSON1 процедуру, позволяющую модифицировать поле `loved_one`. Вот результат:

```
class PERSON2 feature
  name: STRING
  loved_one, landlord: PERSON2
  set_loved (l: PERSON2) is
    -- Присоединить поле loved_one текущего объекта к объекту l.
    do
      loved_one := l
    end
  end
```

Процедура `set_loved` присваивает ссылочному полю `loved_one` текущего экземпляра PERSON2 значение другой ссылки l. Ссылочное присваивание (левая и правая части являются ссылками) присваивает значение источника (правой части) целевой ссылке (слева).

Эффект ссылочного присваивания очевиден: целевая ссылка становится присоединенной к объекту, к которому присоединен источник - или становится void, если такое значение имеет источник. Предположим, например, что мы начинаем с ситуации, изображенной на [рис.8.12](#), где поля `landlord` и `loved_one` всех изображенных объектов пока пусты.

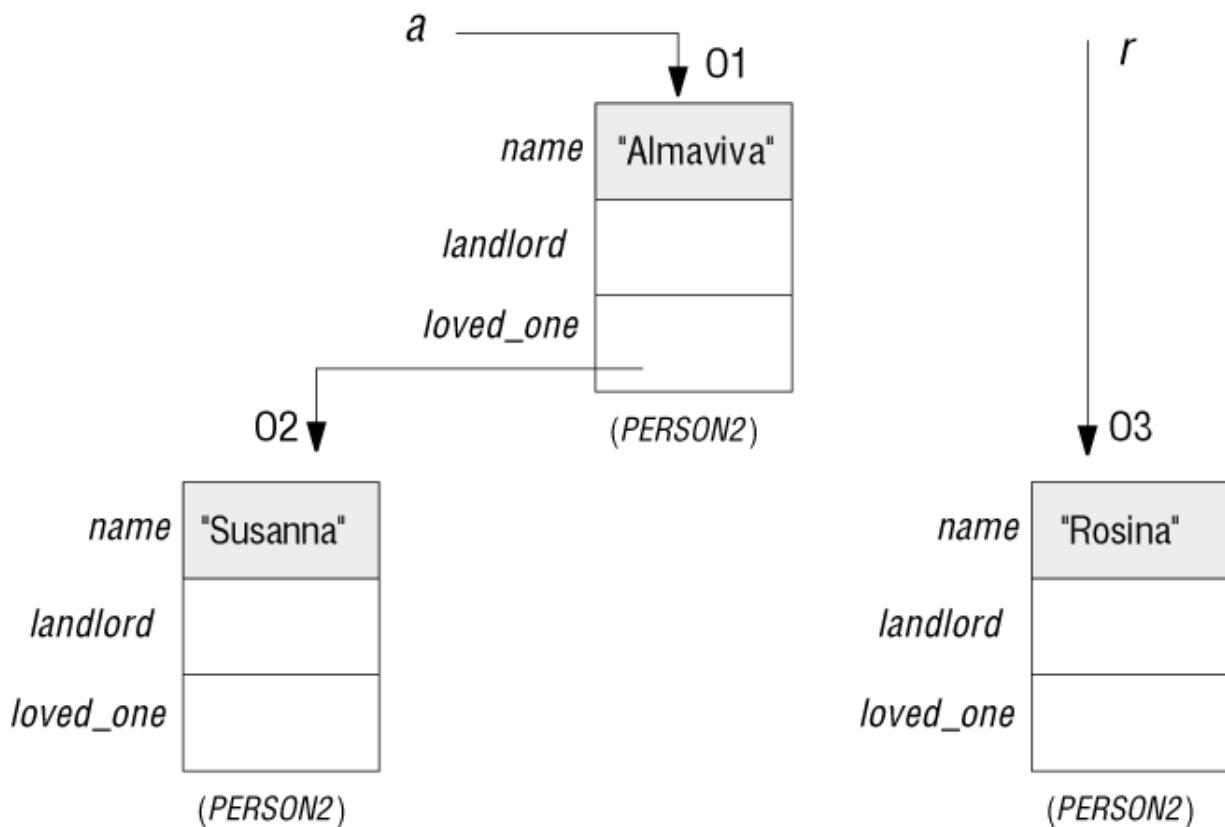


Рис. 8.12. Перед присваиванием ссылке

Предположим, что выполняется вызов процедуры:

```
a.set_loved (r)
```

Сущность *a* присоединена к объекту 01, а сущность *r* - к 03. В результате выполнения процедуры *set_loved* выполнится присваивание:

```
loved_one := 1
```

Здесь в роли текущего объекта выступает объект 01, сущности 1 и *r* имеют одинаковое значение - ссылки на объект 03. В результате изменится значение поля *loved_one* объекта 01 - ссылка присоединится к другому объекту 03, как показано на следующем рисунке:

Если бы *r* было пустой ссылкой, то такой же в результате присваивания стала бы и ссылка в поле *loved_one* объекта 01.

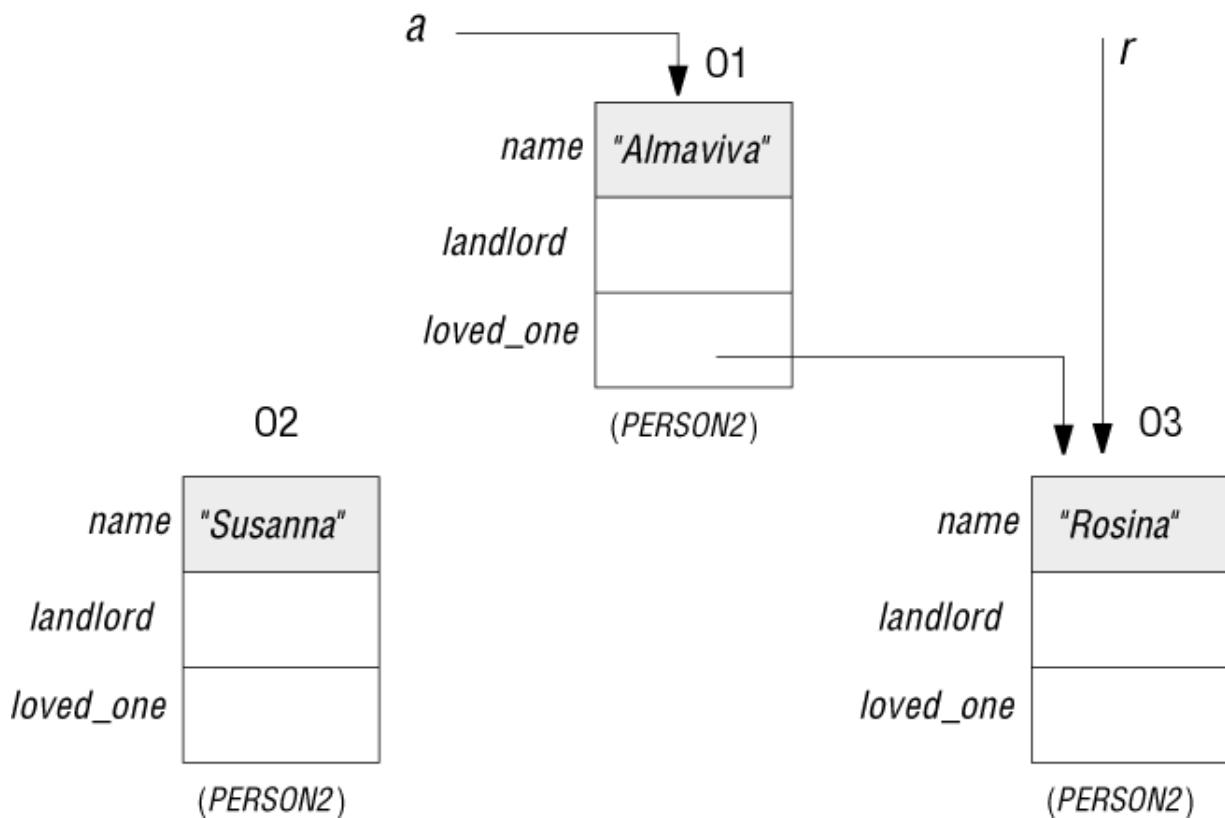


Рис. 8.13. После присваивания ссылки

Сравнение ссылок

Наряду с присваиванием возникает необходимость и в тесте - проверить, присоединены ли две ссылки к одному и тому же объекту. Для этого есть оператор эквивалентности `=`.

Если `x` и `y` - сущности ссылочного типа, то выражение:

`x = y`

истинно тогда и только тогда, когда обе ссылки пусты или присоединены к одному и тому же объекту. Противоположный оператор "не эквивалентно" записывается как `/=`.

Выражение:

`r = a.loved_one`

истинно в ситуации, представленной на [рис.8.5](#) и ложно для ситуации [рис.8.12](#).

Заметьте, в операциях эквивалентности сравниваются ссылки, а не объекты, к которым они присоединены. Так что если две ссылки присоединены к разным объектам, результатом операции эквивалентности будет `false`, даже если объекты имеют все поля с одинаковыми значениями. Операции, сравнивающие объекты, а не ссылки, будут введены позднее.

Значение void

Получить пустую ссылку достаточно просто - при инициализации по умолчанию все ссылки пусты. Однако удобно иметь специальное имя для ссылки, доступной в любом контексте и имеющей значение `void`. Предопределенный компонент

`Void`

играет эту роль.

Обычно компонент `Void` используется в тестах, проверяющих пустоту ссылок:

```
if x = Void then ...
```

и для того, чтобы присвоить некоторой ссылке это значение:

```
x := Void
```

В результате последнего присваивания происходит отсоединение ссылки от объекта, и она становится пустой. Эта ситуация показана на следующем рисунке:

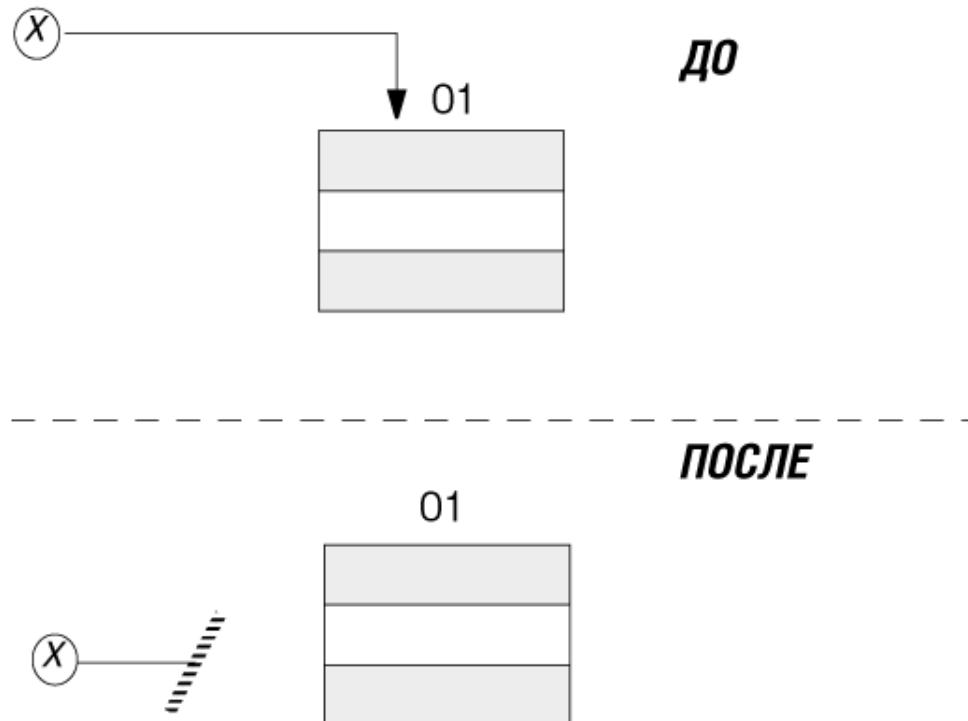


Рис. 8.14. Отсоединение ссылки от объекта

Присваивание `Void` ссылке не оказывает никакого влияния на объект, ранее присоединенный к ссылке, - разрывается только связь между ссылкой и объектом. Было бы некорректно рассматривать эту операцию как освобождение памяти, так как другие ссылки могут продолжать быть связанными с объектом (на рисунке **X** может быть отсоединен от объекта **01**, но другие ссылки могут быть еще присоединены к нему). Об управлении памятью смотри следующую лекцию.

Клонирование и сравнение объектов

Ссыльное присваивание приводит к тому, что две или несколько ссылок присоединяются к одному объекту. Иногда необходима другая форма присваивания, в результате которой мы хотим получить не копию ссылки, а копию объекта. Эта цель достигается при вызове функции клонирования `clone`.

Если у присоединено к объекту **OY**, выражение

`clone (y)`

означает создание нового объекта **OX**, такого, что он имеет те же поля, что и **OY**, и все соответствующие поля имеют идентичные значения. Если **y** равно `void`, то значение `clone (y)` также `void`.

Скопировать присоединенный к объекту и связать копию со ссылкой **x** позволяет присваивание:

[1]
`x := clone (y)`

Вот иллюстрация этого механизма:

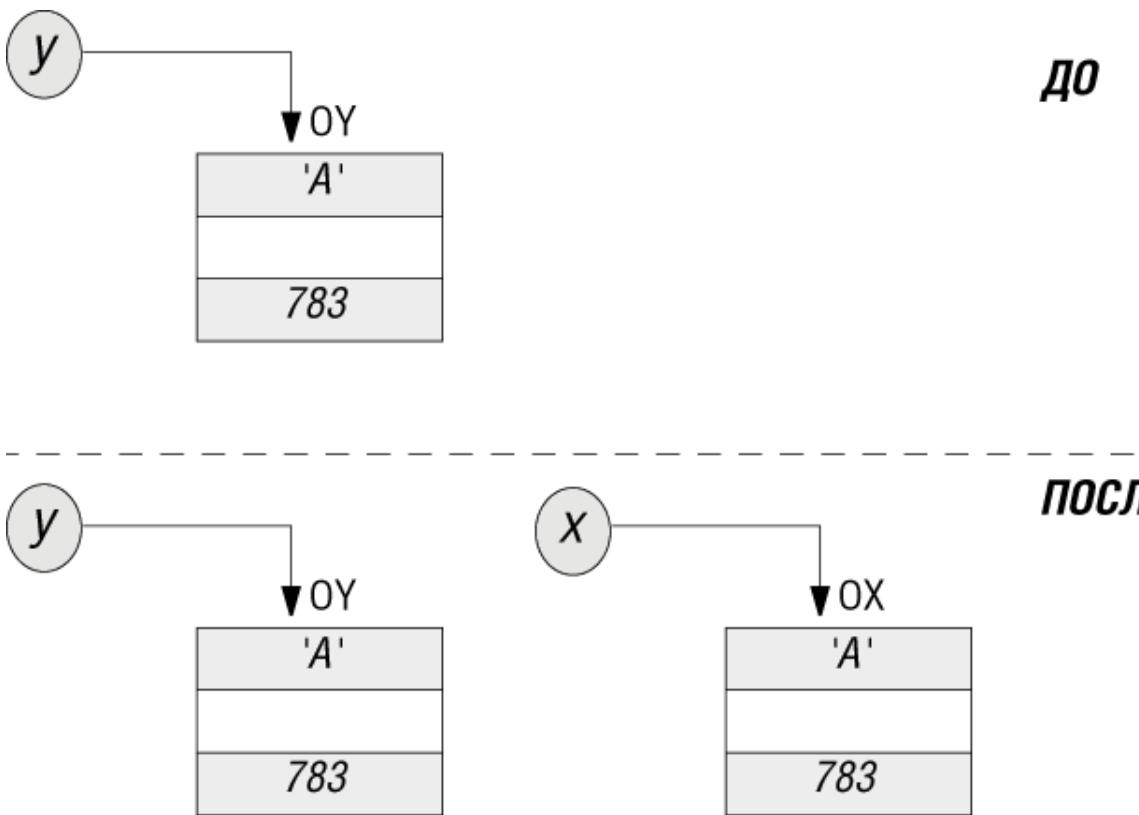


Рис. 8.15. Клонирование объекта

Наряду со сравнением ссылок необходим механизм, позволяющий сравнивать объекты. Этой цели служит функция `equal`. Вызов:

```
equal (x, y)
```

возвращает значение `true`, если и только если *x* и *y* оба имеют значение `void` или присоединены к двум объектам с идентичными полями. После выполнения присваивания с клонированием [1], состояние, непосредственно следующее за присваиванием, удовлетворяет `equal (x, y)`.

Возможно, вы удивляйтесь, почему у функции `clone` есть аргумент, а у функции `equal` - их два. Для ОО-стиля характерен квалифицируемый вызов в форме: *y.twin* и *x.is_equal (y)*. Ответ появится в разделе обсуждения, но это будет еще не скоро, так что попытайтесь догадаться сами.

Копирование объектов

Функция `clone` создает новую копию существующего объекта. Иногда целевой объект уже существует, и все, что необходимо, это скопировать значения полей. Процедура `soru` выполняет эту работу. Она вызывается обычным образом:

```
x.soru (y)
```

Сущности *x* и *y* должны быть одного и того же типа; эффект от выполнения - копирование полей объекта, присоединенного к *y*, в соответствующие поля объекта, присоединенного к *x*.

Как и во всех вызовах компонента, вызов `soru` требует, чтобы целевой объект *x* был не пуст. Дополнительно требуется, чтобы *y* был не пуст. Эта неспособность иметь дело с пустыми ссылками отличает `soru` от `clone`.

Требование не пустоты *y* настолько важно, что должен существовать способ для его формального выражения. Фактически речь идет о более общей проблеме: как программа может задать предусловия на аргументы, передаваемые клиентом при ее вызове. Такие предусловия, являясь частным случаем общего понятия "утверждение" в деталях будут обсуждаться в последующих лекциях. Аналогично, нам хотелось бы уметь выражать в виде постусловия семантическое свойство, отмеченное выше, - результат выполнения `clone` удовлетворяет `equal`.

Процедура `soru` может считаться более фундаментальной, чем функция `clone` в том смысле, что, по меньшей мере, для класса без процедуры создания можно выразить `clone` в терминах `soru` следующим образом:

```
clone (y: SOME_TYPE) is
    -- Void если y равно void; иначе дублировать присоединенный к y объект
    do
        if y /= Void then
            create Result --Правильно только в отсутствие процедур создания
            Result.copy (y)
        end
    end
```

При вызове функции сущность Result автоматически инициализируется в соответствии с общими правилами для атрибутов. Вот почему нет необходимости в ветви else условного оператора: Result инициализируется значением Void , так что результатом функции будет void , если значение у это void .

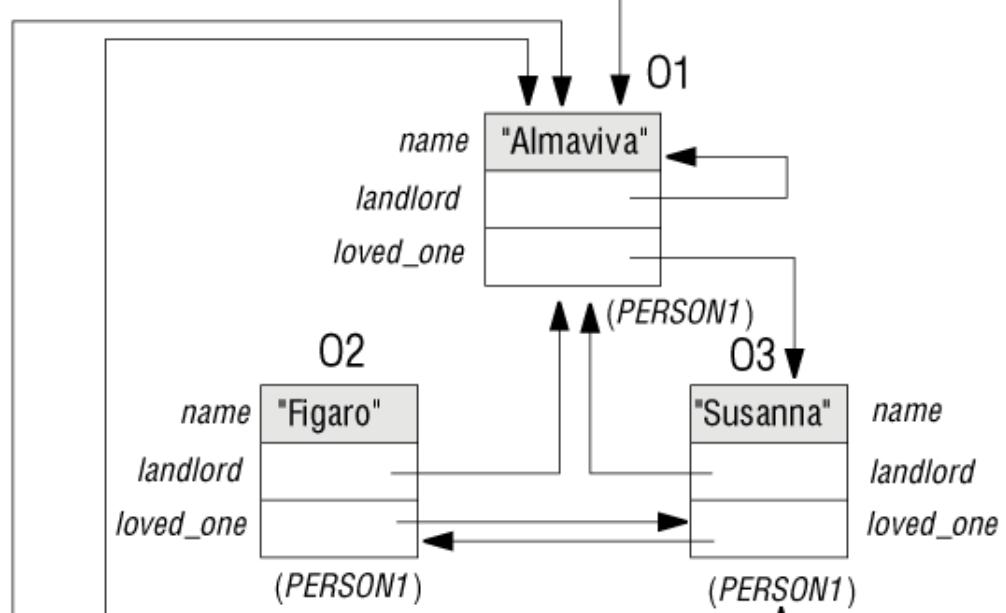
Глубокое клонирование и сравнение

Формы копирования и сравнения, реализуемые подпрограммами clone, equal и copy , называются поверхностными, поскольку они работают с объектами только на первом уровне, никогда не пытаясь следовать вглубь по ссылкам. Возникает необходимость для глубоких вариантов этих операций, рекурсивно дублирующих полную структуру.

Для понимания разницы рассмотрим пример, показанный на [рис.8.16](#). Предположим, что мы начинаем в начальном состоянии A, где сущность A присоединена к объекту 01.

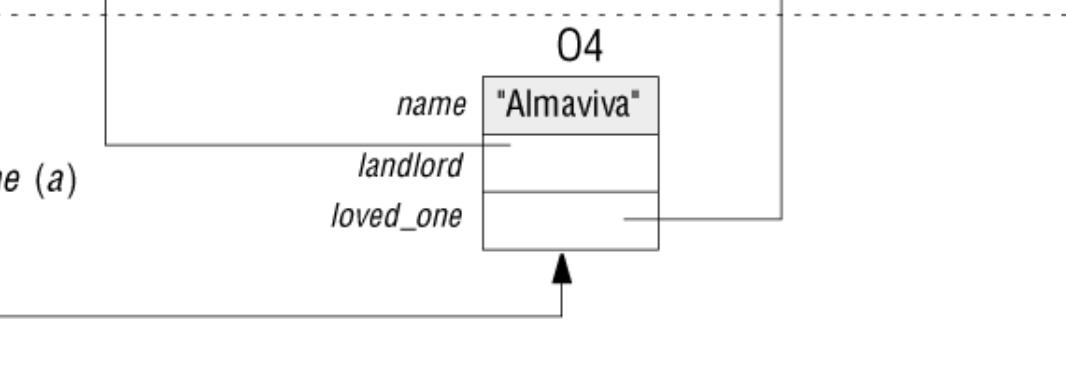
a

A Initial state



b

B Effect of $b := a$



c

C Effect of $c := \text{clone}(a)$

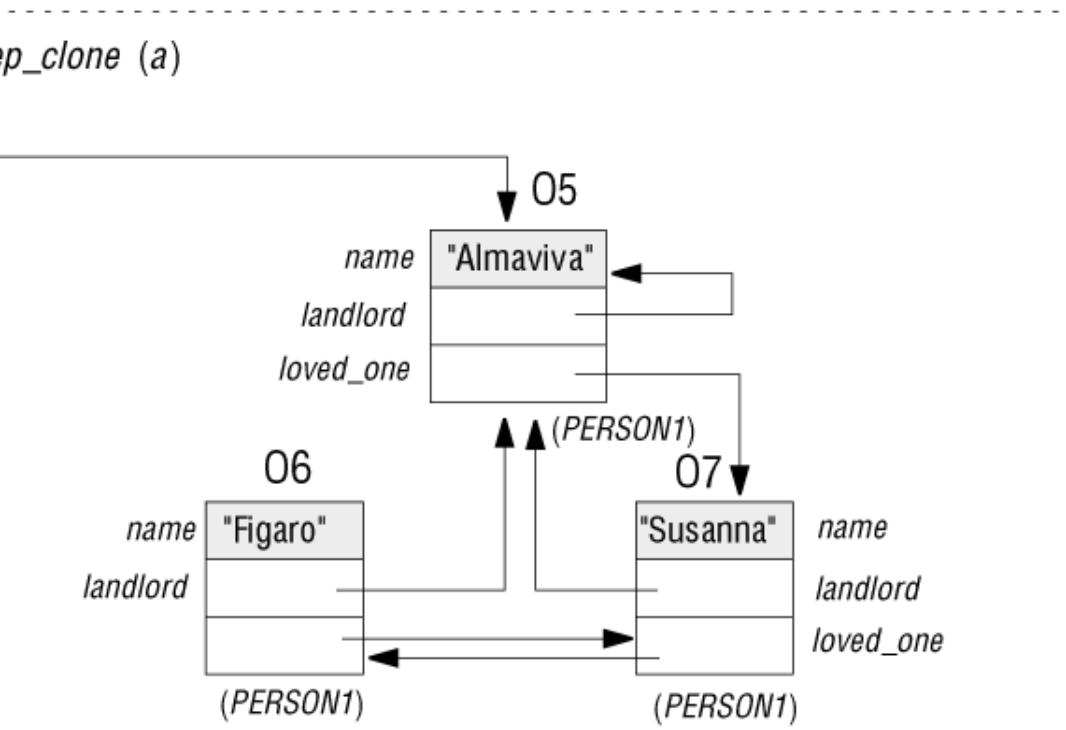


Рис. 8.16. Различные формы присваивания и клонирования

Рассмотрим простое присваивание ссылки:

```
b := a
```

В состоянии В, показанном на рисунке, цель b в результате присваивания присоединена к объекту 01, к которому присоединен источник a. Никаких новых объектов не создается.

Далее рассмотрим операцию клонирования:

```
c := clone (a)
```

Эта инструкция, как показывает раздел С нашего рисунка, создает новый объект 04, с полями, идентичными полям объекта 01. Будут скопированы два ссылочных поля, и значения ссылок будут указывать на те же объекты 01 и 03, как и поля оригинального объекта 01. Но, заметьте, не происходит дублирования самого объекта 03, и никакого другого объекта помимо дублирования 01. По этой причине базисная операция `clone` называется поверхностным клонированием, - она останавливается на первом уровне объектной структуры.

Заметьте, при клонировании исчезли ссылки на себя. Ссылка `landlord` объекта 01 была присоединена к самому объекту 01. У объекта 04 это поле становится ссылкой на оригинал 01.

В некоторых ситуациях вы, возможно, захотите пойти дальше и дублировать структуру рекурсивно без введения разделяемых ссылок. Функция глубокого клонирования `deep_clone` позволяет достичь цели. Процесс создания `deep_clone` (у) рекурсивно следует за всеми ссылочными полями, содержащимися в объекте, дублируя полную структуру. (Если у этого `void`, то и результат будет также `void`.) Эта функция будет, конечно же, правильно обрабатывать циклические ссылочные структуры.

Нижняя часть на рисунке - раздел D - иллюстрирует выполнение этой операции:

```
d := deep_clone (a)
```

В этом случае не появляются новые разделяемые ссылки. Все объекты, прямо или косвенно доступные объекту 01, будут дублированы, создавая новые объекты 05, 06 и 07. Нет никаких связей между старыми объектами (01, 02 и 03) и новыми. Объект 05, дублирующий 01, имеет собственные ссылки на себя.

Так же, как необходимы операции глубокого и поверхностного клонирования, необходимо иметь глубокий вариант эквивалентности. Функция `deep_equal` сравнивает две объектные структуры, определяя их структурную идентичность. В примере, показанном на рисунке, `deep_equal` выполнимо для любой пары из a, b и d. В то же время `equal (a, c)` истинно, поскольку поля объектов 01 и 04 идентичны, `equal (a, d)` - ложно. Фактически `equal` не выполнимо ни для одной пары из d и любого элемента оставшейся тройки. В целом имеют место следующие свойства:

- В результате присваивания `x := clone (y)` или вызова `x.copy (y)`, выражение `equal (x, y)` имеет значение `true` (в случае присваивания это свойство имеет место независимо от того, имеет ли у значение `void`).
- В результате присваивания `x := deep_clone (y)`, выражение `deep_equal (x, y)` имеет значение `true`.

Эти свойства будут отражены в постусловиях соответствующих подпрограмм.

Глубокое хранилище: первый взгляд на сохраняемость

Изучение глубокого копирования и эквивалентности приводит к механизму, обеспечивающему серьезные практические преимущества ОО-метода, естественно, при условии его доступности в среде разработки.

До сих пор обсуждение не затрагивало вопросов ввода и вывода. Но, конечно, ОО-системе необходимо общаться с внешним миром и другими системами. Такое общение предполагает возможность чтения и записи объектов в различные хранилища - файлы, базы данных, сеть.

Для простоты в этом разделе будем предполагать, что проблема сводится к чтению и записи файлов. Для этих операций будем использовать термины "возвратить" (`retrieval`) и "сохранить" (`storage`), адекватные терминам ввод и вывод (`input`, `output`). Изучаемые механизмы должны быть применимыми при использовании других средств коммуникации, например при посылке и получении объектов по сети.

Для экземпляров таких классов, как `POINT` или `BOOK1` сохранение и возвращение объектов не является какой-либо новинкой. Эти классы, используемые в качестве первых примеров этой лекции, имеют атрибуты таких типов, как `INTEGER`, `REAL` и `STRING`, для которых доступно хорошо понятное внешнее представление. Сохранение или возвращение экземпляра такого класса из файла подобно выполнению операций ввода-вывода записей в языке Паскаль или структур языка С. Для этих хорошо известных технических проблем существуют стандартные решения. Поэтому резонно ожидать, что объектам в хорошем ОО-окружении можно предоставить процедуры общего назначения, скажем `read` и `write`, которые подобно `clone` и `copy` будут доступны для всех классов.

Но такие механизмы не могут нас полностью устраивать, поскольку они не управляют главным элементом объектной структуры - ссылками. Так как ссылки могут быть представлены адресами памяти или чем-то подобным, то и для них можно найти подходящее внешнее представление. Это не самая трудная часть проблемы. Сложнее обстоит дело с передачей смысла самих ссылок. Ссылки присоединены к объектам и бесполезны в отсутствие этих объектов. Так что,

как только мы начинаем иметь дело с нетривиальными объектами - объектами, содержащими ссылки, нас перестают устраивать старые механизмы сохранения и возвращения, работающие только со значениями. Механизмы должны вместе с объектом обрабатывать и всех его **связников** (**dependents**) в соответствии со следующим определением:

Определение: связники, прямые связники

Прямыми связниками объекта являются объекты, присоединенные к его ссылочным полям, если таковые имеются.

Связниками объекта являются сам объект и (рекурсивно) связники его прямых связников.

Для структуры объектов, показанной на [рис.8.17](#), было бы бессмысленно сохранить в файле или передать по сети только объект 01. Операция должна включать связников 01 - объекты 02 и 03.

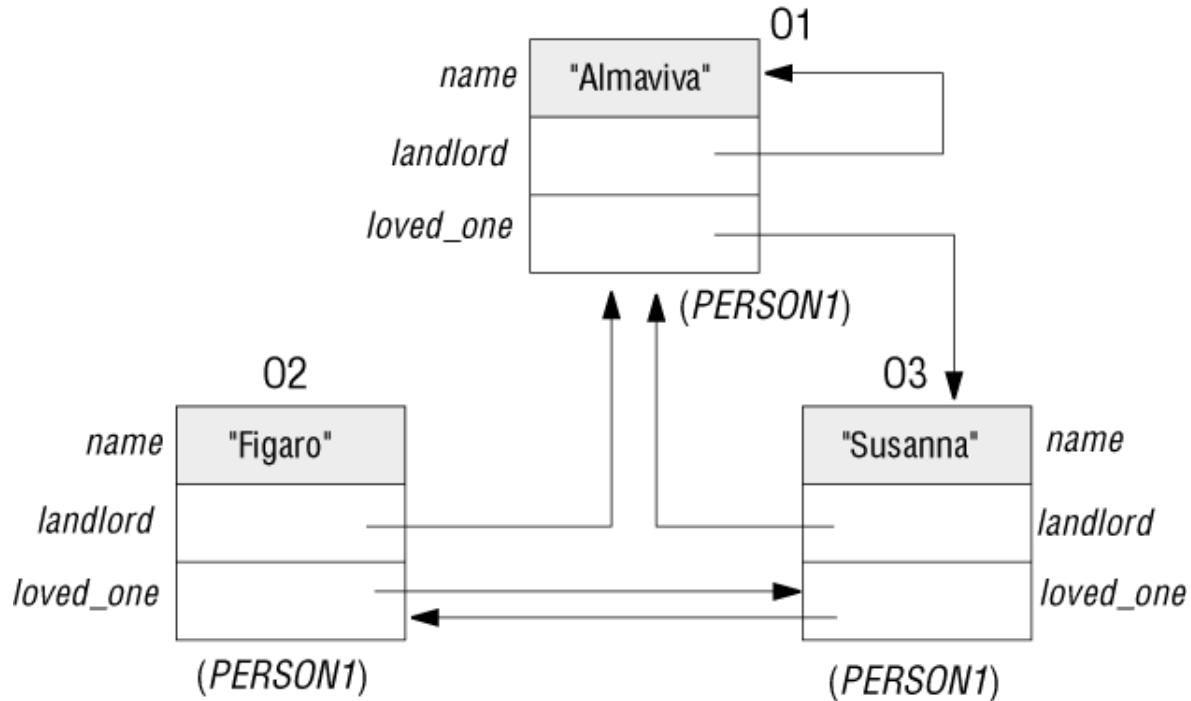


Рис. 8.17. Три взаимно зависимых объекта

В этом примере любой из трех объектов рассматривает оставшиеся два как своих связников. В примере, показанном на [рис.8.18](#), объект W1 можно сохранить независимо, но сохранение объектов B1 или B2 требует сохранения также и W1.

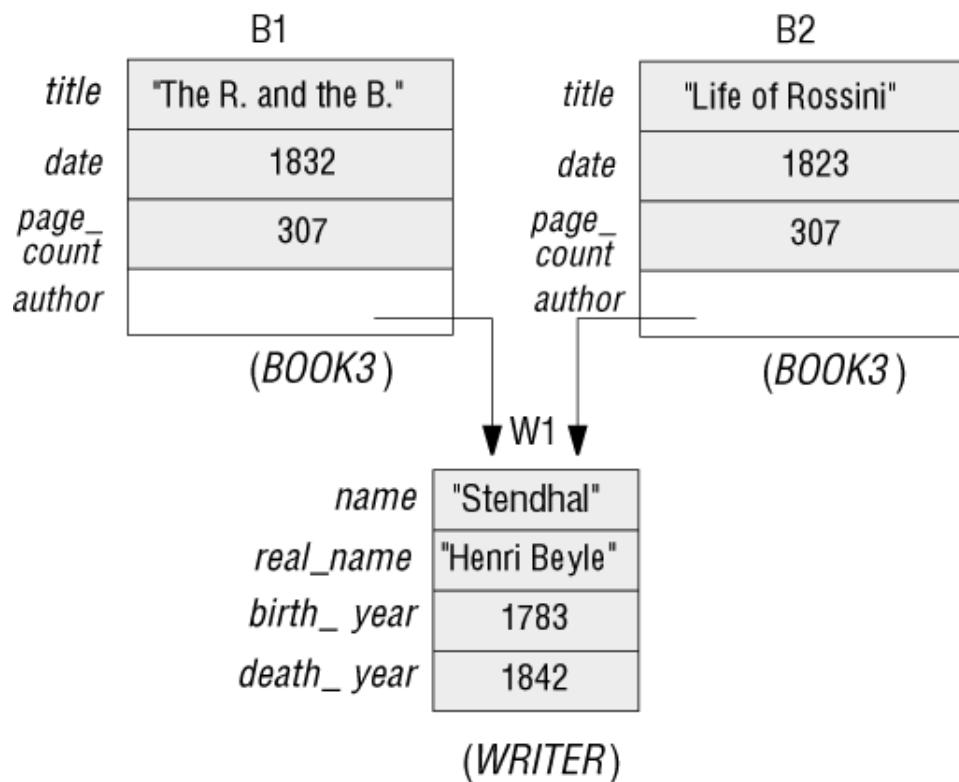


Рис. 8.18. Объекты «Book» и «Writer»

Понятие связников неявно присутствует в представлении `deep_equal`. Вот общее правило:

Принцип Замыкания Сохраняемости (Persistence Closure principle)

Всякий раз, когда механизм сохранения сохраняет объект, он должен сохранять и связников этого объекта.
Всякий раз, когда механизм возвращения возвращает объект, он должен возвращать и связников этого объекта, если они еще не были возвращены.

Базисным механизмом, реализующим эти цели, является библиотечный класс STORABLE, включенный в библиотеку Base. Основными компонентами класса STORABLE являются:

```
store (f: IO_MEDIUM)
retrieved (f: IO_MEDIUM): STORABLE
```

Вызов `x.store (f)` сохраняет в файле, связанном с `f`, объект, присоединенный к `x`, вместе со всеми его связниками. Объект, присоединенный к `x`, называют головным объектом хранимой структуры. Порождающий класс для `x` должен быть потомком STORABLE. Это требуется только для класса головного объекта и не распространяется на связников.

Класс `IO_MEDIUM` это еще один класс библиотеки Base, предназначенный для работы не только с файлами, но и для передачи данных по сети. Очевидно, `f` не должно быть `void`, а присоединенный файл или сетевое устройство должны допускать запись.

Вызов `retrieved (f)` возвращает структуру объектов, идентичную структуре, сохраняемой в `f` предыдущим вызовом `store`. Компонент `retrieved` - это функция, возвращающая в качестве результата ссылку на головной объект возвращаемой структуры объектов.

Механизм STORABLE это наш первый пример важного свойства сохраняемости (persistence) объектов. Объект сохраняется, если он продолжает существовать по окончании очередной сессии работы системы. Класс STORABLE обеспечивает только частичное решение проблемы, накладывая ряд ограничений:

- В сохраняемой и возвращаемой структуре только один объект известен индивидуально - головной объект. Было бы желательно уметь идентифицировать и другие объекты.
- Как следствие, механизм не позволяет выборочное получение объектов, через запросы или по ключу, как это делается, например, в базах данных.
- Вызов `retrieved` воссоздает полную структуру объектов. Это означает невозможность использовать два или более таких вызовов для получения отдельных частей структуры, если только структуры не являются независимыми.

В развитие этой темы следует перейти от понятия механизма сохранения к общему понятию ОО-базы данных, подробно рассматриваемому в [лекции 13](#) курса "Основы объектно-ориентированного проектирования". В ней обсуждаются проблемы механизмов сохранения STORABLE и другие проблемы, такие как эволюция схем и идентичность объектов сохранения.

Отмеченные выше ограничения механизма STORABLE нисколько не умаляют его практическую ценность. Хорошо известно, что отсутствие подобного механизма - одно из главных препятствий на пути широкого использования сложных структур данных в традиционном окружении. Без него хранение данных требует значительных программистских усилий: для каждого вида структуры приходится писать несколько взаимосвязанных рекурсивных процедур, реализующих операции ввода-вывода, также как и специальные механизмы обхода динамических структур данных. Но хуже всего - при изменении структуры данных приходится вновь обращаться к этим программам, внося соответствующие исправления.

Предопределенный механизм STORABLE позволяет решить все эти проблемы независимо от того, какова структура объектов, ее сложность, учитывая при этом эволюцию ПО.

Типичным приложением механизма STORABLE является свойство SAVE. Рассмотрим интерактивную систему, например текстовый редактор, графический редактор или систему компьютерного проектирования. Во всех случаях пользователю необходимо предоставить команду `SAVE`, сохраняющую состояние текущей сессии в файле. Хранимая информация должна быть достаточной для продолжения работы, так что она должна включать все важные структуры данных системы. Механизм STORABLE и хороший выбор головного объекта позволяет реализовать свойство `SAVE` одной командой:

```
head.store (save_file)
```

Уже одной этой причины достаточно для рекомендации выбора ОО-окружения в сравнении с другими более традиционными средами разработки.

Составные объекты и развернутые типы

Обсуждение структуры объектов времени выполнения показало важную роль ссылок. Для завершения картины необходимо выяснить, как работать со значениями, представляющими собой не ссылки на объекты, а непосредственно сами объекты.

Ссылок не достаточно

До сих пор все значения целочисленных, булевых и других аналогичных типов рассматривались как ссылки на объекты. Однако по двум причинам необходимы сущности, значениями которых являются объекты:

- В предыдущей лекции была поставлена важная цель - построение полностью унифицированной системы типов. В этой схеме базовые типы (`BOOLEAN`, `INTEGER` и др.) обрабатываются аналогично типам, введенным разработчиком

(POINT, BOOK и др.). Тем не менее, если используется сущность *n* типа INTEGER, то в большинстве случаев удобнее полагать, что значение *n* - целое число, а не ссылка на объект содержащий целое число. Это удобнее отчасти по соображениям эффективности. Понятно, что для размещения целочисленных объектов необходимо больше памяти, а на обработку косвенного доступа к ним - дополнительное время. Кроме того, концептуально целое число и ссылка на целое число - совершенно различные понятия. Этот довод важен, если нашей целью является построение точной модели.

- Даже в случае сложных, определенных программистом объектов, может оказаться предпочтительным включение в объект O1 подобъекта O2, а не ссылки на внешний объект O2. Причиной такого подхода могут быть повышение эффективности, точное моделирование или и то, и другое.

Развернутые типы

Удовлетворить потребность в составных объектах очень просто. Пусть С- класс, определенный так, как это делалось до сих пор

```
class C feature
  ...
end
```

Класс С может использоваться в качестве типа. Любая сущность типа С является ссылкой. По этой причине С называется **ссылочным типом (reference type)**.

Теперь предположим, что нам необходима сущность *x*, значение которой во время выполнения будет экземпляром С, а не ссылкой на такой экземпляр. Это достигается следующим объявлением *x*:

```
x : expanded C
```

Эта нотация использует новое ключевое слово **expanded (развернутый)**. Нотация **expanded C** означает, что экземпляры этого типа в точности соответствуют экземплярам С. Единственное отличие от обычного объявления типа состоит в том, что сущности типа С обозначают ссылки, которые могут быть присоединены к экземплярам С, а сущности типа **expanded C** обозначают непосредственно экземпляры С.

Таким образом, к структуре, определенной в предыдущих разделах, добавлено понятие **составного объекта (composite object)**. Объект О называется составным, если одно или более его полей являются объектами - **подобъектами (subobjects) O**. Следующий класс является примером описания составных объектов:

```
class COMPOSITE feature
  ref: C
  sub: expanded C
end
```

Класс COMPOSITE имеет два атрибута: *ref*, обозначающий ссылку, и *sub*, обозначающий подобъект. Вот как выглядит прямой экземпляр COMPOSITE.

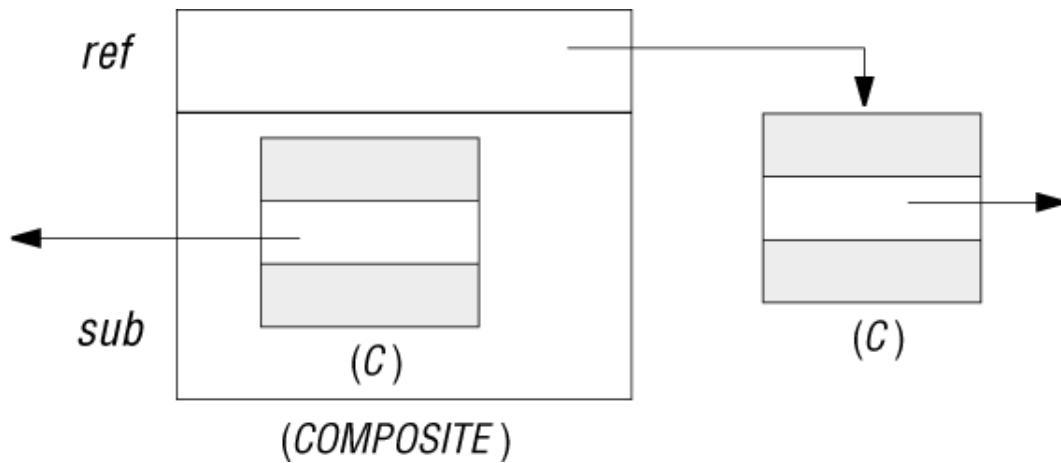


Рис. 8.19. Составной объект с одним подобъектом

Поле *ref* является ссылкой, присоединенной к экземпляру С (возможно, пустой ссылкой). Поле *sub* содержит экземпляр С и не может быть пустым.

Удобно несколько расширить нотацию. Иногда при проектировании класса, например Е, хотелось бы установить, что все экземпляры класса должны быть развернутыми. Чтобы сделать это требование явным, следует объявить класс в следующей форме:

```
expanded class E feature
  ...
  ... Далее все аналогично любому другому классу ...
end
```

Так определенный класс называется развернутым классом. Такое объявление класса никак не отражается на экземплярах

класса, они остаются такими же, как если бы класс был объявлен просто `class E`. Но сущности типа `E` изменяются - теперь их значения не ссылки, а сами объекты. Как следствие этой новой возможности понятие развернутого типа включает два случая:

Определение: развернутый тип

Тип является развернутым в двух случаях:

Он задан в форме: **expanded** С

Он задан в форме Е, где Е - развернутый класс.

Объявление вида

`x: expanded E`

где Е - развернутый класс, не будет ошибкой, поскольку эквивалентно

`x: E`

Таким образом, имеется два вида типов. Тип, не являющийся развернутым, является ссылочным типом. Этую терминологию можно использовать и для сущностей - ссылочные сущности и развернутые сущности. Аналогично и классы могут быть ссылочными и развернутыми.

Роль развернутых типов

Почему нам нужны развернутые типы? Они играют три важные роли:

- улучшают эффективность;
- обеспечивают лучшее моделирование;
- поддерживают базисные типы в унифицированной ОО-системе типов.

Первое применение наиболее очевидно: без развернутых типов каждый раз необходимо использовать ссылки для описания составных объектов. Это означало бы при каждом обращении к подобъекту выполнения операции, называемой "разыменование" (*dereferencing*), что влекло бы к временным потерям. Помимо этого, есть и потери в памяти, поскольку нужно отводить память не только объектам, но и самим ссылкам.

Аргумент производительности, однако, не является ключевым. ОО-конструирование ПО зачастую рассматривается как моделирование. Для отражения реальности необходимо моделировать объект как составной, а не как объект со ссылками. Это концептуальная проблема, а не проблема реализации.

Рассмотрим два объявления атрибутов:

D1. `ref: S`
D2. `exp: expanded S`

Объявления появляются в классе С, предполагается также, что S это ссылочный класс. Объявление D1 отражает тот факт, что каждый экземпляр класса С "знает о" существовании некоторого экземпляра S (если только `ref` не является `void`). Объявление D2 более требовательное: оно устанавливает тот факт, что каждый экземпляр класса С "содержит" экземпляр S. Даже если не думать о проблемах реализации, следует понимать, что речь идет о двух разных отношениях.

Отношение "содержит", поддерживаемое развернутыми типами, не допускает никакого **разделения** встроенного объекта, в то время как отношение "знает о" допускает несколько ссылок, присоединенных к объекту.

Вот пример объявления класса:

```
class WORKSTATION feature
  k: expanded KEYBOARD
  c: expanded CPU
  m: expanded MONITOR
  n: NETWORK
  ...
end
```

Рабочая станция имеет клавиатуру, ЦПУ, монитор и подключена к сети. Клавиатура, ЦПУ и монитор являются частью данного компьютера и не могут разделяться двумя или несколькими рабочими станциями. Однако несколько рабочих станций подключены к одной и той же сети. Эти особенности проявляются в определении класса, использующем развернутые типы для первых трех атрибутов и ссылочный тип для атрибута "сеть".

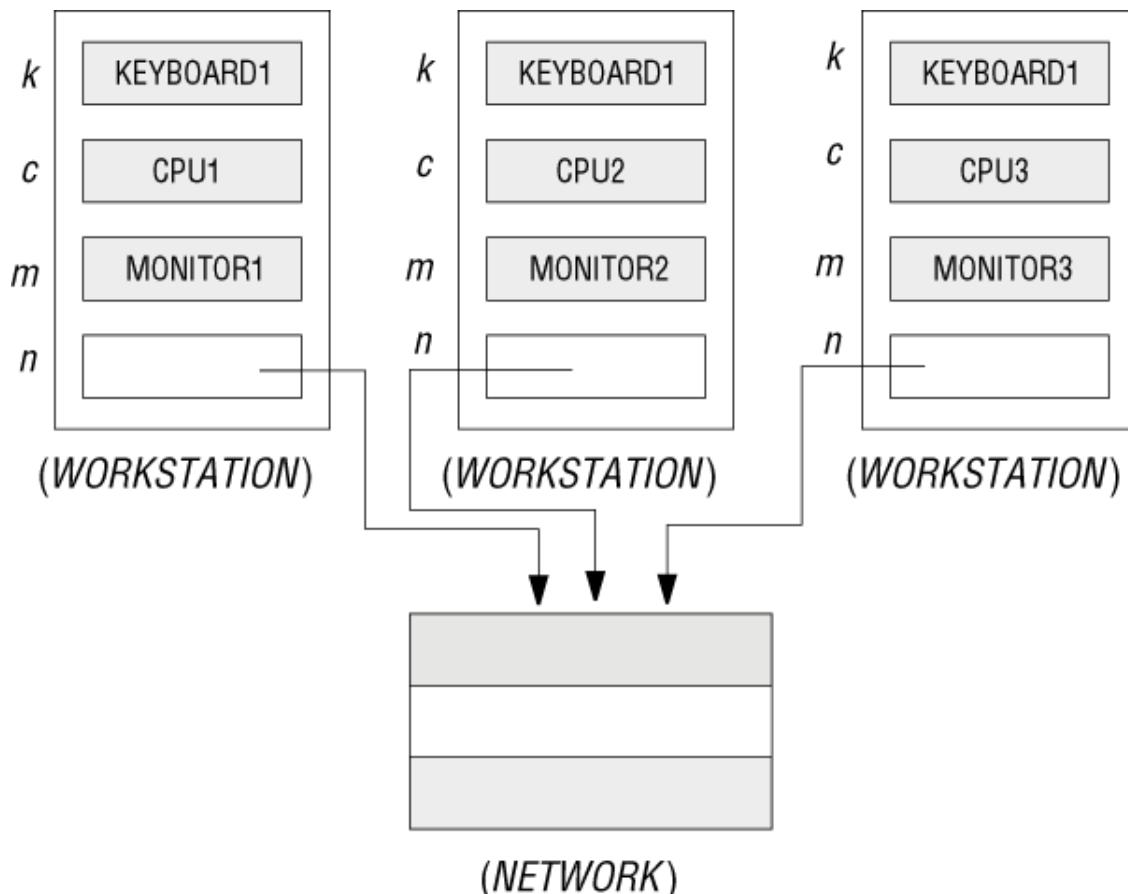


Рис. 8.20. Отношения между объектами: «знает о» и «содержит»

Итак, концепция развернутого типа, появившаяся вначале как техника уровня реализации, фактически помогла описать некоторые из отношений, используемых при информационном моделировании. Отношение "содержит" и обратное к нему отношение "быть частью" являются центральными при построении моделей внешних систем; они появляются в методах анализа и при моделировании баз данных.

Третье важное приложение развернутых типов фактически является частным случаем второго. В предыдущей лекции подчеркивалась желательность унифицированной системы типов, включающей как встроенные, так и пользовательские типы. Пример REAL использовался, чтобы показать, как с помощью инфиксных и префиксных компонентов можно промоделировать понятие вещественного числа как класса. То же самое нетрудно проделать и для других базисных типов: BOOLEAN, CHARACTER, INTEGER, DOUBLE. Но проблема все же остается. Если классы рассматривать как ссылочные, то сущности базисных типов, такие как

r : REAL

будут в период выполнения ссылками на возможные объекты, содержащие значение (в данном случае REAL). Это неприемлемо: чтобы соответствовать общей практике программирования значение должно быть не ссылкой, а самим вещественным числом. Решение проблемы немедленно следует из обсуждения - класс REAL следует объявить как развернутый. Его объявление должно быть таким:

```

expanded class REAL feature
    ... Объявления компонент такие же как и ранее ...
end

```

Все другие базисные типы объявляются подобным образом как развернутые.

Агрегирование

В некоторых областях информатики - базах данных, моделировании, анализе требований - разработана классификация отношений, имеющих место между элементами моделируемой системы. В этих контекстах часто встречается отношение "агрегирования" (aggregation), выражющее тот факт, что каждый объект некоторого типа является агрегатом - содержит в своем составе ноль или более объектов, каждый из которых имеет свой собственный тип. Например: автомобиль является агрегатом, содержащим мотор, кузов и другие детали.

Развернутые типы обеспечивают эквивалентный механизм. Мы можем, например, объявить класс CAR с компонентами развернутых типов: expanded ENGINE и expanded BODY. Другой способ основан на том, что агрегирование представляется отношением "развернутый клиент". Говорят, что класс S является развернутым клиентом класса S, если он содержит объявление компонента типа expanded S (или просто S, если S развернут). Одно из преимуществ такого модельного подхода в том, что развернутый клиент - это частный случай общего отношения "быть клиентом", так что можно использовать общие рамки и нотацию, комбинируя зависимости, подобные агрегированию с зависимостями, допускающими разделение. Примером могут служить с одной стороны - отношение между WORKSTATION и KEYBOARD, с другой - отношение между WORKSTATION и NETWORK.

Используя ОО-подход, можно избежать множественности отношений, используемых в литературе по информационному моделированию, - все покрывается двумя отношениями: клиент (развернутый или нет) и наследование.

Свойства развернутых типов

Рассмотрим развернутый тип E (в любой форме) и развернутую сущность x типа E.

Так как значение x это всегда объект, то он не может быть void. Так что булево выражение:

```
x = Void
```

будет всегда вырабатывать значение false, и вызов в форме x.some_feature(arg1, ...) никогда не приведет к возбуждению исключения из-за void цели, что могло случиться для ссылочной сущности.

Пусть объект O является значением x. Как и в случае не пустой ссылки, говорят, что x присоединено к O. Итак, для любой сущности, значение которой не void, можем говорить о присоединенном объекте, независимо от типа - ссылочного или развернутого - сущности.

Что можно сказать о создании развернутых объектов? Инструкцию:

```
create x
```

можно применить к развернутому x. Для ссылки x эффект достигался за три шага: (C1) создание нового объекта; (C2) инициализация его полей значениями по умолчанию; (C3) присоединение к x. Для развернутого x, шаг C1 неуместен, а шаг C3 бесполезен; так что единственный эффект состоит в инициализации полей значениями по умолчанию.

В общем случае, в случае присутствия развернутых типов инициализация по умолчанию предполагает выполнение шага C2. Предположим, что класс, развернутый или нет, включает развернутые атрибуты:

```
class F feature
  u: BOOLEAN
  v: INTEGER
  w: REAL
  x: C
  y: expanded C
  z: E
  ...
end
```

Класс E развернут, а класс C нет. Инициализация прямого экземпляра F включает установку поля u в false, v - в 0, w - в 0.0, x - ссылкой void, а экземпляры y и z станут экземплярами классов C и E соответственно, чьи поля будут инициализированы в соответствии со стандартными правилами. Этот процесс инициализации может быть рекурсивно продолжен, поскольку поля экземпляров C и E могут быть в свою очередь развернутыми.

Как можно было понять, использование развернутых типов требует введения некоторых ограничений, гарантирующих, что рекурсивный процесс создания будет конечным. Хотя, как отмечалось ранее, клиентские отношения в общем случае могут включать циклы, такие циклы не должны включать развернутые атрибуты. Например, недопустимо для класса C иметь атрибут типа expanded D, если класс D имеет атрибут типа expanded C. Это означало бы, что каждый объект Свключал бы подобъект D, который бы включал подобъект C и так далее. Сформулируем правило "развернутого клиента", ранее введенное неформально:

Правило Развернутого Клиента

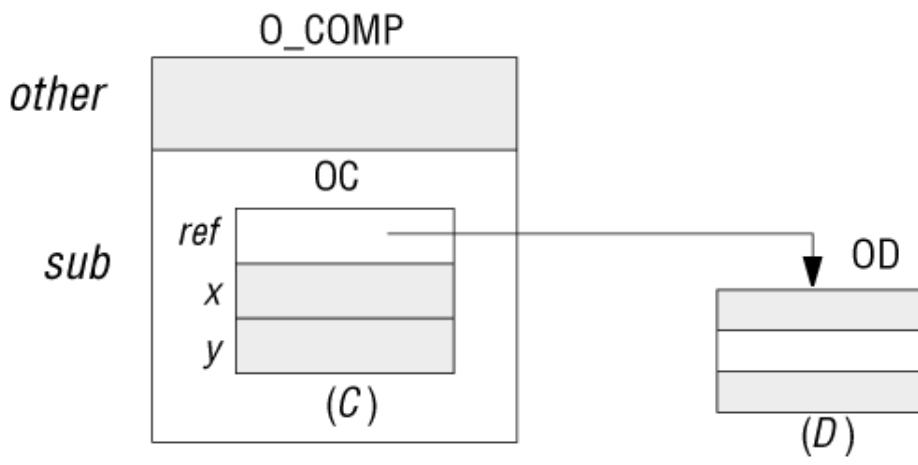
Пусть отношение "развернутый клиент" определяется следующим образом: класс C является развернутым клиентом класса S, если некоторый атрибут C является развернутым типом, основанным на классе S.

Тогда отношение развернутый клиент не может включать никаких циклов.

Другими словами, не может существовать множества классов A, B, C, ... N, где каждый последующий является развернутым клиентом предыдущего, а последний класс N является развернутым клиентом класса A. В частности, класс A не может иметь атрибут типа expanded A, так как это делало бы класс A своим развернутым клиентом.

Недопустимость ссылок на подобъекты

Заключительное замечание ответит на вопрос, как сочетаются ссылки и подобъекты. Развернутый класс или развернутый тип, основанный на ссылочном классе, может иметь ссылочные атрибуты. Вполне допустимо, чтобы подобъект содержал ссылки на объекты, как показано на рисунке:



(COMPOSITE1)

Рис. 8.21. Подобъект со ссылкой на другой объект

Приведенная ситуация предполагает следующие объявления:

```

Class COMPOSITE1 feature
    other: SOME_TYPE
    sub: expanded C
end
class C feature
    ref: D
    x: OTHER_TYPE; y: YET_ANOTHER_TYPE
end
class D feature
    ...
end

```

Каждый экземпляр класса COMPOSITE, такой как O_COMP на рис.8.21, имеет подобъект, (ОС на рисунке) содержащий ссылку ref, которая может быть присоединена к объекту (OD на рисунке).

Но противоположная ситуация, где ссылка становится присоединенной к объекту, невозможна. Это будет следовать из правил присваивания и передаче аргументов, изучаемых в следующем разделе. Итак, структура времени выполнения никогда не может находиться в ситуации, показанной на рис.8.22, где ОЕ содержит ссылку на ОС, - подобъект O_CMP1, и ОС содержит ссылку на себя.

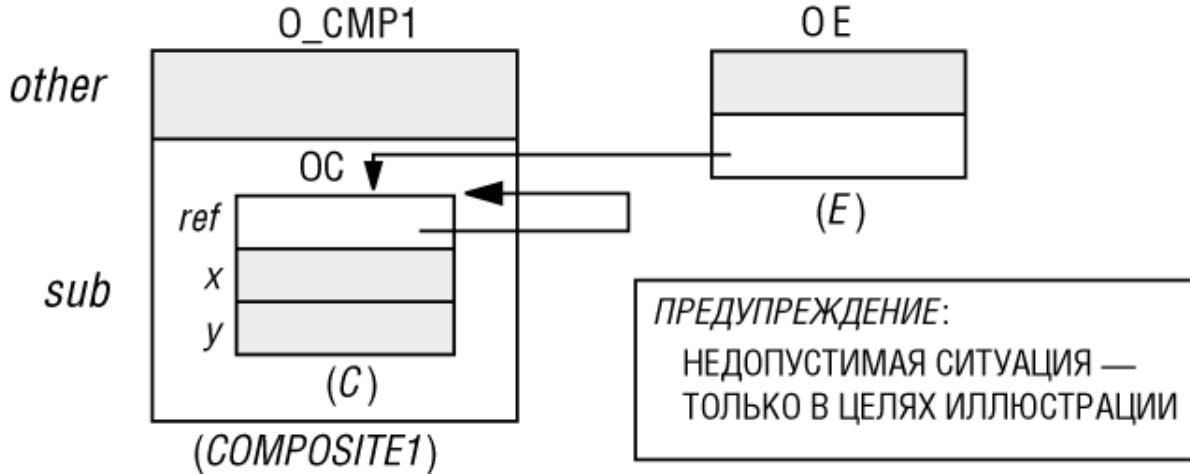


Рис. 8.22. Ссылка на подобъект

Это правило открыто для критики, поскольку оно ограничивает моделирующие возможности подхода. Предыдущая версия нотации книги допускала ссылки на подобъекты. Но эта возможность порождала больше проблем, чем она того стоит:

- С позиций реализации: механизм сборки мусора в этом случае должен быть готов справляться со ссылками на подобъекты, даже если в текущем выполнении будет всего несколько подобных ссылок или их вообще не будет. Это приводит к существенной потере производительности.
- С позиций моделирования: ссылки на подобъекты заставляют отказаться от упрощения описания системы, что можно сделать, определив единственную ссылочную единицу - объект.

Присоединение: две семантики - ссылок и значений

В этом разделе рассматривается специальная информация, и он может быть пропущен при первом чтении.

Введение развернутых типов требует возвращения к рассмотрению двух фундаментальных операций, уже рассмотренных в этой лекции, - присваивания и сравнения. Так как сущности теперь могут обозначать объекты, а не только ссылки, следует точно определить, каков смысл присваивания и эквивалентности в первом из этих случаев.

Присоединение

Семантика присваивания, как отмечалось, распространяется еще на одну операцию - передачу аргумента при вызове подпрограмм. Предположим, существует подпрограмма (процедура или функция) в форме:

```
r (... , x: SOME_TYPE, ...)
```

Здесь сущность *x* это один из формальных аргументов *r*. Рассмотрим теперь некоторый вызов *r* в любой из двух возможных форм - квалифицированный или неквалифицированный вызов:

```
r (... , y, ...)  
t.r (... , y, ...)
```

Выражение *y* является фактическим аргументом, передаваемым формальному аргументу *x*.

Выполнение *r* при **любом из** этих вызовов начинается с инициализации формальных аргументов значениями соответствующих фактических аргументов. Для простоты и согласованности правила, определяющие передачу аргументов, те же, что и правила присваивания. Другими словами, инициализация формального аргумента эквивалентна выполнению присваивания:

```
x := y
```

Это правило приводит к определению:

Определение: Присоединение

Присоединение *y* к *x* является результатом выполнения следующих двух операций:

Присваивания в форме *x := y*

Инициализации *x* при вызове подпрограммы, где *x* - формальный аргумент, а *y* - фактический аргумент вызова.

В обоих случаях *x* является целью присоединения, а *y* - источником.

Одни и те же правила действуют в обоих случаях для определения корректности присоединения (в зависимости от типов цели и источника). При условии корректности одни и те же правила определяют, каков будет эффект присоединения в период выполнения.

Присоединение: ссылочное и копии

При изучении ссылочного присваивания мы уже познакомились с эффектом присоединения. Если источник и цель являются ссылками, то эффект присваивания:

```
x := y
```

и соответствующей передачи аргументов состоит в том, что *x* получает значение ссылки *y*. Это иллюстрировалось несколькими примерами. Если значением *y* является *void*, то операция вместо присоединения сделает и *x* равным *void*; если *y* присоединен к объекту, то и *x* будет присоединен к этому же объекту.

Что происходит, когда типы *x* и *y* развернуты? Ссылочное присваивание не имеет смысла, а вот поверхностная форма копирования вполне возможна. Так и происходит. Рассмотрим объявления:

```
x, y: expanded SOME_CLASS
```

Присваивание *x := y* будет копировать каждое поле объекта, присоединенного к *y*, в соответствующие поля объекта, присоединенного к *x*, создавая тот же эффект, что и выполнение:

```
x.copy (y)
```

Копирование также является легальной операцией, эквивалентной в этом случае присваиванию. (В случае ссылок копирование и присваивание тоже легальны, но имеют разный эффект.)

Семантика копирования для развернутых типов дает ожидаемый эффект для всех базисных типов, которые, как отмечалось выше все относятся к развернутым типам. Например, если *m* и *n* типа INTEGER, то мы ожидаем от присваивания *m := n*, (или от соответствующей передачи аргументов) копирования значения *n* в *m*.

Проведенный анализ применим и к связанной с присваиванием операции эквивалентности. Рассмотрим булевые выражения: *x = y* и *x /= y*. Для *x* и *y* ссылочных типов, как уже отмечалось, истинность первого выражения (ложность второго) достигается только тогда, когда источник и цель оба имеют значение *void* или оба присоединены к одному и тому же объекту. Для развернутых *x* и *y*, такая семантика неприемлема, - здесь действует другая семантика, основанная на последовательном сравнении значений соответствующих полей, так что в этом случае выражение *x = y* имеет то же

значение, что и `equal (x, y)`.

Разрешается, как мы увидим позже при обсуждении наследования, изменить семантику `equal` для придания специальному смысла эквивалентности экземпляров некоторого класса. Это никак не отразится на операции эквивалентности `=`, которая по соображениям безопасности и простоты всегда имеет смысл оригинальной функции `standard_equal`.

Правило присваивания и сравнения обобщается в следующем замечании.

Присоединение `y` к `x` означает копирование объекта `x`, если `x` и `y` принадлежат развернутым типам. Это ссылочное присоединение, если `x` и `y` ссылочного типа. Аналогично, тесты: `x=y` и `x/=y` означают сравнение объектов для `x` и `y` развернутых типов; это ссылочное сравнение, если `x` и `y` ссылочного типа.

Гибридное присоединение

В рассматриваемых до сих пор случаях источник и цель принадлежали одной категории - оба развернутого или ссылочного типа. Что если они из разных категорий?

Вначале рассмотрим ситуацию, когда в присваивании `x := y` цель `x` развернутого типа, а источник `y` - ссылочного типа. Единственно приемлемой в этом случае является семантика копирования: копирование полей объекта, присоединенного к `y`, в поля объекта, присоединенного к `x`. Все хорошо, если `y` не `void` в период выполнения. Если `y` - `void`, то результатом будет включение исключения. (Исключения изучаются в [лекции 12](#))

Для развернутого `x` тест `x = Void` не является причиной появления исключительной ситуации; он просто дает значение `false`. Но нет приемлемой семантики для присваивания `x := Void`, так что всякая подобная попытка приводит к появлению исключения.

Рассмотрим теперь другой случай присваивания: `x := y`, где `x` ссылочного типа, а `y` - развернутого. Тогда в период выполнения `y` всегда присоединен к объекту, который мы можем назвать `OY`, и присоединение также должно присоединить `x` к объекту. Казалось бы, что можно присоединить `x` непосредственно к `OY`. Однако это привело бы к созданию ссылки на подобъект, а подобные ссылки запрещены нашими правилами. Поэтому правильной стратегией является клонирование источника `OY` и присоединение `x` к созданной копии. Рассмотрим пример:

```
class C feature
  ...
end
class COMPOSITE2 feature
  x: C
  y: expanded C
  reattach is
    do x := y end
end
```

При вызове компонента `reattach` в результате присваивания `x` будет присоединен к объекту, являющемуся клоном объекта `y`.

Следующая таблица обобщает семантику присоединения изученных случаев:

Таблица 8.1. Эффект присоединения `x:=y`

Тип цели <code>x</code>	Тип источника <code>y</code>	
	Ссылочный	Развернутый
Ссылочный	Ссылочное присоединение	Клонирование: эффект <code>x := clone(y)</code>
Развернутый	Копирование: эффект <code>x.copy(y)</code>	Ошибка, если <code>y</code> - <code>void</code>

Проверка эквивалентности

Семантика операций, проверяющих эквивалентность (`=` и `/=`) должна быть совместимой с семантикой присваивания. Наряду с операцией `=` можно использовать и `equal`. Какую из этих операций следует применять, зависит от обстоятельств.

- (E1) Если `x` и `y` - ссылки, их можно тестировать как на ссылочную эквивалентность, так и на объектную эквивалентность при условии, что ссылки не `void`. Мы определили операцию `x = y`, как обозначающую ссылочную эквивалентность в этом случае. Функция `equal`, введенная для проверки объектной эквивалентности, дополнена и применима, когда `x` или `y` - `void`.
- (E2) Если `x` и `y` - развернутого типа, единственный смысл имеет объектное сравнение.
- (E3) Если `x` - ссылка, `y` - развернутого типа, объектное сравнение - единственный возможный смысл операции и в данном случае. Сравнение расширяется, допуская случай, когда `x` - `void`, возвращая значение `false` в этой ситуации, поскольку `y` не может быть `void`.

Этот анализ дает желаемую интерпретацию равенства = во всех случаях. Для объектного сравнения всегда доступна функция `equal`, расширенная на случаи, когда один или оба операнда принимают значение `void`. Следующая таблица

подводит итог семантике сравнения:

Таблица 8.2. Семантика сравнения $x=y$

Тип цели x	Тип источника y	
	Ссылочный	Развернутый
Ссылочный	Ссылочное сравнение	equal(x,y) объектное сравнение, если x не void, иначе - false
Развернутый	equal(x,y) объектное сравнение, если y не void, иначе - false	equal(x,y) объектное сравнение

Сравнение таблиц 8.1 и 8.2 показывает совместимость присваивания и операций сравнения в упоминавшемся уже смысле. Напомним, в частности, что equal (x, y) будет истинно после выполнения $x := \text{clone} (y)$ или $x. \text{copy} (y)$.

Обсуждаемые проблемы возникают во всех языках, включающих ссылки и указатели, таких как Pascal, Ada, Modula-2, C, Lisp и другие. Они особенно актуальны для ОО-языков, в которых все создаваемые пользователем типы являются ссылочными. В дополнение к причинам, объясняемых в разделе обсуждения, в синтаксисе явно не отражается факт представления объектов ссылками, так что следует быть особо внимательными при проверке эквивалентности объектов.

Работа со ссылками: преимущества и опасности

В предыдущих разделах отмечалось, что два свойства модели времени выполнения заслуживают дополнительного внимания. Во-первых, важная роль ссылок. Во-вторых, двойственность семантики базовых операций (присваивания, передачи параметров, проверки на равенство), имеющих различный смысл для ссылок и развернутых операндов.

Динамические псевдонимы

Для x и y ссылочного типа при непустом значении y присваивание $x := y$ или соответствующее присоединение в результате вызова приведут к тому, что x и y будут присоединены к одному и тому же объекту.

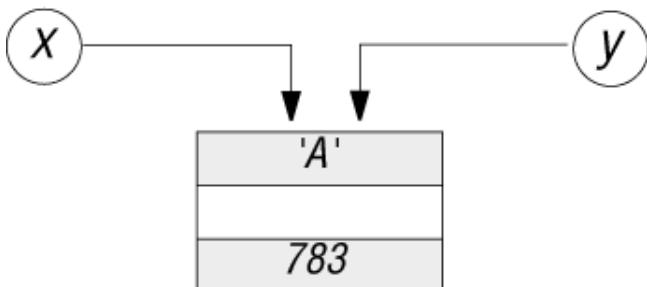


Рис. 8.23. Разделение как результат присоединения

В результате x и y становятся тесно связанными до тех пор, пока x или y не будет присвоено новое значение. В частности любая операция вида $x. f$, где f некоторый компонент соответствующего класса, приведет к тому же результату, что и $y. f$, поскольку воздействует на тот же объект.

Присоединение x к тому же объекту, что и y , известно как назначение **динамического псевдонима (dynamic aliasing)**. Псевдоним является динамическим, поскольку существует только во время выполнения.

Статические псевдонимы закрепляют два имени за одним и тем же программным элементом в исходном тексте, и они всегда обозначают одно и то же значение вне зависимости от событий, происходящих во время выполнения. Этот прием включен в некоторые языки программирования. В Fortran директива EQUIVALENCE означает, что две переменные разделяют содержимое одной и той же области памяти. Директива препроцессора C #define x y определяет, что любое упоминание x в тексте программы эквивалентно y .

Наличие динамических псевдонимов оказывает более серьезное влияние на операции присваивания с участием сущностей ссылочного типа, нежели с участием сущностей развернутого типа. В случае x и y развернутого типа INTEGER присваивание $x := y$ просто устанавливает для x значение y и никакого связывания x и y не происходит. После подобного присваивания с участием ссылочных типов x и y становятся псевдонимами одного объекта.

Семантика использования псевдонимов

Неприятным последствием применения псевдонимов (и статических, и динамических) является воздействие операций на сущности, даже не упоминаемые в операциях.

Модель вычислений без псевдонимов обладает приятным свойством: приведенный ниже фрагмент всегда справедлив

[БЕЗ СЮРПРИЗОВ]
-- Предположим, что свойство $P(y)$ выполняется
 $x := y$
 $C(x)$
-- $P(y)$ останется выполнимым.

Этот пример подразумевает, что Р (у) это частное свойство у, а С (х) некая операция с участием х, но не у. В этом случае никакие действия над х не влияют на значение у.

Для сущностей развернутых типов это действительно так. Приведем типичный пример с х и у типа INTEGER:

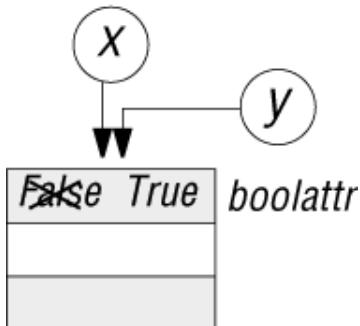
```
-- Предположим, что здесь у "= 0
x := y
x := -1
-- По-прежнему у "= 0.
```

В этом случае нет никакого способа изменить у путем присваивания значения х. Обратимся теперь к аналогичной ситуации с участием динамических псевдонимов. Пусть х и у экземпляры следующего класса С:

```
class C feature
  boolattr: BOOLEAN
    -- Булев атрибут для описания некоторого свойства объекта.
  set_true is
    -- Установка boolattr в true.
    do
      boolattr := True
    end
  ... Другие компоненты ...
end
```

Теперь предположим, что тип у это С, и что у в определенный момент времени выполнения не является пустой ссылкой. Тогда следующий пример уже не обладает свойством "БЕЗ СЮРПРИЗОВ":

```
[СЮРПРИЗ, СЮРПРИЗ!]
  -- Предполагаем, что у.boolattr равно false.
  x := y
  -- Значение у.boolattr по-прежнему false.
  x.set_true
  -- Но теперь у.boolattr равно true!
```



Последняя инструкция данного фрагмента никоим образом не содержит у, однако одним из ее результатов является изменение свойств у.

Выработка соглашений для динамических псевдонимов

Отмеченные тревожные последствия операций присваивания с участием ссылок порождают законный вопрос о целесообразности сохранения динамических псевдонимов в нашей модели вычислений.

Ответ - частично теоретический и частично практический:

- Операции присваивания необходимы для использования всех преимуществ мощи ОО-метода, в частности для описания сложных структур данных. Необходимо постоянно помнить, что рассматриваемый подход предназначен для решения задач моделирования.
- В практике разработки ОО-ПО для устранения опасностей, связанных с манипулированием ссылками, можно использовать инкапсуляцию.

Поочередно рассмотрим оба указанных аспекта.

Псевдонимы в ПО и за его пределами

Предварительное рассмотрение свидетельствует о том, что сами ссылки и их разделение необходимы во многих случаях. Некоторые стандартные структуры данных содержат циклически связанные элементы, которые невозможно реализовать без ссылок. В представлениях списков и деревьев удобно предоставить возможность узлам содержать ссылки на своих соседей или родителей. На [рис.8.24](#) приведен циклический список, использующий обе эти идеи. Открыв любую книгу по фундаментальным структурам данных и алгоритмам, можно найти массу таких примеров. В объектной технологии хотелось бы использовать и более сложные структуры.

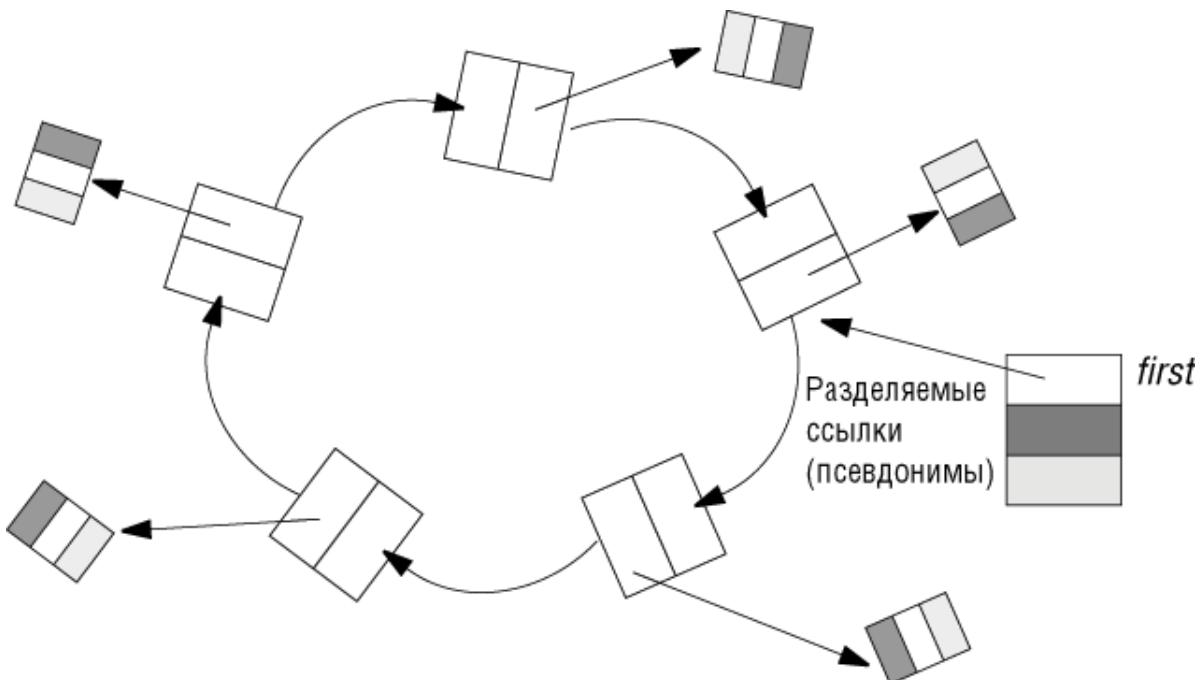


Рис. 8.24. Связный циклический список

На самом деле необходимость в ссылках, присоединении и разделении ссылок возникает и в не слишком сложных ситуациях. Вернемся к одному из вариантов класса описывающего книгу

```
class BOOK3 feature
  ... Остальные компоненты ...
  author: WRITER
end
```

Здесь необходимость разделения ссылок обусловлена тем, что две книги или более могут быть написаны одним и тем же автором. Во многих примерах данной лекции подразумевается разделение, - так в случае PERSON у нескольких персон может быть один лендлорд. Это вопрос потребностей моделирования, а не реализации.

Если b1 и b2 два экземпляра BOOK3 одного автора, то b1.author и b2.author - псевдонимы, то есть ссылки, присоединенные к одному объекту, и использование любой из них в качестве цели вызова даст в точности одинаковый эффект. Рассмотренные в таком свете динамические псевдонимы выглядят скорее не как потенциально опасная возможность программирования, а как факт из реальной жизни. Это цена, которую необходимо заплатить за возможности использования нескольких имен при обращении к одному объекту.

Можно легко найти нарушения приведенного выше свойства "БЕЗ СЮРПРИЗОВ", не обращаясь к области ПО. Пусть для некоторой книги b определены следующие свойства и операции:

- NOT_NOBEL (b) обозначает: "автор никогда не получал Нобелевскую премию".
- NOBELIZE (b) обозначает: "Присудить Нобелевскую премию автору книги b".

Теперь предположим, что rb обозначает книгу "Красное и черное", а cp - "Пармская обитель". Последующие действия вполне корректны:

```
[СЮРПРИЗ В ОСЛО]
-- Предположим, что сейчас выполняется NOT_NOBEL(rb)
NOBELIZE(cp)
-- Теперь свойство NOT_NOBEL(rb) уже несправедливо!
```

Операция над cp изменяет свойство другой сущности rb, которая не упоминается в инструкции! Последствия могут быть весьма значительными (редкая книга Нобелевского лауреата будет переиздана, ее цена возрастет и т.д.). В данной ситуации, не связанной с ПО, произошло в точности то же, что и в предыдущем программном примере после операции x.set_true, повлиявшей на состояние у без упоминания у.

Таким образом, динамические псевдонимы вовсе не являются результатом гнусных трюков программистов со ссылками и указателями. Это следствие свойственного человеку стремления давать имена вещам ("объектам" в наиболее общем смысле этого слова), а иногда и несколько имен одному предмету. В классической риторике эти явления известны как полионимия (polyonymy), например, использование имен "Кибела" (Cybele), "Деметра" (Demeter) и "Церера" (Ceres) "для одной и той же богини, и антономазия (antonomasia) - возможность ссылаться на объект, косвенно именуя его, как, например, в фразе "прекрасная дочь Агамемнона", обращаясь к прекрасной Елене из Трои.

Инкапсуляция действий со ссылками

Теперь накоплено достаточно подтверждений того, что любая система моделирования и разработки ПО должна поддерживать понятие ссылки, а, следовательно, и динамические псевдонимы. Как теперь справиться с неприятными последствиями? Невозможность обеспечить свойство "БЕЗ СЮРПРИЗОВ" показывает, что ссылки и псевдонимы

подвергают опасности саму возможность систематического рассмотрения ПО. Это означает, что, изучая исходный текст, нельзя надежно и просто сделать какие либо выводы о свойствах ПО времени выполнения.

Для поиска решения необходимо сначала понять, является ли данная проблема специфической для ОО-метода. Знакомство с другими языками программирования, такими как Pascal, C, PL/I, Ada и Lisp убеждает в том, что и там ведутся подобные дискуссии. Все языки располагают средствами динамического размещения объектов и разрешают объектам содержать ссылки на другие объекты. Существенно различаются лишь уровни абстракции: указатели C и PL/I фактически являются машинными адресами, а Pascal и Ada наряжают указатели в более респектабельные одежды, используя правила типизации.

Что тогда нового в ОО-разработке? Ответ связан не с теоретическими возможностями метода (за исключением важных отличий, связанных со сборкой мусора, ОО-структуры времени выполнения идентичны своим аналогам в Pascal и Ada), а в практике разработки ПО. ОО-разработка подразумевает повторное использование. В частности, любой проект, в котором многочисленные прикладные классы выполняют хитрые манипуляции со ссылками, является примером некорректного использования ОО-подхода. Такие операции должны быть включены в библиотечные классы.

Для любой системы подавляющее большинство объектных структур, требующих нетривиальных операций со ссылками, не зависит от области приложения и представляет хорошо известные и часто используемые структуры: списки всевозможных типов, деревья в различных представлениях, хэш-таблицы и некоторые другие. В хорошей ОО-среде разработки библиотека должна быть легко доступной и предоставлять реализации подобных структур. В качестве иллюстрации в приложении А приведен эскиз библиотеки Base. Классы в таких библиотеках могут содержать множество ссылочных операций. Примером могут служить действия над ссылками, необходимые для вставки и удаления элемента связного списка или узла дерева.

Если же при разработке приложения потребность в сложных объектных структурах, не представленных адекватно в имеющихся библиотеках, то это следует рассматривать как потребность в новых классах общего назначения. Их необходимо разработать, потратить необходимое время на их тщательное тестирование и затем включить в соответствующую библиотеку. Такая ситуация в терминах одной из предшествующих лекций является примером перехода с позиций потребителя по отношению к повторному использованию на позиции производителя.

Оставшиеся операции со ссылками в классах, зависимых от конкретного приложения, должны быть ограничены простыми и безопасными операциями. В библиографических заметках упомянута статья Suzuki, развивающая эту идею.

Обсуждение

В данной лекции введены некоторые правила и нотация для работы с объектами и соответствующими сущностями. Некоторые из этих соглашений могут вызвать удивление. Поэтому полезно завершить изучение объектов и их свойств рассмотрением спорных вопросов и доводов в пользу выбранных решений. Автор, естественно, надеется, что читатели полностью одобрят его выбор. Основная цель данной дискуссии - добиться полного понимания основополагающих проблем. В этом случае, если кто-то предпочитает другое решение, то сможет выбрать его вполне осознанно.

Графические соглашения

Для разминки начнем с небольшой проблемы, связанной с нотацией. Это конечно деталь, но из деталей складывается общая картина. Речь идет о наборе соглашений, используемых для графического представления классов и объектов.

В предшествующей лекции отмечалось насколько важно различать понятия класс и объект. Соответственно, должны отличаться и графические обозначения. Классы на диаграммах, представляющих системную архитектуру, представлены в виде эллипсов. Они соединяются стрелками для обозначения отношений между классами, обычными стрелками отмечаются отношения наследования и двойными - клиентские отношения.

Классы и объекты существуют в различных контекстах. Эллипсы классов являются частью диаграмм, представляющих структуру программной системы. Прямоугольники-объекты используются на моментальных снимках состояния системы в процессе выполнения. Поскольку указанные виды диаграмм имеют совершенно разное назначение, то в бумажном представлении, как в данной книге, они не появляются одновременно в одном контексте. Но для интерактивных CASE-средств ситуация принципиально меняется. В процессе отладки программной системы возникает необходимость отобразить объект, а затем - порождающий класс для изучения его компонент, родителей и других свойств.

Используемые для классов и объектов графические соглашения совместимы со стандартом, установленным методом BON (Nerson и Walden). В методе BON, (Business Object Notation) предназначенном для использования в интерактивных CASE-средствах и в документации, классы отображаются в виде пузырьков, растягиваемых по вертикали, показывающих компоненты класса, инварианты и другие свойства.(О BON см. библиографические заметки и [лекцию 9](#) курса "Основы объектно-ориентированного проектирования")

В развитие нашего соглашения поля развернутых типов отличаются от ссылочных затенением, а подобъекты присутствуют в виде вложенных прямоугольников, содержащих собственные поля. Все эти соглашения вытекают из решения отображать объекты в виде прямоугольников.

Трудно удержаться от того, чтобы не процитировать следующий ненаучный аргумент, заимствованный из рецензии Ian Graham на книгу по ОО-анализу, использующую другие графические соглашения:

Мне не нравятся классы, изображаемые в виде треугольников с острыми углами. Мне кажется, что это их экземпляры имеют острые углы, так что можно пораниться, уронив их на ногу, а классы безопасны и поэтому у них должны быть скругленные углы.

Ссылки и простые значения

Важный синтаксический вопрос - должны ли мы установить синтаксическое различие при работе со ссылками и простыми значениями. Как отмечалось, присваивание и эквивалентность имеют различный смысл для ссылок и значений развернутых типов. Но синтаксис этого не различает, - в обоих случаях используются одинаковые символы (`:=`, `=`, `/=`). Не опасно ли это? Не предпочтительнее ли использовать различные наборы символов, напоминая тем самым, что они имеют разный смысл?

Два набора символов использовались в языке Simula 67. Немного изменив нотацию для совместимости с настоящей книгой (в языке Simula `reference` сокращается до `ref`), в Simula можно записать объявление сущности ссылочного типа Стак:

```
x: reference C
```

Ключевое слово `reference` указывает, что экземпляры `x` будут ссылками. Рассмотрим объявления:

```
m, n: INTEGER
x, y: reference C
```

Нотация Simula, используемая для операций с простыми и ссылочными типами, приведена в таблице.

Таблица 8.3. Нотация в стиле Simula для операций со ссылками и значениями развернутых типов

Операция	Операнды развернутых типов	Операнды-ссылки
Присваивание	<code>m := n</code>	<code>x := y</code>
Проверка на равенство	<code>m = n</code>	<code>x == y</code>
Проверка на неравенство	<code>m /= n</code>	<code>x /= y</code>

Соглашения Simula лишены неоднозначности. Почему бы их не сохранить? К сожалению, эти соглашения могут служить примером благих намерений, приносящих скорее вред, нежели пользу. Проблемы начинаются в прозаической области поиска ошибок. Два набора символов похожи, - это провоцирует синтаксические ошибки, подобные использованию `:=` вместо `:-`.

Такие ошибки будут обнаружены компилятором. Но хотя ограничения, проверяемые компилятором, предназначены помочь программисту, - здесь это может не сработать. Либо вы знаете разницу между семантикой ссылок и значений, и тогда подсказки компилятора о необходимости проверки, каждый раз, когда вы написали присваивание или равенство, могут показаться раздражающими. Либо вы не понимаете этой разницы, тогда его подсказки немногим могут помочь.

Но самый важный аспект соглашений Simula в том, что он не оставляет выбора: для ссылок нет доступной конструкции, дающей семантику значений. Представляется разумным для сущностей `a` и `b` ссылочного типа иметь два множества операций:

- `a :- b` для ссылочных присваиваний и `a == b` сравнения ссылок;
- `a := b` для присваивания путем копирования (эквивалент `a := clone (b)` или `a . copy (b)` в нашей нотации) и `a = b` для сравнения объектов (эквивалент нашего `equal (a, b)`).

Но за одним исключением, Simula поддерживает только первое множество операций. Если необходимы операции второго множества (`copy` или `clone`, сравнение объектов), придется написать специальные подпрограммы для каждого целевого класса. Исключением является `TEXT`, для которого Simula предлагает оба множества операций.

Кстати, при дальнейшем анализе идея предоставления двух множеств операций для всех ссылочных типов не кажется уже столь разумной. Это бы означало, что тривиальная описка - использование `:=` вместо `:-`, теперь уже не обнаруживалась бы компилятором, а приводила бы к результату, далекому от ожидаемого, например, к клонированию вместо ссылочного присваивания.

Как результат этого анализа, нотация этой книги использует соглашения, отличные от тех, что используются в Simula: одни и те же символы применимы для развернутых и ссылочных типов с различной семантикой. Эффекта семантики значений можно достигнуть для объектов ссылочного типа при использовании предопределенных подпрограмм, применимых для всех типов:

- `a := clone (b)` или `a . copy (b)` для объектного присваивания;
- `equal (a, b)` для сравнения объектов на идентичность всех полей.

Эта нотация существенно отличается от нотации, применяемой для их ссылочных двойников, (`:=` и `=`, соответственно) что снижает риск появления недоразумений.

Помимо чисто синтаксических аспектов, эта проблема интересна и тем, что она представляет типичный образец компромиссов, возникающих при проектировании языка, когда требуется найти баланс между конфликтующими критериями. Один из критериев, победивших в Simula, - может быть сформулирован следующим образом:

- "Выражайте различные концепции с помощью различных символов".

Но другие силы, доминирующие в нашей нотации, требуют:

- "Не создавайте разработчику лишних проблем".
- "Тщательно взвешивайте все "за и против" любой новинки, обращая особое внимание на безопасность и качество".
- "Убедитесь, что общие операции могут быть выражены в простой и ясной форме". Применение этого принципа требует особой тщательности, поскольку проектировщик языка может ошибаться в своих оценках того, что же является наиболее общим случаем. Но в данной ситуации все кажется проще. Для сущностей развернутого типа (таких как INTEGER) присваивание и сравнение значений представляются наиболее употребительными операциями. Для ссылок, в то же время, ссылочное сравнение и присваивание используется чаще, чем клонирование, копирование и сравнение объектов. Поэтому в обоих случаях предпочтительнее использовать := и = для фундаментальных операций.
- "Для сохранения компактности и простоты языка вводите новые обозначения, только если это абсолютно необходимо". Это справедливо в частности для приведенного примера - существующая нотация работает и не существует опасности путаницы.
- "Если вы знаете, что существует риск возникновения недоразумений между двумя возможностями, то соответствующие нотации должны различаться очевидным образом". Так что необходимо избегать использования символов, близких по написанию (: - и :=), но с различной семантикой.

Еще одна причина играет роль в данном случае, хотя она включает механизм, пока еще не изученный. В последующих лекциях мы познакомимся с родовыми или универсальными классами, такими как LIST [G], где G, известно как формальный родовой параметр, представляющий произвольный тип. Такой класс может манипулировать сущностями типа G и использовать их в присваиваниях и проверках на равенство. Клиенты, нуждающиеся в использовании такого класса, должны позаботиться о создании типа, служащего в качестве фактического родового параметра. Например, они могут использовать LIST [INTEGER] или LIST [POINT]. Как показывают эти примеры, фактический родовой параметр может быть развернутого типа, как в первом случае, так и ссылочного типа - во втором случае. В подпрограммах такого родового класса, еслии b имеют тип G, то часто полезно использовать присваивания в форме a := b или тесты в форме a = b с намерением получить семантику значений, когда фактический параметр принадлежит развернутому типу, такому как INTEGER, и ссылочную семантику - для ссылочного типа, такого как POINT.

Примером подпрограммы, нуждающейся в таком дуальном поведении, является процедура вставки элемента x в список. Процедура создает новый элемент списка. Если x целое, элемент должен содержать копию значения x. Если x является ссылкой, то элемент списка должен содержать ссылку на объект, присоединенный к x .

В таких случаях, правила, определенные выше, гарантируют желаемое дуальное поведение, что было бы недостижимо, если бы требовался различный синтаксис для двух видов семантики. С другой стороны, если во всех случаях требуется единая семантика, то и это достичимо: такое поведение может быть только семантикой значений (так как семантика ссылок не имеет смысла для развернутых типов); поэтому в соответствующих подпрограммах следует использовать clone (или copy) и equal, а не (= и =).

Форма операций клонирования и эквивалентности

Форма вызова подпрограмм clone и equal является стилевой особенностью, которая может вызвать удивление. На первый взгляд нотация:

```
clone (x)
equal (x, y)
```

выглядит не слишком объектно-ориентированной. Догматичное следование принципу "ОО-стиля вычислений" из предыдущей лекции предполагает другую форму (См. "Объектно-ориентированный стиль вычислений", [лекция 7](#)):

```
x.twin -- twin это двойник - клон.
x.is_equal (y)
```

В первой версии нотации так и делалось, однако возникла проблема пустых ссылок. Вызов компонента вида x.f (...) не может быть корректно выполнен в случае пустого x во время выполнения. В этом случае вызов инициирует исключение, которое повлечет аварийное завершение всей системы, если в соответствующем классе не предусмотрена обработка исключений. Поскольку во многих случаях x может быть действительно пустой ссылкой, то это означало бы, что каждый вызов twin должен предусматривать охрану и выглядеть так:

```
if x = Void then
    z := Void
else
    z := x.twin
end
```

Соответственно, реализация вызова is_equal должна выглядеть (**and then** является вариантом **and**). См. "Нестрогие булевые операции", [лекция 13](#):

```
if
  ((x = Void) and (y = Void)) or
  ((x /= Void) and then x.is_equal (y))
then
  ...

```

Излишне говорить, что не следует придерживаться этих соглашений. Нам быстро надоест писать подобные витиеватые фрагменты, а когда мы забудем это сделать, то результатом будет ошибка времени выполнения. Окончательный вариант

соглашений, сформулированный в данной лекции, замечателен еще и тем, что дает ожидаемые результаты для `x`, равного `void`, - `clone (x)` вернет `void`, а `equal (x, y)` вернет `true`, если и `y` - `void`.

Вызов процедуры `copy` в форме `x.copy (y)` не создает подобных проблем, поскольку требует непустых `x` и `y`. Это следствие семантики процедуры `copy`, копирующей поля одного объекта в поля другого, и имеющей смысл, только если существуют оба объекта. Как показано далее, такое условие для `y` фиксируется формальным предусловием `copy`, заданным в явном виде в документации.

Отметим, что введенная выше функция `is_equal` существует в библиотеке системы. Причина в том, что часто удобнее определить специфические варианты эквивалентности элементов конкретного класса, перекрыв семантику по умолчанию. Для достижения этого эффекта достаточно переопределить функцию `is_equal` в соответствующем классе. Функция `equal` определяется в терминах `is_equal` (выражением, показанным выше при иллюстрации использования `is_equal`), и поэтому следует за всеми переопределениями `is_equal`.

Когда есть функция `clone`, то нет необходимости в `twin`. Это связано с тем, что функция `clone` определена как создание объекта с последующим вызовом `copy`. Поэтому для адаптации `clone` к специфике класса достаточно переопределить процедуру `copy` данного класса. (См. также [лекция 16](#))

Статус универсальных операций

Последние комментарии частично прояснили вопрос о статусе универсальных операций `clone`, `copy`, `equal`, `is_equal`, `deep_clone`, `deep_equal`.

Эти операции не являются языковыми конструкциями, невзирая на их фундаментальную значимость для практики. Они поставляются классом ANY основной библиотеки Kernel. Этот класс имеет то специальное свойство, что каждый класс, созданный разработчиком, автоматически становится наследником (прямым или косвенным) класса ANY. Вот почему становится возможным переопределить вышеупомянутые компоненты для поддержки специального вида эквивалентности или копирования. (См. "Глобальная структура наследования", [лекция 16](#))

Сейчас нет необходимости в деталях, поскольку мы еще вернемся к этой проблеме при изучении наследования. Но уже теперь полезно знать, что благодаря механизму наследования, мы можем полагаться на библиотечные классы, поддерживающие свойства, доступные всем классам, - и каждый класс может изменить их, приспособливая к своим, специфическим целям.

Ключевые концепции

- ОО-вычисления характеризуются высокой динамичной структурой времени выполнения, в которой объекты создаются только по запросу.
- Некоторые объекты, используемые ПО, являются моделями внешних объектов (обычно косвенными). Другие объекты служат только для целей проектирования и реализации.
- Объект состоит из ряда значений, называемых полями. Каждое поле соответствует атрибуту генератора объекта (класса, прямым экземпляром которого является объект).
- Значение, в частности поле объекта, является объектом или ссылкой.
- Ссылка может быть пустой (`void`) или присоединенной к объекту. Проверка условия `x = Void` позволяет определить текущее состояние ссылки. Корректное выполнение вызова `x.f (. . .)` возможно, если `x` не пустая ссылка.
- Если объявление класса начинается с предложения `class C . . .`, то сущность типа `C` будет обозначать ссылку, которая может быть присоединена к экземпляру `C`. Если начало объявления выглядит как `expanded class D . . .`, то сущность типа `D` будет обозначать объект (экземпляр `D`) и никогда не может быть пустой ссылкой.
- Базовые типы (`BOOLEAN`, `CHARACTER`, `INTEGER`, `REAL`, `DOUBLE`) определены как развернутые классы.
- Разворнутые объявления дают возможность определять составные объекты: объекты с подобъектами.
- Объектные структуры могут содержать циклические цепочки ссылок.
- Инструкция создания `create x` создает объект, инициализирует его поля значениями по умолчанию и присоединяет к нему `x`. Если в классе определены порождающие процедуры создания, то выполнение инструкции вида `create x.createproc (. . .)` приведет, кроме того, к заданной специфической инициализации полей.
- Для сущностей ссылочного типа присваивание (`:=`) и проверка эквивалентности (`=`) являются ссылочными операциями. Для сущностей развернутых типов используется семантика значений. Соответствующая семантика распространяется и на смешанные операнды.
- В результате ссылочных операций появляются динамические псевдонимы. Они затрудняют получение выводов о работе системы при анализе ее текста. На практике большинство нетривиальных действий со ссылками можно инкапсулировать в библиотечные классы.

Библиографические замечания

Понятие идентичности объекта играет важную роль для баз данных, особенно объектно-ориентированных. Смотри [лекцию 13](#) курса "Основы объектно-ориентированного проектирования", посвященную таким базам данных, и библиографию к ней.

Графические обозначения метода BON (Business Object Notation) разработаны Jean-Marc Nerson и Kim Walden [Walden 1995]. James McKim и Richard Bielak детально рассмотрели преимущества альтернативных порождающих процедур [Bielak 1994].

Риски, связанные с нетипизированными указателями и ссылочными операциями, уже долгое время волнуют специалистов

в области методологии программирования, порождая намеки на то, что в области данных это аналог ненавистной операции **goto** в области управления выполнением кода. В удивительно малоизвестной статье Nori Suzuki [Suzuki 1982] обсуждается возможность избежать в рамках строгого подхода с использованием высокоуровневых операций проблем с динамическими псевдонимами (как избавляются от применения **goto**, используя приемы "структурного программирования"). Хотя по признанию автора выводы неутешительны, данная статья весьма полезна.

Я признателен Ross Scaife из Университета Кентукки за помочь по вопросам риторики. См. его страницу <http://www.uky.edu/ArtsSciences/Classics/rhetoric.html>.

Упражнения

У8.1 Книги и авторы

Напишите классы BOOK and WRITER описывающие книги и их авторов, используя заготовки из данной лекции. Обратите внимание на необходимость включения всех важных подпрограмм, а не только атрибутов.

У8.2 Личности

Напишите класс PERSON включающий простое понятие личности с атрибутами mother, father и sibling (следующий по старшинству брат или сестра, если они есть). Включите подпрограммы возвращающие списки имен родителей, двоюродных братьев и сестер, дядюшек и тетушек, свекра и свекрови, тестя и тещи данного лица. **Совет:** пишите рекурсивные процедуры, но внимательно избегайте бесконечных рекурсий для отношений, например, двоюродный брат или сестра, являющихся циклическими.

У8.3 Проектирование нотации

Предположим, вы часто используете сравнение в форме `x.is_equal(y)`, и хотите упростить нотацию, используя преимущества инфиксной записи (применимой здесь, поскольку наша функция имеет один аргумент). Для инфиксного компонента используйте некоторый оператор `§`, вызов тогда будет записываться в виде `x § y`. Это маленькое упражнение потребует выбора для оператора `§`, подходящего для данной ситуации символа, совместимого с правилами инфиксных операторов. Конечно, здесь может существовать много возможных ответов, выбор одного из которых частично (но только частично) дело вкуса. (См. "Компоненты-операторы", [лекция 7](#))

Основы объектно-ориентированного программирования

9. Лекция: Управление памятью

Честно говоря, было бы неплохо забыть про память. Программы создавали бы объекты по мере надобности. Неиспользованные объекты исчезали бы в небытие, а необходимые медленно передвигались бы вверх. Этот процесс подобен движению по служебной лестнице работника большой корпорации, в конце карьеры достигшего уровня руководства. Но это не так. Память не безгранична и не организуется в непрерывный ряд слоев с уменьшающейся скоростью доступа. Нам необходимо увольнять наших бестолковых работников даже, если мы должны называть это ранним уходом на пенсию, продиктованным общей экономической ситуацией. Эта лекция изучает, кто все же должен быть сокращен, кем и как.

Что происходит с объектами

ОО-программа создает объекты. Предыдущая лекция показала, как полезно полагаться на динамическое создание для получения гибких объектных структур, подстраивающихся автоматически к нуждам системы.

Создание объектов

Мы рассмотрели базовые операции размещения новых объектов. Простейший способ размещения записывается как

```
create x
```

и его эффект был определен триадой: создать новый объект; связать его со ссылкой x; и инициализировать его поля.

Вариант этой инструкции вызывает процедуру инициализации; можно также создать новый объект с помощью подпрограмм `clone` и `deep_clone`. Так как все эти формы размещения основаны на одной и той же базисной инструкции создания, можно без потери общности ограничиться рассмотрением `create x`.

Рассмотрим эффект, создаваемый инструкциями управления памятью.

Три режима управления объектами

Во-первых, будет полезным расширить рамки дискуссии. Форма управления объектами, используемая для ОО-вычислений, может поддерживаться одним из трех обычно встречаемых режимов: **статическим, стековым и динамически распределяемым**. Выбор режима определяет, как сущности присоединяются к объектам.

Напомним, что сущность - это имя в тексте программы, представляющее некоторое значение или совокупность значений в период выполнения. Такие значения являются либо объектами, либо (возможно неопределенными) ссылками на объект. Сущностями являются атрибуты, формальные аргументы подпрограмм, локальные переменные подпрограмм и Result. Термин **присоединение** описывает связь между сущностью и объектом: на определенном этапе выполнения программы сущность x присоединяется к объекту O, если значение x есть либо O (для x развернутого типа), либо ссылка на O (для x ссылочного типа). Если x присоединен к O, часто говорят также, что O присоединен к x. Ссылка может быть присоединена не более чем к одному объекту, объект может быть присоединен к двум и более ссылкам. Проблема динамических псевдонимов обсуждалась в предыдущей лекции.

В статическом режиме сущность может быть присоединена максимум к одному объекту в процессе выполнения программы. Эта схема, поддерживаемая в таких языках как Fortran, резервирует место для всех объектов и присоединяет объект к имени раз и навсегда при загрузке программы или в начале ее выполнения.

ФИКСИРОВАННАЯ ОБЛАСТЬ ПАМЯТИ

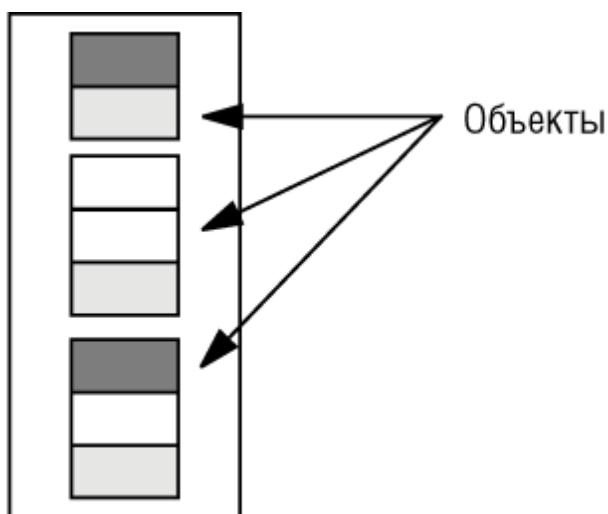


Рис. 9.1. Статический режим

Статический режим прост и эффективно реализуем архитектурой обычного компьютера. Но он имеет серьезные ограничения:

- Препятствует рекурсии. Рекурсивной программе необходимо иметь несколько одновременно активных копий, каждой со своими экземплярами сущностей.
- Препятствует созданию динамических структур данных. Компилятор должен уметь определять точный размер каждой структуры данных из текста программы. Каждый массив, например, должен в этом случае объявляться статично со своим строгим размером. Это серьезно ограничивает мощность языка: становится невозможным оперировать структурами, растущими в ответ на события выполнения. Приходится резервировать максимально возможную память для каждой из структур - это не только неэффективно, но и довольно опасно. Если размер одной из структур данных недооценен, это, скорее всего, вызовет ошибку выполнения системы.

Второй режим размещения объектов - режим стека. Здесь сущность может быть в реальном времени последовательно присоединяться к нескольким объектам. Механизм выполнения размещает и удаляет эти объекты в порядке "последним пришел, первым ушел". Когда объект удаляется, относящаяся к нему сущность присоединяется вновь к объекту, с которым она была связана до появления нового элемента, если, конечно, такой объект существует.

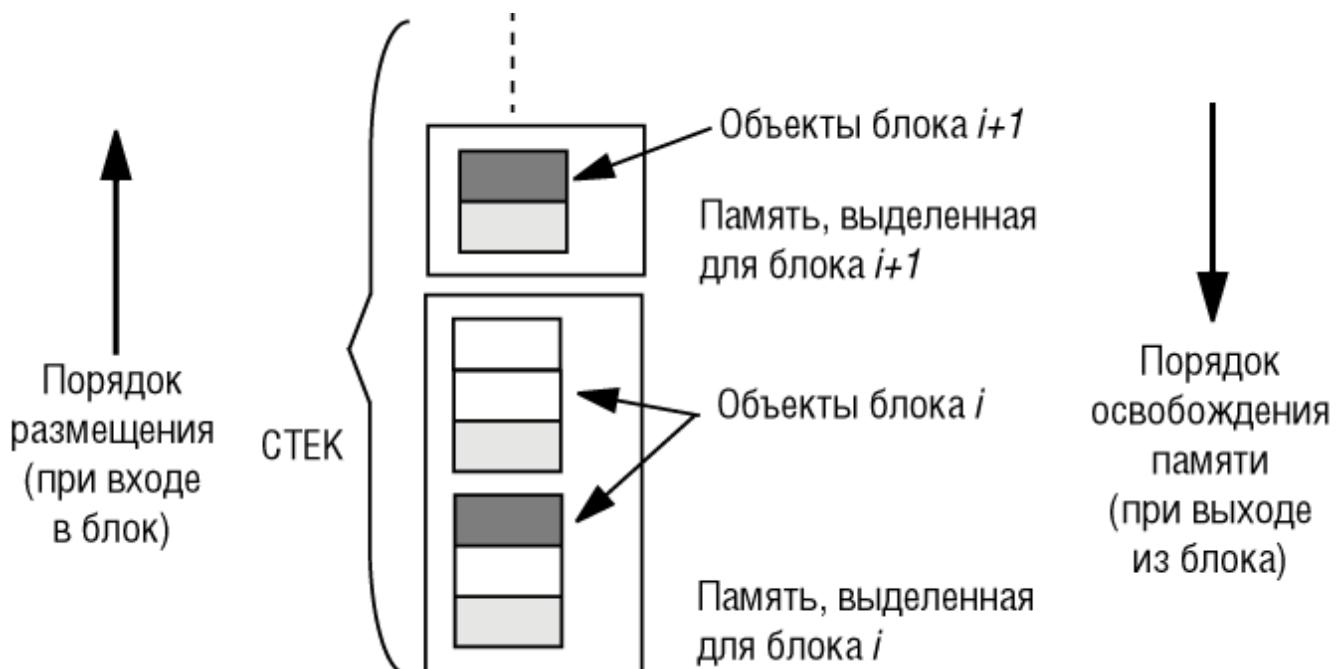


Рис. 9.2. Режим, основанный на стеке

Основанное на стеке управление объектами сделало популярным Algol 60 и с тех пор поддерживается (часто вместе с другими двумя режимами) в большинстве языков. Такой способ поддерживает рекурсию и динамические массивы, границы которых выясняются в процессе выполнения. В Pascal и C этот механизм не применяется к массивам, как это делается в Algol. Однако разработчикам хотелось бы чаще всего именно массивы распределять таким способом. Тем не менее, даже если этот механизм и может быть применен к массивам, размещение в стеке большинства сложных структур данных невозможно.¹¹

Для сложных структур данных нам нужен третий и последний режим: динамическая память, называемая также "кучей", из-за способа ее использования. Это память, в которой объекты создаются динамически по запросу. Сущности могут динамически присоединяться к разным объектам. Во время компиляции обычно нельзя предсказать, какие объекты будут созданы и присоединены к сущности. Кроме того, объекты могут содержать ссылки на другие объекты.

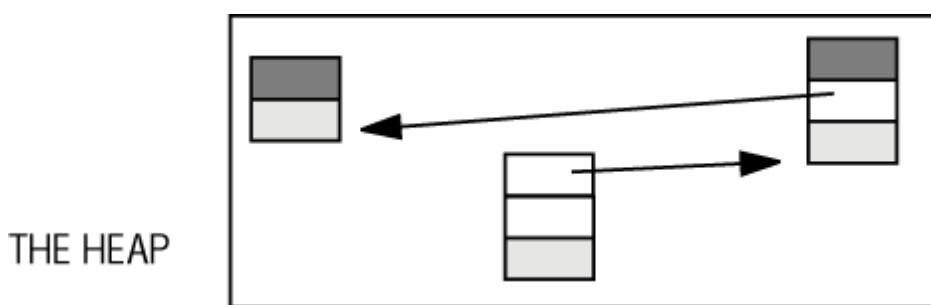


Рис. 9.3. Динамический режим

Динамическая память позволяет создавать сложные динамические структуры данных, необходимые когда, как обсуждалось в предыдущей лекции, ПО требуется вся мощь методов моделирования.

Использование динамического режима

Динамический режим, очевидно, наиболее общий, и он необходим для ОО-программирования. Его используют многие не ОО-языки. В частности:

- Pascal использует статический режим для массивов, режим, основанный на стеке, для переменных, не являющихся массивами и указателями, динамический режим для указателей. В последнем случае создание объекта выполняется с помощью вызова специальной процедуры создания new.
- Язык С похож на Pascal, но дополнительно вводит динамические массивы и статические переменные, не являющиеся массивами, Язык С динамически размещает переменные типа указатель и массивы, используя библиотечную функцию malloc.
- PL/I поддерживает все модели.
- Lisp системы традиционно были высоко динамичны и полагались большей частью на динамический режим распределения памяти. Одна из наиболее важных операций Lisp, используемая многократно для представления списков, - CONS, создает структуру из двух полей. В первом поле хранится значение элемента, а во втором - указатель на следующий элемент. Здесь CONS, скорее источник новых объектов, чем инструкция их создания.

Повторное использование памяти в трех режимах

Для объектов, созданных как в основанном на стеке режиме, так и в динамическом режиме, возникает вопрос, что делать с неиспользуемыми объектами? Возможно ли память, занятую таким объектом, повторно использовать в более поздних инструкциях создания новых объектов?

В статической модели проблемы не существует: для каждого объекта есть одна навсегда присоединенная сущность. Выполнение требует поддерживать связь с объектом все время, пока сущность активна. Поэтому повторное использование памяти невозможно в настоящей трактовке этого понятия. Однако при острой нехватке памяти похожая технология иногда используется. Если вы уверены, что объекты, присоединенные к двум сущностям, никогда не нужны одновременно, и эти сущности не должны сохранять свои значения между последовательными использованиеми, то можно на одной и той же памяти размещать две или более сущности, будучи совершенно уверены в безопасности того, что вы делаете. Эта техника, известная как **перекрытие (overlay)**, достаточно ужасная, все еще практикуется при работе вручную.

Если все-таки использовать перекрытие, то, конечно, его следует выполнять автоматически, используя специальные инструменты, - слишком велика вероятность ошибки. Главной проблемой остается возможность изменений: решение о перекрытии двух переменных может быть корректным на определенном этапе жизни программы. Неожиданное изменение может сделать его неправильным. Мы столкнемся с похожей проблемой ниже, в технологии сборки мусора.

В режиме, основанном на стеке, объекты, присоединенные к сущностям, могут быть размещены в стеке. В языках с блочной структурой ситуация упрощается: размещение объектов происходит одновременно для всех сущностей данного блока, допуская использование одного стека для всей программы. Схема действительно элегантна, потому что использует два множества сопутствующих событий:

Таблица 9.1. Размещение и удаление объектов в языках с блочной структурой

Динамическое свойство (событие времени выполнения)	Статическое свойство (положение в тексте программы)	Техника реализации
Размещение объекта	Начало блока	Выталкивание объектов (один для каждой локальной сущности блока) в стек
Удаление объекта	Конец блока	Выталкивание объектов из стека

Простота и эффективность этой техники реализации является одной из причин успешности языков с блочной структурой. В динамическом режиме все не так просто. Проблема связана с мощью самого механизма: в период компиляции ничего нельзя сказать о создании объекта, невозможно предсказать, когда данный объект может стать ненужным.

Отсоединение

В динамическом режиме объекты могут стать ненужными в непредсказуемые моменты периода выполнения; раз так, то некоторый механизм (определенный позже в этом обсуждении) может освобождать занятую ими память.

Причина - присутствие в этом режиме выполнения операции **отсоединения (detachment)**, обратной к операции присоединения. В предыдущей лекции изучалось, как сущности присоединяются к объектам, но не рассматривались детали отсоединения. Пора это исправить.

Отсоединение распространяется только на объекты x ссылочного типа. Если x развернутого типа - значением x

является объект О, то нет способа отсоединить х от О. Заметьте, однако, если х развернутый атрибут некоторого класса, О представляет подобъект некоторого большого объекта ВО. Тогда ВО, а вместе с ним и О, может стать недостижимым по одной из причин, изучаемых ниже. Посему в оставшейся части этой лекции можно ограничиться рассмотрением сущностей ссылочного типа.

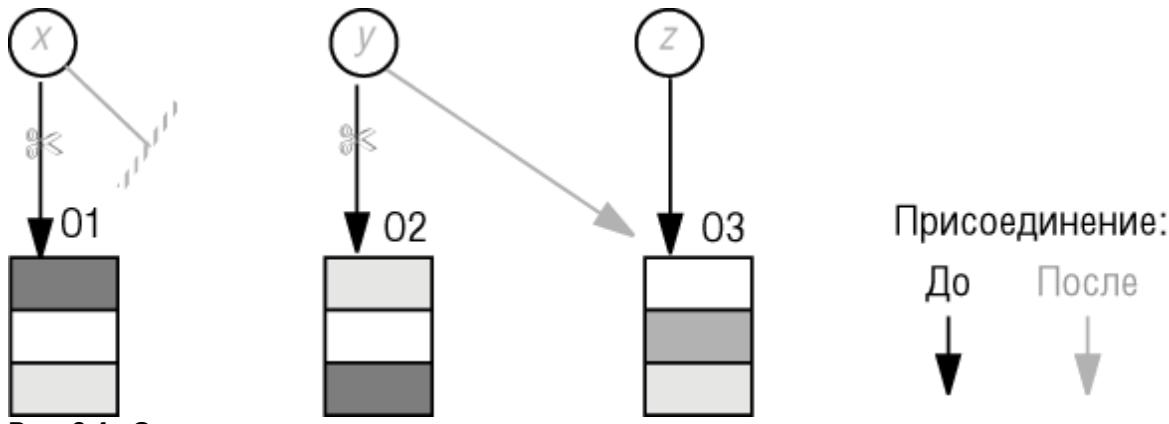


Рис. 9.4. Отсоединение

Основные причины отсоединения следующие. Предположим, х и у сущности ссылочного типа вначале присоединены к объектам О1 и О2. Рисунок иллюстрирует случаи D1 и D2.

- (D1) Присваивание вида $x := \text{Void}$, или $x := v$ где v типа void, отсоединяет х от О1.
- (D2) Присваивание вида $y := z$, где z не присоединен к объекту О2, отсоединяет у от О2.
- (D3) Завершение подпрограммы отсоединяет формальные аргументы от присоединенных к ним объектов.
- (D4) Инструкция создания **create** x , присоединяет х к вновь созданному объекту и, следовательно, отсоединяет х, если он ранее был присоединен к объекту О1.

Случай D3 соответствует ранее данному правилу: инициализация формального аргумента а подпрограммы r во время вызова $t.r(\dots, b, \dots)$, где позиция b в вызове соответствует позиции а в объявлении r, в точности соответствует семантике присваивания $a := b$.

Недостижимые объекты

Значит ли отсоединение объектов, например О1 или О2 ([рис.9.4](#)), что они становятся бесполезными и, следовательно, механизмы периода исполнения могут освободить занимаемое ими место в памяти? Это было бы слишком просто! Сущность, для которой объект был первоначально создан, могла уже потерять интерес к объекту, но из-за динамических псевдонимов другие ссылки могут быть все еще подсоединенны к нему.

Например, [рис.9.4](#) возможно отражает лишь частное видение связей между объектами. Рассматривая более широкий контекст, ([рис.9.5](#)) можно обнаружить, что О1 и О2 все еще достижимы для других объектов.

Но и эта картина все еще не дает полного видения структуры всех связей между объектами. Расширяя контекст, можно, например, выяснить, что О4 и О5 сами не нужны, так что в отсутствии других ссылок, О1 и О2 не нужны тоже.

Таким образом, ответ на вопрос: "Какие объекты можно удалить?" должен следовать из глобального анализа множества всех созданных объектов. Выделим три типа объектов:

- (C1) Объекты, напрямую присоединенные к сущностям, известны (из правил языка программирования) как необходимые.
- (C2) Зависимые от объектов категории C1. (Напомним, наряду с непосредственно зависимыми объектами, имеющими ссылки на объекты C1, зависимые объекты могут рекурсивно иметь ссылки на непосредственно зависимые объекты.) Здесь рассматривается прямая и косвенная зависимость.

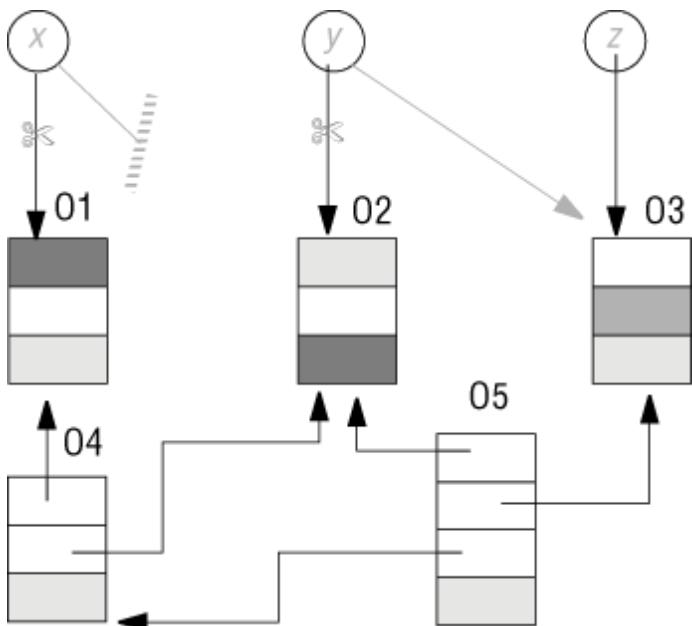


Рис. 9.5. Отсоединение - не всегда смерть объекта

- С3 Объекты, не относящиеся к предыдущим двум категориям.

Объекты первой категории могут называться оригиналами (origins). Вместе с объектами категории С2 они составляют множество достижимых (reachable) объектов. Объекты категории С3 недостижимы (unreachable). Они ранее неформально назывались ненужными или бесполезными. В другой более мрачной терминологии используются термины "мертвые объекты" для категории С3 и "живые" для первых двух. (У программистов принята более прозаическая терминология, и процесс удаления мертвых объектов, изучаемый ниже, называется просто сборкой мусора.)

Для объектов наряду с термином "оригинал" используется термин "корень". Первый термин предпочтительнее, поскольку сама ОО-система имеет "корневой объект" и "корневой класс". Однако результат возможной двусмыслиности не сильно вредит делу, потому что корневой объект, как будет видно далее, является одним из оригиналов.

Первый шаг к решению проблемы управления памятью при использовании динамического режима - выделение достижимых и недостижимых объектов. Для идентификации достижимых объектов нужно начать с оригиналов и пройти по всем многократно возникающим ссылкам. Так что первый вопрос, - как найти оригиналы? Ответ зависит от структуры периода выполнения, определяемой лежащим в ее основе языком программирования.

Достижимые объекты в классическом подходе

Поскольку проблема недостижимости рассматривается в таких классических подходах, как Pascal, С и Ada, разумно начать с этих случаев. (Читатели, незнакомые ни с одним из этих подходов, могут пропустить этот раздел и перейти к следующему, рассматривающему ОО-программирование.)

Все подходы используют стековое размещение объектов и размещение в динамической памяти. Языки С и Ada поддерживают также статическую модель, но для упрощения ее можно проигнорировать, рассматривая статику как специальный случай размещения в стеке. Можно полагать, что статические объекты размещаются в начале выполнения и находятся в конце стека. В языке Pascal они объявляются в самом внешнем блоке.

Общим свойством этих подходов является то, что сущности могут задаваться указателями. В ОО-подходе вместо указателей используются ссылки - более абстрактное понятие (эта тема обсуждалась в предыдущей лекции). Позвольте сделать вид, что указатели есть в действительности ссылки, игнорируя слабо типизируемую природу указателей в С.

При этих допущениях и упрощениях на следующем рисунке показаны оригиналы, размещенные в стеке или присоединенные к ссылке, размещенной в стеке, достижимые и недостижимые объекты.

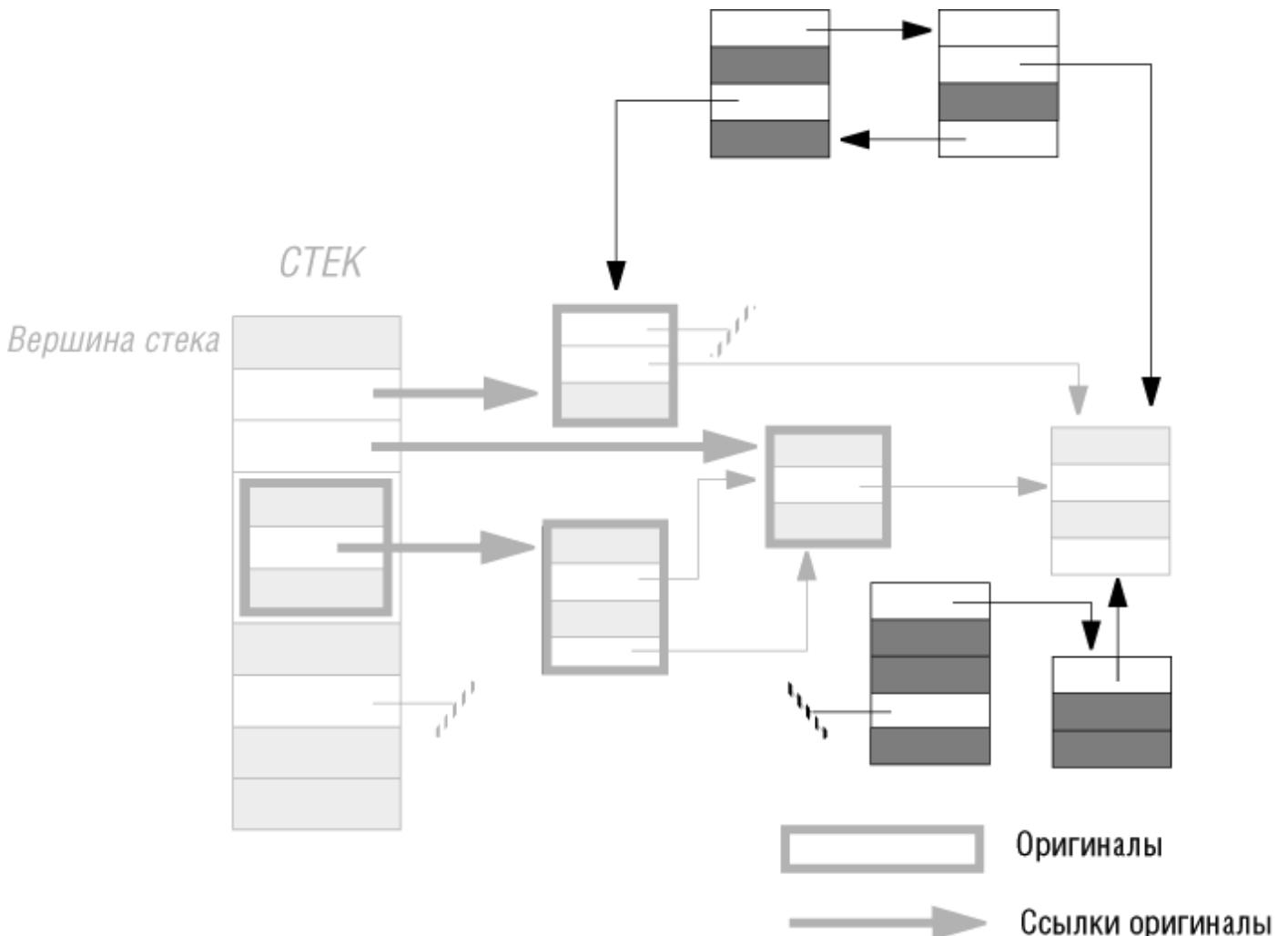


Рис. 9.6. Живые и мертвые объекты в комбинированной модели - стек и динамическая память (живые объекты окрашены в серый цвет)

Проблема недостижимости возникает только для объектов, размещенных в динамической памяти. Такие объекты всегда подсоединяются к сущностям ссылочного типа. Поэтому удобно игнорировать проблему повторного использования памяти для объектов, непосредственно размещенных в стеке. Она может быть решена просто при помощи освобождения стека при окончании блока. Начнем рассмотрение со ссылок, размещенных в стеке. Мы можем назвать их **ссылками оригиналами (reference origins)**. Они изображены толстыми стрелками на рисунке и представляют:

- (O1) Значение локальных сущностей или аргументов функции ссылочного типа (как две верхних начальных ссылки на рисунке).
- (O2) Поля ссылочного типа объектов, расположенных в стеке (ниже лежащая ссылка на рисунке).

Рассмотрим пример объявления типа и процедуры, написанный на смеси Pascal и нотации, используемой в этой книге (**reference G** - ссылка, которая может быть подсоединенена к объекту типа G):

```

type
COMPOSITE =
  record
    m:INTEGER
    r:reference COMPOSITE
  end
  ...
procedure p is
  local
    n: INTEGER
    c: COMPOSITE
    s: reference COMPOSITE
  do
    ...
  end

```

При каждом вызове процедуры p три значения вталкиваются в стек:

СТЕК

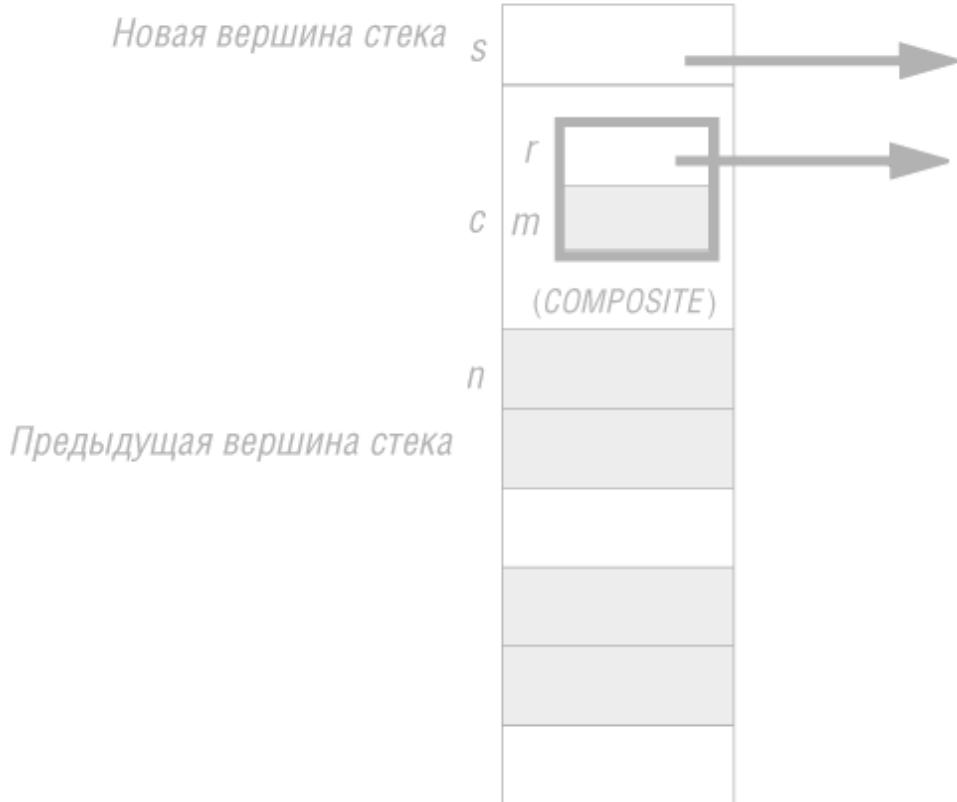


Рис. 9.7. Размещение сущностей для процедуры

Тремя новыми значениями являются: целое *n*, не влияющее на проблему управления объектами (оно исчезнет при завершении процедуры и не ссылается на другие объекты); ссылка *s*, являющаяся примером категории О1; и объект с типа COMPOSITE. Сам объект содержится в стеке и занятая объектом память может быть использована по завершении работы процедуры. Но он содержит ссылочное поле *r*, являющееся примером категории О2.

Итак, для определения достижимости объекта в классическом подходе, комбинирующем стековую и динамическую память, следует начать со ссылок в стеке (переменные ссылочного типа и ссылочные поля комбинированных объектов), и последовательно просмотреть все ссылочные поля присоединенных объектов, если они существуют.

Достижимые объекты в ОО-модели

ОО-структура данных, представленная в предыдущих лекциях, имеет некоторые отличия от рассмотренной выше структуры.

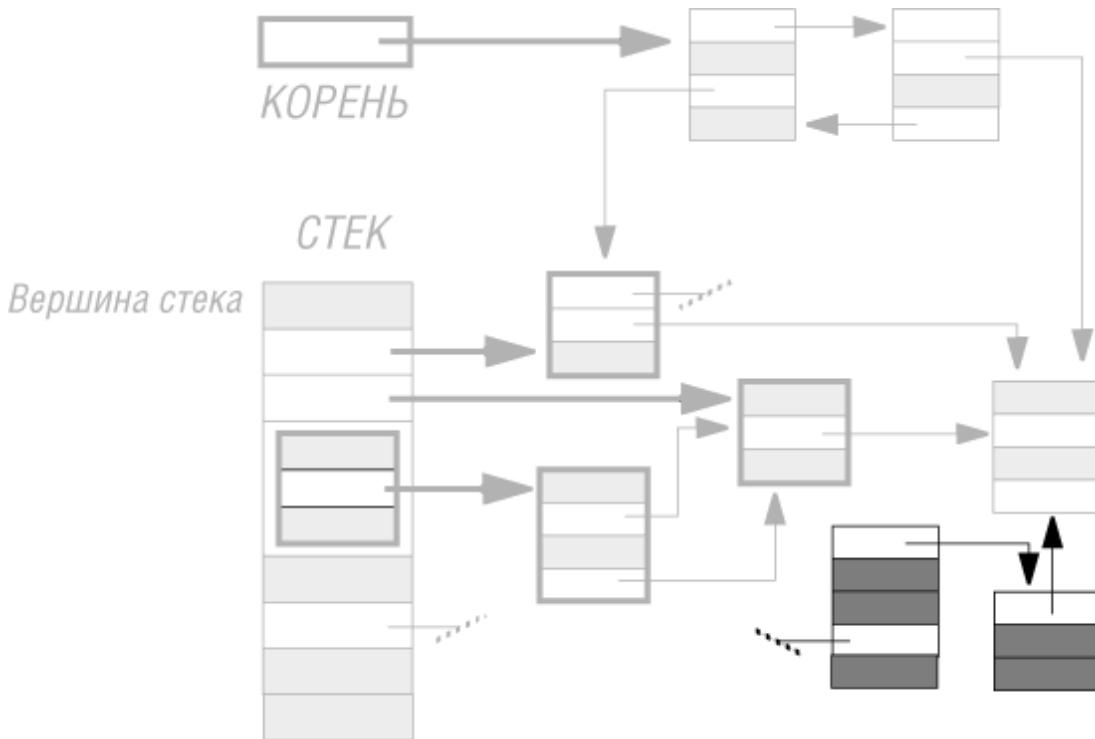


Рис. 9.8. Достижимость в ОО-модели

Работа любой системы начинается с создания объекта, называемого корневым объектом системы, или просто корнем (когда нет путаницы с корневым классом, задаваемым статически). Корень в этом случае является одним из оригиналов.

Другое множество оригиналов возникает из-за возможного присутствия локальных переменных в подпрограмме. Рассмотрим подпрограмму вида

```
some_routine is
    local
        rb1, rb2: BOOK3
        eb: expanded BOOK3
    do
        . . .
        create rb1
        . . . Операции, возможно использующие rb1, rb2 и eb . . .
    end
```

При любом вызове и выполнении подпрограммы `some_routine`, инструкции в теле подпрограммы могут ссылаться на `rb1`, `rb2`, `eb` и на присоединенные к ним объекты, если они есть. Это значит, что такие объекты должны быть частью множества достижимых объектов, но не обязательно зависимы от корня. Заметим, для `eb` всегда есть присоединенный объект, а `rb1` и `rb2` могут при некоторых запусках иметь значение `void`.

Локальные сущности ссылочного типа, такие как `rb1` и `rb2`, подобны переменным подпрограммы, которые в предыдущей модели были размещены в стеке. Локальные сущности развернутого типа, как `eb`, подобны объектам, расположенным в стеке.

Когда завершается очередной вызов `some_routine`, исчезают сущности `rb1`, `rb2` и `eb` текущей версии. В результате все присоединенные объекты перестают быть частью множества оригиналов. Это не значит, что они становятся недостижимыми, - они могут тем временем стать зависимыми от корня или других оригиналов.

Допустим, например, что `a` - это атрибут рассматриваемого класса и что полный текст подпрограммы имеет вид:

```
some_routine is
    local
        rb1, rb2: BOOK3
        eb: expanded BOOK3
    do
        create rb1;create rb2
        a := rb1
    end
```

На следующем рисунке показаны объекты, создаваемые вызовом `some_routine`, и ссылки с присоединенными объектами.

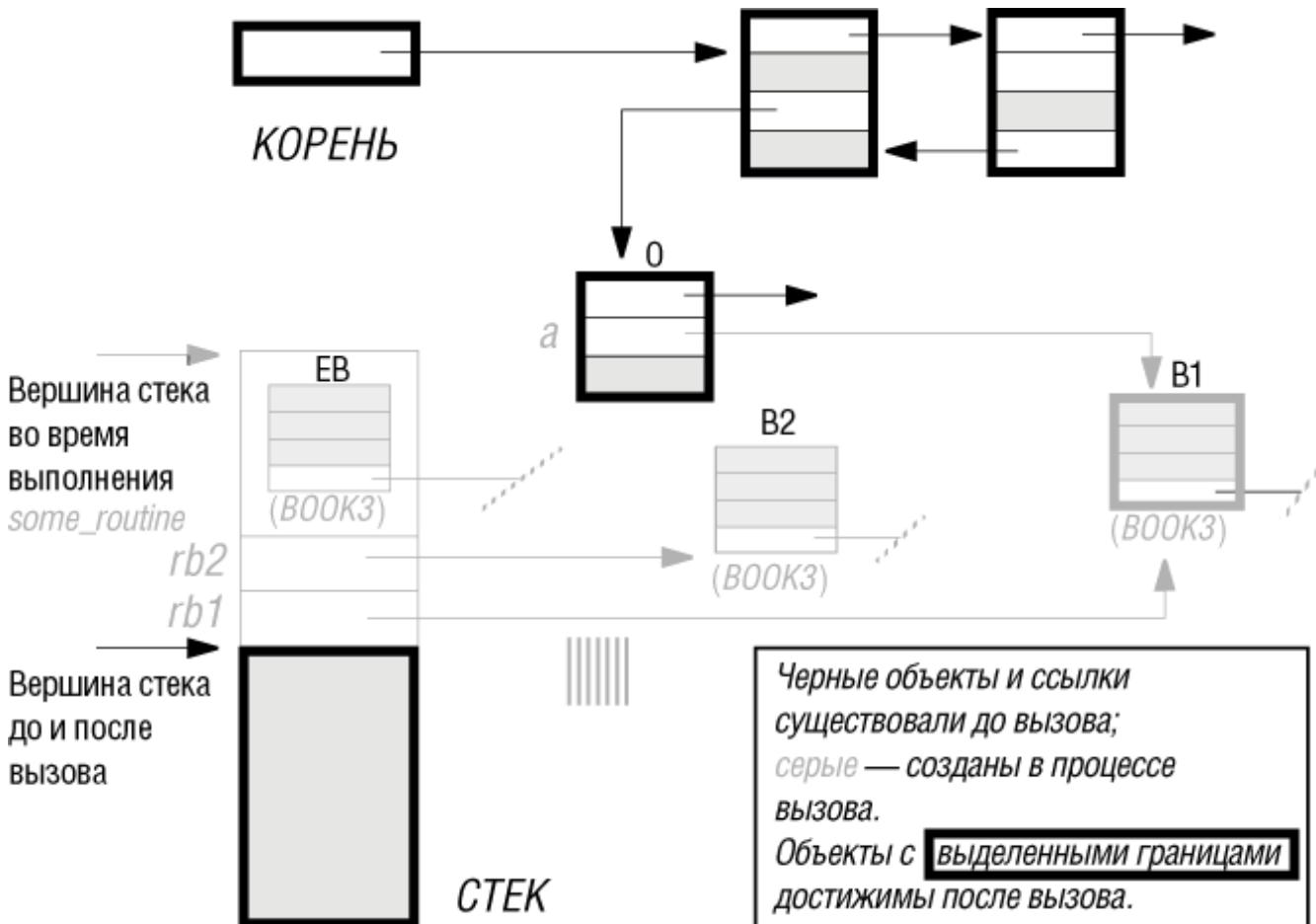


Рис. 9.9. Объекты, присоединенные к локальным сущностям

Когда вызов `some_routine` завершается, объект `O`, представляющий цель вызова, все еще доступен (иначе не было бы этого вызова). Поле `a` этого объекта `O` в результате вызова присоединено к объекту `B1` класса `BOOK3`, созданного первой инструкцией создания нашей подпрограммы. Поэтому объект `B1` остается достижимым по завершении вызова. Напротив, объекты `B2` и `EB`, которые были присоединены к `rb2` и `eb` во время вызова, теперь становятся недостижимыми: в соответствии с текстом процедуры невозможно, чтобы какой-либо другой объект "запомнил" `B2` или `EB`.

Проблема управления памятью в ОО-модели

Подводя итог предшествующего анализа, определим оригиналы и соответственно достижимые объекты:

Определение: начальные, достижимые и недостижимые объекты

В каждый момент времени выполнения системы множество оригиналов включает:

- Корневой объект системы.
- Любой объект, присоединенный к локальной сущности, или формальному аргументу, выполняемой в данный момент подпрограммы (для функции включается локальная сущность `Result`).

Любые объекты, прямо или косвенно зависящие от оригиналов, **достижимы**. Любые другие объекты недостижимы. Память, занятую недостижимыми объектами можно восстановить, (например, выделить ее другим объектам) сохраняя корректность семантики выполнения программы.

Проблема управления памятью возникает из-за непредсказуемости операций, влияющих на множество достижимых объектов: создание и отсоединение.

Такое предсказание возможно в некоторых случаях для строго управляемых структур данных. Примером является библиотечный класс, задающий список, - `LINKED_LIST`, рассматриваемый позже, и связанный с ним класс `LINKABLE`, описывающий элементы этого списка. Экземпляр `LINKABLE` создается только с помощью специальных процедур класса `LINKED_LIST` и может становиться недостижимым только в результате выполнения процедуры `remove`, удаляющей элементы списка. Для подобных классов можно представить себе особенную процедуру управления памятью. (Такой подход будет изучен позднее в этой лекции.)

Приведенный пример, хотя и важен, но только для специальных случаев. В общем случае приходится отвечать на сложный вопрос - что делать с недостижимыми объектами?

Три ответа

С недостижимыми объектами можно поступать тремя способами:

- Проигнорировать проблему и надеяться, что хватит памяти для размещения всех объектов: достижимых и недостижимых. Это можно назвать **несерьезным (casual) подходом**.
- Предложить разработчикам включать в каждое приложение алгоритм, ищащий недостижимые объекты, и дать им механизм освобождения соответствующей памяти. Такой подход называется **восстановлением вручную(manual reclamation)** .
- Включить в среду разработки (как часть исполняемой системы (runtime system)) механизм, автоматически определяющий и утилизирующий недостижимые объекты. Этот подход принято называть **автоматической сборкой мусора (automatic garbage collection)**.

Остаток лекции посвящен этим подходам.

Несерьезный подход (тривиальный)

Первый подход заключается в игнорировании проблемы: предоставлять мертвые объекты их судьбе. Создаются объекты как обычно, но никто не волнуется о том, что может потом случиться с объектами.

Может ли быть оправдан несерьезный подход?

Несерьезный подход не создает проблем в системах, создающих небольшое число объектов, например, при проведении простых тестов и экспериментов.

Более интересен случай, когда система может создавать много объектов, гарантируя, что ни один или немногие из них станут недостижимыми. Этот случай аналогичен статической схеме размещения, в которой ни один объект не удаляется. Разница только в том, что создание происходит динамически во время выполнения программы. Несерьезный подход в этом случае оправдан, поскольку практически не возникает необходимости утилизации объектов.

Некоторые программы реального времени следуют этой схеме: по причине эффективности, создавая все необходимые объекты статично или во время инициализации, избегая непредсказуемых моделей динамического создания.

Этот метод применяется в "жестких" системах реального времени ("hard-real-time"), требующих гарантированное микросекундное время отклика на внешние события (например, системы обнаружения ракет). В таких системах время выполнения каждой операции должно быть полностью предсказуемо. Но тогда приходится отказываться не только от управления памятью, но и от динамического создания объектов, рекурсии, вызова процедур с локальными сущностями и так далее. Работа с такими системами подразумевают специализированную машину с одним исполняемым процессом, фактически без операционной системы в обычном понимании этого термина. В таких средах люди предпочитают писать на языках ассемблера, из-за страха дополнительных неожиданностей от генерированного компилятором кода. Все это сводит обсуждение к малой, хотя и стратегически важной области мира программ.

Надо ли заботиться о памяти?

Другой аргумент, который можно услышать в оправдание несерьезному подходу, - это постоянный рост объема доступной памяти компьютера и уменьшение цены памяти.

Используемая память может быть как виртуальной, так и реальной. В системах виртуальной памяти первичная и вторичная память делится на блоки, называемые страницами. Вновь требуемые страницы первичной памяти вытесняют во вторичную память редко используемые страницы первичной памяти. Если такая система используется для работы с ОО-системами, страницы, содержащие недостижимые объекты, будут вытесняться и освободят основную память для часто используемых объектов.

Если бы действительно в нашем распоряжении было почти безграничное количество почти свободной памяти, можно было бы удовлетвориться несерьезным подходом.

К несчастью, это не так.

Первая причина в том, что на практике виртуальная память не эквивалентна реальной. Если хранить большое количество объектов в виртуальной памяти, в которой меньшинство достижимых объектов рассыпано среди большинства недостижимых, то процесс выполнения будет постоянно вызывать перемещение страниц памяти, феномен, известный как **пробуксовка (trashing)**, приводящий к драматическому увеличению времени выполнения. Действительно, системы виртуальной памяти усложняют эффективное разделение двух основных аспектов - пространства и времени. (См. "Эффективность", [лекцию 1](#).)

Но есть более важное ограничение применения несерьезного подхода. Даже большая память имеет границы. Удивительно, как быстро программисты к ним подходят. Как только мы выходим за пределы систем с небольшим числом недостижимых объектов, лицом к лицу сталкиваемся с проблемой восстановления памяти.

Байт здесь, байт там, и реальные покойники

Пора послушать печальную и поучительную историю Лондонской службы скорой помощи.

Лондонская служба скорой помощи, как говорят, самая большая в мире, обслуживает территорию около 1500 кв. км, с постоянным населением почти в семь миллионов человек и с еще большим количеством населения в дневное время. Каждый день эта служба обслуживает пять тысяч пациентов и получает от двух до трех тысяч звонков.

Как можно догадаться по мрачному заголовку, в этой работе используются компьютеры (точнее ПО). Вначале было несколько неудачных разработок, даже не введенных в действие, как несоответствующих требованиям, несмотря на то что на разработку этих систем затрачивались значительные финансовые средства. Наконец, в 1992 была введена в эксплуатацию новая система, разработанная за миллион фунтов. Скоро о ней снова заговорили. 28 и 29 октября на телевидении и в прессе сообщалось, что из-за неадекватной работы системы были потеряны двадцать жизней. Говорили, что в одном конкретном случае врачи скорой помощи по радио сообщили на базу, что прибыли на место назначения и спросили, почему владелец похоронного бюро прибыл раньше.

Исполнительный директор службы ушел в отставку, была назначена следственная комиссия.

Служба скорой помощи после трагедии не сразу отказалась от компьютерной системы, а переключилась на гибридную модель - частично ручную, частично полагающуюся на систему. Согласно официальным отчетам:

Эта гибридная система действовала с переменным успехом с 27 октября 1992 г. до раннего утра 4 ноября. Однако после двух часов 4 ноября работа системы значительно замедлилась, вскоре после этого система вышла из строя совсем. Перезагрузка не смогла решить проблему. Управление и персонал вернулись к бумаге и телефонным звонкам.

Что привело систему к краху, так что ее не могли сохранить даже как дополнение к ручным операциям? Отчет определил несколько причин. Вот основная:

Следственная команда пришла к выводу, что крах системы был вызван незначительной ошибкой программного обеспечения. Программист ХХ ("XX" здесь и далее заменяет название компании, разрабатывающей программное обеспечение данной системы) тремя неделями раньше оставил в системе кусок программного кода, который, используя небольшой файл на сервере, записывал и не стирал информацию о выезде машины на вызов. Через три недели память переполнилась. Эта ошибка была вызвана беспечностью и отсутствием проверки качества программного кода. Такого рода неисправности вряд ли могут быть обнаружены обычным тестированием программистом или пользователем.

Читатель должен сам решить, какую программную ошибку стоит называть незначительной, особенно принимая во внимание последний комментарий о трудностях тестирования, который еще будет подробно обсуждаться ниже.

Для тех, кто все еще думает, что можно пользоваться несерьезным подходом, и для тех, кто относится к управлению памятью только как к проблеме реализации, двадцать жертв лондонской службы скорой помощи должны служить грустным напоминанием о серьезности рассматриваемой проблемы.

Восстановление памяти: проблемы

Если уйти от несерьезного подхода и его упрощающих допущений, то предстоит решить, как и когда восстанавливать память. Возникают две проблемы:

- **Обнаружение (detection).** Как найти мертвые элементы?
- **Восстановление (reclamation).** Как восстановить для повторного использования память, присоединенную к этим элементам?

Для каждой из этих задач можно искать решение на одном из двух возможных уровнях:

- **Реализации языка** - компилятор и среда исполнения обеспечивают общую поддержку любому ПО, создаваемому на этом языке и в данной среде.
- **Приложения** - приложение само решает возникающие проблемы.

В первом случае управление выделенной памятью происходит автоматически с помощью программно-аппаратных средств. Во втором случае каждый разработчик приложения должен позаботиться об этом сам.

Фактически, существует еще третий возможный уровень, нечто среднее между этими двумя, - фабрика компонентов. Функции управления памятью возлагаются на общечелевые повторно используемые классы библиотеки ОО-среды. Подобно уровню приложения, можно использовать только разрешенные конструкции языка программирования, не имея прямого доступа к аппаратуре и функциям операционной системы. Подобно уровню реализации языка, проблемы управления памятью решаются один раз и для всех приложений.

Даны две проблемы и три способа решения каждой, в итоге - девять возможных вариантов. Только четыре из них имеют практический смысл. Рассмотрим их.

Удаление объектов, управляемое программистом

Одно популярное решение - обнаружение мертвых элементов возложить на разработчика программы, а восстановление памяти решать на уровне реализации языка.

Это простейшее решение для реализаторов языка: все, что от них требуется, - это ввести в язык примитив, скажем, `reclaim`, такой что `a.reclaim` сообщает системе, что объект, присоединенный к `a`, не нужен, и соответствующие ячейки памяти можно освободить для новых объектов.

Это решение реализовано в не ОО-языках, таких как Pascal (`dispose` процедура), C (`free`), PL/I (`FREE`), Modula-2 и Ada. Оно есть в большинстве "гибридных ОО" языков, в частности в C++.

Такое решение особенно приветствуется в мире С-программистов, любящих полностью контролировать происходящее. Обычная реакция таких программистов на тезис о том, что Objective-C может давать преимущества, благодаря автоматическому восстановлению памяти, следующая:

Я говорю, НЕТ! Оставлять недостигимые объекты - ПЛОХОЙ СТИЛЬ ПРОГРАММИРОВАНИЯ. Если вы создаете объект, вы должны отвечать за его уничтожение, если вы им не пользуетесь. Разве мама не учила вас убирать свои игрушки после игры? (Послано Яном Стефенсоном (Ian Stephenson), 11 мая 1993.)

Для серьезной разработки программ эта позиция не позовительна. Хорошие разработчики должны разрешать кому-либо другому играть со своими "игрушками" по двум причинам: надежности и простоты разработки.

Проблема надежности

Допустим, разработчик управляет утилизацией объектов с помощью механизма `reclaim`. Возможность ошибочного вызова `reclaim` всегда существует; особенно при наличии сложных структур данных. В жизненном цикле ПО `reclaim`, бывшее когда-то правильным, может стать некорректным.

Такие ошибки приводят к проблеме висячих ссылок, - когда в одном из полей существующего объекта хранится ссылка на удаленный объект. Если система, после того как область памяти, занимаемая этим объектом, была утилизирована и использована для хранения другой информации, попытается использовать ссылку, то результатом будет крах программы или (еще хуже) ее ошибочное или неуправляемое поведение.

Этот тип ошибки известен, как источник появления самых частых и неприятных жучков в практике языка С и производных языков. Программисты боятся таких жучков из-за трудности обнаружения их источника. Если программист не заметил, что определенная ссылка еще присоединена к объекту и как результат - ошибочно выполняет `reclaim`, то это часто происходит из-за того, что ссылка находится в другой части программы. Если так, то должна быть большая физическая и концептуальная дистанция между ошибкой - вызовом `reclaim` и ее проявлением - крах или другое ненормальное поведение из-за попытки применения некорректной ссылки. Проявиться ошибка может значительно позже и, по-видимому, совсем в другой части программы. К тому же, ошибка может быть плохо воспроизводимой, поскольку распределение памяти операционной системой не всегда происходит одинаково и может зависеть от внешних по отношению к программе факторов.

Сказать, что причиной этих ошибок является "плохой стиль программирования", как в письме, упомянутом выше, это не сказать ничего. Человеку свойственно ошибаться; ошибки при программировании неизбежны. Даже в приложениях средней сложности, нет разработчиков, которым можно доверять, нельзя доверять самому себе в способности проследить за всеми объектами периода выполнения. Это работа не для человека, с ней может справиться только компьютер.

Многие из С или C++ программистов ночи проводят, пытаясь понять, что произошло с одной из их игрушек. Нередко, что проект задерживается из-за загадочных ошибок при работе с памятью.

Проблема простоты разработки

Даже если можно было бы избежать ошибочных вызовов `reclaim`, остается вопрос - сколь реально просить разработчиков управлять удалением объектов? Загвоздка в том, что даже при обнаружении объекта, подлежащего утилизации, обычно просто удалить его недостаточно, он может сам содержать ссылки на другие объекты и нужно решить, что с ними делать.

Рассмотрим структуру, показанную на [рис. 9.10](#), ту же, что использовалась в предыдущей лекции для описания динамической природы объектных структур. Допустим, выяснилось, что можно утилизировать самый верхний объект. Тогда в отсутствии каких-либо других ссылок можно удалить и другие два объекта, на один из которых он ссылается прямо, а на другой - косвенно. Не только можно, но и **нужно**: разве хорошо удалять только часть структуры? В терминологии Pascal это иногда называется рекурсивной проблемой удаления: если операции утилизации имеют смысл, они должны быть рекурсивно применены ко всей структуре данных, а не только к одному индивидуальному объекту. Но конечно, необходимо быть уверенным, что на объекты удаляемой структуры нет ссылок из внешних объектов. Это трудная и чреватая ошибками задача.

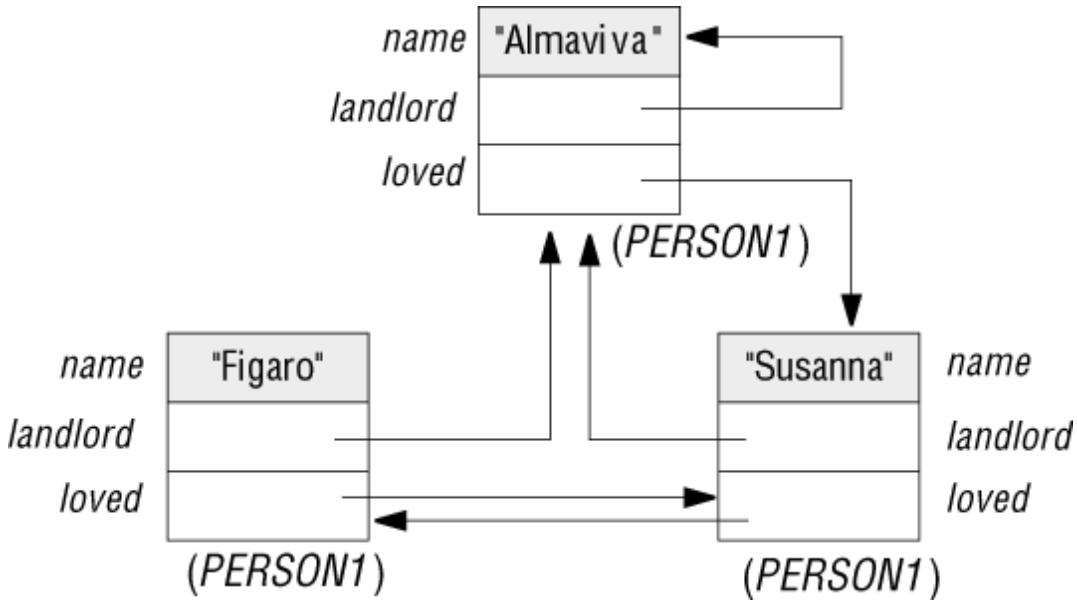


Рис. 9.10. Прямые и косвенные взаимные ссылки

На этом рисунке все объекты одного типа PERSON1. Предположим, что сущность x присоединена к объекту О типа MY_TYPE , объявленным как класс:

```

class MY_TYPE feature
    attr1: TYPE_1
    attr2: TYPE_2
end

```

Каждый объект типа MY_TYPE , такой как О, содержит две ссылки, которые (кроме void) присоединены к объектам типа TYPE_1 и TYPE_2 . Утилизация О может предполагать, что эти два объекта тоже должны быть утилизированы, также как и зависимые от них объекты. Выполнение рекурсивной утилизации, в этом случае, предполагает написание множества процедур утилизации, - по одной для каждого типа объектов, которые, в свою очередь, могут содержать ссылки на другие объекты. Результатом будет множество взаимно рекурсивных процедур большой сложности.

Все это ведет к катастрофе. Нередко, в языках, не поддерживающих автоматическую сборку мусора, в приложения включаются специально разработанные системы управления памятью. Такая ситуация неприемлема. Разработчик приложения должен иметь возможность сконцентрироваться на своей работе, а не стать счетоводом или сборщиком мусора.

Возрастающая сложность программы из-за ручного управления памятью приводит к падению качества. В частности, она затрудняет читаемость и такие свойства как простота обнаружения ошибок и легкость модификации. В результате, к сложности конструкции добавляется проблема надежности. Чем сложнее система, тем больше вероятность содержания ошибок. Дамоклов меч ошибочного вызова `reclaim` всегда висит над головой и, скорее всего, упадет в наихудшее время: когда система пройдет тестирование и начнет использоваться, создавая большие и замысловатые структуры объектов.

Вывод очевиден. Кроме жестко контролируемых ситуаций (рассмотренных в следующем разделе), ручное управление памятью не подходит для разработки серьезных систем, как минимум, по соображениям качества.

Подход на уровне компонентов

(Этот раздел описывает решение, полезное только для специального случая; его можно пропустить при первом чтении книги.)

Перед тем как перейти к амбициозным схемам, таким как автоматическая сборка мусора, стоит посмотреть на решение, которое может быть альтернативой предыдущему, исправляя некоторые его недостатки.

Это решение применимо только для ОО-программирования "снизу-вверх", где структуры данных создаются не для нужд конкретной программы, а строятся как повторно используемые классы.

Что предлагает ОО-подход по отношению к управлению памятью? Одна из новинок скорее организационная, чем техническая: в этом подходе большое внимание уделяется повторному использованию библиотек. Между разработчиками приложения и создателями системных средств - компилятора и среды разработки - стоит третья группа людей, отвечающих за написание повторно используемых компонентов, реализующих основные структуры данных. Членов третьей группы, которые, конечно могут иногда выступать и в двух других ипостасях, принято называть производителями компонентов (component manufacturers).

Производители компонентов имеют полный контроль над любым использованием данного класса и потому

находятся в лучшем положении при поиске приемлемого решения проблемы управления памятью для всех экземпляров этого класса.

Если модель размещения и удаления объектов класса достаточно проста, разработчики компонентов могут найти эффективное решение, возможно, даже не требующее специальной подпрограммы `reclaim`. Они могут выразить все в терминах понятий высокого уровня. Это и называется подходом на уровне компонентов.

Управление памятью связного списка

Приведем пример подхода на уровне компонентов. Рассмотрим класс `LINKED_LIST`, описывающий список, состоящий из заголовка (header) и набора связанных ячеек, являющихся экземплярами класса `LINKABLE`. Модель размещения и удаления для связного списка проста. Объектами рассмотрения являются связанные ячейки. В этом примере производители компонентов (люди, отвечающие за классы `LINKED_LIST` и `LINKABLE`) знают точно, как создаются и как становятся "мертвыми" экземпляры класса `LINKABLE` - процедурами вставки и удаления. Поэтому они могут управлять соответствующей памятью особенным способом.

Допустим, класс `LINKED_LIST` имеет только две процедуры вставки: `put_right` и `put_left`, вставляющие новый элемент справа или слева от текущей позиции курсора. Каждой процедуре вставки необходимо создать ровно один новый `LINKABLE` объект. Типичная реализация приведена ниже:

```
put_right (v: ELEMENT_TYPE) is
    - Вставка элемента со значением v правее позиции курсора.
    require
        ...
    local
        new: LINKABLE
    do
        create new.make (v)
        active.put_linkable_right (new)
        ... Инструкции по изменению других связей...
    end
```

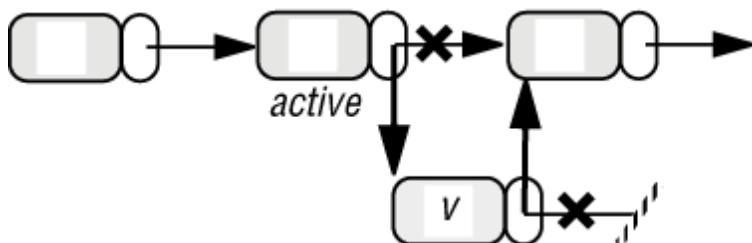


Рис. 9.11. Связный список

Инструкция создания `create new.make (v)` дает указание уровню реализации языка разместить в памяти новый объект.

Точно так же, как мы управляем тем, где создавать объекты, мы точно знаем, где они становятся недостижимыми, - в процедурах удаления. Пусть в нашем классе три такие процедуры: `remove`, `remove_right`, `remove_left`. Могут быть также и другие процедуры, такие как `remove_all_occurrences` (которая удаляет все экземпляры с определенным значением) и `wipe_out` (удаляет все элементы списка). Допустим, что если они присутствуют, то используют первые три процедуры удаления. Процедура `remove`, например, может иметь следующую форму:

```
remove is
    - удаляет элемент текущей позиции курсора.
    do
        ...
        previous.put_linkable_right (next)
        ... Инструкции по изменению других связей...
        active := next
    end
```

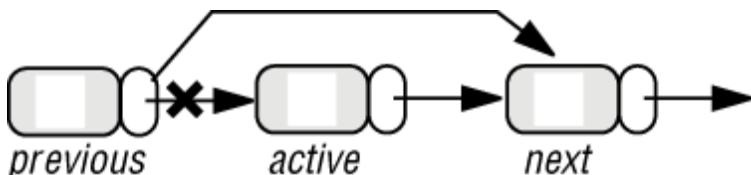


Рис. 9.12. Удаление объекта

Эти процедуры удаления представляют точный контекст обнаружения недостижимых объектов и, при желании, предоставят эти объекты для последующего использования. В отсутствие какой-либо автоматической схемы освобождения памяти разработчик компонентов может безопасно резервировать освобождающуюся память. Если предыдущее удаление создало недостижимые LINKABLE объекты и разместило их где-то для последующего использования, то можно их использовать, когда вставка требует создания новых элементов.

Предположим, экземпляры LINKABLE хранятся в структуре данных, называемой available. Она будет представлена ниже. Тогда можно заменить инструкции создания типа **create new.make (v)** в **put_right** и **put_left** на

```
new := fresh (v)
```

где **fresh** закрытая функция класса LINKED_LIST, возвращающая готовый к использованию экземпляр linkable. Функция **fresh** пытается получить память из available списка, и выполнит создание нового элемента только, когда этот список пуст.

Элементы будут попадать в available в процедурах удаления. Например, тело процедуры **remove** теперь должно быть:

```
do
    recycle (active)
        - остальное без изменений:
    ... Инструкции по обновлению связей: previous, next, first_element, active...
```

где **recycle** новая процедура LINKED_LIST играет роль, противоположную **fresh**, добавляя свой аргумент в available. Эта процедура будет закрытой, она нужна только для внутреннего использования.

Работа с утилизированными объектами

Для реализации **fresh** и **recycle**, можно среди других возможных вариантов представить available как стек: **fresh** будет удалять элемент из стека, а **recycle** будет помещать элемент в стек. Создадим класс STACK_OF_LINKABLES для этого случая и добавим следующие закрытые компоненты в класс LINKED_LIST (В упражнении У23.1. требуется определить, будет ли корректным появление у функции **fresh** побочных эффектов.):

```
available: STACK_OF_LINKABLES
fresh (v: ELEMENT_TYPE): LINKABLE is
    - Новый элемент со значением v, для повторного
    - использования во вставке
do
    if available.empty then
        - Создание нового элемента
        create Result.make (v)
    else
        - Повторное использование linkable
        Result := available.item; Result.put (v); available.remove
    end
end
recycle (dead: LINKABLE) is
    - Возвращает dead в список достижимых элементов.
require
    dead /= Void
do
    available.put (dead)
end
```

Мы можем объявить класс STACK_OF_LINKABLES следующим образом:

```
class
    STACK_OF_LINKABLES
feature {LINKED_LIST}
    item: LINKABLE
        - Элемент в вершине стека
```

```

empty: BOOLEAN is
    - нет элементов в стеке?
do
    Result := (item = Void)
end
put (element: LINKABLE) is
    - Добавить элемент в вершину стека.
require
    element /= Void
do
    element.put_right (item); item := element
end
remove is
    - Удалить последний добавленный элемент.
require
    not empty
do
    item := item.right
end
end

```

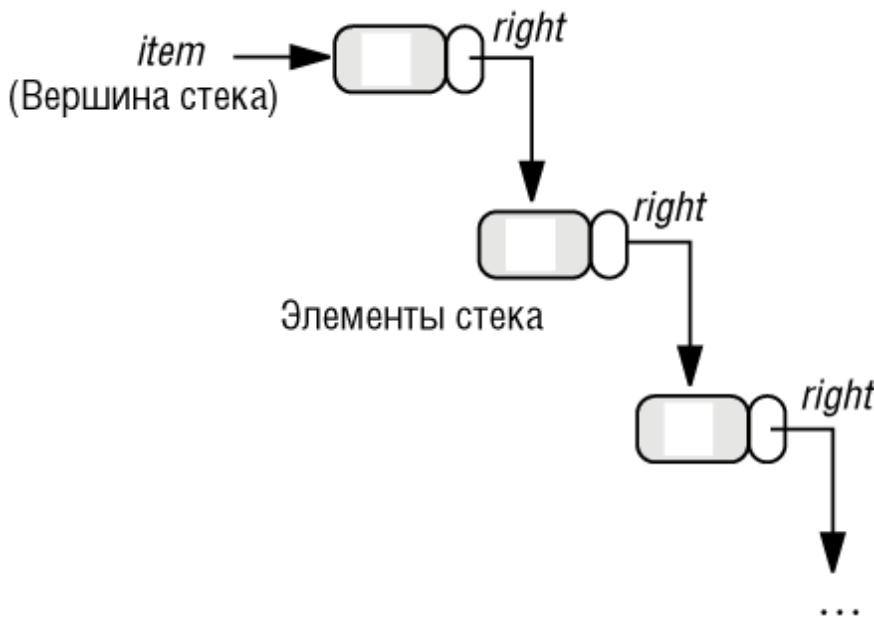


Рис. 9.13. STACK_OF_LINKABLES

Представление стека использует все преимущества поля `right`, присутствующего в каждом элементе `LINKABLE`, связывая все утилизированные элементы и предоставляя, тем самым, дополнительную память для размещения новых элементов списка `LINKED_LIST`. Класс `LINKABLE` должен экспортировать свои компоненты `right` и `put_right` в класс `STACK_OF_LINKABLES`.

Компонент `available` является атрибутом класса. Это означает, что каждый связный список будет иметь свой собственный стек. Конечно, память можно было бы использовать эффективнее в системе, содержащей несколько списков и единственный стек для всех удаленных элементов. Такая техника однократных функций (*once functions*), будет представлена позже; применение ее для `available` означает, что только один экземпляр класса `STACK_OF_LINKABLES` будет существовать до конца выполнения системы, что означает достижение поставленной цели. (Упражнение У9.3. и У9.4. Об однократных функциях см. [лекцию 18](#))

Дискуссия

Этот пример показывает, как подход на уровне компонентов может облегчить проблему восстановления памяти. Подразумевается, что реализации языка не предоставляет автоматического механизма сборки мусора, описанного в следующих разделах. Не обременяя приложение проблемами управления памятью, решение передается повторно используемым библиотечным классам, созданных производителями компонентов.

Недостатки и польза - понятны. Проблемы ручного управления памятью (угроза ненадежности, монотонность) не исчезают магически. Защищенная от неправильного использования схема управления памятью, например, для связного списка, - трудна. Но вместо того, чтобы бороться с проблемой каждому разработчику приложений, работа возлагается на производителя компонентов. Чрезмерные усилия, затрачиваемые производителями компонент, окупаются тем, что созданные компоненты многократно используются различными приложениями.

Автоматическое управление памятью

Ни один из рассмотренных подходов не является полностью удовлетворительным. Общее решение проблемы управления памятью предполагает серьезную работу на уровне реализации языка.

Необходимость автоматических методов

Хорошая ОО-среда должна предлагать механизм автоматического управления памятью, который обнаруживал бы и утилизировал недостижимые объекты, позволяя разработчикам приложений концентрироваться на своей работе - разработке приложений.

Предыдущее обсуждение достаточно ясно показало, как важно иметь возможность управлять памятью. По словам Михаила Швейцера (Michael Schweitzer) и Ламберта Стреттера (Lambert Strether): "ОО-программа без автоматического управления памятью то же самое, что сковорочка без клапана безопасности: рано или поздно она взорвется!" (Из [Schweitzer 1991])

Многие среды разработки, разрекламированные как ОО, не поддерживают такие механизмы. Они могут иметь свойства, делающие их привлекательными на первый взгляд. Они даже могут безупречно работать в малых системах. Но в серьезном проекте вы рискуете разочароваться в среде, когда приложение достигнет реального размера. В заключение конкретный совет:

При выборе ОО-среды - или просто компилятора ОО-языка - для разработки программного продукта ограничьте ваше внимание только теми решениями, которые предлагают автоматическое управление памятью.

Два главных подхода применимы при автоматическом управлении памятью: подсчет ссылок и сборка мусора. Они оба достойны внимания, хотя второй намного мощнее и обще применим.

Что в точности понимается под восстановлением?

Прежде чем рассмотреть подсчет ссылок и сборку мусора, займемся одной технической деталью. В любой форме автоматического управления памятью возникает вопрос, - каков механизм утилизации объекта, определенного как недостижимый? Возможны две интерпретации:

- Механизм может добавить память, занимаемую объектом, к постоянно поддерживаемому "списку свободных ячеек", в духе техники, использованной при рассмотрении подхода на уровне компонентов. Последующая инструкция создания (`create` x . . .) вначале обратится к этому списку для выделения памяти новому объекту. Только если этот список пуст или нет подходящих ячеек, инструкция запросит память у операционной системы. Этот подход может быть назван **внутренний список свободной памяти**.
- Альтернативой является возвращение занимаемой объектом памяти операционной системе. На практике это решение включает в себя некоторые аспекты первого: для избежания переизбытка системных вызовов, утилизированные объекты могут временно храниться в списке, возвращаемого операционной системе при достижении определенного предела. Этот подход может быть назван **реальным восстановлением**.

Хотя возможны оба решения, долго работающие системы требуют реального восстановления. Причина очевидна. Рассмотрим приложение, которое никогда не останавливается. Оно создает объекты, большинство из которых становятся недостижимыми. Существует верхняя граница количества одновременно достижимых объектов, в то время как общее количество созданных с начала работы объектов не ограничено. Тогда при подходе **внутренних списков свободной памяти** возможна ситуация, когда приложение постоянно запрашивает большую, чем нужно, память. В упражнении У9.1 этой лекции требуется создать образец программы, демонстрирующий такое поведение.

Было бы большим разочарованием иметь автоматическое управление памятью и оказаться в ситуации лондонской службы скорой помощи, - посягая без причин байт за байтом на доступную память, пока выполнение не выйдет за рамки памяти и не закончится катастрофой.

Подсчет ссылок

Простая идея лежит в основе первого метода управления памятью - подсчета ссылок. Каждый объект хранит текущее число сделанных на него ссылок. Когда оно становится равным нулю, объект можно утилизировать.

Это решение не сложно для реализации на уровне языка. Нужно обновлять число ссылок любого объекта в ответ на операции, создающие новый объект, присоединения и отсоединения объекта.

Любая операция, создающая объект, инициализирует число ссылок, делая его равным единице. В частности, так должно происходить с инструкцией создания `create` a , создающей объект и присоединяющей его к a. (Ситуация с инструкцией `clone` вкратце будет обсуждена позже.)

Любая операция, присоединяющая новую ссылку к объекту O, должна увеличивать число ссылок O на единицу. Имеются два вида операций присоединения, в которых значение a представляет ссылку, присоединенную к O:

A1 L b := a (присваивание).

A2 L x.r(..., a, ...) , где r - некоторая подпрограмма (передача аргумента).

Любая операция, отсоединяющая ссылку от объекта O, должна уменьшать число ссылок O на единицу. Имеется два вида операций отсоединения:

- (D1) Любое присваивание a := b. Заметим, что это также присоединяющая операция (A1) для объекта, присоединенного к b. (Поэтому если b также присоединен к O, необходимо как увеличить, так и уменьшить счетчик O, т.е. оставить его без изменения - приятный результат.)
- (D2) Завершение вызова подпрограммы вида x.r(..., a, ...). (Если a встречается более одного раза в списке фактических аргументов, необходимо считать отсоединением каждое вхождение a.)

После таких операций, реализация должна также проверять, не является ли значение счетчика, равным нулю, если да, то можно утилизировать объект.

В заключение рассмотрим ситуацию с clone, требующую особого внимания. Операция a := clone (b) создает копию объекта OB, присоединенного к b, если OB существует. Вновь созданный объект OA присоединяется к a. Счетчик ссылок OA инициализируется единицей, естественно, не копируя счетчик OB. Если OB имеет непустые ссылочные поля, то при его копировании следует увеличить на единицу счетчик ссылок каждого объекта, присоединенного к каждому ссылочному полю, не исключено, что некоторые поля могут быть присоединены к одному и тому же объекту.

Очевидным недостатком подсчета ссылок являются издержки выполнения как временные, так и по объему памяти. Для всех операций со ссылками реализация языка должна выполнять арифметическую операцию, а в случае отсоединения, - условный оператор. К тому же, к каждому объекту добавляется поле счетчика ссылок.

Но есть более серьезная проблема, делающая подсчет ссылок, к сожалению, мало используемым. ("К сожалению", поскольку эта техника легко реализуема.) Проблема связана с циклическими структурами. Рассмотрим в очередной раз наш основной пример структуры с взаимосвязанными объектами:

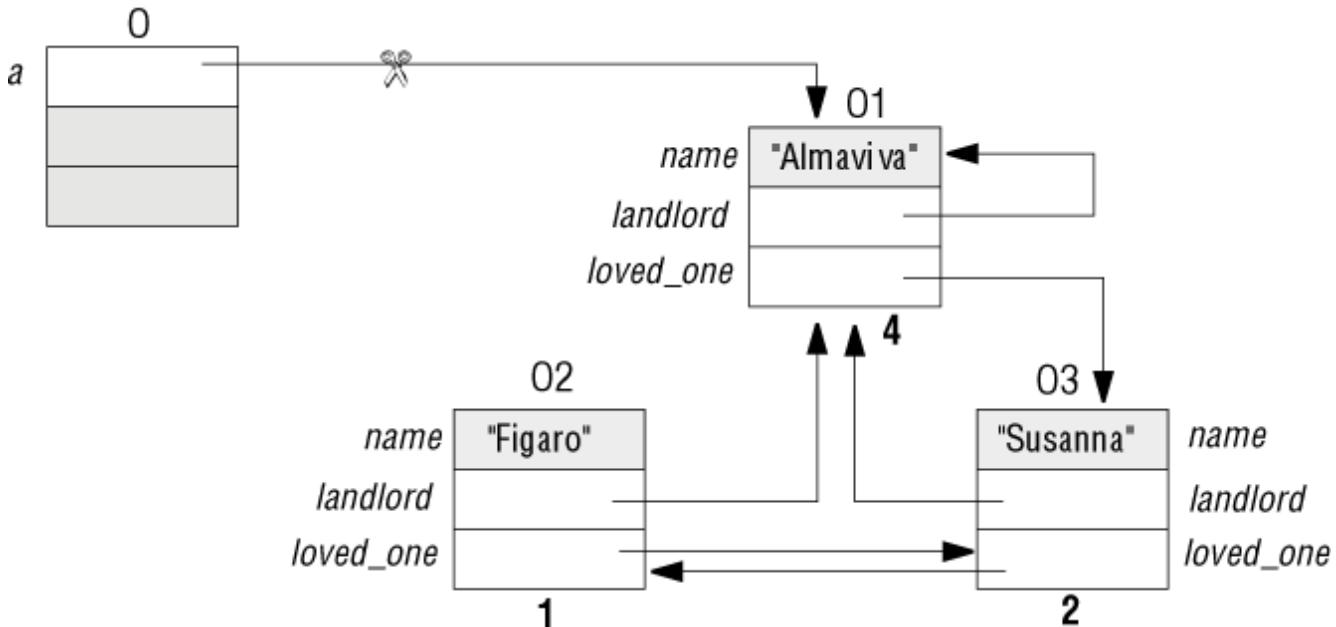


Рис. 9.14. Неудаляемая при подсчете ссылок циклическая структура

Объекты O1, O2 и O3 содержат циклические ссылки друг на друга. Допустим, что нет объектов вне структуры кроме O, содержащих ссылки на какой-либо из объектов структуры. Соответствующий счетчик ссылок показан под каждым объектом.

Теперь допустим, что ссылка от O к O1 отрезана, например потому что подпрограмма вызываемая с целью O выполняет инструкцию:

a:=void

Тогда объекты O1, O2, O3 станут недостижимыми, но механизм подсчета ссылок не определит эту ситуацию: вышеуказанная инструкция уменьшит счетчик ссылок O1 до трех и только. Счетчики всех трех объектов останутся положительными, что не позволит определить необходимость их утилизации.

Из-за этой проблемы, подсчет ссылок применим только к структурам, гарантированно не использующим циклы. Это делает его неподходящим в качестве универсального механизма на уровне реализации языка. Невозможно гарантировать, что произвольная система не создает циклических структур. Поэтому метод может быть применен только при создании библиотек компонентов. К сожалению, если методы уровня компонентов, рассмотренные в

предыдущем разделе, не применимы, то это происходит потому, что используемые структуры слишком сложны и, чаще всего, по причине наличия циклов.

Сборка мусора

Наиболее общей и полностью удовлетворительной техникой является лишь автоматическая сборка мусора или просто сборка мусора.

Механизм сборки мусора

Сборщик мусора (garbage collector) - это функция **исполнительной системы (runtime system)** языка программирования. Сборщик мусора выполняет обнаружение и утилизацию недостижимых объектов, не нуждаясь в управлении приложением, хотя приложение может иметь в своем распоряжении различные средства контроля работы сборщика.

Детальное рассмотрение всех проблем сборки мусора требует отдельной книги. (В конце лекции приведена библиография по этой проблеме.) Рассмотрим общие принципы и возникающие проблемы, концентрируя внимание на свойствах, важных для разработчиков программ.

Требования к сборщику мусора

Сборщик мусора, несомненно, должен быть корректным, удовлетворяя двум требованиям:

Свойства сборщика мусора

Качественность: каждый собираемый объект должен быть недостижимым.

Полнота: каждый недостижимый объект должен быть собран.

Качественность - абсолютное требование: лучше не собирать мусор, чем выбрасывать нужный объект. Нужна полная уверенность в том, что управлению памятью можно слепо доверять. Фактически надо забыть о нем почти навсегда, будучи уверенным, что кто-то как-то убирает беспорядок в вашей программе, также как кто-то как-то убирает мусор в вашем офисе, когда вас нет, но не убирает при этом ваши книги, компьютер и семейные фотографии со стола.

Полнота желательна - без нее все равно можно столкнуться с проблемой, которую сборщик мусора должен решить: память тратится на бесполезные объекты. Но здесь можно не требовать безупречности: сборщик может быть полезным, если он собирает основную часть мусора, иногда пропуская один или два объекта.

Это замечание требует детализации. На практике любой сборщик промышленного масштаба должен обладать полнотой. Полнота на практике также необходима, как качественность, но менее жестка, если перефразировать ее определение: "каждый недостижимый объект должен быть, в конце концов, собран". Предположим, что мы можем сделать процесс сборки более эффективным, благодаря алгоритму, который собирает каждый недостижимый объект, но может запоздать с обращением к некоторым из них: такая схема будет приемлемой для большинства приложений. В этом идея обсуждаемого далее алгоритма "сборки мусора поколений", который в целях эффективности чаще сканирует области памяти, содержащие с большей вероятностью недостижимые объекты, и реже обращает внимание на другие участки памяти.

При таком компромиссном подходе для сборщика мусора необходимо будет ввести не только бинарные критерии полноты и качественности, но и критерий, называемый своевременность (**timeliness**). Его значением является интервал времени от момента, когда объект становится недостижимым, до момента его утилизации, причем важно как среднее значение времени, так и верхняя его граница.

Определение качественности выясняет трудности, связанные со сборкой мусора для некоторых языков программирования, и соответствующие роли языка и его реализации. Почему, например, сборка мусора обычно неприменима для C++? Обычно приводимые причины связаны с культурой: в мире С каждый разработчик должен сам заботиться о своих "игрушках" (по словам Стеффенсона); он просто не доверяет какому-либо автоматическому механизму управлять его делами. Но, если бы это было действительной причиной, а не апостериорным оправданием, среди C++ могли бы, как минимум, предложить сборку мусора как подключаемую возможность, но большинство реализаций этого не делают.

Действительная проблема лежит в структуре языка, а не в технологии компиляции или культурных традициях. Язык C++, следуя С, слабо типизирован; он предоставляет возможность преобразования типа, благодаря которой на объект одного типа можно ссылаться как на сущность другого типа. Конструкция:

```
(OTHER_TYPE) x
```

означает, что теперь x рассматривается как сущность типа OTHER_TYPE, связанного или несвязанного с истинным типом x. Хорошие книги по C++ предостерегают приложения от применения подобных конструкций. Но разработчикам компилятора деваться некуда, - они обязаны реализовать язык в соответствии с его определением. Теперь представьте следующий сценарий. Ссылка на объект какого-либо полезного типа, скажем

NUCLEAR_SUBMARINE, временно приведена к типу `integer`. Сборщик мусора, работающий в этот момент, не видит ссылки, а видит только целое типа `integer`. Не находя других ссылок на объект, сборщик утилизирует подлодку. Когда, через некоторое время программа выполнит обратное преобразование целого в ссылку типа NUCLEAR_SUBMARINE, будет уже поздно, - подлодка уничтожена.

Для решения этой проблемы предлагались разные методы. Широкого применения они не получили из-за накладываемых ограничений. Язык Java может рассматриваться как язык семейства C++, в котором введены существенные ограничения на систему типов, вплоть до удаления множественного наследования и универсализации, чтобы сделать, наконец, возможной сборку мусора в мире программ, основанных на С.

При тщательно спроектированной системе типов, конечно, можно сочетать мощь множественного наследования и универсализации с безопасностью типов и поддержкой эффективной сборки мусора.

Основа сборки мусора

Рассмотрим работу сборщика мусора.

Основной алгоритм включает две фазы: пометка и чистка. Фаза пометки, начиная с оригиналов, рекурсивно следует ссылкам, проходит активную часть структуры и помечает как достижимые все встреченные объекты. Фаза чистки обходит всю структуру объектов, утилизируя все не помеченные объекты и удаляя все пометки. (Об оригиналах см. раздел "Достижимые объекты в ОО-модели этой лекции".)

Как и в случае с подсчетом ссылок, объекты включают дополнительное поле, используемое здесь для пометки. Но требуемая для этого поля память незначительна, - достаточно одного бита для каждого объекта. Как будет видно при изучении динамического связывания, реализация ОО-возможностей требует, чтобы объект имел дополнительную внутреннюю информацию (например, тип). Эта информация обычно занимает одно или два слова в каждом объекте. Бит пометки может быть частью служебного слова, и не будет занимать дополнительную память.

Сборка по принципу "все-или-ничего"

Когда нужно приводить в действие сборщик мусора?

Классические сборщики мусора активизируются по требованию и работают до завершения. Другими словами, сборщик мусора не работает, пока остается память для работы приложения. Как только ее не хватает, приложение запускает полный цикл сборки мусора - фаза пометки и следом фаза чистки.

Эта техника может быть названа "все-или-ничего". Преимущество ее в том, что она не вызывает перегрузки пока достаточно памяти. Когда программа выходит за пределы достижимых ресурсов, в наказание вызывается сборщик мусора.

Но сборка мусора по принципу "все-или-ничего" имеет серьезный недостаток: полный цикл пометки-чистки может занять много времени - особенно в среде виртуальной памяти, большое пространство виртуальных адресов которого сборщик мусора должен обойти полностью, прерывая на это время выполнение приложения.

Для пакетных приложений такая схема еще может быть приемлема. Но и здесь при высоком коэффициенте отношения виртуальной памяти к реальной перегрузка может стать причиной серьезной потери производительности, если система создает большое число объектов, лишь малая часть из которых является в каждый момент достижимыми.

Сборка мусора по принципу "все-или-ничего" не будет работать для интерактивных систем или систем реального времени. Представим систему обнаружения ракетного нападения, которая имеет 50-миллисекундный интервал для реагирования на запуск ракеты. Допустим, программа прекрасно работала, пока система не вышла за пределы памяти, но, к несчастью это событие произошло в момент запуска ракеты, когда вместо системы начал свою работу неспешный сборщик мусора.

Даже в менее жизненно важных приложениях, таких как интерактивные системы, неприятно использовать инструмент, например, редактор текста, который иногда непредсказуемо зависает на 10 минут, потому что у него начался цикл сборки мусора.

В таких случаях проблема заключается не в глобальном эффекте временных потерь, связанных со сборкой мусора: определенная потеря производительности может быть вполне допустимой для разработчиков и пользователей, как плата за надежность и удобство, предоставляемое автоматической сборкой мусора. Но временные потери должны быть равномерно распределены. Неприемлемы непредсказуемые всплески активности сборщика мусора. Лучше черепаха, чем заяц, время от времени без предупреждения засыпающий на полчаса. Подсчет ссылок, если бы не его фатальный порок, удовлетворял бы лозунгу: "лучше ехать медленно, но с постоянной скоростью, чем быстро, но с неожиданными и непредсказуемыми остановками".

Конечно, временные потери должны быть не только постоянными, но и небольшими. Если приложение без сборщика мусора - заяц, никто не согласится заменить его черепахой. Хороший сборщик мусора должен обеспечивать задержку, не превышающую 5-15%. Хотя некоторые скажут, что и это неприемлемо, я знаю совсем

немного приложений, которым нужны меньшие издержки. Необходимо учитывать также, что в отсутствии сборщика мусора потребуется ручная утилизация, также не обходящаяся без издержек. Несмотря на все издержки, сборка мусора необходима.

В ходе обсуждения выявлены две дополнительные проблемы эффективности работы сборщика мусора: производительность глобальная (overall performance) и в стартстопном режиме (incrementality).

Продвинутый (Advanced) подход к сборке мусора

Хороший сборщик должен обеспечивать хорошую производительность, работая как постоянно, так и в стартстопном режиме, становясь приемлемым для интерактивных приложений и даже для систем реального времени.

Отсюда первое требование - необходимо дать возможность разработчикам управлять запуском и выключением циклов работы сборщика. В частности, библиотеки должны предоставлять процедуры:

```
collection_off  
collection_on  
collect_now
```

Вызов первой прекращает циклическую работу по сборке мусора до особого распоряжения; второй - включает сборщик, восстанавливая нормальное состояние работы; третьей - заставляет сборщик немедленно выполнить полный цикл работы. Пусть некоторая система содержит критический по времени выполнения раздел, в котором не должно быть никаких непредсказуемых временных задержек. В этом случае разработчик может вызвать collection_off в начале этого раздела и collection_on в его конце; в любой другой точке, где приложение работает вхолостую (например, во время ввода или вывода), можно запустить collect_now.

Более продвинутая техника, используемая в большинстве современных сборщиков мусора, известна как **сборка мусора поколений (generation scavenging)**. Она исходит из следующего наблюдения: чем больше циклов сборки мусора объект пережил, тем больше вероятность, что он доживет до следующего цикла или всегда будет достижимым. Отсюда принцип работы сборщика мусора: "старые объекты оставляй нетронутыми". Сборщику полезна любая информация, позволяющая сканировать определенные категории объектов реже, чем остальные. Сборка мусора поколений обнаруживает объекты, существующие более чем определенное количество циклов. Такие объекты получают статус **постоянной должности (tenuring)** по аналогии с механизмом, защищающим экземпляры класса реальной жизни PROFESSOR, прошедших несколько циклов переизбрания и получивших, наконец, постоянную позицию. Объекты-долгожители будут рассматриваться отдельным сборщиком, работающим реже, чем сборщик "молодых" объектов.

Практическая реализация сборки мусора поколений имеет много вариаций. В частности, обычно делят объекты не только на молодые и старые, но на большее число поколений с разными стратегиями сборки мусора различных поколений.

Алгоритмы параллельной сборки мусора

Для получения полного решения проблемы работы в стартстопном режиме крайне привлекательно выделить сборщику мусора отдельный поток выполнения, конечно, при условии поддержки многозадачности операционной системой. Этот прием известен, как сборка мусора "на лету" (on-the fly) или параллельная.

Во время сборки мусора на лету выполнение ОО-системы использует два отдельных потока (часто соответствующих двум отдельным процессам операционной системы): приложение и сборщик. Только приложение выделяет память объектам с помощью инструкций создания; только сборщик освобождает память с помощью reclaim операций.

Сборщик будет работать непрерывно, повторяя фазу пометки и следом фазу чистки для обнаружения и удаления недостижимых объектов.

Отдельные потоки не обязательно должны быть отдельными процессами. Они могут быть, во избежание дополнительных расходов на переключение между процессами или даже потоками, плоскими сопрограммами. (Подробнее сопрограммы будут рассмотрены в [лекции 12](#) курса "Основы объектно-ориентированного проектирования", рассматривающей "параллелизм")

Даже при этих условиях сборка мусора на лету на практике имеет неудовлетворительную полную производительность. Это печально, поскольку сам метод достаточно хорош, особенно при условии использования алгоритма Дейкстры (см. библиографическую ссылку).

По моему мнению (мой комментарий отражает надежду, а не научно установленный результат) параллельная сборка мусора - решение будущего, требующее кооперации с аппаратными средствами. Вместо того, чтобы воровать время у процессора, выполняющего приложение, сборка мусора должна управляться отдельным процессором, предназначенным только для решения этой задачи и сконструированным так, чтобы как можно меньше влиять на процессор(ы), работающие с приложением.

Эта идея требует изменения доминирующей аппаратной архитектуры и, вероятно, вряд ли найдет скорое применение. Я надеюсь, что ответом на иногда задаваемый вопрос -

"Какой тип аппаратного обеспечения наиболее пригоден для объектной технологии?" -

первым пунктом в списке пожеланий будет наличие отдельного процессора для сборки мусора.

Практические проблемы сборки мусора

Среда исполнения, обеспечивающая управление памятью, должна не только использовать хороший алгоритм сборки мусора, но и поддерживать несколько свойств, которые, хотя и не главные в теории управления памятью, являются существенными для практического использования среды.

Класс MEMORY

Наиболее удобный подход - представить эти свойства в виде класса, который назовем MEMORY. Класс приложения, нуждающийся в свойствах, будет наследником MEMORY.

Аналогичный подход будет использован для механизма обработки исключений (класс EXCEPTIONS, [лекция 12](#)) и для управления параллелизмом (класс CONCURRENCY, [лекция 12](#) курса "Основы объектно-ориентированного проектирования")

Среди компонентов класса MEMORY будут представлены рассмотренные ранее процедуры: collection_off, collection_on, collect_now.

Механизм освобождения

Другой важной процедурой класса MEMORY является dispose (не путайте с тезкой Pascal, которая освобождает память). Она связана с важной практической проблемой, иногда называемой **финалом** или **окончательным завершением (finalization)**. Если сборщик мусора утилизирует объект, связанный с внешними ресурсами, вы можете пожелать включить в его спецификацию некоторое дополнительное действие, такое как освобождение ресурсов, выполняемое параллельно с утилизацией. Типичный пример - класс FILE, экземпляр которого представляет файлы операционной системы. Желательно иметь возможность в случае утилизации недостижимого экземпляра класса FILE вызвать некоторую процедуру, закрывающую соответствующий физический файл.

Обобщая сказанное, рассмотрим процедуру dispose, выполняющую во время утилизации необходимые объекту операции. Это могут быть не только операции по освобождению ресурсов, но и любые операции, определяемые спецификацией класса.

При ручном управлении памятью проблем не возникает: достаточно включить вызов dispose до вызова reclaim. Деструктор класса в C++ включает в себя две операции dispose и reclaim. Однако при наличии сборщика мусора приложение напрямую не контролирует момент утилизации объекта, поэтому невозможно вставить dispose в нужное место.

Решение проблемы использует мощь объектной технологии и, в частности, наследование и переопределение. (Эта техника изучается в последующих лекциях, но ее применение здесь достаточно просто и понятно без детального ознакомления.) Класс MEMORY включает процедуру dispose, в теле которой никакие действия не выполняются:

```
dispose is
    - Действия, которые следует выполнить в случае утилизации;
    - по умолчанию действия отсутствуют.
    - Вызывается автоматически сборщиком мусора.
do
end
```

Тогда любой класс, требующий специальных действий всякий раз, когда сборщик утилизирует один из его экземпляров, должен переопределить процедуру dispose так, чтобы она выполняла эти действия. Например, представим, что класс FILE имеет логический атрибут opened и процедуру close. Он может переопределить dispose следующим образом:

```
dispose is
    - Действия, которые следует выполнить в случае утилизации:
    - закрыть связанный файл, если он открыт.
    - Вызывается автоматически сборщиком мусора.
do
    if opened then
        close
    end
```

end

Комментарии описывают используемое правило: при утилизации объекта вызывается `dispose` - либо изначально пустую процедуру (что далеко не самый общий случай), либо версию, переопределенную в классе, представляющего потомка `MEMORY`.

Сборка мусора и внешние вызовы

Хорошо спроектированная ОО-среда со сборкой мусора должна решать еще одну практическую проблему. Во многих случаях ОО-программы взаимодействуют с программами, написанными на не ОО-языках. В следующих лекциях будет рассмотрено, как лучше обеспечить такое взаимодействие. (См. "Взаимодействие с не ОО-программой", [лекцию 13](#))

Если ПО включает вызовы подпрограмм, написанных на других языках (называемых далее внешними программами), возможно, этим подпрограммам необходимо будет передавать ссылки на объекты. Это потенциально опасно для управления памятью. Предположим, что внешняя подпрограмма имеет следующий вид (преобразованная в соответствии с синтаксисом языка внешней программы):

```
r (x: SOME_TYPE) is
  do
    ...
    a := x
    ...
  end
```

где `a` сущность, сохраняющая значение между последовательными вызовами `r`. Например, `a` может быть глобальной или статической переменной в традиционных языках, или атрибутом класса в нашей ОО-нотации. Рассмотрим вызов `r(y)`, где `y` связан с некоторым объектом `O1`. Возможно, что через некоторое время после вызова, `O1` становится недостижимым в объектной части нашей программы, но ссылка на него (от сущности `a`) остается во внешней программе. Сборщик мусора может - и в конце концов должен - утилизировать `O1`, но в данном случае это неправильно.

Для таких ситуаций необходимы процедуры, вызываемые из внешней программы, которые защищают нужный объект от сборщика. Эти процедуры могут быть названы:

```
adopt (a) - усыновлять
wean (a)   - отнимать от груди, отлучать
```

и должны быть частью интерфейса любой библиотеки, обеспечивающей взаимодействие ОО-программ и внешних программ. В следующем разделе описан подобный механизм для языка С. "Усыновление" объекта забирает его из области действия механизма утилизации; "отлучение" - возвращает возможность утилизации.

Передача объектов в не ОО-языки и удерживание ссылки на них внешней программой - дело рискованное. Но избежать этого возможно не всегда. Например, ОО-проект нуждается в специальном интерфейсе между ОО-языком и имеющейся системой управления БД. В этом случае, можно разрешить внешней программе сохранять информацию об объектах. Такие низкоуровневые манипуляции никогда не должны появляться в нормальном программном продукте. Они должны содержаться в обслуживающем классе, написанном с единственной целью - скрыть детали от остальной части программы и защитить ее от возможных неприятностей.

Среда с управлением памятью

В заключение рассмотрим, не вдаваясь в детали, как одна специфическая среда, представленная более широко в последней лекции этой книги, управляет памятью. Это даст пример практического, реально достижимого подхода к проблеме.

Основы

Управление памятью - автоматическое. Среда включает сборку мусора, существующую по умолчанию. Вполне естественен вопрос пользователя "как включить сборщик мусора?". Ответ - он уже включен! В обычном использовании, в том числе и в интерактивных приложениях, он незаметен. Его можно отключить с помощью `collection_off`.

В отличие от сборщиков в других средах, сборщик мусора не просто освобождает память для повторного использования объектами того же приложения, а возвращает память операционной системе для ее использования другими приложениями (по крайней мере, операционными системами, поддерживающими механизм освобождения памяти навсегда). Ранее показано, как важно это свойство, особенно для систем, работающих долгое время или постоянно.

Дополнительные инженерные цели, возложенные на сборщика мусора при его проектировании: эффективная сборка памяти, небольшие накладные расходы, стартстопный режим работы, позволяющий предотвратить

блокировку приложения в критические моменты его работы.

Сложные проблемы

Сборщик мусора сталкивается со следующими проблемами, вызванными практическими ограничениями на размещение объектов в современной ОО-среде:

- ОО-подпрограммы могут вызывать внешние программы, в частности, С-функции, которые могут, в свою очередь, размещать нечто в памяти. Поэтому нужно рассматривать два различных вида памяти: память для объектов и внешнюю память.
- Объекты создаются по-разному. Массивы и строки имеют переменный размер; экземпляры других классов имеют фиксированный размер.
- Наконец, как отмечалось, недостаточно освобождать память для повторного использования в самом ОО-приложении, - нужно возвращать ее навсегда операционной системе.

По этим причинам размещение объектов в памяти не может полагаться на стандартный системный вызов `malloc`, который, наряду с другими ограничениями, не возвращает память операционной системе. Вместо этого среда запрашивает у ядра операционной системы участки памяти и распределяет объекты в этих участках с помощью собственных механизмов.

Перемещение объектов

Необходимость возвращать память операционной системе порождает одну из самых утонченных частей механизма: сборщик мусора может при необходимости перемещать объекты.

Это свойство вызывает головную боль при реализации сборщика, но оно делает этот механизм устойчивым и практичным. Без него невозможно было бы использовать сборку мусора для долго работающих, критически важных систем.

Внутри ОО-мира нет необходимости задумываться о перемещении объектов, если гарантируется, что система не имеет тенденции постоянного расширения (подразумевается, что общий размер достижимых объектов ограничен). При использовании внешних программ и передачи им объектов эту проблему необходимо рассматривать. Внешняя программа может сохранять ссылки на объекты из ОО-мира в виде простых адресов (указателей в языке С). При попытке использовать эти объекты, находящиеся без защиты, например, в течение 10 минут, возникнут трудности: за это время объект может быть перемещен и по его адресу лежит нечто другое или вообще ничего. Простой библиотечный механизм решает эту проблему: С-функции должны получать сам объект и доступ к нему через специальный макрос, который находит объект, где бы он ни находился.

Механизм сборки мусора

Приведем схему алгоритма, используемого сборщиком мусора.

Решение представляет собой не единственный алгоритм, а основано на комбинации основных алгоритмов, часть из которых используется совместно, часть - независимо друг от друга. Каждая активизация сборщика выбирает алгоритм или сочетание алгоритмов, основанных на критерии запроса необходимой памяти. Основные алгоритмы включают: сборку мусора поколений, пометку-чистку и сжатие памяти, плюс несколько других, в меньшей степени относящихся к данному обсуждению.

Идея сборки мусора поколений описана в этой лекции ранее: следует сосредоточиться на молодых объектах, - именно они с большой вероятностью могут быть недостижимыми, и собраны мусорщиком. Основное преимущество этого алгоритма в том, что он просматривает не все объекты, а только те, которые могут быть достижимы из локальных сущностей и из старых объектов, содержащих ссылки на молодые объекты. Всякий раз по завершению обработки поколения все выжившие объекты становятся старше; когда они достигают определенного возраста, они переходят на постоянную должность в другое поколение. Алгоритм ищет компромисс, устанавливающий границу переходного возраста. Ее снижение приводит к росту старых объектов, увеличение - к частой сборке мусора.

Алгоритм время от времени нуждается в выполнении полной **пометки-сборки** для поиска любых недостижимых объектов, пропущенных сборщиком поколений. **Пометка-сборка** состоит из двух шагов: **пометка** - рекурсивный обход и пометка достижимых объектов; **чистка** - полный обход памяти и сборка непомеченных объектов.

Алгоритм сжатия памяти возвращает неиспользуемые участки памяти операционной системе, работая с наименьшими временными затратами. Он разбивает память на **n** блоков и за **n-1** циклов сжимает их.

Повышенное чувство голода и потеря аппетита (*Bulimia and anorexia*)

Алгоритм сжатия предохраняет от частых, дорогих по времени вызовов операционной системы - выделить или возвратить память. Вместо возвращения всех освобожденных блоков он сохраняет некоторые из них для построения небольшого резерва памяти, доступной приложению без обращения к операционной системе.

Эта техника крайне полезна для часто встречающегося класса приложений с повышенным чувством голода и потерей аппетита, у которых период кутежа с массовым созданием объектов сменяется постом, в течение которого происходит избавление от ненужных объектов; затем все повторяется.

Операции сборщика мусора

Сборщик мусора включается одной из двух требующих память операций: инструкцией создания (`create x`) или клонирования. Сборщик запускается не только, когда программе не хватает памяти: механизм может активизироваться, когда он определяет некоторые условия, за которыми последует нехватка памяти.

Если первичная память заполнена, сборщик начнет сборку мусора поколений. В большинстве случаев освободится достаточно памяти для текущих нужд. Если этого не произошло, следующий шаг - полный цикл пометки-чистки с последующим сжатием памяти. Если и в этом случае памяти не достаточно, сборщик запросит память у операционной системы.

Основные алгоритмы являются стартстопными; их время выполнения обычно составляет несколько процентов от времени выполнения приложения. Внутренняя статистика ведет учет занятой памяти и помогает определить подходящий для вызова алгоритм.

Можно настроить работу сборщика, задавая различные параметры, в частности, включение параметра `speed` заставит алгоритм не собирать всю доступную память с помощью механизма сжатия, а сразу использовать возможности операционной системы. Устанавливая другие параметры, можно включать механизмы: `collection_off`, `collect_now` и `dispose` из класса `MEMORY`.

Механизм управления памятью, построенный на основе всех этих методов, сделал возможным разработку и выполнение больших приложений, создающих много объектов, создающих их быстро, не заботясь об используемой памяти, поручая кому-то другому заботу о последствиях.

Ключевые концепции

- Существует три основных режима создания объектов: статический, основанный на стеке, динамический. Последний характерен для ОО-языков, но встречается везде, например, в Lisp, Pascal (указатели и `new`), C (`malloc`), Ada (типы доступа).
- В программах, создающих много объектов, объекты могут становиться недостижимыми. Их память теряется, приводя, в худших случаях, к сбою из-за нехватки памяти, при том что часть памяти остается неиспользованной.
- Эту проблему можно игнорировать в тех случаях, когда программа почти не создает недостижимых объектов или создает всего лишь несколько объектов, общий размер которых сравним с доступной памятью.
- Во всех других случаях (динамические структуры данных, ограниченные ресурсы памяти) любое решение проблемы включает два компонента: обнаружение мертвых объектов и восстановление занятой ими памяти.
- Каждая из задач может быть решена на одном из трех уровней: реализации языка, разработки компонентов, приложения.
- Вменять в обязанность приложения обнаружение мертвых объектов и восстановление памяти - опасно и обременительно. Эта проблема должна решаться на уровне языка.
- В некоторых специальных случаях можно управлять памятью на уровне компонентов. Обнаружение выполняется компонентами, восстановление памяти - компонентами, либо средствами, реализованными на уровне языка.
- Подсчет ссылок не работает для циклических структур.
- Общепринимаемой техникой решения проблемы является сборка мусора. Ее накладные издержки приемлемы малы в нормальных ситуациях и, благодаря алгоритмам, работающим в стартстопном режиме, невидимы в нормальных интерактивных приложениях.
- Сборка мусора поколений увеличивает эффективность алгоритма, используя тот факт, что недостижимыми становятся, в первую очередь, новые объекты.
- Хороший механизм управления памятью должен возвращать неиспользуемую память не только текущему приложению, но и операционной системе.
- Описанная схема реальной системы управления памятью предлагает комбинацию алгоритмов и способов, позволяющих разработчикам приложений производить настройку механизмов системы, в том числе позволяя включать и отключать сборку мусора в критических разделах приложения.

Библиографические заметки

Различные модели создания объектов, обсуждаемые в начале этой лекции, поддерживаются "контурной моделью" выполнения языка программирования, которая может быть найдена в [Johnston 1971].

Информация о фiasco Лондонской службы скорой помощи получена из множества сообщений, присланных на форум Risks.

Алгоритм параллельной сборки мусора представлен в [Dijkstra 1978]. Проблемы производительности подобных алгоритмов рассматривал [Cohen 1984]. Сборка мусора поколений представлена в [Ungar 1984].

Механизм сборки мусора ISE's среды, описанный в конце этой лекции, был создан Рафаэлем Манфреди (Raphael Manfredi) и усовершенствован Ксавьером Ле Вурч (Xavier Le Vourch) и Фабрис Францески (Fabrice Franceschi) (чей технический отчет служил основой данного здесь описания).

Упражнения

У9.1 Модели создания объектов

При обсуждении автоматического управления памятью рассмотрен подход, основанный на создании внутренних списков свободной памяти. В этом случае память, занимаемая утилизированными объектами, не возвращается операционной системе, а остается в создаваемом списке. Разработайте модель системы, демонстрирующую постоянный рост занимаемой памяти, хотя фактически требуемая приложению память ограничена.

Вы можете описать эту модель как последовательность o_1, o_2, o_3, \dots , где o_i либо 1, (что показывает выделение памяти одному объекту), либо $(-n)$, показывающее восстановление n единиц памяти.

У9.2 Какой уровень утилизации?

Подход на уровне компонентов, если программировать на языке типа Pascal или C, где операционная система предоставляет dispose или free, может напрямую использовать эти операции вместо создания своего списка свободной памяти для каждого типа структур данных. Рассмотрите плюсы и минусы двух подходов.

У9.3 Совместное использование стека достижимых элементов

(Это упражнение подразумевает знакомство с результатами [лекции 18](#)) Перепишите компонент available, задающий стек достижимых элементов при подходе на уровне компонентов. Единственный стек должен совместно использоваться всеми связанными списками одного и того же типа. (Указание: используйте функцию once.)

У9.4 Совместное использование

(Это упражнение подразумевает, что вы выполнили предыдущее и прочитали все лекции, включая [лекцию 18](#)) Можно ли сделать available стек разделяемым всеми связанными списками произвольных типов?

¹⁾ Динамические массивы могут создаваться в языке C, используя функцию malloc - механизм, подобный динамическому распределению, описываемому ниже; некоторые расширения Pascal поддерживают динамические массивы.

10. Лекция: Универсализация

Слияние двух концепций - модуля и типа - позволило разработать мощное понятие класса, послужившее основой ОО-метода. Уже в таком виде оно позволяет делать многое. Однако для достижения наших целей - расширяемости, возможности повторного использования, надежности необходимо сделать конструкцию класса более гибкой. Развитие может идти в двух направлениях. Один, представленный вертикалью на следующем рисунке, показывает абстракцию и специализацию; он ведет к изучению наследования в последующих лекциях. В данной лекции изучается другая размерность (горизонталь на рисунке), параметризация (тип как параметр), известная также как универсализация.

Горизонтальное и вертикальное обобщение типа



Рис. 10.1. Размерности обобщения

Уже изученные механизмы позволяют написать класс, помещенный в центр рисунка - `LIST_OF_BOOKS`, экземпляр которого представляет список книг. У класса следующие компоненты: `put` для вставки элемента, `remove` для удаления элемента, `count` для подсчета числа элементов и т.д. Очевидны два пути обобщения понятия `LIST_OF_BOOKS`.

- Списки являются специальным видом структур, представляющих контейнеры. Из многих других примеров контейнеров отметим деревья, стеки и массивы. В сравнение со списком `LIST_OF_BOOKS`, более абстрактным вариантом контейнера является класс `SET_OF_BOOKS` (множество книг). Более специализированным вариантом является класс `LINKED_LIST_OF_BOOKS`, определяющий связанный список книг - специализированный вариант списка. Три класса задают вертикальную размерность на рисунке - размерность наследования.
- Списки книг являются, с другой стороны, специальным случаем списков объектов некоторого вида. Из многих других видов отметим списки журналов, списки людей, списки целых чисел. Это горизонтальная размерность на нашем рисунке - размерность универсализации, тема нашего изучения в последующей части этой лекции. Если задать параметр класса, представляющий произвольный тип, то можно не создавать почти идентичные классы, такие как `LIST_OF_BOOKS` и `LIST_OF_PEOPLE`, не жертвуя при этом безопасностью, вносимой статической типизацией.

Отношение между двумя этими механизмами - трудный вопрос для изучающих ОО-концепции. Как рассматривать наследование и параметризацию, как соперников или как соратников, когда целью является построение более гибкого ПО?¹

Данная лекция посвящена универсализации, кроме того, мы подробно рассмотрим один из наиболее общих примеров родовых структур: массивы. Заметьте, термины универсальный класс, родовой класс, параметризованный класс являются синонимами. Во всех случаях речь идет о классе с параметрами, задающими некоторый тип (класс).

Необходимость параметризованных классов

Универсализация уже рассматривалась в данной книге, но не применялась для классов. Мы столкнулись с ней при обзоре традиционных подходов к повторному использованию и при изучении математической модели класса - АТД, где была показана необходимость определения параметризованного АТД.

Родовые АТД

Наш работающий пример АТД, STACK, был объявлен с параметром, как STACK [G]. Любое его использование заставляет специфицировать "фактический родовой параметр", представляющий тип хранимого в стеке объекта. Имя G, используемое в спецификации АТД, - **формальный родовой параметр** класса. Оно указывает, что элементы стека могут иметь любой возможный тип. При таком подходе можно использовать одну спецификацию для всех возможных стеков. Альтернативой, вряд ли приемлемой, было бы введение множества классов: INTEGER_STACK, REAL_STACK и т.д.

Любые АТД, описывающие контейнеры: множества, списки, матрицы, массивы и многие другие, очевидно, также должны быть родовыми.

Это решение применимо к контейнерам классам, также как к контейнерам АТД.

Проблема

Рассмотрим пример стека, но уже не как АТД, а как класс. Мы знаем, как написать класс INTEGER_STACK, задающий стек объектов типа INTEGER. Компоненты будут включать count (число элементов), put (вталкивание элемента), item (элемент в вершине), remove (выталкивание элемента), empty (пустой ли стек?).

Тип INTEGER будет часто использоваться в объявлениях этого класса. Например, это тип аргумента для put и результата для item:

```
put (element: INTEGER) is
    -- Втолкнуть элемент (в вершину стека).
    do ... end
item: INTEGER is
    -- Элемент в вершине стека
    do ... end
```

Эти появления типа INTEGER следуют из правила явного объявления, используемого при разработке нотации: всякий раз при введении сущности, обозначающей возможные объекты времени выполнения, необходимо явное указание ее типа, такое как element: INTEGER. Здесь это означает, что необходимо указать тип для запроса item, для аргумента element процедуры put и для других сущностей, обозначающих возможные элементы стека.

Как следствие, придется писать различные классы для каждого сорта стека: INTEGER_STACK, REAL_STACK, POINT_STACK, BOOK_STACK... Все эти стековые классы будут одинаковыми за исключением объявления типов item, element и некоторых других сущностей. Основные операции над стеком не зависят от типа элементов стека и реализуются одинаково. Для всех, заинтересованных в повторном использовании, такое дублирование классов представляется мало привлекательным.

Проблема возникает из-за противоречия двух основных требований, предъявляемых к классам и сформулированных в начале этой книги.

- Надежность: сохранение преимуществ безопасности типов с помощью явного объявления типа.
- Повторное использование: возможность написать один программный элемент, покрывающий многие варианты одного понятия.

Роль типизации

Зачем настаивать на явном объявлении типов (первое из двух требований)? Это часть главного вопроса о типизации, которому в этой книге посвящена отдельная лекция ([лекция 17](#)). Но уже сейчас можно указать две основные причины, по которым ОО-программа должна быть статически типизирована.

- Читаемость: явное объявление четко сообщает читателю о том, как предполагается использовать каждый элемент. Это важно как для автора, так и для того, кому нужно понять часть программы, чтобы отладить или расширить ее.
- Надежность: благодаря явному объявлению типов компилятор сможет найти ошибочные операции еще на этапе компиляции, не допуская их проявления при выполнении. В фундаментальных операциях ОО-вычислений вызов компонента имеет форму $x.f(a, \dots)$, где x - некоего типа TX. Причины возникновения ошибок могут быть разными: соответствующий класс TX может не иметь метода f; метод может существовать, но быть скрытым; количество аргументов при вызове может не совпадать с объявленным в описании класса; тип a или другого аргумента может не совпадать с ожидаемым. В языке Smalltalk, в котором отсутствует статическая типизация, любая такая ситуация приведет к краху на этапе выполнения с выдачей, например, сообщения: "Message not understood", в то время как компилятор языка с явной типизацией не пропустит ошибочной конструкции.

Ключ к надежности - следование принципу "предотвратить, а не лечить". Исследования показали, что стоимость исправления ошибки астрономически возрастает, когда затягивается ее обнаружение. Статическая типизация, позволяющая ранее обнаружение ошибок, - фундаментальный инструмент в борьбе за надежность.

Без учета требований надежности явное объявление типов было бы не нужно так же как универсализация. Остаток этой лекции обращается к языкам со статической типизацией, т.е. языкам, которые требуют объявления каждой сущности и задают правила, позволяющие компиляторам обнаруживать несоответствие типов до выполнения. В не статически типизированных языках, таких как Smalltalk, универсализация не имеет смысла. Язык упрощается, но не защищает от схем вида:

```
my_stack.put (my_circle)
my_account := my_stack.item
my_account.withdraw (5000)
```

где элемент, полученный из вершины стека, рассматривается как банковский счет, хотя в действительности это круг, что можно понять из первой инструкции. Выполнение программы закончится, при попытке получить пять тысяч долларов от "дырки от бублика".

Статическая типизация защищает от подобных неудач. Совмещение типизации с требованием повторного использования приведет нас к механизму универсализации.

Родовые классы

Согласование статической типизации с требованием повторного использования для классов, описывающих контейнерные структуры, означает, как показано на примере стека, что мы хотим одновременно иметь возможность:

- Объявить тип каждой сущности, появляющейся в классе стека, включая сущности, представляющие элементы стека.
- Написать класс так, чтобы он не содержал никаких намеков на тип элемента стека, и следовательно, мог использоваться для построения стеков с элементами произвольных типов.

На первый взгляд эти требования кажутся несовместимыми, но на самом деле это не так. Первое требование заставляет нас объявить тип. Но вовсе не требуется, чтобы тип в объявлении был конкретным! Назвав имя типа, мы умножим механизм проверки. ("Назови свой страх - и он уйдет"). В этом идея универсализации: получить класс с параметром, задающим тип, снабдить его именем, назвав его формальным родовым параметром.

Объявление родового класса

По соглашению родовой параметр обычно, использует имя G (от **Generic**). Это неформальное правило. Если нужны еще родовые параметры, они будут называться H, I и т.д.

Согласно синтаксису, формальные родовые параметры заключаются в квадратные скобки, следующие за именем класса, подобно синтаксису параметризованного АТД в предыдущей лекции. Например:

```
indexing
    description: "Стек элементов произвольного класса G"
class STACK [G] feature
    count: INTEGER
        -- Количество элементов в стеке
    empty: BOOLEAN is
        -- Есть ли элементы?
    do ... end
    full: BOOLEAN is
        -- Стек заполнен?
    do ... end
    item: G is
        -- Вершина стека
    do ... end
    put (x: G) is
        -- Втолкнуть x в стек.
    do ... end
    remove is
        -- Вытолкнуть элемент из стека.
    do ... end
end -- class STACK
```

Формальный родовой параметр G можно использовать в объявлениях класса не только для результата функций (как в item) и формальных аргументов подпрограмм (как в put), но и для атрибутов и локальных сущностей класса.

Использование родового класса

Клиент может использовать родовой класс для объявления собственных сущностей, задающих стек. В этом случае в момент объявления следует задать фактический тип элементов стека - фактический родовой параметр, например:

```
sp: STACK [POINT]
```

Если у класса несколько родовых параметров, то соответственно столько же необходимо задать и фактических параметров.

Предоставление фактических родовых параметров родовому классу для создания типа называется **родовым порождением (generic derivation)**, а полученный в результате класс, такой как STACK [POINT], называют **параметрически порожденным классом**.

Родовому порождению требуется тип, родовое порождение создает новый тип:

- Результат порождения STACK [POINT] является типом.
- Для получения такого результата, необходим уже существующий тип, используемый в качестве фактического параметра (POINT в примере).

Фактический параметр может быть произвольным типом. Ничто не мешает выбрать тип, который сам по себе параметрически порожден. Предположим, что мы определили другой родовой класс LIST [G], тогда можно определить стек, элементы которого являются списками точек:

```
slp: STACK [LIST [POINT]]
```

или, используя STACK [POINT] как фактический родовой параметр, - стек стеков точек:

```
ssp: STACK [STACK [POINT]]
```

Нет предела глубины таких вложений, кроме естественной необходимости сохранять простоту программного текста.

Терминология

Обсуждая универсализацию, необходимо уточнить используемые термины.

- Процесс порождения нового типа, такого как STACK [POINT], из типов POINT и STACK, можно было бы называть созданием экземпляра типа "generic instantiation". Но этот термин мог бы ввести в заблуждение, поскольку в названии неявно предполагается процесс периода выполнения ПО. Заметьте, родовое порождение - статический механизм, действующий на текст программы, а не на ее выполнение.
- В этой книге термин "параметр" и "аргумент" используются по-разному. Первый для универсальных классов, второй - для подпрограмм. В традиционной программистской терминологии параметры и аргументы чаще всего синонимы.

Проверка типов

Используя универсализацию, можно гарантировать, что структура данных будет содержать элементы определенного типа. Допустим, класс содержит объявления:

```
sc: STACK [CIRCLE]; sa: STACK [ACCOUNT]; c: CIRCLE; a: ACCOUNT.
```

Тогда в программах этого класса допустимы следующие инструкции:

```
sc.put (c) -- Втолкнуть круг в стек кругов  
sa.put (a) -- Втолкнуть счет в стек счетов  
c := sc.item -- Сущности круг присвоить вершину стека кругов.
```

Но каждая из следующих инструкций недопустима и будет отвергнута:

```
sc.put (a); -- Попытка: Втолкнуть счет в стек кругов.  
sa.put (c); -- Попытка: Втолкнуть круг в стек счетов.  
c := sa.item -- Попытка: Дать кругу значение счета.
```

Это исключает ошибочные операции, подобные попытке вычитания денег из круга.

Правило типизации

Правило типизации, делающее допустимым первый набор и недопустимым второй, интуитивно понятно, но его надо уточнить.

Вначале рассмотрим обычные, не родовые классы. Пусть С такой класс. Рассмотрим объявление его компонента, не использующее, естественно, никаких формальных родовых параметров:

```
f(a:T):U is ...
```

Тогда вызов вида `x.f(d)`, появляющийся в произвольном классе В, где x типа С будет мудро корректен по типу, тогда и только тогда, когда:

- f доступен классу В, - экспортован всем классам или множеству классов, включающих В;
- d принадлежит типу T. Если учитывать возможность наследования, то d может принадлежать потомкам Т.
- Результат вызова имеет тип U. В этом примере предполагается, что компонент f является функцией.

Теперь предположим, что С родовой класс с формальным родовым параметром G имеет компонент:

```
h (a: G): G is...
```

Вызов h имеет вид `y.h(e)`, где y сущность, объявленная как

```
y: C [V]
```

Тип V - некоторый ранее определенный тип. Теперь правило типизации - двойник неродового правила - требует, чтобы е имело тип V или при наследовании было потомком V. Аналогичное требование к результату выполнения функции h.

Требования правила понятны: V - фактический параметр, заменяющий формальный родовой параметр G параметризованного класса С, поэтому он заменяет все вхождения G при вызове компонент класса. Все предыдущие примеры следовали этой модели: вызов `s.put(z)` требует параметра z типа POINT, если s типа STACK [POINT]; INTEGER если s типа STACK [INTEGER]; и s.item возвращает результат типа POINT в первом случае и типа INTEGER во втором.

Операции над сущностями родового типа

В родовом классе С [G, H, ...] рассмотрим сущность, чей тип - один из формальных родовых параметров, например x типа G. Когда класс используется клиентом для объявления сущностей, G, разумеется, может представлять любой тип. Поэтому любая операция, которую выполняют подпрограммы С над x, должна быть применима ко всем типам. Это ограничение позволяет выполнять только пять видов операций:

Использование сущностей формального родового типа

Корректно использовать сущность x, чей тип задан формальным родовым параметром G, можно следующим образом.

1. Слева от оператора присваивания `x := y`, где выражение y также имеет тип G.
2. Справа от оператора присваивания `y := x`, где сущность y также типа G.
3. В логических выражениях вида `x = y` или `x /= y`, где y также типа G.
4. Как фактический аргумент в вызове подпрограммы на месте формальных параметров типа G, или типа ANY.
5. Как цель вызова компонента класса ANY.

В частности, инструкция создания вида `create x` неприменима, так как нам ничего неизвестно о процедурах создания, если таковые есть, для класса, определенного возможным фактическим родовым параметром, соответствующим G.

Случаи (4) и (5) ссылаются на класс ANY. Упомянутый уже несколько раз, этот класс содержит компоненты, наследуемые всеми классами. Поэтому можно быть уверенным, что независимо от фактического типа G при родовом порождении компоненты будут доступны. Компонентами класса ANY являются все основные операции копирования и сравнения объектов: `clone`, `copy`, `equal`, `deep_clone`, `deep_equal` и др. Поэтому для x и y формального родового типа G корректно использовать следующие инструкции:

```
x.copy (y)
x := clone (y)
if equal (x, y) then ...
```

Случай (4) разрешает вызов `a.f(x)` в родовом классе С [G], если f имеет формальный аргумент типа G. В частности, возможна ситуация, когда a может быть типа D [G], где D другой родовой класс. В классе D [G] объявлен компонент f, требующий аргумента типа G, обозначающий в этом случае формальный родовой параметр

класса D, а не класса C. (Если предыдущая фраза не совсем понятна, перечитайте ее еще раз, и, надеюсь, она покажется столь же прозрачной²⁾, как горный ручей.)

Типы и классы

Мы уже научились смотреть на класс - центральное понятие объектной технологии, - как на продукт слияния двух концепций: модуля и типа. До введения универсализации можно было говорить, что класс - это модуль, но это и тип данных.

С появлением универсализации второе утверждение перестало быть буквально истинным, хотя нюанс невелик. Родовой класс, объявленный как C [G], является не типом, а шаблоном типа, задающим бесконечное множество возможных типов. Любой тип из этого множества можно получить, предоставив фактический родовой параметр, который, в свою очередь, является типом.

Это приводит к более общему и гибкому понятию. Но за выигрыш в мощности приходится немногого пожертвовать простотой: только при небольшом насилии над языком можно продолжать говорить о "компонентах класса T" или о "клиентах T", если х объявлен, как имеющий тип T. Теперь T может быть параметрически порожденным типом C [U] из некоторого родового класса C и некоторого типа U. Конечно, основой типа остается родовой класс C, поэтому насилие над языком приемлемо.

Если требовать буквальной строгости, то терминология следующая. Любой тип T ассоциируется с базовым классом T, поэтому всегда можно говорить о компонентах и клиентах базового класса T. Если T неродовой класс, то он же является и базовым классом. Если T родовое порождение C [U, ...], то C является базовым классом T.

Базовые классы будут использоваться при введении еще одного вида типов, основанного также (как и все остальное в ОО-подходе) на классе, но косвенно: закрепленного типа (см. гл. 16.7).

Массивы

В заключение этой дискуссии полезно рассмотреть пример контейнерного класса ARRAY, представляющего одномерный массив.

Массивы как объекты

Понятие массив обычно является частью определения языка программирования. В объектной технологии нет необходимости нагружать нотацию специальными заранее определенными конструкциями: массив - контейнерный объект, экземпляр класса, который можно назвать ARRAY.

ARRAY хороший пример родового класса. Рассмотрим первый набросок этого класса:³⁾

```
indexing
    description: "Последовательность значений одного типа или согласуемых типов, %
                  %доступных через целые индексы в заданном диапазоне"
class ARRAY [G] creation
    make
feature
    make (minindex, maxindex: INTEGER) is
        -- Размещение массива с границами minindex и maxindex
        -- (пустой, если minindex > maxindex)
        do ... end
    lower, upper, count: INTEGER
        -- Минимальный и максимальный допустимый индекс; размер массива.
    put (v: G; i: INTEGER) is
        -- Присвоить v элементу массива с индексом i
        do ... end
    infix "@", item (i: INTEGER): G is
        -- Элемент с индексом i
        do ... end
end -- класса ARRAY
```

Для создания массива a с границами m и n, тип объявления которого ARRAY [T] с заданным типом T, нужно выполнить инструкцию создания

```
create a.make (m, n)
```

Для задания значений элементов массива используется процедура put: вызов a.put(x, i) присваивает значение x i-ому элементу. Для доступа к элементам можно использовать функцию item (синоним инфиксной операции @, поясняемой позже), например:

```

x := a.item (i)
Вот схема того, как этот класс может быть использован клиентом:
pa: ARRAY [POINT]; p1: POINT; i, j: INTEGER
...
create pa.make (-32, 101) -- Разместить массив с указанными границами.
pa.put (p1, i) -- Присвоить значение p1 элементу с индексом i.
...
p1 := pa.item (j) -- Присвоить сущности p1 значение элемента с индексом j.

```

В обычной нотации (скажем, в Pascal) нужно писать:

```

pa [i] := p1 вместо pa.put (p1, i)
p1 := pa [i] вместо p1 := pa.item (i)

```

Свойства массива

Некоторые замечания о классе.

- Подобные классы существуют для массивов большей размерности: ARRAY2 и т. д.
- Компонент Count может быть реализован и как атрибут и как функция, поскольку count = upper - lower+1. В реальном классе это выражается инвариантом, как объясняется в следующей лекции.
- Техника утверждений позволяет связывать точные условия согласования с put и item, отражая тот факт, что вызовы допустимы, только если индекс i лежит между lower и upper.
- Идея описания массивов как объектов (и ARRAY как класс) - хороший пример мощности унификации и упрощения объектной технологии, позволяющей сократить нотацию до минимума и уменьшить количество узкоспециализированных конструкций. Здесь массив рассматривается как обычный пример контейнерной структуры с собственными методами доступа, представленными компонентами put и item.
- Так как ARRAY - обычный класс, он может участвовать во всем, что в предыдущих лекциях называлось ОО-играми; в частности, другие классы могут быть его наследниками. Класс ARRAYED_LIST, описывающий реализацию абстрактного понятия - списка массивов может быть наследником классов LIST и ARRAY. Подобные конструкции будут рассматриваться далее.

Как только мы изучим механизм утверждений, этот унифицированный подход даст возможность развития нашего класса. Предусловия позволяют управлять проверкой корректного задания индексов, что обычно требует узко специализированных механизмов.

Размышления об эффективности

- Может ли элегантность и простота нанести удар по эффективности выполнения? Одна из причин широкого использования массивов состоит в том, что основные операции - доступ и изменение элемента - проходят быстро. Надо ли платить за каждый вызов подпрограммы при использовании item или put? Нет. То, что ARRAY для ничего не подозревающего разработчика выглядит как нормальный класс, не запрещает компилятору опираться на скрытую информацию. Она позволяет компилятору находить вызовы item и put и переопределять их так, чтобы сгенерировать такой же код, как сделает компилятор Fortran, Pascal или C для эквивалентных инструкций (p1 := pa [i] и pa [i] := p1 в синтаксисе Pascal). Поэтому разработчик получит лучшее: универсальность, общность, упрощенность, простоту использования ОО-решения, сочетаемую с сохранением производительности традиционного решения.
- Работа компилятора не тривиальна. Как выяснится при изучении наследования, для потомка класса ARRAY возможно переопределить любой компонент класса и эти переопределения могут быть косвенно вызваны через динамическое связывание. Поэтому компилятор должен выполнять тщательный анализ для проверки корректности изменений массива. Для научных приложений, интенсивно использующих массивы, современные компиляторы от ISE и других компаний сегодня могут генерировать код, столь же эффективный, как написанный вручную на С или Fortran.

Синонимичная инфиксная операция

Класс ARRAY предоставляет возможность, косвенно относящуюся к вопросам этой лекции, но полезную на практике. Объявление компонента item фактически определяет и его синоним - инфиксную операцию⁴⁾ следующим образом:

```
infix "@", item (i: INTEGER): G is...
```

Здесь задаются два имени компонента: infix "@" и item как синонимы, обозначающие один и тот же компонент, заданный определением.

В общем, объявление компонентов в форме:

```
a, b, c... "Описание компонента"
```

рассматривается как краткая форма записи последовательности объявлений:

```
a "Описание компонента"  
b "Описание компонента"  
c "Описание компонента"  
...
```

с одним и тем же "Описанием компонента".

Это применимо как для атрибутов (где "Описание компонента" имеет форму: **некоторый_тип**), так и для подпрограмм (**is тело_программы**).

Нотация, применяемая в этом примере для доступа к массиву, достаточно проста. Она совместима с механизмами доступа для других структур, хотя, заметим, инструкция `a.item(i)` более многословна, чем традиционное `a[i]`, встречающееся с некоторыми вариациями в Pascal, C, Fortran и других языках. Определяя "@" как синоним `item`, можно превзойти традиционные языки в их собственной игре за краткость записи. Написав `a @ i`, реализуем мечту, - запись требует на одно нажатие клавиши меньше, чем даже C++!. Заметим снова, что это не специальный механизм языка, но прямое применение общей ОО-концепции, компонент-оператора, скомбинированного с нотацией синонима.

Стоимость универсализации

Как всегда нужно убедиться, что ОО-техника, введенная в интересах повторного использования, расширяемости и надежности, не влечет потерю производительности. Этот вопрос уже поднимался при рассмотрении массивов.

Теперь необходимо с этих позиций проэкзаменовать механизм универсализации в целом. Какова цена универсализации?

В частности, этот вопрос возникает из-за опыта C++, где универсализация, известная как механизм шаблонов, представляла одно из поздних добавлений к языку. Выяснилось, что некоторые компиляторы воспринимают введение универсализации буквально, генерируя различные копии методов класса для каждого фактического родового аргумента! В результате в литературе по C++ предупреждают программистов об опасности широкого использования шаблонов:

Число создаваемых экземпляров шаблона - уже проблема для некоторых пользователей C++. Если пользователь создает `List<int>`, `List<String>`, `List<Widget>` и `List<Blidjet>` (где `Widget` и `Blidjet` классы, определенные пользователем) и вызывает `head`, `tail` и `insert` для всех четырех объектов, то каждая из этих функций будет создана в четырех экземплярах (из-за родового порождения). Вместо этого широко применимый класс `List` мог бы создать единственный экземпляр каждой функции применимый для различных типов.⁵⁾

Авторы этого предупреждения (C++ эксперты из AT&T, один из них соавтор официальной C++ документации [Ellis 1990]) продолжают предлагать различные способы, позволяющие избежать порождения шаблонов. Но универсализация не предполагает дублирование кода. При хорошо спроектированном языке и хорошем компиляторе можно генерировать единый код компонентов родового класса, так что последующие добавления потребуют минимальных затрат:

- времени компиляции;
- размера сгенерированного кода;
- времени выполнения;
- памяти, требуемой для выполнения.

Работая в такой среде, можно использовать всю мощь универсализации, не опасаясь потери производительности, как на этапе компиляции, так и выполнения.

Обсуждение: что все-таки не сделано

Основные идеи универсализации уже представлены, но как вы могли заметить, на два важных вопроса не даны ответы.

Первое: в наших усилиях гарантирования безопасности типов мы заняли чересчур консервативную позицию. Конечно, некорректно пытаться втолкнуть круг в стек банковских счетов. Трудно вообразить, какому приложению нужен стек, содержащий точки и банковские счета. Но рассмотрим графическое приложение, для которого вполне естественен стек, содержащий круги, прямоугольники, точки. Такая потребность кажется довольно разумной, но пока мы не можем удовлетворить ее. Система типов, определенная до сих пор, отвергнет вызов `figure_stack.put(that_point)` если тип `figure_stack` был объявлен как `STACK [FIGURE]`, а `that_point` - тип, отличный от `FIGURE`. Дадим пока имя рассматриваемым структурам и назовем их **полиморфными структурами данных (polymorphic data structure)**. Вызов, стоящий перед нами - как поддержать эти структуры без потери преимуществ безопасности типов.

Второе: родовые параметры представляют произвольные типы. Это хорошо для стеков и массивов, поскольку объекты любого типа по своей сути являются хранимыми в различных контейнерах. Но при работе, например, с

векторами, хотелось бы иметь возможность складывать элементы векторов или сами векторы. При работе с классом, задающим хеш-таблицы, хотелось бы быть уверенным, что хеш-функция применима к любому элементу таблицы. Такая форма универсализации, где формальный родовой параметр уже не может быть произвольным типом, а является типом, гарантирующим предоставление ряда операций, называется **ограниченной универсализацией (constrained genericity)**.

Для обеих этих проблем ОО-метод обеспечивает простые и элегантные решения, оба основанные на комбинировании универсализации и наследования.

Ключевые концепции

- Классы могут иметь формальные родовые параметры, представляющие типы.
- Родовые классы служат для описания общих контейнерных структур данных, реализуемых одинаково, независимо от элементов, которые они содержат.
- Универсализация требуется только в типизированном языке, гарантирующем статически проверяемую безопасность типов.
- Клиент родового класса должен предоставлять фактические типы для формальных параметров.
- Единственные допустимые операции над сущностью, чей тип является формальным родовым параметром, - это операции, применимые к любому типу. Сущность может быть правой и левой частью оператора присваивания, фактическим аргументом подпрограммы или операндом теста на равенство или неравенство.
- Понятие массива не требует специального языкового механизма и вполне укладывается в обычную схему родового библиотечного класса.
- Более гибкое и продвинутое использование универсализации - полиморфные структуры данных и ограниченная универсализация - требует введения наследования.

Библиографические замечания

Один из первых языков, поддерживающий универсализацию - LPG [Bert 1983]. Язык Ada сделал эту концепцию широко известной введением механизма родовых пакетов.

Универсализация была также введена в языки формальной спецификации, такие как Z, CLEAR и OBJ-2, на которые были ссылки в [лекции 6](#) по АТД. Родовой механизм, описанный здесь, был построен на основе механизма, представленного в ранней версии Z [Abrial 1980] [Abrial 1980a] и расширенного в [M 1985b].

Если не считать эту книгу, то одним из первых ОО-языков, поддерживающих универсализацию, был DEC's Trellis язык [Schaffert 1986].

Упражнения

У10.1 Ограниченнная универсализация

Это упражнение немного специфично - оно ставит вопрос, детальный ответ на который будет дан позднее в этой книге. Его цель - дать возможность сравнить ваше решение с решением, предложенным в книге. Оно особенно полезно, если вы не знакомы с решениями, предлагаемыми в ОО-языках. Подход языка Ada может помочь в поиске решения, но и без него можно обойтись.

Продумайте механизм ограниченной универсализации, совместимый по духу с ОО-подходом. Он должен позволять автору родового класса указать, что правильные фактические родовые параметры должны обладать определенным набором операций.

У10.2 Двумерные массивы

Используя класс ARRAY как источник вдохновения и как основу реализации, напишите класс ARRAY2, описывающий двумерные массивы.

У10.3 Использование своего формального родового параметра фактически как чужого

Сконструируйте пример, в котором подпрограмма родового класса C [G] вызывает подпрограмму, объявленную в другом родовом классе D [G], имеющую формальный параметр типа G.

¹⁾ Полный ответ можно найти в приложении В "Genericity versus inheritance".

²⁾ Для проверки прозрачности выполните упражнение У10.3.

³⁾ Улучшенная версия класса рассмотрена в [лекции 11](#)

⁴⁾ Нотация инфиксных операций была введена в [лекции 7](#)

⁵⁾ Martin Carroll, Margaret Ellis, "Reducing Instantiation Time", in "C++ Report", vol. 6, no. 5, July-August 1994, pages 14, 16 and 64.

Основы объектно-ориентированного программирования

11. Лекция: Проектирование по контракту: построение надежного ПО

Вооруженные базисными концепциями класса, объекта, параметризации вы можете теперь создавать программные модули, реализующие возможно параметризованные типы структур данных. Мои поздравления! Сделан важный шаг в битве за лучшую программную архитектуру. Но рассмотренных методов явно недостаточно для реализации всеобъемлющего вида качества, введенного в начале книги. Факторы качества, на которых было сконцентрировано наше внимание, - повторное использование, расширяемость, совместимость - не должны достигаться ценой надежности (корректность и устойчивость). Хотя концепция надежности просматривалась по ходу обсуждения, мы добиваемся большего. Необходимо уделить больше внимания семантическим свойствам классов становятся особенно очевидной, если вспомнить что класс - это реализация АТД. Рассматриваемые до сих пор классы состояли из атрибутов и программ, реализующих функции спецификации АТД. Но АТД это не просто список операций: вспомните роль семантических свойств, выражаемых аксиомами и предусловиями. Они являются основой, проясняющей природу экземпляров данного типа. В классах мы - временно - потеряли этот семантический аспект концепции АТД. Необходимо вернуться назад, чтобы наше ПО было не только гибким и повторно используемым, но и корректным и устойчивым. Утверждения и связанные с ними концепции, проясняемые в этой лекции, частично дают ответы. Не являясь полным доказательством, представленные ниже механизмы снабжают программиста основными средствами для формулирования и проверки аргументов корректности. Ключевой концепцией будет Проектирование по контракту (Design by Contract) - установление отношений между классом и его клиентами в виде формального соглашения, недвусмысленно устанавливающее права и обязанности сторон. Только через точное определение для каждого модуля требований и ответственности можно надеяться на достижение существенной степени доверия к большим программным системам. При обзоре концепций мы впервые столкнемся с ключевой проблемой программной инженерии - как справиться с ошибками периода выполнения, возникающими при нарушении контракта. Этой теме - обработке исключительных ситуаций посвящена следующая лекция. Распределение ролей между двумя главами примерно отражает разницу между двумя компонентами надежности: корректностью и устойчивостью. Корректность - это возможность ПО выполнять свои задачи в соответствии со спецификациями, устойчивость - способность должным образом реагировать на ситуации, выходящие за пределы спецификации. Утверждения (этота лекция), как правило, покрывают корректность, а исключения (следующая лекция) - устойчивость. Некоторые важные расширения основных идей проектирования по контракту должны ожидать введения наследования, полиморфизма и динамического связывания, что позволит нам перейти от контрактов к выдаче субподрядов.

Базисные механизмы надежности

Технические приемы, введенные в предыдущих лекциях, были направлены на создание надежного ПО. Дадим их краткий обзор - было бы бесполезно рассматривать более продвинутые концепции до приведения в порядок основных механизмов надежности. Первым и определяющим свойством объектной технологии является почти навязываемая структура программной системы - простая, модульная, расширяемая, - проще гарантирующая надежность, чем в случае "кривых" структур, возникающих при применении ранних методов разработки. В частности, усилия по ограничению межмодульного взаимодействия, сведения его к минимуму, были в центре дискуссии о модульности. Результатом стал запрет общих рисков, снижающих надежность, - отказ от глобальных переменных, механизм ограниченного взаимодействия модулей, отношения наследования и вложенности. Общее наблюдение: самый большой враг надежности (и качества ПО в целом) - это сложность. Создавая наши структуры настолько простыми, сколь это возможно, мы достигаем необходимого, но не достаточного условия, гарантирующего надежность. Прежнее обсуждение служит лишь верной отправной точкой в последующих систематических усилиях.

Заметьте, необходим, но также недостаточен, постоянный акцент на создание элегантного и читабельного ПО. Программные тексты не только пишутся, они еще читаются и переписываются по много раз. Ясность и простота нотации языковых конструкций - основа любого изощренного подхода к надежности.

Еще одно необходимое оружие - автоматическое управление памятью, в особенности сборка мусора. В лекции, посвященной этой теме, в деталях пояснено, почему для любой системы, оперирующей динамическими структурами данных, столь опасно опираться на управление этим процессом вручную. Сборка мусора не роскошь - это ключевой компонент ОО-среды, обеспечивающий надежность.

Тоже можно сказать об еще одном, сочетающемся с параметризацией механизме, - статической типизации. Без правил строгой статической типизации пришлось бы лишь надеяться на снисхождение многочисленных ошибок, возникающих в период выполнения.

Все эти механизмы дают необходимую основу для более полного взгляда на то, что следует предпринять для обеспечения устойчивости и корректности ПО.

О корректности ПО

Зададимся вопросом, что означает утверждение - программный элемент корректен? Наблюдения и рассуждения, отвечающие на это вопрос, могут показаться тривиальными. Но, как заметил один известный ученый, таковы все научные результаты, - они начинаются с обычных наблюдений и продолжаются путем простых рассуждений, но все это нужно делать упорно и настойчиво.

Предположим, некто пришел к вам с программой из 300 000 строк на С и спрашивает, корректна ли она? Если вы консультант, то взыщите высокую плату и ответьте - "нет". Вы, вероятно, окажетесь правы.

Для того чтобы можно было дать разумный ответ на подобный вопрос, одной программы недостаточно, необходима еще и ее спецификация, точно описывающая, что должна делать программа. Оператор

`x := y+1`

сам по себе не является ни корректным, ни не корректным. Эти понятия приобретают смысл лишь по отношению к ожидаемому эффекту присваивания. Например, присваивание корректно по отношению к утверждению: "Переменные x и y имеют различные значения". Но не гарантируется его корректность по отношению к высказыванию: "переменная x отрицательна", поскольку результат присваивания зависит от значения y , которое может быть положительным.

Эти примеры иллюстрируют свойство, служащее отправной точкой в обсуждении проблемы корректности: программная система или ее элемент сами по себе ни корректны, ни не корректны. Корректность подразумевается лишь по отношению к некоторой спецификации. Строго говоря, мы и не будем обсуждать проблему корректности программных элементов, а лишь их **согласованность (consistent)** с заданной спецификацией. В наших обсуждениях мы будем продолжать использовать хорошо понимаемый термин "корректность", но всегда при этом помнить, что этот термин не применим к программному элементу, он имеет смысл лишь для пары - "программный элемент и его спецификация".

Свойство корректности ПО

Корректность - понятие относительное.

В этой лекции мы научимся выражать спецификации через **утверждения (assertions)**, что поможет оценить корректность разработанного ПО. Но пойдем дальше и перевернем проблему, - разработка спецификации является первым, важнейшим шагом на пути, гарантирующем, что ПО действительно соответствует спецификации. Существенную выгоду можно получить, когда спецификации пишутся одновременно с написанием программы, а лучше, до ее написания. Среди следствий такого подхода можно отметить следующее.

- Разработка ПО корректного с самого начала, проектируемого так, чтобы быть корректным. Один из создателей структурного программирования Харлан Д. Миллс в семидесятые годы написал статью со знаменательным названием "Как писать корректные программы и знать это". Слово "знать" в данном контексте означает снабжать программу в момент ее написания аргументами, характеризующими корректность.
- Значительно лучшее понимание проблемы и достижение ее решения.
- Упрощение задачи создания программной документации. Как будет позже показано, утверждения будут играть важную роль в ОО-подходе к документации.
- Обеспечение основ для систематического тестирования и отладки.

Оставшаяся часть лекции посвящена исследованию этих вопросов. Одно предупреждение: языки программирования C, C++ и другие имеют оператор утверждения `assert`, динамически проверяющий истинность заданного утверждения в момент выполнения программы и останавливающий вычисление, если утверждение является ложным. Эта концепция, хотя и имеет отношение к предмету обсуждения, но является лишь малой частью использования утверждений в ОО-методе. Поэтому, если подобно многим разработчикам вы знакомы с этим оператором, не обобщайте ваше знание на всю картину, почти все концепции этой лекции, возможно, будут новыми.

Выражение спецификаций

От неформальных высказываний перейдем к простой математической нотации, принятой в теории формальной проверки правильности программ и имеющей ценность при доказательстве корректности программных элементов.

Формула корректности

Пусть A - это некоторая операция (оператор или тело программы). **Формула корректности (correctness formula)** - это выражение в форме:

$\{P\} A \{Q\}$

Формула выражает свойство, которое может быть или не быть истинным:

Смысл формулы корректности $\{P\} A \{Q\}$

Любое выполнение A , начинающееся в состоянии, где P истинно, завершится и в заключительном состоянии будет истинно Q .

Формула корректности, называемая также триадой Хоара, - математическое понятие, а не программистская конструкция. Она не является частью языка программирования и введена для того, чтобы выражать свойства программных элементов. В этой формуле A , как было сказано, обозначает операцию, P и Q - свойства вовлекаемых в рассмотрение сущностей, называемые утверждениями (точный смысл этого термина будет определен ниже). Утверждение P называется предусловием, а Q - постусловием.

С этого момента обсуждение корректности ПО будет связываться не с программным элементом A, а с триадой, содержащей этот элемент A, предусловие P и постусловие Q. Единственной целью становится установление того, что триада Хоара $\{P\} \ A \ \{Q\}$ выполняется (истинна).

Вот пример выполняемой тривиальной формулы, в которой полагается, что x имеет тип integer:

```
{x>=9} x := x+5 {x>=13}
```

Число 13 в постусловии не опечатка. Предполагая корректную реализацию целочисленной арифметики, данная формула действительно выполняется. Если предусловие $x >= 9$ выполняется перед присваиванием, то $x >= 13$ будет истинным по завершении оператора присваивания. Конечно, можно утверждать более интересную вещь: при заданном предусловии сильнейшим, насколько это возможно, будет постусловие $x >= 14$. В свою очередь, при заданном постусловии $x >= 13$ слабейшим предусловием будет $x >= 8$. Из выполняемой формулы корректности всегда можно породить новые выполняемые формулы, ослабляя постусловие или усиливая предусловие. Займемся теперь выяснением того, что означают термины "сильнее" и "слабее" в пред- и постусловиях.

Сильные и слабые условия

Понятия "сильнее" и "слабее" пришли из логики. Говорят, что P1 сильнее, чем P2, а P2 слабее, чем P1, если P1 влечет P2 и они не эквивалентны. Каждое утверждение влечет True, и из False следует все что угодно. Можно говорить, что True является слабейшим, а False сильнейшим из всех возможных утверждений.

Давайте взглянем на формулу корректности с позиций человека, собирающегося наняться на работу по выполнению операции A. Каковы с его точки зрения наилучшие предусловие P и постусловие Q, если у него есть возможность выбора? Возможность усиления предусловия означает, что можно предъявлять более жесткие требования к работодателю, что можно уменьшить число ситуаций, в которых следует приступать к выполнению работы. Так что сильное предусловие это "хорошие новости" для работника. Наилучшей для него работой - синекурой является работа, чья спецификация выражается формулой:

Синекура 1

```
{False} A {...}
```

Постусловие здесь не специфицировано, поскольку не имеет значения каково оно. К выполнению работы можно вообще не приступать, поскольку нет ни одного начального состояния, в котором предусловие было бы истинным. Так что если вам предложат такую синекуру, немедленно соглашайтесь, не глядя на постусловие - требования, предъявляемые к выполненной работе.

Именно такую спецификацию работ имел в виду начальник полиции одного из американских городов. Когда его спросили в интервью, почему он выбрал именно эту работу, он ответил: "Это единственная работа, где заказчик всегда неправ!"

Для постусловия ситуация меняется на противоположную. Лучшими для работника являются более слабые условия - это "хорошие новости"; в этом случае хорошо уметь делать очень немногое. Наилучшей работой - второй синекурой является работа, заданная спецификацией:

Синекура 2

```
{...} A {True}
```

Как бы не была выполнена работа, постусловие в этом случае будет истинным по определению. Кстати, почему эта работа является все-таки второй по предпочтительности? Причина, как можно видеть из определения триады Хоара, в завершаемости (**terminate**). Определение устанавливает, что выполнение должно завершиться в состоянии, удовлетворяющем Q, всякий раз, когда оно начинается в состоянии, удовлетворяющем P. Для синекуры 1, где нет состояний, удовлетворяющих P, не имеет значения, что делает A даже если программный текст приводит к выполнению бесконечного цикла, или ломает компьютер. Любое A будет корректным по отношению к данной спецификации. Для синекуры 2, однако, требуется завершение работы, должно существовать заключительное состояние, не важно, что делает A, но то, что делается, должно быть выполнено за конечное время.

Читатели, знакомые с теорией, могли заметить, что формула $\{P\} \ A \ \{Q\}$ определяет **тотальную (total correctness)** или **полную корректность**, включающую завершаемость наряду с соответствием спецификации. Свойство, устанавливающее, что программа удовлетворяет спецификации при условии ее завершения, известно, как частичная корректность. См. [М 1990] для детального знакомства с этими концепциями.

Обсуждение того, будет ли усиление или ослабление утверждений "хорошей" или "плохой" новостью, шло с позиций работника, нанимающегося для выполнения работы. Обратим ситуацию, и рассмотрим ее с позиций работодателя. В этом случае слабое предусловие станет "хорошой" новостью, поскольку означает выполнение работы для большего множества входных случаев; более предпочтительным теперь является сильное постусловие, поскольку оно расширяет получение важных результатов. Эта двойственность критериев типична в рассмотрении корректности ПО. Она вновь появится в качестве центрального понятия этой лекции при обсуждении темы: контракты между модулями - клиентами и поставщиками, в установлении которых преимущества, приобретаемые одним участником, становятся обязательствами для другого. Производство эффективного и надежного ПО проходит через составление контрактов, представляющих возможные наилучшие компромиссы во всех межмодульных коммуникациях клиентов и поставщиков.

Введение утверждений в программные тексты

Как только корректность ПО определена как согласованность реализации с ее спецификацией, следует предпринять шаги по включению спецификации в сам программный продукт. Для большинства в программистском сообществе это все еще новая идея. Привычно писать программы, устанавливая тем самым, - как делать (**the how**); менее привычно рассматривать описание целей - что делать (**the what**) - как часть программного продукта.

Спецификации будут основываться на утверждениях - выражениях, включающих сущности нашего ПО. Выражение задает свойство, которому эти сущности могут удовлетворять на некоторых этапах выполнения программы. Типичное утверждение может выражать тот факт, что определенное целое имеет положительное значение, или что некоторая ссылка не определена.

Ближайшим к утверждению математическим понятием является предикат, хотя используемый язык утверждений обладает лишь частью выразительной силы полного исчисления предикатов.

Синтаксически утверждения в нашей нотации будут обычными булевыми выражениями с небольшими расширениями. Одним из расширений является введение в нотацию термина "**old**", другим - введение символа ";" для обозначения конъюнкции (логического И). Вот пример:

```
n>0; x /= Void
```

Как между объявлениями и операторами, стоящими на разных строках, символ ";" является возможным, но не обязательным, так и в последовательности утверждений, записанных на разных строках, он может быть опущен, подразумеваясь по умолчанию. Эти соглашения облегчают идентификацию индивидуальных компонентов утверждения, которым обычно даются имена:

```
Positive: n > 0
Not_void: x /= Void
```

Метки, такие как **Positive** и **Not_void**, в период выполнения играют роль утверждений, что будет еще обсуждаться в этой лекции. В данный момент они введены, главным образом, для ясности и документирования. В нескольких последующих разделах будет дан обзор принципиальных возможностей применения утверждений: как концептуального средства, позволяющего создавать корректные системы, и как документирование того, почему они корректны.

Предусловия и постусловия

Первое использование утверждений - семантическая спецификация программ. Программа - это не просто часть кода, она задает реализацию функции, входящей в спецификацию АТД. Задачу, выполняемую функцией, необходимо выразить точно, как в интересах проектирования, так и как цель последующей реализации и понимания программного текста. Два утверждения связываются с программой - предусловие и постусловие. Предусловие устанавливает свойства, которые должны выполняться всякий раз, когда программа вызывается; постусловие определяет свойства, гарантируемые программой по ее завершению.

Класс стек

Этот пример даст возможность ознакомиться с практическим использованием утверждений. В предыдущей лекции была дана схема параметризованного класса "стек" в форме:

```
class STACK [G] feature
    ... Обявление компонент:
    count, empty, full, put, remove, item
end
```

Реализация появится ниже. До рассмотрения проблем реализации важно отметить, что программы характеризуются строгими семантическими свойствами, не зависящими от специфики реализации. Например:

- Программы `remove` и `item` применимы, только если число элементов стека не равно нулю.
- `put` увеличивает, `remove` - уменьшает число элементов на единицу.

Такие свойства являются частью спецификации АТД, и даже люди далекие от использования любых формальных подходов неявно их понимают. Но в общих подходах к разработке ПО в программных текстах нельзя обнаружить следов спецификации. Предусловие и постусловие программы можно сделать явными элементами ПО. Так и поступим. Введем предусловие и постусловие как специальный вид объявлений с помощью ключевых слов `require` и `ensure` соответственно. Для класса "стек" это приведет к следующей записи, где временно оставлены пустые места для реализации:

```

indexing
  description: "Стеки: Структуры с политикой доступа Last-In, First-Out %
    %Последний пришел - Первый ушел"
class STACK1 [G] feature - Access (доступ)
  count: INTEGER
    -- Число элементов стека
  item: G is
    -- Элемент вершины стека
    require
      not empty
    do
      ...
    end
  feature - Status report (Отчет о статусе)
    empty: BOOLEAN is
      -- Пуст ли стек?
    do ... end
    full: BOOLEAN is
      -- Заполнен ли стек?
    do
      ...
    end
  feature - Element change (Изменение элементов)
    put (x: G) is
      -- Добавить элемент x на вершину.
    require
      not full
    do
      ...
    ensure
      not empty
      item = x
      count = old count + 1
    end
    remove is
      -- Удалить элемент вершины.
    require
      not empty
    do
      ...
    ensure
      not full
      count = old count - 1
    end
  end
end

```

Оба предложения `require` и `ensure` являются возможными; когда они присутствуют, то появляются в фиксированных местах, `require` - перед предложением `local`.

Обратите внимание на разделы **feature**, группирующие свойства по категориям, снабженных заголовками в виде комментариев. Категории Access, Status report, Element change - это несколько примеров из десятков стандартных категорий, используемых в библиотеках и применяемых повсеместно в примерах этой книги.

Предусловия

Предусловия выражают ограничения, выполнение которых необходимо для корректной работы функции. Здесь:

- `put` не может быть вызвана, если стек заполнен;

- `remove` и `item` не могут быть применены к пустому стеку.

Предусловия применяются ко всем вызовам программы, как внутри класса, так и у клиента. Корректная система никогда не вызовет программу в состоянии, в котором не выполняется ее предусловие.

Постусловия

Постусловие выражает свойство состояния, завершающего выполнение программы. Здесь:

- После завершения `put` стек не может быть пуст; на его вершине находится только что втолкнутый элемент, число его элементов увеличилось на единицу.
- После `remove` стек не может быть полон, число его элементов на единицу уменьшилось.

Постусловие в программе выражает гарантию, предоставленную создателем программы, что выполнение программы завершается и приводит к состоянию с заданными свойствами, в предположении, что программа была запущена в состоянии, удовлетворяющем предусловию.

В постусловиях доступна специальная нотация `old`. Она используется, например, в программах `remove` и `item` для выражения изменения значения `count`. Запись `old e`, где `e` - выражение (в большинстве случаев - атрибут) обозначает значение, которое данное выражение имело на входе программы. Любое вхождение `e`, которому не предшествует `old`, означает значение выражения на выходе программы.

Постусловие программы `put` включает предложение:

```
count = old count + 1
```

устанавливающее, что `put`, примененное к любому объекту, должно увеличить на единицу значение поля `count` этого объекта.

Педагогическое замечание

Понятно ваше нетерпение и желание незамедлительно узнать, каков же эффект от утверждений при выполнении программы; что произойдет при вызове `put` при заполненном стеке, или что будет, когда `empty` дает `true` по завершении вызова `put`? Полный ответ на этот вопрос дать еще слишком рано, но предварительный использует любимое словечко адвокатов - это зависит (*it depends*).

Это зависит от того, что вы хотите. Можно рассматривать утверждения просто как комментарии, и тогда их нарушение не обнаруживается в период выполнения. Но их можно использовать для проверки того, что все идет по плану. Тогда во время выполнения окружение автоматически следит за выполнением утверждений и включает исключение при возникновении нарушений, завершая обычно выполнение и выводя сообщение об ошибке. Можно включить в программу обработку исключений, пытающуюся восстановить ситуацию и продолжить выполнение. Эта тема будет детально обсуждаться в следующей лекции. Для указания желаемой политики используются параметры компиляции, которые можно установить независимо для каждого класса.

Все детали мониторинга утверждений периода выполнения появятся чуть позже в этой лекции. Но было бы ошибкой на данном этапе уделять им много внимания. Другие аспекты утверждений сейчас важнее. Мы еще только приступили к рассмотрению этой техники, предназначеннной, прежде всего, для создания корректного ПО; нам еще нужно многое открыть в их методологической роли встроенных стражей надежности. Вопрос о том, что случится, если возникнет ошибка, тоже важен, но рассматривать его следует после того, как мы сделаем все, чтобы предотвратить ее появление.

Посему, хотя и следует думать о будущем, не следует забивать себе голову вопросами о возможной потере производительности из-за введения конструкции `old`. Должна ли система сохранять значения перед запуском программы, чтобы иметь возможность вычислять `old` выражения? Это зависит: в некоторых обстоятельствах (например, при тестировании и отладке) полезно вычислять утверждения; в других - (для полностью проверенных систем) их можно рассматривать как аннотации программного текста.

Все это учитывается в следующих разделах, являясь методологическим вкладом утверждений и метода Проектирование по Контракту - концептуального средства анализа, проектирования, реализации и документирования, помогающего нам построить ПО **со встроенной надежностью (reliability is built-in)**, в терминологии Миллса строить корректную программу и знать это.

Контракты и надежность ПО

Предусловие и постусловие программы определяют контракт со всеми ее клиентами.

Права и обязательства

Связывая с программой `g` предложения `require pre` и `ensure post`, класс говорит своим клиентам:

"Если вы обещаете вызвать `g` в состоянии, удовлетворяющем `rge`, то я обещаю в заключительном состоянии выполнить `post`".

В отношениях между людьми и компаниями контракт - это письменный документ, фиксирующий отношения. Удивительно, что в программной индустрии, где точность так важна и двусмысленность так рискована, эта идея так долго не появлялась. Любой хороший контракт устанавливает для обоих участников как обязательства, так и приобретаемую выгоду; обычно обязательства одного обрачиваются выгодой для другого участника, и это взаимно. Все это верно и для контрактов между классами.

- Предусловие связывает клиента: определяются условия, при которых вызов программы клиентом легитимен. Обязательства клиента приносят пользу поставщику.
- Постусловие связывает класс: программа обязана обеспечить условия по ее завершению. Здесь польза клиента обрачивается обязательствами поставщика класса.

Вот пример контракта для одной из программ нашего примера:

Таблица 11.1. Контракт программы: программа `put` класса стек

<code>put</code>	Обязательства	Преимущества
Клиент	(Выполнить предусловие:) Вызывать <code>put (x)</code> только для непустого стека.	(Из постусловия:) Получить обновленный стек: не пустой, <code>x</code> на вершине, (<code>item</code> дает <code>x</code> , <code>count</code> увеличилось на единицу).
Поставщик	(Выполнить постусловие:) Обновить представление стека: иметь <code>x</code> на вершине (<code>item</code> возвращает <code>x</code>), <code>count</code> увеличить на единицу, стек не пуст.	(Из предусловия:) Упрощающее обработку предположение о том, что стек не пуст.

Интуиция (Дзен) и искусство программной надежности: больше гарантий и меньше проверок

Возможно, вы не заметили, что контракт противоречит мудрости, бытующей в программной инженерии. Поначалу это шокирует, но контракт - один из главных вкладов в надежность ПО.

Правило контракта говорит, что предусловие дает преимущество поставщику, если клиентская часть контракта не выполняется, то класс перестает быть связан постусловием. В этом случае программа может делать все что угодно, например зациклиться, не нарушая при этом контракт. Это тот самый случай, когда "заказчик виноват".

Первое преимущество от такого соглашения в том, что стиль программирования существенно упрощается. Разработчик класса при написании тела программы смело может предполагать, что все ограничения, заданные предусловием, выполняются; ему нет нужды проверять их в теле программы. Так для функции, вычисляющей квадратный корень:

```
sqrt (x: REAL): REAL is
  -- Квадратный корень из x
  require
    x >= 0
  do ... end
```

можно смело применять алгоритм, не учитывающий случай отрицательного `x`, поскольку это предусмотрено предусловием, и ответственность за его выполнение несут клиенты программы. С первого взгляда это может показаться опасным, но читайте дальше. Фактически метод Проектирования по Контракту идет дальше. Предположим, что мы написали в предложении `do` предыдущей программы следующий текст:

```
if x < 0 then
  "Обработать ошибку как-нибудь"
else
  "Выполнить нормальное вычисление квадратного корня"
end
```

Заметьте, в этом не только нет никакой необходимости, но это и неприемлемо! Этот факт можно отразить в следующем методологическом правиле:

Принцип Нет-Избыточности

Ни при каких обстоятельствах в теле программы не должно проверяться ее предусловие

Это правило противоречит тому, чему учат во многих учебниках по программирования, где необходимость проверок часто выступает под знаменами "защитного программирования" (**defensive programming**). Его идея в том, что для получения надежного ПО каждая программа должна защищать себя настолько, насколько это возможно. Лучше больше проверок, чем недостаточно; нельзя доверять незнакомцам; еще одна проверка может и не поможет, но и не навредит делу.

Проектирование по контракту утверждает противное: избыточные проверки могут нанести вред. Конечно, это кажется странным, на первый взгляд. Это естественная реакция, полагать, что дополнительная проверка в худшем случае может быть бесполезной, но не может быть причиной неполадок. Возьмем, например, программу `sqrt`, включившую проверку `x < 0`, хотя ее клиенты были проинструктированы о необходимости обеспечения `x >= 0`. Что в этом плохого? С микроскопической точки зрения, ограничив наше видение узким мирком `sqrt`, кажется, что включение проверки делает программу более устойчивой. Но мир системы не ограничивается одной программой - он содержит множество программ в множестве классов. Для получения надежной системы необходимо перейти к макроскопическому видению проблемы, обобщающему всю архитектуру.

С этой глобальной точки зрения **простота** становится критическим фактором. Сложность - главный враг качества. Когда в этот концерн привносятся излишние проверки, то это уже не покажется столь безобидным делом. Экстраполируйте на тысячи программ в системе среднего размера (или на десятки и сотни тысяч в большой системе) проверку (`if x < 0 then ...`), столь безобидную с первого взгляда, - все это начнет выглядеть подобно монстру бесполезной сложности. Добавляя избыточные проверки, добавляете больше кода. Больше кода - больше сложности, отсюда и больше источников ошибок, приводящих к тому, что все пойдет не так, это приведет к дальнейшему разрастанию кода и так до бесконечности. Если пойти по этой дороге, то определенно можно сказать одно - мы никогда не достигнем надежности. Чем больше пишем, тем больше придется писать.

Этот бег с препятствиями не для нас, нас ждет другая дорога. Проектирование по Контракту приглашает идентифицировать согласованные условия, необходимые для правильного функционирования каждого контракта в кооперации клиенты - поставщики. Метод вынуждает для каждого соглашения установить, кто несет ответственность - клиент или поставщик. Ответ может быть разный, частично он определяется стилем проектирования; позже будет дан ответ, как это делать лучшим образом. Но когда решение принято, нужно его придерживаться. Если требования корректности появляются в предусловии, определяя тем самым ответственность клиента, то в программе не должно быть соответствующих проверок. Требования, не указанные в предусловии, должны проверяться и выполняться в программе.

Следует отметить, что избыточные проверки широко применяются в процессе функционирования компьютеров и другой электронной аппаратуры. Физическая система, нормально функционирующая, со временем может терять целостность; причины разные - износ, разрыв, внешние воздействия. Поэтому нормальной практикой является, когда получатель и отправитель сигнала оба проверяют его целостность. Для программных систем феномена износа не наблюдается, нет и необходимости в избыточных проверках.

Можно также заметить, что так называемая избыточная проверка аппаратуры, на самом деле таковой не является: это могут быть различные дополнительные тесты, например, проверка на четность, проверка разных устройств и т.д.

Еще одним недостатком защитного программирования является его стоимость. Потеря производительности - наказание за избыточные проверки. Иногда этого вполне достаточная причина для отказа от защитного программирования, чтобы не писалось в учебниках. Работа по удалению таких проверок может быть довольно утомительной. Приемы, рассматриваемые в этой лекции, оставляют место дополнительным проверкам, но они будут основываться на разработке такого окружения, которое возьмет на себя заботу о подобных проверках. После завершения отладки достаточно будет отключить соответствующий параметр компиляции, чтобы проверки исчезли; в самом программном продукте они не содержатся.

Не говоря уже о потере производительности, принципиальной причиной отказа от защитного программирования является наша цель - получение максимальной надежности. Для систем сколь либо существенных размеров недостаточно обеспечение качества отдельных элементов, - более важно гарантировать, что для каждого взаимодействия двух элементов задан явный список взаимных обязательств и преимуществ - контракт. В заключение сформулируем парадокс Дзен-стиля: меньше проверок - больше надежности.

Утверждения не являются механизмом проверки вводимых данных

Полезно сосредоточиться на некоторых неявно обсуждавшихся свойствах контрактов. Заметьте, контракты описывают только взаимодействие двух программ (программа - программа). Контракты не задают другие виды взаимодействий: человек - программа, внешний мир - программа. Предусловие не заботится о корректировке ввода пользователя, например программа `read_positive_integer`, ожидающая в интерактивном режиме ввода пользователем положительного целого. Включение в такую программу предусловия:

```
require  
    input > 0
```

хотя и желательно, но технически не реализуемо. Полагаться на пользователя в контрактах нельзя. В данной ситуации нет заменителя обычной конструкции проверки условия, включая почтенный **if - then - else**; полезен и механизм обработки исключений.

У утверждений своя роль в решении проблемы проверки ввода данных. При описании критерия Защищенности модуля отмечалось, что Метод поощряет проверку правильности любых объектов, получаемых из внешнего мира - от сенсоров, пользовательского ввода, из сети и т. д. Эта проверка должна быть максимально приближена к источникам объектов, используя при необходимости модули - "фильтры".

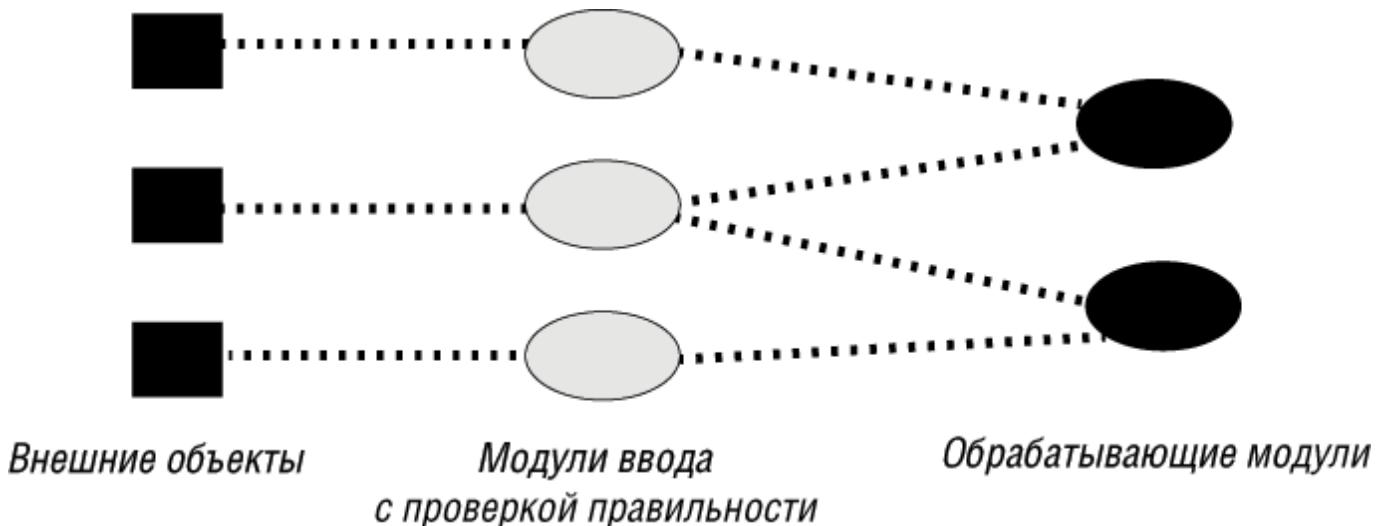


Рис. 11.1. Использование модулей - фильтров

При получении информации извне нельзя опираться на предусловия. Задача модулей ввода - гарантировать, что никакая информация не будет передана обрабатывающим модулям, пока она не будет удовлетворять условиям, требуемым для корректной обработки. При таком подходе утверждения будут широко использоваться в коммуникациях программы - программа. Постусловия модулей ввода должны соответствовать или превосходить предусловия, продиктованные обрабатывающими модулями. Фильтры играют охраняющую роль, обеспечивая корректность входных данных.

Утверждения это не управляющие структуры

Еще одно типичное заблуждение - рассматривать утверждения как управляющую структуру, реализующую разбор случаев. К этому моменту должно быть ясно, что не в этом их роль. Если написать программу `sqrt`, в которой отрицательные значения будут обрабатываться одним способом, а положительные - другим, то писать предусловие - предложение `require` не следует. В этом случае используется обычный разбор случаев: оператор `if - then - else`, или оператор `case` языка Pascal, или оператор `inspect`, введенный в этой книге как раз для таких целей.

Утверждения выражают нечто иное. Они говорят о корректности условий. Если `sqrt` имеет предусловие, то вызов, в котором $x < 0$, это "жучок" (bug).

Правило нарушения утверждения (1)

Нарушение утверждения в период выполнения является проявлением "жучка" в ПО.

Слово "жучок" не принадлежит к научному лексикону, но этот термин понятен всем программистам. Учитывая контракты, это правило можно уточнить:

Правило нарушения утверждения (2)

Нарушение предусловия является проявлением "жучка" у клиента.

Нарушение постусловия является проявлением "жучка" у поставщика.

Нарушение предусловия означает, что вызывающая программа нарушила контракт - "виноват заказчик". С позиций внешнего наблюдателя можно, конечно, критиковать сам контракт, но коль скоро контракт заключен, его следует выполнять. Если есть программа, осуществляющая мониторинг утверждений, то запускать на выполнение программу, чье предусловие не выполняется, не имеет смысла.

Нарушение постусловия означает, что программа, предположительно вызванная в корректных условиях, не выполнила свою часть работы, предусмотренную контрактом. Здесь тоже ясно, кто виноват, а кто нет: "жучок" в программе, клиент не виновен.

Ошибки, дефекты и другие насекомые

Появление слова "жучок" в предыдущем анализе нарушений утверждений - хороший повод прояснить терминологию. Э. Дейкстра полагал, что появление термина "жучок" связано с жалкой попыткой программистов обвинить кого-то в том, что ошибка "закралась" в программу со стороны, пока программисты занимались делом, - как будто не разработчики повинны в ошибках. И все же термин прижился, возможно, из-за эмоциональной окраски и понятности. И в этой книге он свободно используется, но следует дополнить его более специфическими (и более нудными) терминами для случаев, когда необходима более строгая классификация ошибок.

Термины, обозначающие бедствия ПО

Ошибка (Error) - неверное решение, принятое при разработке программной системы.

Дефект (Defect) - свойство программной системы, которое может стать причиной отклонения системы от намеченного поведения.

Неисправность (Fault) - событие в программной системе, приведшее к отклонению от нормального поведения в процессе одного из запусков системы.

Причинные связи понятны: неисправности порождаются дефектами, являющимися, в свою очередь, результатом ошибок.

"Жучок" обычно имеет смысл дефекта ("а вы уверены, что в вашей программе не осталось жучков"?). Такова его интерпретация в этой книге. Но в неформальных обсуждениях он может появляться и как ошибка и как неисправность.

Работа с утверждениями

Давайте займемся дальнейшим исследованием предусловий и постусловий, рассматривая понятные элементарные примеры. Утверждения, некоторые простые, другие более детальные, будут проникать во все примеры в последующих лекциях.

Класс стек

Поставляемый с утверждениями класс STACK был оставлен пока в схематичной форме (STACK1). Теперь на суд предстанет полная версия, включающая реализацию.

Для написания эффективного класса необходимо задать реализацию. В качестве таковой выберем реализацию стека на базе массива, уже обсуждавшаяся при рассмотрении АТД в шестой лекции.

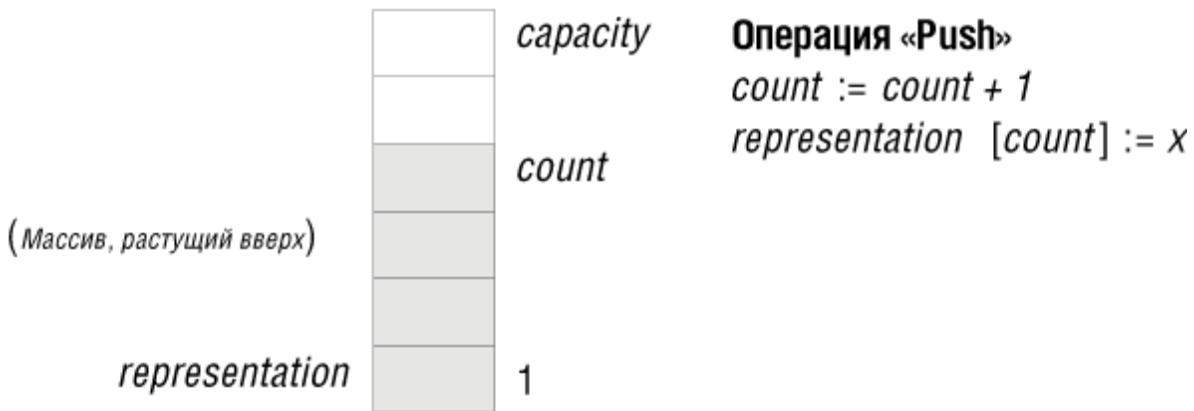


Рис. 11.2. Реализация стека на базе массива

Массив, названный representation, имеет границы 1 и capacity: реализация использует также целочисленный атрибут count, отмечающий вершину стека. Заметьте, после того, как мы откроем для себя наследование, появится возможность писать классы с отложенной реализацией, что позволит покрывать несколько возможных реализаций, а не одну конкретную. Даже для класса с фиксированной реализацией, например, как здесь на базе массива, мы будем иметь возможность строить его как потомка родительского класса Array. В данный момент никакие методы наследования применяться не будут.

Вот он класс. Остается напомнить, что для массива a операция, присваивающая значение x его i-му элементу, записывается так: a.put(x, i). Получить i-й элемент можно так: a.item(i) или a @ i. Если, как здесь, границы заданы, то i во всех случаях лежит между этими границами: $1 \leq i \leq capacity$.

```
indexing
description: "Стеки: Структуры с политикой доступа Last-In, First-Out %
  % Последний пришел - Первый ушел, и с фиксированной емкостью"
class STACK2 [G] creation
  make
```

```

feature - Initialization (Инициализация)
make (n: INTEGER) is
    -- Создать стек, содержащий максимум n элементов
    require
        positive_capacity: n >= 0
    do
        capacity := n
        create representation|make (1, capacity)
    ensure
        capacity_set: capacity = n
        array_allocated: representation /= Void
        stack_empty: empty
    end
feature - Access (доступ)
capacity: INTEGER
    -- Максимальное число элементов стека
count: INTEGER
    -- Число элементов стека
item: G is
    -- Элемент на вершине стека
require
    not_empty: not empty -- i.e. count > 0
do
    Result := representation @ count
end
feature -- Status report (Отчет о статусе)
empty: BOOLEAN is
    -- Пуст ли стек?
do
    Result := (count = 0)
ensure
    empty_definition: Result = (count = 0)
end
full: BOOLEAN is
    -- Заполнен ли стек?
do
    Result := (count = capacity)
ensure
    full_definition: Result = (count = capacity)
end
feature -- Element change (Изменение элементов)
put (x: G) is
    -- Добавить элемент x на вершину
require
    not_full: not full -- т.е. count < capacity в этом представлении
do
    count := count + 1
    representation.put (count, x)
ensure
    not_empty: not empty
    added_to_top: item = x
    one_more_item: count = old count + 1
    in_top_array_entry: representation @ count = x
end
remove is
    -- удалить элемент вершины стека
require
    not_empty: not empty -- i.e. count > 0
do
    count := count - 1
ensure
    not_full: not full
    one_fewer: count = old count - 1
end
feature {NONE} -- Implementation (Реализация)
representation: ARRAY [G]
    -- Массив, используемый для хранения элементов стека
invariant
    ... Будет добавлен позднее ...
end

```

Текст класса иллюстрирует простоту работы с утверждениями. Это полный текст, за исключением предложений **invariant**, задающих инварианты класса, которые будут добавлены позднее в этой лекции. Давайте исследуем различные свойства класса.

Это первый законченный класс этой лекции, не слишком далеко отличающийся от того, что можно найти в профессиональных библиотеках повторно используемых ОО-компонентов, таких как Базовые библиотеки. Одно замечание о структуре класса. Поскольку класс имеет более двух-трех компонентов, возникает необходимость сгруппировать его компоненты подходящим образом. Нотация позволяет реализовать такую возможность введением множества предложений **feature**. Это свойство группировки компонентов, введенное в предыдущей лекции, использовалось там, как способ задания различного статуса экспорта компонентов. И здесь в последней части класса, помеченной **Implementation**, это свойство используется для указания защищенности компонента **representation**. Но преимущества группирования можно использовать и при неизменном статусе экспорта. Его цель - сделать класс простым при чтении и легче управляемым, группируя компоненты по категориям. После каждого ключевого слова **feature** появляется комментарий, называемый комментарием к предложению **Feature** (Feature Clause Comment). Он позволяет дать содержательное описание данной категории - роль компонентов, включенных в этот раздел. Категории, используемые в примере, те же, что и при описании класса STACK1 с добавлением раздела **Initialization** с процедурой создания (конструктором).

Стандартные категории **feature** и связанные с ними комментарии к предложениям **Feature** являются частью общих правил организации повторно используемых библиотек классов.

Императив и апликатив (применимость)

Утверждения из STACK2 иллюстрируют фундаментальную концепцию - разницу между императивным и апликативным видением.

Утверждения **empty** и **full** могут вызвать удивление. Приведу еще раз текст **full**:

```
full: BOOLEAN is
      -- Заполнен ли стек?
    do
      Result := (count = capacity)
    ensure
      full_definition: Result = (count = capacity)
    end
```

Постусловие говорит, что **Result** имеет значение выражения (**count = capacity**). Но оператор присваивания именно это значение присваивает переменной **Result**. В чем же смысл написания постусловия? Не является ли оно избыточным?

Фактически между двумя конструкциями большая разница. Присваивание это команда, отданная виртуальному компьютеру на изменение его состояния. Утверждение ничего не делает, оно специфицирует свойство ожидаемого заключительного состояния, полученное клиентом, вызвавшим программу.

Инструкция предписывает (**prescriptive**), утверждение описывает (**descriptive**). Инструкция описывает "как", утверждение описывает "что". Инструкция является частью реализации, утверждение - элементом спецификации.

Инструкция **императивна**, утверждение - **апликативно**. Эти два термина выражают фундаментальную разницу между программированием и математикой.

- Компьютерные операции могут изменять состояние аппаратно-программной машины. Инструкции в языках программирования являются командами (императивные конструкции), заставляющие машину выполнять такие операции.
- Математические вычисления никогда ничего не меняют. Как отмечалось при рассмотрении АТД, взятие квадратного корня от числа 2 не меняет это число. Вместо этого математики описывают как, используя свойства одних объектов, вывести свойства других, таких как $\sqrt{2}$, полученных применением (**applying** - отсюда и термин "апликативный") математических трансформаций.

То, что две нотации в нашем примере так близки, - присваивание и эквивалентность - не должно затмнять фундаментальное различие. Утверждение описывает ожидаемый результат, инструкция предписывает способ его достижения. Клиенты модуля обычно интересуются утверждениями, а не реализациями.

Причина близости нотаций в том, что присваивание зачастую кратчайший путь достижения эквивалентности. Но при переходе к более сложным примерам концептуальное различие между спецификацией и реализацией будет только возрастать. Даже в простейшем случае вычисления квадратного корня постусловие может быть задано в форме: $\text{abs}(\text{Result}^2 - x) \leq \text{tolerance}$, где abs - обозначает абсолютное значение, а tolerance - допустимое отклонение от точного значения. Инструкции, вычисляющие квадратный корень, могут быть не тривиальными, реализуя определенный алгоритм вычисления квадратного корня.

Даже для `put` в классе `STACK2` одной и той же спецификации могут соответствовать различные алгоритмы, например:

```
if count = capacity then Result := True else Result := False end
```

или упрощенный вариант, учитывающий правила инициализации:

```
if count = capacity then Result := True end
```

В ходе работы мы столкнулись со свойством утверждений, заслуживающим дальнейшей проработки: оно важно для авторов клиентских классов, не интересующихся реализацией, но нуждающихся в абстрактном описании роли программы. Эта идея приведет нас к понятию **краткой формы (short form)**, обсуждаемой далее в этой лекции в качестве основного механизма документирования класса.

Предупреждение: по практическим соображениям допускается включение в утверждение функций - по внешнему виду императивных элементов. Эта проблема исследуется в конце этой лекции.

В заключение обсуждения полезно перечислить слова, используемые по контрасту в двух категориях программных элементов:

Таблица 11.2.

Императивно -

аппликативное

противопоставление

Реализация	Спецификация
Инструкция	Выражение
Как	Что
Императив	Аппликатив
Предписание	Описание

Замечание о пустоте структур

Предусловие в процедуре создания (конструкторе) `make` класса `STACK1` требует комментария. Оно устанавливает $n \geq 0$ и, следовательно, допускает пустые стеки. Если $n=0$, то `make` вызовет процедуру создания для массивов, также имеющую имя `make`, с аргументами 1 и 0 для нижней и верхней границ соответственно. Это не ошибка, это соответствует спецификации процедуры создания массивов, которая в случае, когда нижняя граница на единицу больше верхней, создает пустой массив.

Пустой стек не ошибка, это особый случай. Ошибка может возникнуть при попытке чтения из пустого стека, но этот случай охраняется предусловиями `put` и `item`.

При определении общих структур данных, подобных стеку или массиву, возникает вопрос о концептуальной целесообразности пустой структуры. В зависимости от ситуации ответ может быть разный, например, для деревьев полагается обычно, что дерево должно иметь хотя бы один узел - корень. Но в случае стеков или массивов, когда нет логической невозможности существования пустой структуры, ее следует допускать.

Проектирование предусловий: толерантное или требовательное?

Центральная идея Проектирования по контракту выражена в принципе Нет_Избыточности, суть которого в том, что за выполнение условия, необходимого для правильного функционирования программы, должен нести ответственность только один из партнеров контракта.

Какой же? В каждом случае есть две возможности.

- Ответственность возлагается на клиента. В этом случае условие становится частью предусловия программы.
- За все отвечает поставщик. Тогда условие появится в программе, являясь частью разбора возможных ситуаций.

Первую ситуацию назовем требовательной (**demanding**), вторую - толерантной (**tolerant**). Класс `STACK2` иллюстрирует требовательный стиль, толерантная версия, не содержащая предусловий, может выглядеть так:

```
remove is
    -- Удалить элемент вершины стека
do
```

```

if empty then
    print ("Ошибка: попытка удаления элемента из пустого стека")
else
    count := count - 1
end
end

```

Проводя аналогию с контрактами между людьми, требовательный стиль характерен для опытного поставщика, имеющего хорошо поставленное дело, и требующего от своих клиентов, чтобы они до обращения к нему выполнили всю необходимую предварительную работу. Толерантный стиль вызывает образ вновь образованной фирмы, старающейся завоевать своих клиентов, и выставляющей с этой целью рекламный плакат:



Рис. 11.3. Реклама толерантного стиля

Какой же из стилей лучше? С первого взгляда может казаться, что толерантный стиль лучше, как с позиций повторного использования, так и для повышения надежности. В требовательном стиле на всех клиентов одного поставщика ложится ответственность за выполнение ряда условий; при повторном использовании число таких клиентов только возрастает. Так не эффективнее и надежнее было бы потребовать, чтобы эту ответственность брал на себя поставщик, освобождая клиентов от обязательств?

Исследуем эту проблему чуть глубже. Условие корректности описывает требования, необходимые для успешной работы программы. Толерантная версия программы `remove` является хорошим примером, демонстрирующим слабости этого стиля. Действительно, что может сделать бедная, занимающаяся выталкиванием программа, когда стек пуст? Она делает храбрую попытку выдать явно неинформативное сообщение об ошибке, но на большее она не способна - ей недоступен контекст клиента, она не способна определить, что нужно делать, когда стек пуст. Только клиент - модуль, использующий стек для своих целей, например, модуль разбора текста в компиляторе, - обладает достаточной информацией для принятия нужного решения. Является ли это нормальным, хотя и бесполезным запросом, который следует просто проигнорировать. Если это ошибка, как следует ее обработать: выбросить ли исключение, попытаться скорректировать ситуацию, или, в крайнем случае, выдать информативное для пользователя сообщение об ошибке.

Обсуждая пример с квадратным корнем, приводился такой вариант программы:

```

if x < 0 then
    "Обработайте ошибку как-нибудь"
else
    "Выполнить нормальное вычисление квадратного корня"
end

```

Ключевое слово здесь "как-нибудь". Предложение `then` скорее заклинание, чем программный код: нет разумной, общечелевой техники обработки случая $x < 0$. Только автор клиента может знать, что значит этот случай - ошибка в ПО, возможность замены нулевым значением, причина для возбуждения исключения...

Ситуация, в которую попала толерантная версия `remove`, напоминает почтальона, который должен доставить письмо, не содержащее ни адреса получателя, ни адреса отправителя, - немногое может сделать такой почтальон.

Соответствуя духу Проектирования по контракту, требовательный подход к проектированию предусловий не пытается создавать программы, выполняющие все для своих клиентов. Более того, его суть в том, что каждая программа выполняет только хорошо определенную часть работы, но делает ее хорошо (корректно, эффективно, способную повторно использоваться многими клиентами). Такая программа четко классифицирует случаи, с которыми она не может справиться. Автор программы не должен пытаться быть умнее своих клиентов, если он не уверен, что должна делать программа в некоторой критической ситуации, он должен исключить этот случай из программы и явно включить его в предусловие. Эта позиция является следствием основной темы, проходящей через всю книгу, - создание программных систем как множества модулей, занятых своим делом.

Есть сходство в данном обсуждении и обсуждении использования частичных функций в математических моделях, рассматриваемое в лекции про АТД. Там говорилось, что целесообразнее использовать частичные функции, чем делать функцию всюду определенной, введением специального неопределенного значения - `winteger`. Эти две идеи близки, Проектирование по контракту является частью применения к программным конструкциям концепции частичных функций, замечательно гибкого и мощного аппарата формальных спецификаций.

Предупреждение: требовательный подход применим при условии, что предусловия являются разумными и обоснованными. В противном случае, работа была бы достаточно простой, достаточно для каждого модуля написать предусловие `False`, и любая программа была бы корректной. Дадим более точную характеристику "обоснованности" предусловия:

Принцип обоснованности предусловия

Каждое предусловие программы при требовательном стиле проектирования должно удовлетворять следующим требованиям:

1. Предусловие появляется в официальной документации, поставляемой авторам клиентских модулей.
2. Предусловие формулируется только в терминах спецификации, что делает возможным его вычисление.

Первое требование поддерживается понятием краткой формы, изучаемой позднее в этой лекции. Второе требование исключает появление ограничений, определяемых реализацией поставщика программы. Например, для программы, занимающейся выталкиванием элементов из стека, предусловие `not empty` является требованием, проверяемым в терминах спецификации, и вытекающим из очевидного факта - из пустого стека ничего нельзя вытолкнуть. При вычислении квадратного корня предусловие `x>0` отражает известный математический факт, - отрицательные числа не имеют вещественных квадратных корней.

Некоторые ограничения могут навязываться реализацией. Например, в программе `put` из класса STACK2 присутствие в качестве предусловия `require not full` связано с реализацией стека на основе массива. Но это не является нарушением принципа, поскольку класс STACK2 в полном соответствии с его спецификацией определяет стеки ограниченной емкости, что отражено, например, в предложении `indexing` этого класса. АТД, служащий в роли спецификации этого класса, не задает наиболее общий вид стеков, но является понятием стека ограниченной емкости.

Обычно следует избегать структур ограниченной емкости; даже в случае массивов можно строить стеки на динамических массивах, изменяющих размерность при необходимости. В Базовой библиотеке представлен общий класс, описывающий стеки¹, отличающийся от класса STACK2 тем, что в нем не используется понятие емкости; стек по умолчанию перестраивается, когда текущей емкости недостаточно для хранения очередного поступающего элемента.

Предусловия и статус экспорта

Возможно, вы заметили необходимость дополнительного требования, не отраженного в принципе обоснованности предусловия. Для того чтобы клиент мог проверить предусловие, оно не должно использовать закрытые свойства класса, недоступность которых отражена в статусе экспорта.

Рассмотрим следующую ситуацию:

-- Предупреждение: это неправильный класс, только в целях иллюстрации.
class SNEAKY feature
 tricky is
 require
 accredited
 do
 ...
 end
 feature {NONE}
 accredited: BOOLEAN is do ... end
 end

Спецификация для `tricky` устанавливает, что любой вызов этой процедуры должен удовлетворять условию, выраженному булевой функцией `accredited`. Но при экспорте класса эта функция для клиентов является закрытой, поэтому у них нет способа проверить выполнимость условия перед вызовом `tricky`. Очевидно, подобная ситуация неприемлема.

Причина, по которой принцип Обоснованности предусловия не покрывает подобные ситуации, в том, что это методологический принцип, а мы нуждаемся в правиле языка, заставляющем компилятор контролировать решение проблемы, не полагаясь на разработчиков.

Это правило учитывает все возможные ситуации экспорта, а не только случаи доступности всем клиентам (`tricky`) или полной недоступности (`accredited`). Как отмечалось, при обсуждении проблемы скрытия информации, компонент класса можно сделать доступным для некоторых клиентов, явно перечислив их в `feature` предложении, например `feature {A, B, ...}`, определяющего доступность только для классов A, B, ... и их потомков. Сформулируем правило языка:

Правило Доступности предусловия

Каждый компонент, появляющийся в предусловии программы, должен быть доступен каждому клиенту, которому доступна сама программа.

В соответствии с этим правилом каждый клиент, способный вызвать программу, способен проверить ее предусловие. По этому правилу класс `SNEAKY` является коварным, некорректно построенным, поскольку экспортирует `tricky` с недоступным предусловием. Нетрудно превратить этот класс в правильно построенный, изменив статус экспортации у `accredited`. Если `tricky` появится с предложением `feature` в форме `feature {A, B, C}`, то `accredited` должна экспортироваться, по меньшей мере, клиентам A, B, C, появляясь в той же группе `feature`, что и `tricky`. Можно задать для `accredited` собственное `feature`-предложение в одной из форм: `feature {A, B, C}, feature {A, B, C, D, ...}` или просто `feature`. Любое нарушение этого правила приведет к ошибке в период компиляции. Класс `SNEAKY`, например, будет забракован компилятором.

Подобного правила нет для постусловий. Не является ошибкой в постусловии ссылаться на компоненты, закрытые или экспортируемые избранным клиентам. Просто это означает, что описание эффекта выполнения программы содержит некоторые свойства, непосредственно не используемые клиентом. Подобная ситуация имеет место в процедуре `put` класса `STACK2`:

```
put (x: G) is
    -- Добавить элемент x на вершину
    require
        not full
    do
        ...
    ensure
        ... Другие предложения...
        in_top_array_entry: representation @ count = x
    end
```

Последнее предложение в постусловии устанавливает, что элемент массива с индексом `count` содержит последний втолкнутый в стек элемент. Это свойство реализации, хотя `put` обычно доступно (экспортируется всем клиентам), массив `representation` является закрытым. Но ничего ошибочного в постусловии нет. Оно просто включает наряду со свойствами, полезными для клиентов ("Другие предложения"), свойство, имеющее смысл только для тех, кто знаком с полным текстом класса. Такие закрытые предложения не будут появляться в краткой форме класса - документации, предназначеннной для авторов клиентских модулей.

Тolerантные модули

(При первом чтении этот раздел можно опустить или ограничиться его беглым просмотром.)

Простые, но не защищенные модули могут быть не достаточно устойчивыми для использования их у произвольных клиентов. В таких случаях возникает необходимость создания нескольких классов, играющих роль фильтров. В отличие от ранее рассмотренных фильтров, устанавливаемых между внешним миром и обрабатывающими модулями, новые фильтры будут устанавливаться между "беспечными" клиентами с одной стороны и незащищенными классами с другой стороны.

Хотя было показано, что обычно это не лучший подход к проектированию, полезно рассмотреть, как выглядят классы, если использовать толерантный стиль в некоторых особых случаях. Класс `STACK3`, представленный ниже, иллюстрирует эту идею.

Поскольку классу понадобятся целочисленные коды ошибок, удобно для этой цели использовать ранее не введенную нотацию "unique" для целочисленных констант. Если объявить множество атрибутов следующим

образом:

```
a, b, c, ...: INTEGER is unique
```

то в результате этого объявления *a*, *b*, *c* получат последовательно идущие целочисленные значения. Эти значения будут даваться компилятором с гарантией того, что все объявленные таким образом константы получат различные значения (будут уникальными). По принятому соглашению, всем объявляемым таким образом константам даются имена, начинающиеся с буквы в верхнем регистре и с остальными символами в нижнем регистре, например *Underflow*.

Вот написанная в этом стиле толерантная версия нашего класса стек. Заметьте, что этот текст, возможно пропущенный при первом чтении, включен только для понимания толерантного стиля. Он не является примером рекомендуемого стиля проектирования по причинам, обсуждаемым ниже, но которые достаточно ясны при просмотре этого текста.

```
indexing
  description: "Стеки: Структуры с политикой доступа Last-In, First-Out %
    %Первый пришел - Последний ушел, с фиксированной емкостью; %
    %толерантная версия, устанавливающая код ошибки в случае %
    %недопустимых операций."
class STACK3 [G] creation
  make
feature - Initialization (Инициализация)
  make (n: INTEGER) is
    -- Создать стек, содержащий максимум n элементов, если n > 0;
    -- в противном случае установить код ошибки равным Negative_size.
    -- Без всяких предусловий!
  do
    if capacity >= 0 then
      capacity := n
      create representation.make (capacity)
    else
      error := Negative_size
    end
  ensure
    error_code_if_impossible: (n < 0) = (error = Negative_size)
    no_error_if_possible: (n >= 0) = (error = 0)
    capacity_set_if_no_error: (error = 0) implies (capacity = n)
    allocated_if_no_error: (error = 0) implies (representation /= Void)
  end
feature - Access (Доступ)
  item: G is
    -- Элемент вершины, если существует; в противном случае
    -- значение типа по умолчанию.
    -- с ошибкой категории Underflow.
    -- Без всяких предусловий!
  do
    if not empty then
      check representation /= Void end
      Result := representation.item
      error := 0
    else
      error := Underflow
      -- В этом случае результатом является значение по умолчанию
    end
  ensure
    error_code_if_impossible: (old empty) = (error = Underflow)
    no_error_if_possible: (not (old empty)) = (error = 0)
  end
feature -- Status report (Отчет о статусе)
  empty: BOOLEAN is
    -- Пуст ли стек?
  do
    Result := (capacity = 0) or else representation.empty
  end
  error: INTEGER
    -- Индикатор ошибки, устанавливаемый различными компонентами
    -- в ненулевое значение, если они не могут выполнить свою работу
  full: BOOLEAN is
```

```

-- Заполнен ли стек?
do
    Result := (capacity = 0) or else representation.full
end
Overflow, Underflow, Negative_size: INTEGER is unique
-- Возможные коды ошибок
feature -- Element change (Изменение элементов)
put (x: G) is
    -- Добавить x на вершину, если возможно; иначе задать код ошибки.
    -- Без всяких предусловий!
do
    if full then
        error := Overflow
    else
        check representation /= Void end
        representation.put (x); error := 0
    end
ensure
    error_code_if_impossible: (old full) = (error = Overflow)
    no_error_if_possible: (not old full) = (error = 0)
    not_empty_if_no_error: (error = 0) implies not empty
    added_to_top_if_no_error: (error = 0) implies item = x
    one_more_item_if_no_error: (error = 0) implies count = old count + 1
end
remove is
    -- Удалить вершину, если возможно; иначе задать код ошибки.
    -- Без всяких предусловий!
do
    if empty then
        error := Underflow
    else
        check representation /= Void end
        representation.remove
        error := 0
    end
ensure
    error_code_if_impossible: (old empty) = (error = Underflow)
    no_error_if_possible: (not old empty) = (error = 0)
    not_full_if_no_error: (error = 0) implies not full
    one_fewer_item_if_no_error: (error = 0) implies count = old count - 1
end
feature {NONE} - Implementation (Реализация)
representation: STACK2 [G]
    -- Незащищенный стек используется для реализации
capacity: INTEGER
    -- Максимальное число элементов стека
end - class STACK3

```

Операции этого класса не имеют предусловий (более точно, имеют True в качестве предусловия). Результат выполнения может характеризовать ненормальную ситуацию, поступление переопределено так, чтобы позволить отличать корректную и ошибочную обработку. Например, при вызове `s.remove`, где `s` это экземпляр класса `STACK3`, в корректной ситуации значение `s.error` будет равно 0; в ошибочной - `Underflow`. В последнем случае никакая другая работа выполняться не будет. Клиент несет ответственность за проверку `s.error` после вызова. Как уже отмечалось, у общечелевого модуля, такого как `STACK3` нет способа решить, что делать в ошибочной ситуации: выдать сообщение об ошибке, произвести корректировку ситуации...

Такие модули фильтры служат для отделения нормальных ситуаций от ситуаций, обрабатывающих ошибки. В этом отличие корректности от устойчивости, объясняемое в начале книги: написание модуля корректно выполняющего свою задачу в предусмотренных случаях - одна задача, сделать так, чтобы и в непредусмотренных ситуациях обработка выполнялась сносно - совсем другая задача. Обе они необходимы, но их нужно разделять и управлять ими по-разному. Одна из типичных ошибок, приводящая к безнадежной сложности программных систем, - в алгоритм, делающий действительно нечто полезное, добавляется куча проверок на безнадежные ситуации и из лучших побуждений делается попытка управлять ими. В таких системах путаница начинает расти как грибы после дождя.

Несколько технических замечаний к приведенному примеру класса.

- Экземпляр `STACK3` - содержит атрибут `representation`, представляющий ссылку на экземпляр `STACK2`, содержащий, в свою очередь, ссылку на массив. Эти обходные пути пагубно отражаются на

- эффективности, избежать этого можно введением наследования, изучаемого в последующих лекциях.
- Булева операция **or else** подобна **or**, но если первый операнд равен `True`, игнорирует второй операнд, возможно неопределенный в такой ситуации.
 - Инструкция **check**, используемая в `put` и `remove`, служит для проверки выполнения некоторых утверждений. Она будет изучаться позднее в этой лекции.

В заключение: вы, наверное, отметили тяжеловесность STACK3 в сравнении с простотой STACK2, достигнутой благодаря предусловиям. Это хороший пример, показывающий, что толерантный стиль может приводить к бесполезно усложненному ПО. Требовательный стиль, по контрасту, вытекает из общего духа Проектирования по контракту. Попытка управлять всем, - и возможными и невозможными случаями - совсем не лучший способ помочь вашим клиентам. Если вместо этого вы построите классы, влекущие возможно более строгие условия на их использование, точно опишите эти условия, включив их в документацию класса, вы реально облегчите жизнь вашим клиентам. Требовательная любовь (*tough love*) может быть лучше всепрощающей; лучше эффективная поддержка функциональности с проверяемыми ограничениями, чем страстная попытка предугадать желания клиентов, принятие возможно неадекватных решений, жертвой чего становятся простота и эффективность.

Для модулей, чьими клиентами являются другие программные модули, требовательный подход обычно является правильным выбором. Возможным исключением становятся модули, предназначенные для клиентов, чьи авторы используют не ОО-языки и могут не понимать основных концепций Проектирования по контракту.

Толерантный подход остается полезным для модулей, принимающих данные от внешнего мира. Как отмечалось, в этом случае строятся фильтры, отделяющие внешний мир от обрабатывающих модулей. Класс STACK3 иллюстрирует идеи построения подобных фильтров.

Инварианты класса

Предусловия и постусловия описывают свойства отдельных программ. Но экземпляры класса обладают также глобальными свойствами. Их принято называть инвариантами класса (class invariants), и они определяют более глубокие семантические свойства и ограничения целостности, характеризующие класс.

Определение и пример

Рассмотрим снова реализацию стека классом STACK2:

```
class STACK2 [G] creation
  make
feature
  ? make, empty, full, item, put, remove?
  capacity: INTEGER
  count: INTEGER
feature {NONE} -- Implementation
  representation: ARRAY [G]
end
```

Атрибуты класса - массив `representation` и целые `capacity`, `count` - задают представление стека. Хотя предусловия и постусловия программ отражают семантику стека, их недостаточно для выражения важных свойств, связывающих атрибуты. Например, `count` всегда должно удовлетворять условию:

```
0 <= count; count <= capacity
```

из которого следует, что `capacity >= 0` и что `capacity` задает размер массива:

```
capacity = representation.capacity
```

Инвариант класса это утверждение, выражающее общие согласованные ограничения, применимые к каждому экземпляру класса как целому. Этим они отличаются от предусловий и постусловий, характеризующих отдельные программы.

Выше приведенные примеры инвариантов включали только атрибуты. Инварианты могут выражать отношения между функциями и между функциями и атрибутами. Например, инвариант STACK2 может включать следующее свойство, описывающее связь между функцией `empty` и `count`:

```
empty = (count = 0)
```

Этот пример не показателен, он повторяет утверждение, заданное постусловием `empty`. Более полезные утверждения те, которые включают только атрибуты или более чем одну функцию.

Вот еще один типичный пример. Предположим, что мы имеем дело с банковскими счетами, и есть класс `Bank_Account` с компонентами: `deposits_list`, `withdrawals_list` и `balance`. Тогда инвариантом такого класса может быть утверждение в форме:

```
consistent_balance: deposits_list.total - withdrawals_list.total = balance
```

где функция `total` дает суммарное значение списка всех операций (приходных или расходных). Инвариант определяет основное условие согласования всех банковских операций над счетом, связывая баланс, приходные и расходные операции.

Форма и свойства инвариантов класса

Синтаксически инвариант класса является утверждением, появляющимся в предложении **invariant**, стоящим после всех предложений **feature**, и перед предложением **end**. Вот пример:

```
class STACK4 [G] creation
  ...Как в STACK2...
feature
  ...Как в STACK2...
invariant
  count_non_negative: 0 <= count
  count_bounded: count <= capacity
  consistent_with_array_size: capacity = representation.capacity
  empty_if_no_elements: empty = (count = 0)
  item_at_top: (count > 0) implies (representation.item (count) = item)
end
```

Инвариант класса С это множество утверждений, которым удовлетворяет каждый экземпляр класса во все "стабильные" времена. В эти времена экземпляр класса находится в наблюдаемом состоянии:

- на момент создания экземпляра, сразу после выполнения `create a` или `create a.make(...)`, где `a` класса С;
- перед и после каждого удаленного вызова `a.r(...)` программы `r` класса С.

Следующий рисунок, показывающий жизнь объектов, поможет разобраться в инвариантах и стабильных временах:

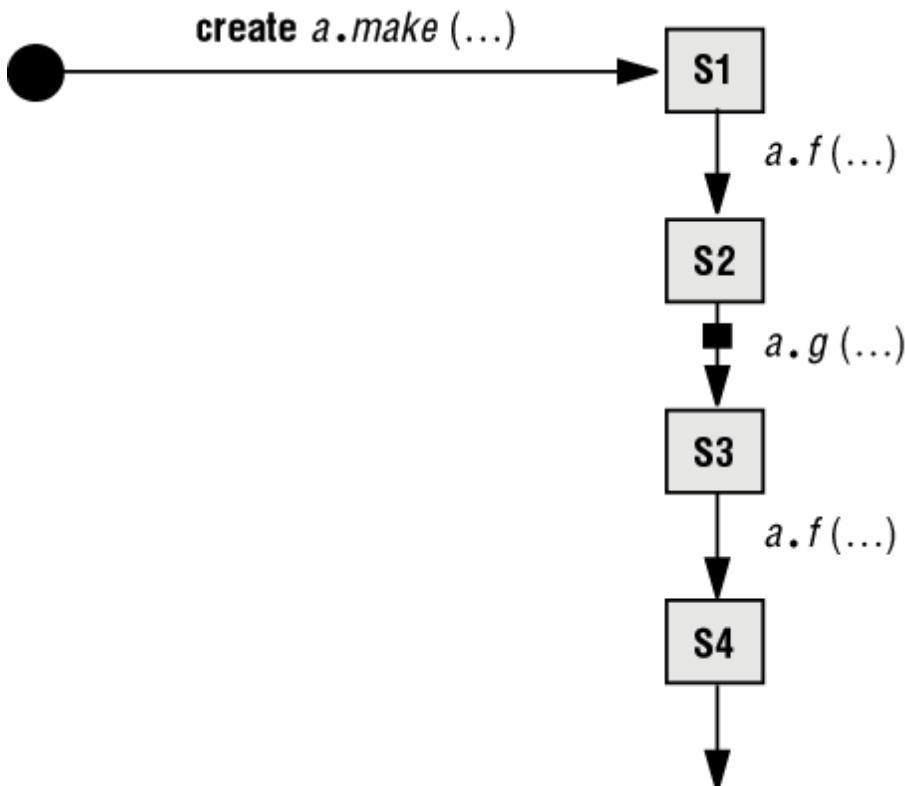


Рис. 11.4. Жизнь объектов

Жизнь объектов не столь уж захватывающая. Вначале - слева на рисунке - он просто не существует. При выполнении инструкции `create a` или `create a.make(...)` или `clone` объект создается и достигает первой станции `S1` в своей жизни. Затем идет череда довольно скучных событий: клиенты, для которых доступен объект,

один за другим вызывают его компоненты в форме $a.f(\dots)$. Так все продолжается, пока не завершится вычисление.

Инвариант является характеристическим свойством состояний, представленных большими квадратиками на рисунке: S1, S2, S3 и т.д. Эти состояния соответствуют стабильным временам, упомянутым выше.

Здесь рассматриваются последовательные вычисления, но все идеи легко переносятся на параллельные вычисления, что и будет сделано в соответствующей лекции.

Инвариант в момент изменения

Несмотря на свое имя, инвариант не должен выполняться во все времена. Вполне законно, что некоторая процедура g , начиная выполнять свою работу, разрушает инвариант, а, завершая работу, восстанавливает его истинность. В промежуточном состоянии, показанном на рисунке маленьким квадратиком, инвариант не выполняется, но инвариант всегда должен выполняться в заключительном состоянии каждой процедуры. И в человеческом сообществе многие, стараясь сделать что-либо полезное, начинают с того, что разрушают существующий порядок вещей.

Кто должен обеспечить сохранность инвариантов

Квалифицированные вызовы в форме $a.f(\dots)$, выполняемые на стороне клиента, всегда начинаются и заканчиваются в состоянии, удовлетворяющем инварианту. Подобного правила нет для неквалифицированных вызовов в форме $f(\dots)$, недоступных для клиентов, но используемых в квалифицированных вызовах для служебных целей. Как следствие, обязанность управлять инвариантами возлагается только на модули, экспортируемые всем клиентам или выборочно. Закрытые методы, недоступные клиентам, не обязаны беспокоиться об инвариантах.

Закончим обсуждение правилом, точно определяющим, когда утверждение является корректным инвариантом класса:

Правило инварианта

Утверждение Inv является корректным инвариантом класса, если и только если оно удовлетворяет следующим двум условиям:

1. Каждая процедура создания, применимая к аргументам, удовлетворяющим ее предусловию в состоянии, в котором атрибуты имеют значения, установленные по умолчанию, вырабатывает заключительное состояние, гарантирующее выполнение Inv .
2. Каждая экспортируемая процедура класса, примененная к аргументам в состоянии, удовлетворяющем Inv и предусловию, вырабатывает заключительное состояние, гарантирующее выполнение Inv .

Заметьте, в этом правиле:

- Предполагается, что каждый класс обладает процедурой создания, задаваемой конструктором по умолчанию, при отсутствии явного ее определения.
- Состояние объекта определяется значениями всех его полей (значениями атрибутов класса для этого конкретного экземпляра).
- Предусловие программы может включать начальное состояние и аргументы.
- Постусловие может включать только заключительное состояние, начальное состояние, (используя нотацию old) и, в случае функций, возвращаемое значение, заданное предопределенной сущностью Result .
- Инвариант может включать только состояние.

Утверждения могут использовать функции, но такие функции фактически являются ссылками на атрибуты - состояния.

Математическое выражение правила Инварианта появится позже в этой лекции.

Можно использовать правило Инварианта как основу для ответа на вопрос, что означает нарушение инварианта в период выполнения системы? Мы уже установили, что нарушение предусловия означает ошибку (жучок) клиента, нарушение постусловия - ошибка поставщика. Для инвариантов ответ такой же, как и для постусловий²⁾.

Роль инвариантов класса в программной инженерии

Свойство (2) правила инвариантов показывает, что неявно их можно рассматривать как добавления к предусловиям и постусловиям каждой экспортируемой программы класса. Посему принципиально понятие инварианта класса избыточно - это часть предусловий и постусловий программ.

Такое преобразование, конечно, не желательно. Это усложнило бы тексты программ, и, что более важно, - был бы утерян глубокий смысл инварианта, выходящий за пределы отдельных программ, применяемый к классу, как

целому. Следует помнить, что инвариант применим не только к уже написанным программам класса, но и к тем, которые еще будут написаны. Он контролирует эволюцию класса, что будет отражено в правилах наследования.

Изменения в ПО неизбежны. Задача в том, чтобы уметь управлять ими. Этот подход соответствует принципам разработки, введенным в начале этой книги. Можно ожидать, что некоторые аспекты программных систем и их компонентов - классов - меняются чаще, чем другие. Добавление, удаление, изменение функциональности явление частое и нормальное. В этом изменчивом процессе все-таки хотелось бы иметь устойчивые свойства, в значительной степени, не подверженные изменениям. Именно эту роль играют инварианты, поскольку в них отражаются фундаментальные соотношения, характерные для класса. Конечно, в программных системах все может изменяться, едва ли можно гарантировать неприкосновенность любого из аспектов системы. Но фундамент остается фундаментом.

Класс STACK2 иллюстрирует базисные идеи, но оценить полную мощь инвариантов можно, лишь ознакомившись со всеми дальнейшими их примерами в остальной части этой книги. Понятие инварианта является одной из наиболее значимых концепций ОО-метода. Только после того, как я написал инвариант, (для разработанного мной класса), только после знакомства и понимания инвариантов (для изучаемого мной класса), только тогда я почувствовал, - я знаю, что такое класс.

Инварианты и контракты

В метафоре контрактов интерпретация инвариантов ясна и понятна. В сообществе людей все контракты часто содержат ссылки на общие правила, регулирующие отношения между партнерами независимо от конкретной области применения контракта. Например правила, установленные для городских зон, справедливы для всех контрактов по строительству жилья. Инварианты класса играют роль общих правил: инвариант класса действует на все контракты между программами класса и клиентами.

Давайте пойдем дальше. Выше отмечалось, что инварианты можно рассматривать как добавки к предусловиям и постусловиям экспортируемых программ. Пусть `body` тело программы, `pre` - предусловие, `post` - постусловие, `Inv` - инвариант программы. Требование корректности программы может быть записано в виде:

```
{INV and pre} body {INV and post}
```

Это означает, что любое выполнение `body`, начинающееся в состоянии, удовлетворяющем `Inv` и `pre`, завершится в состоянии, в котором выполняются `Inv` и `post`. Для человека, создающего `body`, появление инварианта является "хорошей" или "плохой" новостью, облегчается или затрудняется его задача? Ответ, как следует из предыдущих обсуждений, и да и нет! Вспомним ленивого работника, который мечтает о сильном предусловии и слабом постусловии, чтобы можно было бы работать как можно меньше. Инвариант усиливает как предусловие, так и постусловие. Так что, если вы ответственны за реализацию `body`, то добавление инварианта:

- Облегчает работу: накладывая на клиента более жесткие требования, уменьшая тем самым число ситуаций, при которых нужно приступать к работе.
- Усложняет работу: помимо постусловия в заключительном состоянии необходимо гарантировать выполнение инварианта.

Эти наблюдения согласуются с ролью инварианта, задающего общие требования к классу. Приступая к работе над одной из программ класса, вы получаете преимущества, поскольку гарантируется выполнение общих для класса условий. Но на вас возлагается обязанность к концу работы сохранить выполнимость этих условий, чтобы ими могли воспользоваться и другие программы класса.

Класс `BANK_ACCOUNT`, упоминавшийся выше, с инвариантом класса:

```
deposits_list.total - withdrawals_list.total = balance
```

дает хороший пример. При добавлении в класс новой программы гарантируется, что свойства `deposits_list`, `withdrawals_list`, `balance` имеют согласованные значения, посему нет необходимости в проверках согласованности. Но это также означает, что при написании программы следует следить за сохранением согласованности. Так что процедура `withdraw`, которая занимается снятием некоторых сумм со счетов, должна в конце работы изменить соответственно и баланс - атрибут `balance`.

Заметьте, `balance` может быть не атрибутом, а функцией, возвращающей значение, вычисляемое, например, так: `deposits_list.total - withdrawal_list.total`. В этом случае процедуре `withdraw` вообще ничего не нужно делать для обеспечения выполнимости инварианта. Возможность переключаться между двумя представлениями (атрибута и функции) без влияния на клиента обеспечивается принципом Унифицированного доступа.

Когда класс корректен?

Хотя нам еще предстоит ознакомиться с рядом конструкций, связанных с утверждениями, пора сделать паузу и

проезаменовать некоторые из следствий уже изученных понятий - предусловий, постусловий, инвариантов. В этом разделе не вводятся никакие новые конструкции, но описываются теоретические обоснования сделанного. Полагаю, и при первом чтении следует познакомиться с этими идеями, поскольку они являются основополагающими для правильного понимания метода, и будут иметь большую ценность при попытке постигнуть, как использовать наследование должным образом.

Корректность класса

Вооруженные понятиями инварианта, предусловий и постусловий, мы можем теперь точно определить понятие корректности уже не отдельной подпрограммы, а класса в целом.

Класс, подобно всем остальным программным элементам, не может быть корректным или некорректным сам по себе, - только по отношению к некоторой спецификации. Инварианты, предусловия и постусловия, это способ задания спецификации класса. На этой основе можно приступить к определению корректности: класс корректен, если и только если его реализация, заданная подпрограммами, согласована с предусловиями, постусловиями и инвариантами.

Нотация $\{P\} A \{Q\}$ поможет выразить наше определение более строго.

Пусть С обозначает класс, Inv - инвариант, r - программа класса. Для каждой программы Body_r - ее тело, pre_r(x_r), post_r(x_r) - ее предусловие и постусловие с возможными аргументами x_r. Если предусловие или постусловие для программы r опущены, то будем считать их заданными константой True.

Наконец, пусть Default_C обозначает утверждение, выражающее тот факт, что атрибуты класса С имеют значения по умолчанию, определенные их типами. Например, Default_{STACK2} для класса STACK2 является следующим утверждением:

```
representation = Void
capacity = 0
count = 0
```

Эта нотация позволяет дать общее определение корректности класса:

Определение: Корректность класса

Класс С корректен по отношению к своим утверждениям, если и только если:

1. Для любого правильного множества аргументов x_p процедуры создания p:

$$\{\text{Default}_C \text{ and } \text{pre}_p(x_p)\} \text{ Body}_p \{\text{post}_p(x_p) \text{ and } \text{Inv}\}$$

2. Для каждой экспортруемой программы r и для любого множества правильных аргументов x_r:

$$\{\text{pre}_r(x_r) \text{ and } \text{Inv}\} \text{ Body}_r \{\text{post}_r(x_r) \text{ and } \text{Inv}\}$$

Это правило является математической формулировкой ранее рассмотренной неформальной диаграммы, показывающей жизненный цикл типичного объекта ([рис. 11.4](#)). Условие (1) означает, что любая процедура создания при ее вызове с выполняемым предусловием должна вырабатывать начальное состояние (S1 на рисунке), удовлетворяющее постусловию и инварианту. Условие (2) отражает тот факт, что любая экспортруемая процедура r (f и g на рисунке), вызываемая в состояниях (S1, S2, S3), вызываемая в состояниях, удовлетворяющих предусловию и инварианту, должна завершаться в состояниях, удовлетворяющих постусловию и инварианту.

Два практических замечания:

- Если у класса нет предложения **creation**, то можно полагать, что существует неявная процедура создания по умолчанию - nothing с пустым телом. Применение правила (1) к B_{nothing} в этом случае означает, что Default_C влечет Inv; другими словами, значения полей по умолчанию должны удовлетворять инварианту в этом случае.
- Из определения корректности класса следует, что любая экспортруемая программа может делать, все что угодно, если при ее вызове нарушается предусловие или инвариант.

Только что было описано, как определить корректность класса. На практике чаще хочется проверить, что данный класс действительно корректен. Эта проблема будет обсуждаться позднее в этой лекции.

Роль процедур создания

Инвариант класса задает множество свойств объектов (экземпляров класса), которые должны выполняться в

стабильные времена жизни объектов. В частности, эти свойства должны выполняться сразу после создания экземпляра объекта.

Стандартный механизм распределения инициализирует поля значениями по умолчанию соответствующих типов, приписанных атрибутам. Эти значения могут удовлетворять или не удовлетворять инварианту. Если нет, то требуется специальная процедура создания, инициализирующая значения атрибутов таким образом, чтобы инвариант выполнялся. Поэтому процедуру создания можно рассматривать, как операцию, гарантирующую, что все экземпляры класса начинают жить, имея корректный статус, - в котором инвариант выполняется.

При первом представлении процедур создания они рассматривались, как способ ответа на земной (и очевидный) вопрос, как переопределить инициализацию по умолчанию, если она не подходит для моего класса. Другая рассматриваемая проблема, - как задать несколько различных механизмов инициализации. Но теперь, с введением инвариантов и теоретического обсуждения, отраженного в правиле (1), мы видим более весомую роль процедур создания. Теперь они создают уверенность, что любой экземпляр класса, только начиная жить, удовлетворяет фундаментальным правилам своей касты - инварианту класса.

Ревизия массивов

Набросок библиотечного класса ARRAY дан в предыдущей лекции. Теперь мы в состоянии дать ему подходящее определение. Фундаментальное понятие массива требует задания предусловий, постусловий и инварианта.

Приведем улучшенный, но все еще схематичный вариант, включающий утверждения. Предусловия выражают базисные требования к доступу и модификации элементов: индексы должны быть в допустимой области. Инвариант задает отношение, существующее между count, lower и upper. Компонент count разрешается реализовать функцией, а не задавать атрибутом.

```
indexing
  description: "Последовательности значений одного типа или %
  %согласованных типов, доступных по индексам - целым из заданного интервала %"
class ARRAY [G] creation
  make
feature - Initialization (Инициализация)
  make (minindex, maxindex: INTEGER) is
    -- Создать массив с границами minindex и maxindex
    -- (пустой если minindex > maxindex).
  require
    meaningful_bounds: maxindex >= minindex - 1
  do
    ...
  ensure
    exact_bounds_if_non_empty: (maxindex >= minindex) implies
      ((lower = minindex) and (upper = maxindex))
    conventions_if_empty: (maxindex < minindex) implies
      ((lower = 1) and (upper = 0))
  end
feature -- Access (Доступ)
  lower, upper, count: INTEGER
    -- Минимальное и максимальное значение индекса; размер массива.
  infix "@", item (i: INTEGER): G is
    -- Элемент с индексом i
  require
    index_not_too_small: lower <= i
    index_not_too_large: i <= upper
  do ... end
feature -- Element change (Изменение элементов)
  put (v: G; i: INTEGER) is
    -- Присвоить v элементу с индексом i
  require
    index_not_too_small: lower <= i
    index_not_too_large: i <= upper
  do
    ...
  ensure
    element_replaced: item (i) = v
  end
invariant
  consistent_count: count = upper - lower + 1
  non_negative_count: count >= 0
end
```

Единственное, что не конкретизировано в описании этого класса, это реализация программ `item` и `put`. Поскольку эффективная манипуляция с массивом требует доступа к системам низкого уровня, то эти программы будут реализованы с использованием внешних классов, что будет рассмотрено в последующих лекциях.

Связывание с АТД

Класс, как неоднократно говорилось, является реализацией АТД, заданного формальной спецификацией или неявно подразумеваемого. В начале лекции отмечалось, что утверждения можно рассматривать, как способ введения в класс семантических свойств, лежащих в основе АТД. Давайте уточним наше понимание концепции утверждений, прояснив их связь с компонентами спецификации АТД.

Не просто коллекция функций

Как отмечалось в лекции про АТД, они включают четыре элемента:

- имя типа, возможно с родовым параметром (раздел TYPES);
- список функций с их сигнатурами (раздел FUNCTIONS);
- аксиомы, выражающие свойства результатов функций (раздел AXIOMS);
- ограничения применимости функций (раздел PRECONDITIONS).

При поверхностном применении АТД часто опускают две последние части. Во многом, это лишает данный подход привлекательности, поскольку предусловия и аксиомы выражают семантические свойства функций. Если их опустить и просто рассматривать стек как инкапсуляцию операций `put`, `remove` и других, то преимущества от скрытия информации останутся, но это все. Понятие стека становится пустой оболочкой без семантики, кроме той, что остается в именах функций. (В этой книге имена функций менее информативны по причине согласованности и повторного использования, - мы сознательно выбрали общие имена - `put`, `remove`, `item`, а не те, которые применяются обычно для стеков - `push`, `pop`, `top`).

Этот риск потери семантики переносится на программирование: программы, реализующие операции соответствующего АТД, в принципе могут выполнять нечто отличное от задуманного. Утверждения предотвращают этот риск, возвращая семантику классу.

Компоненты класса и АТД функции

Для понимания отношений между утверждениями и АТД необходимо, прежде всего, установить отношение между компонентами класса и их двойниками - АТД функциями. В свете прежних обсуждений функции подразделяются на три категории: создатели, запросы и команды. Возвращаясь назад, напомню, категория функции f

$$f : A \times B \times \dots \rightarrow X$$

зависит от того, где имя АТД, скажем T , встречается среди типов A , B , \dots X , включенных в эту сигнатуру:

- Если T появляется только справа от стрелки, f является создателем; в классе это соответствует процедуре создания.
- Если T появляется только слева от стрелки, f является запросом, обеспечивающим доступ к свойству экземпляра класса. Для класса запрос соответствует атрибуту или функции; термин запрос сохраняется и для класса, когда нет необходимости различать, как он реализован.
- Если T появляется как слева, так и справа от стрелки, f является командой, вырабатывающей новый объект из одного или нескольких уже существующих. На этапе реализации f часто задается процедурой (также называемой командой), которая модифицирует существующий объект, не создавая новый, как это делают функции.

Выражение аксиом

Из соответствия между АТД функциями и компонентами класса можно вывести соответствие между утверждениями класса и семантическими свойствами АТД.

- Предусловие для специфицированной в АТД функции появляется как предусловие программы, соответствующей данной функции.
- Аксиома, включающая команду, и, возможно, одну или более функций запросов, появится как постусловие соответствующей процедуры.
- Аксиомы, включающие только запросы, появятся как постусловия соответствующих функций или как инвариант. Последнее обычно имеет место, если более чем одна функция включена в аксиому, или, по меньшей мере, один из запросов реализован в виде атрибута.
- Аксиомы, включающие функцию создатель, появятся в постусловии соответствующей процедуры создания.

В этот момент следует вернуться назад и сравнить аксиомы АТД STACK с утверждениями класса STACK4 (включая и те, которые даны для класса STACK2).

Функция абстракции

Этот раздел требует от читателя определенной математической подготовки.

Полезно рассмотреть предшествующее обсуждение в терминах следующего рисунка, навеянного работой [Hoare 1972a], в которой описывается понятие "С является корректной реализацией А".

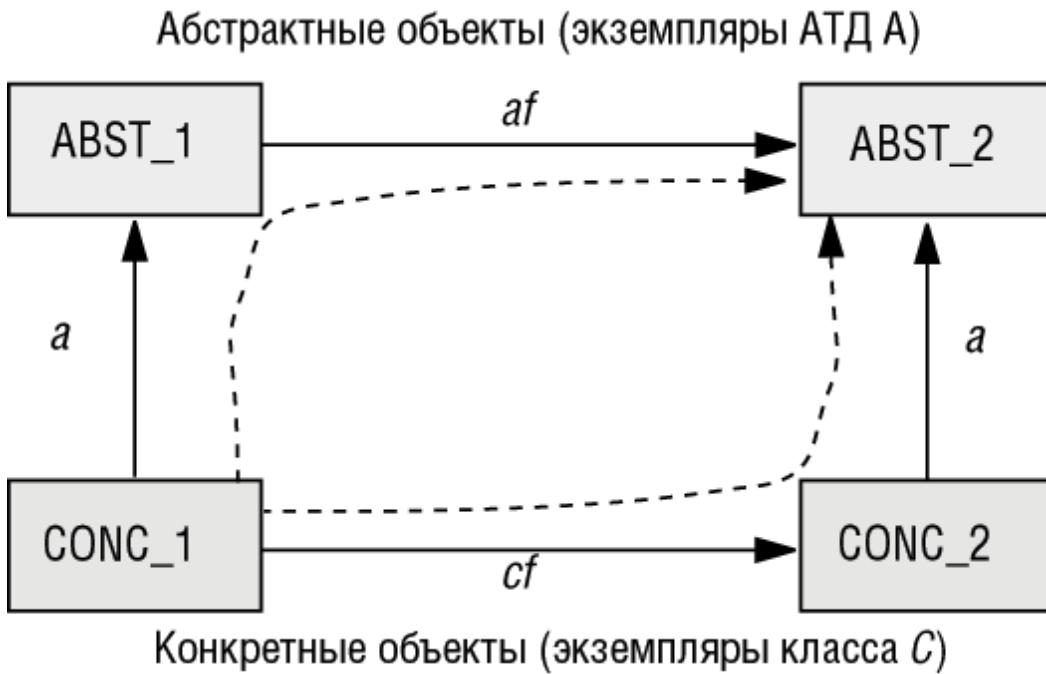


Рис. 11.5. Преобразования между абстрактными и конкретными объектами

Здесь А - АТД, С - класс, его реализующий. Абстрактной функции *af* из спецификации АТД соответствует в классе конкретная функция *cf*. Для простоты, полагаем, что абстрактная функция *af* из А возвращает результат того же типа А.

Стрелка, помеченная *a*, представляет функцию абстракции (**abstraction function**), которая для любого экземпляра класса - конкретного объекта - возвращает соответствующий абстрактный объект (экземпляр АТД). Как будет видно, функция обычно бывает частичной, а обратное отношение обычно не является функцией.

Реализация корректна, если (для всех функций *af* из АТД и их реализаций *cf*) диаграмма коммутативна, или, как говорят, имеет место:

Свойство согласованности Класс-АТД

$$(cf; a) = (a; af)$$

где символ ";" обозначает композицию функций. Другими словами, для любых двух функций *f* и *g*, их композиция: *f ; g* задает функцию *h*, такую что $h(x) = g(f(x))$ для каждого применимого *x*.

(Композицию *f ; g* также записывают в виде: $g \circ f$, с обратным порядком применения операндов.)

Свойство устанавливает, что для каждого конкретного объекта CONC_1 не имеет значения, в каком порядке применяются преобразования (функция абстракции, а затем *af* или вначале *cf*, а потом функция абстракции); оба пути, помеченные на рисунке штрихованными линиями, ведут к одному и тому же значению - абстрактному объекту ABST_2. Результат будет одним и тем же, если:

- Применить конкретную функцию класса *cf*, а потом функцию абстракции *a*, получив $a(cf(CONC_1))$.
- Применить функцию абстракции *a*, а потом функцию АТД - *af*, получив $af(a(CONC_1))$.

Инварианты реализации

Некоторые утверждения появляются в реализации, хотя они не имеют прямых двойников в спецификации АТД. Эти утверждения используют атрибуты, включая некоторые закрытые атрибуты, которые, по определению, не имеют смысла в АТД. Простым примером являются свойства, появляющиеся в инварианте STACK4:

```
count_non_negative: 0 <= count
count_bounded: count <= capacity
```

Такие утверждения составляют часть инварианта класса, известную как **инвариант реализации (implementation invariant)**. Они позволяют задать соответствие представления реализации, выбранное в классе, (здесь это атрибуты count, capacity, representation) - визави соответствующего АТД.

[Рис. 11.5](#) помогает понять концепцию инварианта реализации. Он иллюстрирует характеристические свойства функции абстракции, представленной вертикальной стрелкой на рисунке. Об этом стоит поговорить подробнее.

Прежде всего, корректно ли рассматривать a , как функцию? Напомним, что функция (тотальная или частичная) отображает каждый элемент исходного множества ровно в один элемент целевого множества, в противоположность общему случаю отношения, не имеющего такого ограничения. Рассмотрим обратное преобразование (сверху - вниз) от абстрактного объекта к конкретному. Будем называть его **отношением представления (representation relation)**; как правило, это отношение не является функцией, так как существует множество представлений одного и того же абстрактного объекта. В реализации стека массивом, где каждый стек задан парой $\langle \text{representation}, \text{count} \rangle$, абстрактный стек имеет много различных представлений, иллюстрируемых следующим [рис. 11.6](#). Все они имеют одно и то же значение count и одинаковые элементы массива representation для всех индексов в пределах от 1 до count , но размер массивов - capacity - может быть любым значением, большим или равным count ; элементы массива с индексом, большим count могут содержать произвольные значения.

Так как интерфейс класса ограничен компонентами, непосредственно выводимыми из функций АТД, клиенты не имеют способа различать поведение конкретных объектов, представляющих один и тот же абстрактный стек (это и есть причина, по которой все они имеют одну функцию абстракции a). Заметьте, в частности, что процедура `remove` из STACK4 выполняет свою работу, просто изменяя count

```
count := count - 1
```

не пытаясь очистить выше расположенные элементы. Всякое изменение элементов, расположенных выше count , будет модифицировать конкретный стек CS , не оказывая никакого влияния на ассоциированный абстрактный стек $a(CS)$.

Итак, отношение реализации это обычно не функция. Но инверсия этого отношения - функция абстракции - действительно является функцией, так как каждому конкретному объекту ставится в соответствие один абстрактный объект. В примере стека каждой правильной паре $\langle \text{representation}, \text{count} \rangle$ соответствует в точности один абстрактный стек. У него count элементов, растет снизу вверх, элементы representation имеют индексы в пределах от 1 до count .

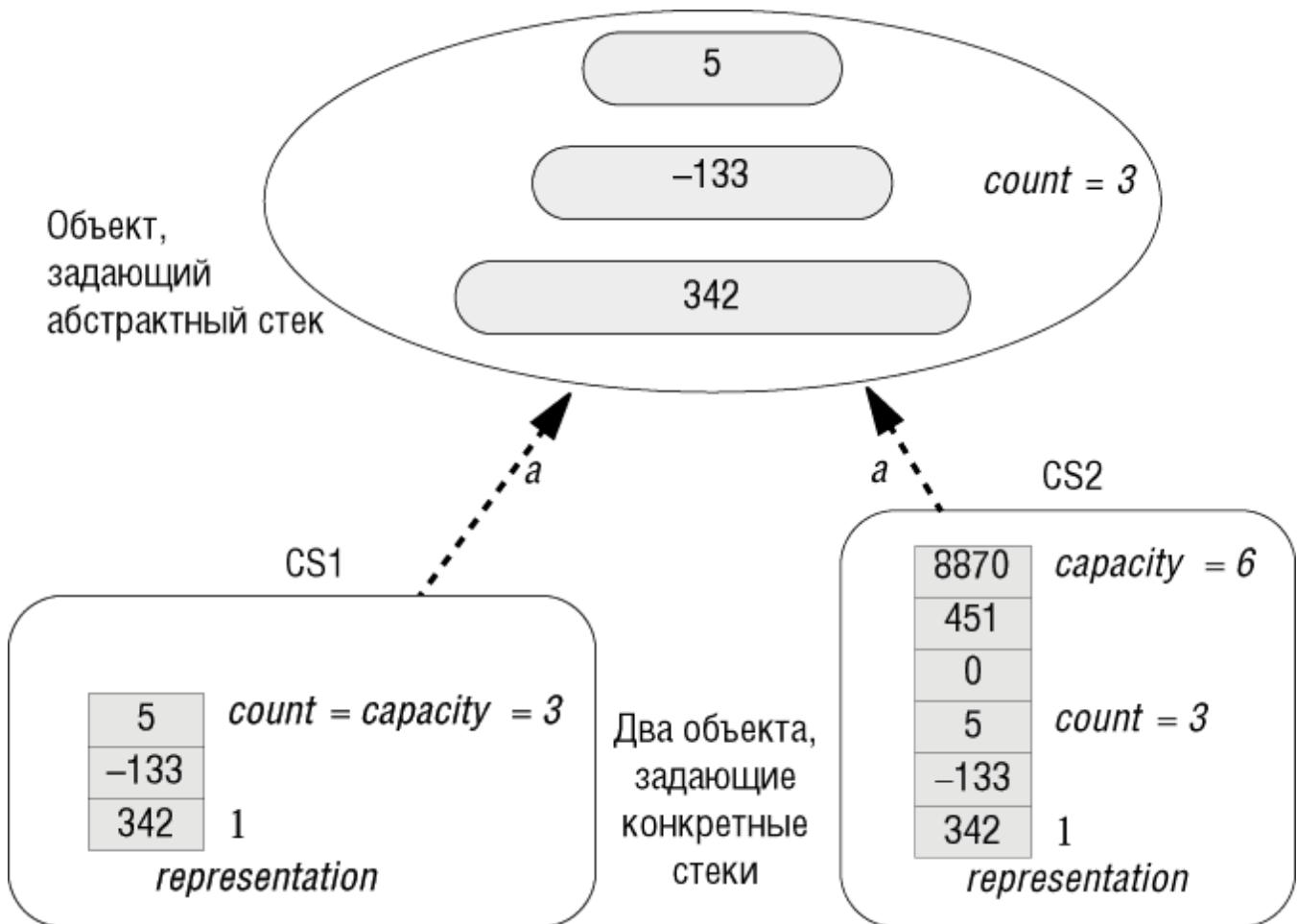


Рис. 11.6. Один абстрактный объект и два его представления

Оба конкретных стека, изображенные на рисунке, являются реализациями абстрактного стека, состоящего из трех элементов со значениями: 342, -133, 5. Отображение *a* должно быть функцией, иначе конкретный объект мог быть интерпретирован как реализация двух или более различных абстракций. В этом случае выбранная реализация двусмысленна и, следовательно, неадекватна. Поэтому стрелка, ассоциированная с *a*, правильно отображает существующую функциональную зависимость между абстрактными и конкретными типами. (Обсуждение наследования будет делаться при тех же предположениях).

Функция абстракции *a* обычно представима частичной функцией: не для каждого возможного конкретного объекта существует правильное представление абстрактного объекта. Например, не каждая пара $\langle \text{representation}, \text{count} \rangle$ является правильным представлением абстрактного стека. Если *representation* является массивом емкости 3 и *count* = 4, то они совместно не представляют абстрактный стек. Правильные представления (члены, входящие в область определения функции абстракции), - только те пары, для которых *count* находится между 0 и размерностью массива. Это свойство является инвариантом реализации.

В математических терминах, инвариант реализации является характеристической функцией области определения абстрактной функции. Другими словами, это булево свойство, определяющее применимость функции. (Характеристическая функция подмножества А задает булево свойство, истинное на А и ложное всюду вне его.)

Инвариант реализации является той частью утверждений класса, у которой нет двойника в спецификации АТД. Он не связан с АТД, и относится только к реализации. Он определяет, при каких условиях кандидат - конкретный объект - действительно является реализацией одного и только одного абстрактного объекта.

Инструкция утверждения

Утверждения, рассматриваемые до сих пор - предусловия, постусловия, инварианты, - это основные составляющие метода. Они устанавливают связь между конструкциями ОО-программных систем и теорией АТД, лежащей в основе метода. Инварианты класса, в частности, не могут быть поняты, и даже обсуждаться вне рамок ОО-подхода.

Можно рассматривать и другие возможности использования утверждений. Хотя они менее специфичны для нашего метода, но тоже играет важную роль, и должны быть частью нашей нотации. Наши расширения будут включать инструкцию проверки **check**, а также конструкции, задающие корректность цикла (инварианты и варианты цикла), рассматриваемые в следующем разделе.

Инструкция **check** выражает уверенность автора программы, что некоторое свойство всегда выполняется, когда вычисление достигает точки, в которой находится наша инструкция. Синтаксически, инструкция записывается в

следующей форме:

```
check
    assertion_clause1
    assertion_clause2
    ...
    assertion_clausen
end
```

Включив эту инструкцию в программный текст, мы говорим, что всякий раз, когда управление достигает этой инструкции, заданное утверждение (предложения утверждения между **check** и **end**) должно выполняться.

Это некоторый способ убеждать самого себя, что некоторые свойства выполняются. Более важно, что это позволяет будущим читателям вашего программного текста понять, на каких гипотезах вы основываетесь. Создание ПО требует многочисленных предположений о свойствах объектов системы. Тривиальный, но типичный пример - вызов `sqrt(x)` предполагает $x \geq 0$. Это предположение может быть очевидным из контекста, например, если вызов является частью условного оператора в форме:

```
if x >= 0 then y := sqrt (x) end
```

Но проверка может быть чуть менее очевидной, если, например:

```
x := a^2 + b^2
```

Инструкция **check** дает возможность выразить наше предположение о свойствах объектов:

```
x := a^2 + b^2
...
Другие инструкции ...
check
    x >= 0
        -- Поскольку x был вычислен как сумма квадратов.
end
y := sqrt (x)
```

Здесь нет конструкции **if... then...**, защищающей вызов `sqrt`; но **check** показывает, что вызов корректен. Хорошей практикой является сопровождать инструкцию комментарием с обоснованием утверждения, как это сделано в примере. Отступы при записи инструкции это тоже часть рекомендованного стиля; они подчеркивают, что при нормальных обстоятельствах инструкция проверки никак не влияет на ход алгоритмического процесса вычислений.

Этот пример типичен для демонстрации того, что наиболее вероятное применение инструкции проверки состоит в добавлении ее, как раз перед вызовом программы, имеющей предусловие. В качестве еще одного примера рассмотрим вызов

```
s.remove
```

в точке, где вы точно знаете, что стек `s` не пуст, поскольку до этого в стек засыпалось `n` элементов, а удалялось `m`, и вам известно, что $n > m$. В этом случае нет необходимости защищать вызов: **if (not s.empty) then ...;** но, если причина корректности вызова непосредственно следует из контекста, то есть смысл напомнить читателю, что "беззащитный" вызов является осознанным решением, а не недосмотром. Этого можно достичь, добавляя проверку:

```
check not s.empty end
```

Вариант такой ситуации встречается, когда пишется вызов в форме `x.f` в полной уверенности, что `x /= Void`, так что нет необходимости заключать этот вызов в оператор **if (x /= Void) then ...;** но, тем не менее, существование `x` не очевидно из контекста. Вернемся к рассмотрению процедур `put` и `remove` нашего "защищенного" класса `STACK3`. Вот текст тела процедуры `put`:

```
if full then
    error := Overflow
else
    check representation /= Void end
    representation.put (x); error := 0
end
```

Здесь читатель может думать, что вызов в `else` ветви: `representation.put(x)` потенциально не безопасен, поскольку ему не предшествует тест: (`representation /= Void`). Но, исследуя текст класса, можно понять, что из условия (`full = false`) следует положительность `capacity`, откуда, в свою очередь, следует, что `representation` не может быть `Void`. Это важное и не совсем тривиальное свойство, которое должно быть частью инварианта реализации класса. Фактически, с полностью установленным инвариантом реализации следует переписать инструкцию проверки следующим образом:

```
check
    representation_exists: representation /= Void
        -- Поскольку предложение representation_exists истинно, когда
        -- full можно, что следует из инварианта реализации.
end
```

В обычных подходах к конструированию ПО, хотя вызовы и другие операции часто основываются на корректности различных предположений, последние, чаще всего, остаются неявными. Разработчик уверяет себя, что некоторое свойство всегда имеет место в некоторой точке, использует этот анализ при написании кода, но после всего этого, не фиксирует этого в тексте программы, в результате смысл работы потерян. Когда некто, возможно, сам автор, несколькими месяцами позже, захочет разобраться в программе, возможно, с целью ее модификации, ему придется начинать работу с нуля, поскольку все предположения остались в сознании автора. Инструкция `check` помогает избежать подобных проблем, требуя документирования нетривиальных предположений.

Механизмы утверждений, рассмотренные в этой лекции, помимо того, что они дают преимущества все вещи делать правильно с самого начала, они еще позволяют найти то, что сделано неверно. Используя параметры компиляции можно включить проверку, и сделать инструкцию `check` реально проверяемой в период выполнения. Если все предположения выполняются, то проверка не оказывает воздействия на процесс вычислений, но, если вы ошиблись, и предположения не выполняются, то в точке их нарушения будет выброшено исключение и процесс остановится. Тем самым появляется возможность быстрого обнаружения содержательных ошибок. Механизм `checking` - включения проверок будет вкратце рассмотрен в дальнейшем.

Инварианты и варианты цикла

Наши следующие и последние конструкции утверждений помогут строить корректные циклы. Эти конструкции являются прекрасным дополнением рассмотренных ранее механизмов. Поскольку они не являются специфической частью ОО-метода, то вы вправе пропустить этот раздел при первом чтении.

Трудности циклов

Возможность повторять некоторые вычисления произвольное число раз, не поддаваясь усталости, без случайных потерь чего-либо важного, - в этом принципиальное отличие компьютерных вычислений от возможностей человека. Вот почему циклы так важны. Трудно вообразить, что можно было бы делать в языках, в которых были бы только две управляющие структуры - последовательность и выбор, - но не было бы циклов и не было бы поддержки рекурсии, еще одного базисного механизма поддержки итеративных вычислений.

Но с мощностью приходят и риски. У циклов дурная слава, - их трудно заставить работать правильно. Типичными для циклов являются:

- Ошибки "больше-меньше" (выполнение цикла слишком много или слишком мало раз).
- Ошибки управления пограничными ситуациями, например пустыми структурами. Цикл может правильно работать на больших массивах, но давать ошибки, когда у массива один элемент или он вообще пуст.
- Ошибки завершения ("зацикливание") в некоторых ситуациях.

Бинарный поиск - один из ключевых элементов базового курса "Введение в информатику" (Computer Science 101) - хорошая иллюстрация "коварства" циклов даже в относительно тривиальной ситуации. Рассмотрим целочисленный, упорядоченный по возрастанию массив `t` с индексами от 1 до `n`. Используем алгоритм бинарного поиска для ответа на вопрос: появляется ли целое `x` среди элементов массива. Если массив пуст, ответ должен быть "нет", если в массиве ровно один элемент, то ответ "да" тогда и только тогда, когда элемент массива совпадает с `x`. Суть бинарного поиска, использующего упорядоченность массива, проста: вначале `x` сравнивается со средним элементом массива, если есть совпадение, то задача решена, если `x` меньше среднего элемента, то поиск продолжается в верхней половине массива, в противном случае - в нижней половине. Каждое сравнение уменьшает размер массива вдвое. Ниже представлены четыре попытки реализации этой простой идеи. К несчастью, все они содержат ошибки. Вам предоставляется случай поупражняться в поиске ошибок и установить, в какой ситуации каждый из алгоритмов не работает нужным образом.

Напомню, `t @ m` означает элемент массива `t` с индексом `m`. Знак операции `//` означает деление нацело, так что `7 // 2` и `6 // 2` дают значение 3. Синтаксис цикла будет дан ниже, но он должен быть и так понятен.

Предложение `from` вводит инициализацию цикла.

Таблица 11.3. Четыре (ошибочных) попытки реализации бинарного поиска

BS1	BS2
<pre> from i := 1; j := n until i = j loop m := (i + j) // 2 if t @ m <= x then i := m else j := m end end Result := (x = t @ i) </pre>	<pre> from i := 1; j := n; found := false until i = j and not found loop m := (i + j) // 2 if t @ m < x then i := m + 1 elseif t @ m = x then found := true else j := m - 1 end end Result := found </pre>
BS3	BS4
<pre> from i := 0; j := n until i = j loop m := (i + j + 1) // 2 if t @ m <= x then i := m + 1 else j := m end end if i >= 1 and i <= n then Result := (x = t @ i) else Result := false end </pre>	<pre> from i := 0; j := n + 1 until i = j loop m := (i + j) // 2 if t @ m <= x then i := m + 1 else j := m end if i >= 1 and i <= n then Result := (x = t @ i) else Result := false end </pre>

Сделаем циклы корректными

Разумное использование утверждений может помочь справиться с такими проблемами. Цикл может иметь связанное с ним утверждение, так называемый инвариант цикла (**loop invariant**), который не следует путать с инвариантом класса. Он может также иметь вариант цикла (**loop variant**), являющийся не утверждением, а, обычно целочисленным выражением. Совместно, инвариант и вариант позволяют гарантировать корректность цикла.

Для понимания этих понятий необходимо осознать, что цикл - это способ вычислить некоторый результат **последовательными приближениями (successive approximations)**.

Рассмотрим тривиальный пример вычисления максимума в целочисленном массиве, используя очевидный алгоритм:

```

maxarray (t: ARRAY [INTEGER]): INTEGER is
  -- Максимальное значение массива t
  require
    t.capacity >= 1
  local
    i: INTEGER
  do
    from
      i := t.lower
      Result := t @ lower
    until i = t.upper loop
      i := i + 1
      Result := Result.max (t @ i)
    end
  end

```

В разделе инициализации *i* получает значение нижней границы массива, а сущность *Result* - будущий результат вычислений - значение первого элемента. Предусловие гарантирует существование хотя бы одного элемента в массиве. Производя последовательные итерации в цикле, мы достигаем верхней границы массива,

увеличивая на каждом шаге i на 1, и заменяя `Result` значением элемента $t @ i$, если этот элемент больше чем `Result`. Для нахождения максимума двух целых используется функция `max`, определенная для класса `integer`: `a.max(b)` возвращает максимальное значение из `a` и `b`.

Это пример вычисления последовательными приближениями. Мы продвигаемся вверх по массиву последовательными нарезками: $[lower, lower]$, $[lower, lower+1]$, $[lower, lower+2]$ и так вплоть до полного приближения $[lower, upper]$.

Свойство инварианта цикла состоит в том, что на каждом шаге прохождения цикла `Result` представляет максимум текущей нарезки массива. Инициализация гарантирует выполнимость этого свойства непосредственно перед началом работы цикла. Каждая итерация увеличивает нарезку, сохраняя истинность инварианта. Цикл завершает свою работу, когда очередная нарезка массива совпадает со всем массивом. В этом состоянии истинность инварианта означает, что `Result` является максимумом массива, что и является требуемым результатом работы.

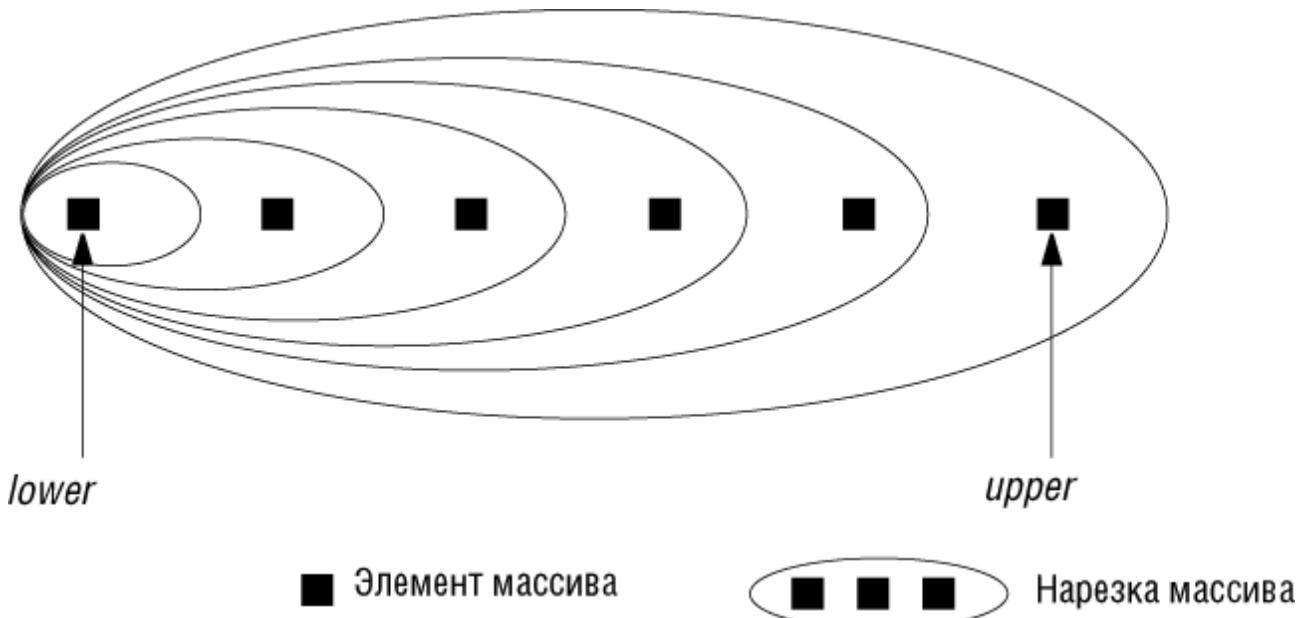


Рис. 11.7. Аппроксимация массива последовательными нарезками

Ингредиенты доказательства корректности цикла

Простой пример вычисления максимума массива иллюстрирует общую схему циклических вычислений, применимую ко многим ситуациям. Вы определяете, что решением некоторой проблемы является элемент, принадлежащий n -мерной поверхности `POST`. В некоторых случаях `POST` может содержать ровно один элемент - решение, но обычно может быть более чем одно приемлемое решение проблемы. Циклы полезны, когда нет прямого способа достичь решения "одним выстрелом". Но у вас есть непрямая стратегия, вы можете, например, прицелиться и попасть в m -мерную поверхность `INV`, включающую `POST` (для $m > n$). Инвариантом является то, что поверхность попадания все время содержит `POST`. Итерация за итерацией приближаемся к `POST`, сохраняя истинность `INV`. Следующий рисунок иллюстрирует этот процесс:

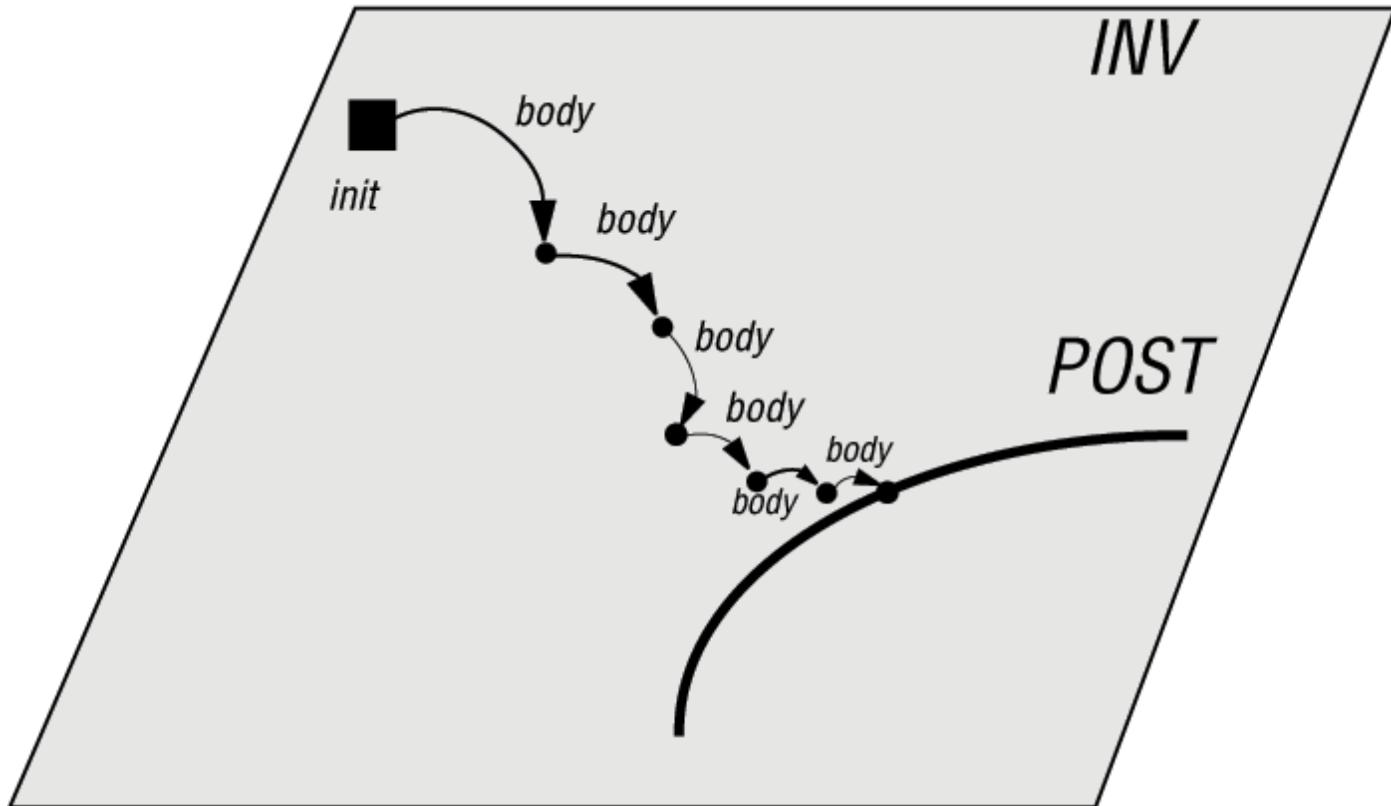


Рис. 11.8. Вычисление цикла (из [М 1990])

Вычисление цикла имеет следующие ингредиенты:

- Цель post, определяемую как свойство, выполняемое в любом допустимом заключительном состоянии. Пример: "Result является максимумом массива". На рисунке цель post представлена множеством состояний POST.
- Инвариант цикла inv, являющийся обобщением цели, так что можно говорить, что цель - это частный случай инварианта. Пример: "Result является максимумом текущей нарезки массива". Инвариант цикла поиска цели, изображенный на рисунке: "Каждая точка лежит на поверхности, содержащей POST".
- Точку инициализации init, о которой известно, что она должна быть в INV, другими словами должна обеспечить выполнение инварианта.
- Преобразование body, начинающееся в INV, но не в POST, вырабатывающее точку более близкую к POST, но все еще остающуюся в INV. Тело цикла функции maxarray является примером подобного преобразования.
- Верхняя граница числа применений body, необходимого для перевода точки из INV в POST. Как будет пояснено ниже, этот параметр необходим для определения варианта.

Последовательные приближения один из главных инструментов численного анализа. Но там эта идея понимается шире. Важная разница состоит в том, что в чистой математике допускаются бесконечные вычисления, последовательность может иметь предел, даже если он не достигается конечным числом приближений. Последовательность $1/n$ имеет предел 0, хотя среди членов последовательности нет числа 0. В компьютерных вычислениях мы хотим видеть результаты на нашем экране еще при нашей жизни, так что мы настаиваем, все аппроксимирующие последовательности достигают своей цели после конечного числа итераций.

Компьютерные реализации численных алгоритмов также требуют конечной сходимости. Даже когда математический алгоритм сходится на бесконечности, мы обрываем процесс сходимости, когда полагаем, что решение найдено с требуемой точностью.

Практический способ гарантии завершения циклического процесса состоит в связывании с итерационным процессом целочисленной величины - варианта цикла, обладающего следующими свойствами:

- Вариант всегда не отрицателен.
- Любое выполнение тела цикла уменьшает вариант.

Так как целочисленная неотрицательная величина не может уменьшаться бесконечно, то наличие варианта позволяет гарантировать завершение цикла. Вариант является верхней границей, максимальным числом применений body, приводящим точку в POST. В задаче нахождения максимума найти вариант просто: $t \cdot \text{upper} - i$. Это выражение удовлетворяет обоим условиям:

- Предусловие программы требует положительности $t \cdot \text{capacity}$; другими словами, программа применима только к непустым массивам. Инвариант класса ARRAY задает: capacity = upper - lower + 1.

Отсюда следует, что свойство $i \leq t.\text{upper}$ будет выполняться после инициализации i значением $t.lower$.

- Любое выполнение тела цикла выполняет инструкцию $i := i + 1$, уменьшая вариант на единицу.

В этом примере цикл является простым итерированием на последовательности целых чисел в конечном интервале, известный в языках программирования, как "цикл For" или "цикл DO", завершение которого не трудно проверить. Для более изощренных циклов число требуемых итераций определить не так просто, выявление завершения становится сложной задачей, единственным универсальным способом является нахождение варианта.

Нам понадобится еще одно понятие, преобразующее только что наброшенную схему в программный текст, описывающий цикл. Мы нуждаемся в простом способе определения того, что текущая итерация достигла цели (постусловия) post . Поскольку итерация конструируется так, чтобы обеспечить выполнение INV , а POST является частью INV , то обычно можно найти условие exit такое, что элемент из INV принадлежит POST тогда и только тогда, когда выполняется exit . Другими словами, постусловие post и инвариант inv связаны соотношением:

```
post = inv and exit
```

так что мы можем остановить цикл, - чьи промежуточные состояния по построению удовлетворяют inv , - как только выполнится exit . В этом состоянии, следовательно, будет выполнено и post .

Синтаксис цикла

Синтаксис цикла непосредственно следует из предшествующих соображений, определяющих ингредиенты цикла. Он будет включать элементы, отмеченные как необходимые.

- Инвариант цикла inv - утверждение.
- Условие выхода exit , чья конъюнкция с inv дает желаемую цель.
- Вариант var - целочисленное выражение.
- Множество инструкций инициализации, которые всегда приводят к состоянию, в котором inv выполняется, а var становится неотрицательным.
- Множество инструкций body , которое (при условии, что оно начинается в состоянии, где var неотрицательно и выполняется inv), сохраняет инвариант и уменьшает var , в то же время следя за тем, чтобы оно не стало меньше нуля.
- Синтаксис цикла честно комбинирует эти ингредиенты:

```
from
    init
invariant
    inv
variant
    var
until
    exit
loop
    body
end
```

Предложения **invariant** и **variant** являются возможными. Предложение **from** по синтаксису требуется, но инструкция **init** может быть пустой.

Эффект выполнения цикла можно описать так: вначале выполняется **init**, затем 0 или более раз выполняется тело цикла, которое перестает выполняться, как только **exit** принимает значение **false**.

В языках Pascal, C и других такой цикл называется "циклом **while**", в отличие от цикла типа "**repeat ... until**", в котором тело цикла выполняется, по меньшей мере, один раз. Здесь же тест является условием выхода, а не условием продолжения, и синтаксис цикла явно содержит фазу инициализации. Потому эквивалент записи нашего цикла на языке Pascal выглядит следующим образом:

```
init;
while not exit do body
```

С вариантами и инвариантами цикл для **maxarray** выглядит так:

```
from
    i := t.lower; Result := t @ lower
invariant
```

```

-- Result является максимумом нарезки массива t в интервале [t.lower, i].
variant
    t.lower - i
until
    i = t.upper
loop
    i := i + 1
    Result := Result.max (t @ i)
End

```

Заметьте, инвариант цикла выражен неформально, в виде комментария. Последующее обсуждение в этой лекции объяснит это ограничение языка утверждений.

Вот еще один пример, ранее показанный без вариантов и инвариантов. Целью следующей функции является вычисление наибольшего общего делителя - НОД (**gcd** - **greatest common divisor**) двух положительных целых a и b, следуя алгоритму Эвклида:

```

gcd (a, b: INTEGER): INTEGER is
    -- НОД a и b
    require
        a > 0; b > 0
    local
        x, y: INTEGER
    do
        from
            x := a; y := b
        until
            x = y
        loop
            if x > y then x := x - y else y := y - x end
        end
        Result := x
    ensure
        -- Result является НОД a и b
    end

```

Как узнать, что функция gcd удовлетворяет своему постусловию и действительно вычисляет наибольший общий делитель a и b? Для проверки этого следует заметить, что следующее свойство истинно после инициализации цикла и сохраняется на каждой итерации:

```

x > 0; y > 0
-- Пара <x, y> имеет тот же НОД, что и пара <a, b>

```

Это и будет служить инвариантом цикла **inv**. Ясно, что **inv** выполняется после инициализации. Если выполняется **inv** и условие цикла $x \neq y$, то после выполнения тела цикла:

```
if x > y then x := x - y else y := y - x end
```

инвариант **inv** остается истинным, замена большего из двух положительных неравных чисел их разностью не меняет их **gcd** и оставляет их положительными. Тогда по завершению цикла следует:

```
x = y and «Пара <x, y> имеет тот же НОД, что и пара <a, b>»
```

Отсюда, в свою очередь, следует, что x является наибольшим общим делителем. По определению НОД (x, x) = x.

Как узнать, что цикл всегда завершается? Необходим вариант. Если x больше чем y, то в теле цикла x заменяется разностью x - y. Если y больше x, то y заменяется разностью y - x. Нельзя в качестве варианта выбрать ни x, ни y, поскольку для каждого из них нет гарантии уменьшения. Но можно быть уверен, что максимальное из них обязательно будет уменьшено. Поэтому разумно выбрать в качестве варианта $x . max(y)$. Заметьте, вариант всегда остается положительным. Теперь можно написать цикл со всеми предложениями:

```

from
    x := a; y := b
invariant

```

```

x > 0; y > 0
-- Пара <x, y> имеет тот же Нод, что и пара <a, b>
variant
    x.max (y)
until
    x = y
loop
    if x > y then x := x - y else y := y - x end
end

```

Как отмечалось, предложения **invariant** и **variant** являются возможными. Когда они присутствуют, то помогают прояснить цель цикла и проверить его корректность. Для любого не тривиального цикла характерны интересные варианты и инварианты; многие из примеров в последующих лекциях включают варианты и инварианты, обеспечивая глубокое понимание корректности лежащих в основе алгоритмов.

Использование утверждений

Теперь мы уже познакомились со всеми конструкциями, содержащими утверждения. Разумно, еще раз взглянуть на те преимущества, которые мы можем получить от этого. Выделим четыре основных применения.

- Помощь в создании корректного ПО.
- Поддержка документирования.
- Поддержка тестирования, отладки и гарантия качества.
- Поддержка приемлемого способа обработки неисправностей.

Только два последних пункта предполагают мониторинг утверждений в период выполнения.

Утверждения как средство для написания корректного ПО

Первое использование является чисто методологическим и, вероятно, самым важным. В деталях оно рассматривалось в предыдущих разделах: точные требования к каждой программе, глобальные свойства классов и циклов - все это помогает разработчикам производить программный продукт, корректный с самого начала в противоположность подходу, пытающемуся добиться корректности в процессе отладки. Преимущества точной спецификации и систематического подхода к конструированию программ не могут быть преувеличены. Во всей этой книге всякий раз при встрече с программным элементом его формальные свойства выражались точно, насколько это было возможным.

Ключевая идея этой лекции - Проектирование по контракту. Использование компонент некоторого модуля является контрактом с его службами. Хорошие контракты точно специфицируют и ограничивают права и обязанности каждого участника. В проектировании ПО, где корректность и устойчивость так важны, необходимо раскрытие терминов контракта, как предварительное условие их следование. Утверждения дают способ точно установить, что ожидается и что гарантируется каждой стороне в этом соглашении.

Использование утверждений для документирования: краткая форма класса

Второе использование является основным в производстве повторно используемых программных элементов и, более обще, в организации интерфейсов модулей в большой программной системе. Постусловия, предусловия, инварианты классов обеспечивают потенциальных клиентов модуля необходимой информацией о предлагаемых модулем службах, выраженной в соответствующей и точной форме. Никакое количество описательной документации не может заменить множества аккуратно выраженных утверждений, являющихся частью самого ПО.

В самом последнем разделе этой лекции можно ознакомиться с проектом, где эти правила были проигнорированы, что обошлось в \$500 миллионов и привело к провалу космического проекта.

Средство автоматической документации **short** использует утверждения, как важную компоненту при извлечении из класса информации, значимой для потенциальных клиентов. Краткая форма класса - его описание на более высоком уровне. Она включает только ту информацию, которая полезна авторам клиентских классов, ничего не показывая из скрытых компонент, и не раскрывая реализации открытых. Но краткая форма сохраняет утверждения, составляющие основу документации, устанавливая контракты, которые класс предлагает своим клиентам.

Вот пример краткой формы класса STACK4:

```

indexing
description: "Стеки: Структуры с политикой доступа Last-In, First-Out %
%Первый пришел - Последний ушел, с фиксированной емкостью"
class interface STACK4 [G] creation
make

```

```

feature -- Initialization (Инициализация)
make (n: INTEGER) is
    -- Создать стек, содержащий максимум n элементов
    require
        non_negative_capacity: n >= 0
    ensure
        capacity_set: capacity = n
    end
feature -- Access (Доступ)
capacity: INTEGER
    -- Максимальное число элементов стека
count: INTEGER
    -- Число элементов стека
item: G is
    -- Элемент в вершине стека
    require
        not_empty: not empty -- i.e. count > 0
    end
feature -- Status report (Отчет о статусе)
empty: BOOLEAN is
    -- Пуст ли стек?
    ensure
        empty_definition: Result = (count = 0)
end
full: BOOLEAN is
    -- Заполнен ли стек?
    ensure
        full_definition: Result = (count = capacity)
    end
feature -- Element change (Изменение элементов)
put (x: G) is
    -- Втолкнуть x в вершину стека
    require
        not_full: not full
    ensure
        not_empty: not empty
        added_to_top: item = x
        one_more_item: count = old count + 1
    end
remove is
    -- Удалить элемент вершины стека
    require
        not_empty: not empty -- i.e. count > 0
    ensure
        not_full: not full
        one_fewer: count = old count - 1
    end
invariant
    count_non_negative: 0 <= count
    count_bounded: count <= capacity
    empty_if_no_elements: empty = (count = 0)
end

```

Эта краткая форма не является синтаксически правильным текстом класса, посему здесь используется термин **class interface** вместо обычного термина **class**. Хотя достаточно просто превратить эту форму в правильный отложенный класс, известное понятие, рассматриваемое в деталях при изучении наследования.

В среде ISE получить краткую форму можно одним щелчком соответствующей кнопки в Class Tool; можно генерировать либо плоский текст, либо текст в форматах HTML, RTF, MML (FrameMaker), TEX и других. Можно определить и свой собственный формат.

Если сравнить краткую форму утверждений с их оригиналами в классе, то можно заметить, что исчезли все предложения, включающие `representation`, так как этот атрибут не экспортируется.

Краткая форма документации особенно интересна по следующим причинам:

- Документация является более высокой формой абстракции, чем объект, который она описывает. Это основное требование, предъявляемое к качественной документации. Фактическая реализация, описывающая "как", удаляется. Утверждения, объясняющие "что", а в некоторых случаях и "почему",

остаются. Сохраняются заголовочные комментарии к программам и описания, включаемые в предложение **indexing**, дополняющие в менее формальной форме утверждения, поясняя цель и назначение программы.

- Являясь прямым следствием принципа Самодокументирования, изучаемого в нашем обзоре концепций модульности, краткая форма рассматривает документацию как информацию, содержащуюся в самом программном продукте. Это означает, что есть только один сопровождаемый продукт, - важное требование, проходящее через всю книгу. Как результат, появляется больше шансов корректности документации. Сохраняя все в одном месте, вы уменьшаете риск несоответствия документации обновленному продукту.
- Краткая форма может быть извлечена из класса автоматически. Так что документация не есть нечто, требующее специального написания. Вместо этого, когда она необходима, вы просто "просите компьютер" произвести это нечто, щелкнув кнопкой мыши.

Интересно сравнить этот подход с понятием интерфейса пакета в языке Ada, где модуль (пакет) состоит из двух частей - интерфейса и реализации. Java использует подобный механизм. Интерфейс пакета имеет некоторое сходство с краткой формой, но имеет и существенные различия:

- Здесь нет утверждений, так что вся спецификация сводится к объявлению типов и комментариям.
- Интерфейс не создается автоматически, а пишется независимо от реализации. Поэтому разработчик дважды должен задавать многие вещи: заголовки программ, их сигнатуры, комментарии к заголовкам, объявления открытых переменных. Эта связанный избыточность утомительна (вдвое при включении утверждений) и, как обычно, повышает риск несоответствия; всегда есть шанс, обновить одну часть и забыть про другую.

Краткая форма, дополненная ее вариантом - плоско-краткой формой (**flat-short form**), изучаемой при рассмотрении наследования, является принципиальным вкладом в ОО-метод. В повседневной практике ОО-разработки она появляется не только как средство документирования, но и как стандартный формат, в котором разработчики и менеджеры изучают существующие проекты, разрабатывают новые и обсуждают предложения по изменению проектов.

Краткая форма играет центральную роль в ОО-разработке, поскольку она удовлетворяет цели, определенной при анализе требований, обеспечивающих повторное использование. Суть требования: основой повторного использования являются абстрактные модули. Класс в его краткой или плоско-краткой форме является тем самым разыскиваемым абстрактным модулем.

Мониторинг утверждений в период выполнения

Пришло время, дать полный ответ на вопрос: "какой эффект производят утверждения в период выполнения?". Как отмечалось, ответ определяется разработчиком, имеющим возможность управлять параметрами компиляции. Выбор нужных параметров не требует изменения текста класса, вместо этого меняется содержимое Ace файла. Напомню, Ace файл написан на языке Lace, описывающем компиляцию и сборку системы.

Напомню также, что Lace один из возможных языков, позволяющих управлять сборкой системы; он не является неизменяемым компонентом метода. Но всегда необходимо подобное средство для перехода от отдельных компонент к полной компилируемой системе.

Вот пример применения Ace файла, устанавливающего некоторые параметры мониторинга утверждений:

```
system painting root
    GRAPHICS
default
    assertion (require)
cluster
    base_library: "\library\base"
    graphical_library: "\library\graphics"
        option
            assertion (all): BUTTON, color_BITMAP
        end
painting_application: "\user\application"
    option
        assertion (no)
    end
end -- system painting
```

Предложение **default** указывает, что для большинства классов системы проверяться будут только предусловия (**require**). Два кластера переопределяют установки умолчания. Кластер **graphical_library** будет наблюдать за всеми (**all**) утверждениями в классах **BUTTON** и **color_BITMAP**. Кластер **painting_application** вообще отменяет наблюдение за утверждениями во всех его классах. Этот пример иллюстрирует возможности мониторинга на разных уровнях - всей системы, отдельных кластеров, отдельных классов.

Следующие ключевые слова, управляющие проверкой утверждений, могут появиться в круглых скобках `assertion(...)`:

- **no** - не выполнять никакое из утверждений. В этом режиме оказывают на выполнение не больший эффект, чем комментарии;
 - **require** - только проверка выполнимости предусловий на входе программ;
 - **ensure** - проверка выполнимости постусловий на выходе из программы;
 - **invariant** - проверка выполнимости инвариантов класса на входе и выходе программы для квалифицированных вызовов (`obj.f`);
 - **loop** - проверка выполнимости инвариантов цикла перед и после каждой итерации; проверка уменьшения вариантов на каждой итерации с сохранением их не отрицательности;
 - **check** - выполнение предложений **check**, проверяющих выполнимость соответствующих утверждений.
- Ключевое слово **all** является синонимом **check**.

За исключением "**no**" каждый уровень автоматически влечет выполнение всех предыдущих уровней. В частности, не имеет смысла управлять постусловиями, если не проверить выполнимость предусловий. Этим объясняется эквивалентность **check** и **all**.

При включенном мониторинге пока утверждения выполняются никакого видимого эффекта на процесс вычислений они не оказывают, если не считать затрат процессорного времени. Но если одно из утверждений принимает значение `false`, то это довольно серьезное событие, приводящее обычно к завершению работы. Фактически, возбуждается исключение, и, если не принять специальных мер по захвату этого исключения, то выполнение остановится. При этом, однако, будет создана таблица истории исключения (**exception history table**) в ее общей форме:

```
Failure: object: 02 class: YOUR_CLASS routine: your_routine
    Cause: precondition violation, clause: not_too_small
Called by: object: 02 class: YOUR_CLASS routine: his_routine
Called by: object: 01 class: HER_CLASS routine: her_routine
...
...
```

Это дает нам цепочку вызовов, начинающуюся программой, вызвавшей исключение, с указанием всех объектов и их классов - клиентов, в конечном счете, вызвавших эту программу. Показанная здесь форма является только наброском; обсуждение исключений в следующей лекции даст более полный пример таблицы истории исключения.

Возможные метки, допускаемые в утверждениях, такие как `not_too_small` в

```
your_routine (x: INTEGER) is
    require
        not_too_small: x >= Minimum_value
    ...
...
```

перечисляются при трассировке исключения, что помогает идентифицировать, что же именно пошло не так.

Каков оптимальный уровень мониторинга?

Каков уровень трассировки следует включать? Ответ вырабатывается в результате компромисса, с учетом следующих факторов: уровня доверия к корректности ПО, насколько критичны потери эффективности, насколько серьезны последствия не обнаруженных ошибок в период выполнения.

В экстремальных ситуациях все ясно:

- При тестировании системы или очередной ее версии следует включать на самом высоком уровне мониторинг классов (для используемых библиотек это не обязательно). Эта возможность - один из принципиальных вкладов метода, представленного в этой книге. Мало кто из людей осознавал мощь этих идей, и как основательно они влияют на практику разработки ПО. Перелом наступил, когда фактически был получен опыт тестирования больших систем с утверждениями, включающими механизм мониторинга, описанный в этом разделе.
- Для системы с полной степенью доверия в приложениях, критичных по времени выполнения, где каждая микросекунда на счету, - следует полностью удалять мониторинг.

Последний совет парадоксален, при отсутствии формальных доказательств корректности говорить о "полной степени доверия" вряд ли возможно. Стоит привести красноречивое высказывание С. А. Hoare:

Абсурдно выполнять проверку в период отладки, когда не требуется доверие к получаемым результатам, и отключать ее в рабочем состоянии, когда ошибочный результат может стоить дорого или вообще катастрофичен. Что бы вы подумали о любителе плавания, который надевает спас-жилет во время тренировок на берегу и снимает его, бросаясь в море [Hoare 1973].

Интересную возможность дает параметр, включающий проверку предусловий. В рабочем режиме, когда отладка завершена и даны гарантии качества, крайне важно избежать катастроф в результате необнаруженных вызовов программ вне области их применения. Эта проверка обходится намного дешевле, чем проверка постусловий и инвариантов. Инварианты, в частности, особенно дороги, поскольку они проверяются на входе и выходе каждого квалифицированного вызова, и, что более важно, они всегда сложны, поскольку включают условия согласованности компонент класса.

Проверка предусловий, это параметр, устанавливаемый по умолчанию в Ace файле. Его появление в примере не было необходимым.

Этот параметр особенно интересен для библиотек. Вспомните, о чем говорит основное правило нарушения утверждений. За ошибку выполнения предусловия отвечает клиент. Если вы используете, повторно используемые библиотеки, предположительно высокого качества, то обычно мониторинг их постусловий и инвариантов нежелателен. Хотя ошибки в библиотеках, конечно, возможны, но априорно ошибки в клиентском ПО более вероятны. Но даже для совершенных во всех отношениях библиотек следует включать проверку предусловий с единственной целью - найти ошибки клиентов.

Вероятно, наиболее очевидным примером является проверка границ массива. В классе ARRAY мы видели, что `put`, `item` и его синоним - инфиксный знак операции `@`, - все они имеют предусловие:

```
index_not_too_small: lower <= i  
index_not_too_large: i <= upper
```

Включение предусловий для класса решает хорошо известную проблему любого продукта, использующего массивы: возможность выхода индекса за границы массива, что приводит к попаданию в область памяти, отведенную другим данным или коду, и может иметь разрушительные последствия. Большинство компиляторов предлагают специальный параметр компиляции, позволяющий управлять доступом к массиву в период выполнения. Но в объектной технологии массивы рассматриваются с общих позиций класса и объектов, а не как специальные конструкции. Мониторинг границ становится доступным благодаря общему механизму проверки условий. Просто скомпилируйте класс ARRAY, включив `assertion` (`require`).

Следует ли всегда включать проверку границ? Вот что говорит по этому поводу Тони Хоар:

В нашем компиляторе каждое вхождение каждого индекса в каждый массив проверялось во всех случаях в период выполнения. Через много лет мы спросили наших клиентов, не стоит ли ввести в интересах эффективности параметр компиляции, позволяющий отключать эту проверку. Единогласно они убеждали нас, не делать этого, - они уже хорошо знали, как часто встречается эта ошибка и к каким ужасным последствиям она может приводить. Со страхом и ужасом я заметил, что даже сегодня проектировщики языков и пользователи не выучили этот урок. В любой уважающей себя ветви инженерии непринятие предосторожностей такого рода считались бы нарушением закона.

Этот комментарий применим не только к массивам, но и ко всем предусловиям в целом. Если действительно "ошибки задания индекса часто встречаются в работающих системах", то это должно быть истинно и для других нарушений предусловий.

Кто-то может занимать менее экстремальную позицию. Прежде всего, это компании, поставляющие ПО, в котором ошибки предусловий, "часто встречающиеся в работающей системе" связаны и с низким качеством самой системы, не решаемые мониторингом утверждений. Мониторинг фиксирует следствия - неисправности (**fault**), но не причины - ошибки и дефекты. Это правда, что мониторинг полезен конечным пользователям даже в системе низкого качества. Лучше часто получать сообщения об ошибках, чем получать неверные результаты. Есть один неприятный эффект, возникающий у разработчиков, поставляющих системы с некоторым уровнем мониторинга утверждений. У них может возникнуть, даже неосознанная, беззаботная позиция по отношению к корректности. Нестрашно, что есть ошибки в поставляемом ПО - пользователи их обнаружат в процессе мониторинга, и мы исправим их в очередной версии. Так не стоит ли остановить отладку прямо сейчас и начать поставку системы?

Трудно дать абсолютный ответ на вопрос "следует ли оставлять включенным некоторый уровень мониторинга?". Без знания потери производительности на мониторинг утверждений на него не ответить. Если добавление мониторинга увеличивает время работы системы в 10 раз, то немногие поддержат точку зрения Хоара, кроме тех, кто занимается критически важными приложениями, где за ошибки приходится дорого платить. Если потери производительности на мониторинг составляют два процента, то немногие решатся отключить мониторинг. На практике, конечно потери находятся где-то посередине.

Но, между прочим, каковы они? Ясно, что многое зависит от того, что делает ПО, и как много в нем утверждений, но можно сообщить некоторые эмпирические наблюдения. По опыту ISE стоимость мониторинга предусловий (параметр по умолчанию, включающий, конечно, и проверку границ массивов) составляет 50%. Что самое удивительное, - 75% этой стоимости не связано с проверкой предусловий, а идет на поддержку трассировки вызовов, чтобы при нарушении предусловия можно было точно сказать, кто нарушил и где. Это может быть названо Парадоксом Проверки Предусловия: проверка предусловия сама по себе недорого стоит, но, чтобы

получить ее, нужно заплатить за дополнительные услуги. Что касается постусловий и инвариантов, то штраф может достигать от 100% до 200%.

Кому-то может показаться, что привнесение производительности в это обсуждение, означает компромисс с корректностью, что нарушает основной принцип, высказанный еще в начале этой книги:

Как бы ни были необходимы компромиссы между факторами качества, один из факторов стоит в стороне от остальных - корректность. Нет никакого оправдания тому, что корректность подвергается опасности ради других факторов, таких как эффективность. Если программный продукт не выполняет свою функцию, все остальное не имеет смысла.

Рассмотрение производительности, когда мы решаем, оставить ли мониторинг или нет, не является нарушением этого принципа. Вопрос не в том, приносить ли корректность в жертву эффективности, - нужно решить, что делать с не корректной системой, при разработке которой мы, очевидно, не приложили достаточных усилий, чтобы сделать ее корректной.

В действительности, эффективность - часть корректности. Рассмотрим метеорологическую систему, требующую 12 часов работы для выработки прогноза на следующие сутки. Система тщательно оптимизирована, в частности исключены все проверки, в том числе выход индекса за границы и другие подобные неисправности. Она тщательно разрабатывалась и тестировалась. Теперь, предположим, что добавление проверок периода исполнения вдвое увеличит время ее работы. Будет ли включена проверка, - нет!

Давайте не остановимся на этом, а зададим действительно трудный вопрос. Предположим, что 12 часов уходит на работу системы с включенными проверками. Хотелось ли бы вам удалить их, чтобы получить прогноз за 6 часов, а не за 12, или тратить те же 12 часов, но перейти к более сложному алгоритму, дающему лучший прогноз? Я думаю, что если предлагается "возможность выключить проверки в интересах эффективности производственной системы", почти каждый ответит "да".

В конечном итоге, выбор уровня мониторинга в производственных системах не так прост, как предполагает Хоаровское правило. Следует соблюдать несколько точных и строгих правил.

- Помните, программная система должна быть сделана надежной до того, как она начнет свою производственную жизнь. Ключом является применение методов, обеспечивающих надежность, описанные в литературе по программной инженерии, включая методы данной лекции и всей этой книги.
- Если вы являетесь менеджером проекта, никогда не позволяйте своим разработчикам предполагать, что в производственной версии проверки будут включены. Заставьте каждого исходить из того, - все проверки могут быть выключены. Это особенно важно для больших систем, в природе которых устрашающие последствия ошибок.
- Убедитесь, что в процессе разработки системы проверка утверждений всегда включена, по меньшей мере, на уровне предусловий.
- Выполняйте интенсивное тестирование со всеми включенными проверками. Включайте также все проверки при каждом найденном жучке и устранении его последствий.
- Для стандартной производственной версии решите, выберите ли версию без проверок или защищенную версию. Напомню о трех факторах, рассмотренных в самом начале этого раздела, которые следует учитывать при принятии решения.
- Если вы решите выбрать версию без проверок в качестве стандарта, то включите в поставку и версию с проверками, по меньшей мере, предусловий. В случае, если система у пользователей начнет вести себя непредсказуемым способом, вопреки ожиданиям, вы сможете попросить пользователей перейти на защищенную версию, что поможет быстро отыскать неисправности системы.

Такой способ использования мониторинга утверждений обеспечивает замечательную помощь в быстрой прополке всех сорняков - ошибок, сумевших выстоять в процессе систематического конструирования программной системы.

Обсуждение

Механизм утверждений, представленный в этой лекции, привносит несколько тонких проблем, подлежащих исследованию.

Нужен ли мониторинг в период выполнения?

Действительно, нужно ли проверять утверждения в период выполнения? После того, как мы были в состоянии, используя утверждения, дать теоретическое определение корректности класса: каждая процедура создания должна гарантировать инвариант, и тело каждой процедуры, запущенной в состоянии, удовлетворяющим инвариант и предусловию, сохраняет в заключительном состоянии инвариант и гарантирует выполнение постусловия. Теперь мы должны выполнить математическую проверку $m+n$ соответствующих условий (для m процедур создания и n экспортруемых процедур), и тогда долой мониторинг в период выполнения.

Мы должны, но мы не можем. Доказательство правильности программ уже многие годы является активной областью исследований, и достигло определенных успехов. Все же сегодня невозможно проверить корректность

реального ПО, написанного на современных языках программирования.

Для этого необходим, в частности, и более мощный язык утверждений. Язык IFL, обсуждаемый ниже, может быть использован как часть стратегии многоярусного доказательства.

Даже, если со временем методы и инструментальные средства доказательства станут доступными, можно ожидать, что отказаться от мониторинга не удастся. В системе всегда останется место трудно предсказуемым событиям - ошибкам аппаратуры, ошибкам в самом доказательстве. Поэтому следует применять хорошо известную в инженерии технику - множественные, независимые способы проверки.

Выразительная сила утверждений

Как можно было заметить, применяемый язык утверждений является языком обычных булевых выражений, обогащенный некоторыми понятиями, такими как **old**. Как результат, он ограничен и не позволяет включить в наши классы некоторые свойства, достаточно просто выражаемые в математической нотации, используемой при описании АТД.

Утверждения класса стек дают хороший пример того, что выразимо, и что не выразимо в нашем языке. Мы найдем, что многие аксиомы и предусловия из спецификации АТД, приведенной в [лекции 6](#), прямым образом отображаются в утверждения класса. Например, аксиома

```
A4. not empty (put (s, x))
```

задает постусловие **not empty** процедуры `put`. Но в некоторых случаях в классе нет непосредственного двойника. Ни одно из постусловий для `remove`, приводимое до сих пор, не отражает аксиому

```
A2. remove (put (s, x)) = s
```

Мы, конечно, можем ввести эту аксиому неформально, добавив в постусловие комментарий, описывающий это свойство:

```
remove is
    -- Удалить элемент вершины
require
    not_empty: not empty -- i.e. count > 0
do
    count := count - 1
ensure
    not_full: not full
    one_fewer: count = old count - 1
    LIFO_policy: -- item является последним элементом, помещенным в стек
                  -- и еще не удален, если такое имело место.
End
```

Подобные неформальные утверждения, синтаксически выраженные комментариями, появлялись в инвариантах цикла для `maxarray` и `gcd`.

В таких случаях, два из принципиальных использований утверждений, обсуждаемых ранее, остаются применимыми, по крайней мере, частично: помочь в создании корректного продукта и его документации (утверждения, заданные комментариями будут появляться в краткой форме класса). Другие использования, в частности, отладка и тестирование предполагают вычисление выражений, и становятся теперь неприменимыми.

Было бы предпочтительнее, выражать все утверждения формально. Лучший способ достичь этой цели - расширить язык выражений, так чтобы он позволял задавать любые свойства. Это требует возможности описания сложных математических объектов - множеств, последовательностей, функций, отношений. Необходим и мощный по выразительности язык, например, язык логики предикатов первого порядка, допускающий выражения с кванторами всеобщности и существования. Существуют формальные языки спецификаций, обладающие, по крайней мере, частью такой выразительной силы. Наиболее известными являются языки Z, VDM, Larch, OBJ-2; как Z, так и VDM имеют ОО-расширения, например, Object-Z. Библиографические замечания к [лекции 6](#) дают необходимые ссылки.

Включение полного языка спецификаций в язык этой книги полностью изменило бы ее природу. Смысл языка в том, чтобы он был простым, легким в обучении, применимым во всех программистских конструкциях. Он должен допускать быструю компиляцию и эффективную реализацию с производительностью, соизмеримой с C или Fortran.

Вместо этого, в механизме утверждений мы пошли на инженерный компромисс: он включает достаточно формальных элементов, оказывающих существенный эффект на качество ПО, но останавливается в точке

убывания - границе, за которой выгоды от большей формализации, начинают оборачиваться потерями простоты и эффективности.

Определение границы во многом определяется личным выбором. Я был удивлен, для программистского сообщества в целом эта граница не изменилась со временем первого издания этой книги. Наша деятельность требует большего формализма, но профессиональное сообщество еще не осознало этого.

Так что пока и на ближайшее будущее утверждения остаются булевыми выражениями с некоторыми расширениями. Это не такое уж и строгое ограничение, поскольку булевые выражения допускают вызов функций.

Включение функций в утверждения

Булевые выражения не ограничиваются использованием атрибутов и локальных сущностей. Мы уже использовали возможность вызова функций в утверждениях: предусловие для `put` класса стек было `not full`, где `full` - функция

```
full: BOOLEAN is
    -- Is stack full? (Заполнен ли стек?)
    do
        Result := (count = capacity)
    ensure
        full_definition: Result = (count = capacity)
    end
```

В этом наш маленький секрет, - мы вышли из рамок исчисления высказываний, в котором булевые выражения могут строиться только из переменных, констант и знаков логических операций. Благодаря введению функций, мы получили мощный механизм, позволяющий вычислять булевые значения любым, подходящим для нас способом. Не следует беспокоиться о присутствии постусловия самой функции `full`, это не создает никакого пагубного зацикливания. Детали вскоре.

Использование функций ведет к получению более абстрактных утверждений. Например, кто-то предпочитет заменить предусловие в операциях над массивом, ранее выраженное как

```
index_not_too_small: lower <= i
index_not_too_large: i <= upper
```

одним предложением в форме

```
index_in_bounds: correct_index (i)
```

с определением функции

```
correct_index (i: INTEGER): BOOLEAN is
    -- Является ли i внутри границ массива?
    do
        Result := (i >= lower) and (i <= upper)
    ensure
        definition: Result = ((i >= lower) and (i <= upper))
    end
```

Еще одно преимущество использования функций в выражениях в том, что они дают способ обойти ограничения выразительной силы, возникающие из-за отсутствия механизмов логики предикатов первого порядка. Неформальный инвариант нашего цикла для `maxarray`

```
-- Result является максимумом нарезки массива t в интервале [t.lower,i]
```

формально может быть выражен так

```
Result = (t.slice (lower, i)).max
```

в предположении, что `slice` вырабатывает нарезку - массив с индексами от `lower` до `i`, - а функция `max` дает максимальный элемент этого массива.

Этот подход был исследован в [М 1995а] как способ расширения выразительной силы механизма утверждений, возможно ведущий к разработке полностью формального метода, - другими словами, к математическому

доказательству корректности ПО. В этом исследовании есть две центральные идеи. Первая - использование библиотек в процессе доказательства, так что можно его проводить для реальных, широкомасштабных систем, строя многоярусную структуру, использующую условные доказательства. Вторая идея - определение ограниченного языка чисто аппликативной природы - IFL (Intermediate Functional Language), в котором выражаются функции, используемые в выражениях. Язык IFL является подмножеством нотации этой книги, включающий некоторые императивные конструкции, такие как любые присваивания.

Ясно, чем мы рискуем: появление функций в выражениях означает введение потенциально императивных элементов (программ) в чисто аппликативный, до сего времени, мир утверждений. Без функций мы имели ясное и четкое разделение ролей, обсуждаемое ранее: инструкции предписывают, утверждения описывают. Теперь мы открыли ворота аппликативного города императивным полчищам.

Все же трудно сопротивляться мощи использования функций, поскольку все альтернативы имеют свои недостатки.

- Включение полного языка спецификаций, как отмечалось, приводит к потере эффективности и простоты изучения.
- Вероятно, хуже то, что неясно, достаточны ли общепринятые языки утверждений. Возьмем, например, такого естественного кандидата, в которого многие верят, - язык логики предикатов первого порядка. Этот формализм не позволяет нам выразить некоторые свойства, представляющие непосредственный интерес для разработчиков и часто используемые в утверждениях, такие как, например, "граф не имеет циклов" (типичный инвариант цикла). Математически это может быть выражено как $r^+ \cap r = \emptyset$, где r - это отношение на графике, a^+ его транзитивное замыкание. Хотя можно представить себе язык спецификации, поддерживающий эти понятия, большинство языков этого не делают.

Все это создает больше трудностей для программиста, которому проще написать булеву функцию `cyclic`, исходящую из графа и возвращающую `true`, если и только если в графике есть цикл. Такие примеры являются серьезными аргументами в пользу базисного языка утверждений с использованием функций для повышения его выразительной силы.

Но остается необходимость разделять императивные и аппликативные элементы. Любая программно реализованная функция, используемая в утверждениях для специфицирования свойств, должна быть "безупречной", без обвинений ее в императивности, - она не должна быть причиной никаких изменений абстрактного состояния.

Это неформальное требование достаточно ясно на практике; формализм подъязыка IFL исключает все императивные элементы, которые либо изменяют глобальное состояние системы, либо не имеют тривиальных аппликативных эквивалентов, в частности исключаются:

- присваивания атрибутам;
- присваивания в циклах;
- вызовы программ, не входящих в IFL.

Если особо тщательно дирижировать функциями, достаточно простыми с очевидной корректностью, то использование в утверждениях программно реализованных функций дает мощный метод абстракции.

Некоторые технические вопросы могут потребовать внимания. Функция f , используемая в утверждении программы r , может сама иметь утверждения, что демонстрируют примеры функций `full` и `correct_index`. Возникает потенциальная проблема при мониторинге утверждений в период выполнения: если при вызове r мы вычисляем утверждение, вызывающее f , то не придется ли нам вычислять утверждение для f ? Нетрудно сконструировать пример зацикливания, если пойти по этому пути. Но даже и без этого риска было бы неправильно вычислять утверждение для f . Это бы означало, что мы рассматриваем "на равных" программы, являющиеся предметом наших вычислений, такие как r , и их функции утверждения, такие как f . В противовес этому сформулируем правило, согласно которому утверждения должны иметь более высокий приоритет, чем программы, которые они защищают, их корректность должна быть кристально ясной. Приведено простое:

Правило вычисления утверждения

В процессе вычисления утверждений, входящие в них вызовы программ должны выполняться без вычисления ассоциированных утверждений.

Если вызов f встречается как часть проверки утверждения программы r , то слишком поздно спрашивать, удовлетворяет ли f своим утверждениям. Подходящим является время, когда решается вопрос использования f в утверждении, применимом к r .

Рассматривайте f как охранника ядерного предприятия, в обязанности которого входит проверка посетителей. Охранников тоже нужно проверять, но не тогда, когда они сопровождают посетителей.

Инварианты класса и семантика ссылок

ОО-модель, разрабатываемая до сих пор, включала два частично не связанных аспекта, оба из которых полезны:

- Понятие инварианта класса, введенное в этой лекции.
- Гибкая модель периода выполнения, детально рассмотренная в начальных лекциях, существенно использующая ссылки.

К несчастью, эти индивидуально желательные свойства могут стать причиной трудностей при их совместном использовании.

Проблема вновь в динамически создаваемых псевдонимах, предохраняющих нас от проверки корректности класса на том основании, что класс делает это сам. Мы уже видели, что корректность класса означает проверку $m+n$ свойств, выражающих следующее (если мы концентрируем внимание на инвариантах INV, игнорируя предусловия и постусловия, не играющие здесь роли):

1. Каждая из m процедур создания порождает объект, удовлетворяющий INV.
2. Каждая из n экспортруемых программ сохраняет INV.

Кажется, совместно эти два условия гарантируют, что INV действительно инвариант. Доказательство почти тривиально: так как INV удовлетворяется в момент создания и сохраняется при каждом вызове, то по индукции INV истинно во все стабильные времена.

Это неформальное доказательство, однако, не верно в присутствии семантики ссылок и динамических псевдонимов. Проблема в том, что атрибуты объекта могут модифицироваться операциями другого объекта. Даже если $a.g$ сохраняет INV для объекта OA, присоединенного к a , то некоторая операция $b.s$ (для b , присоединенного к другому объекту,) может разрушить INV для OA. Так что условия (1) и (2) могут выполняться, но INV может не быть инвариантом.

Вот простой пример. Предположим, что A и B классы, каждый из которых содержит атрибут другого класса:

```
class A ... feature forward: B ... end
class B ... feature backward: A ... end
```

Потребуем, чтобы ссылки были связаны содержательным условием. Если ссылка forward определена и задает экземпляр класса B, то ссылка backward этого экземпляра, в свою очередь, должна указывать на соответствующий экземпляр класса A. Это может быть выражено как инвариант класса A:

```
round_trip: (forward /= Void) implies (forward.backward = Current)
```

Вот пример ситуации, включающей экземпляры обоих классов и удовлетворяющей инварианту:

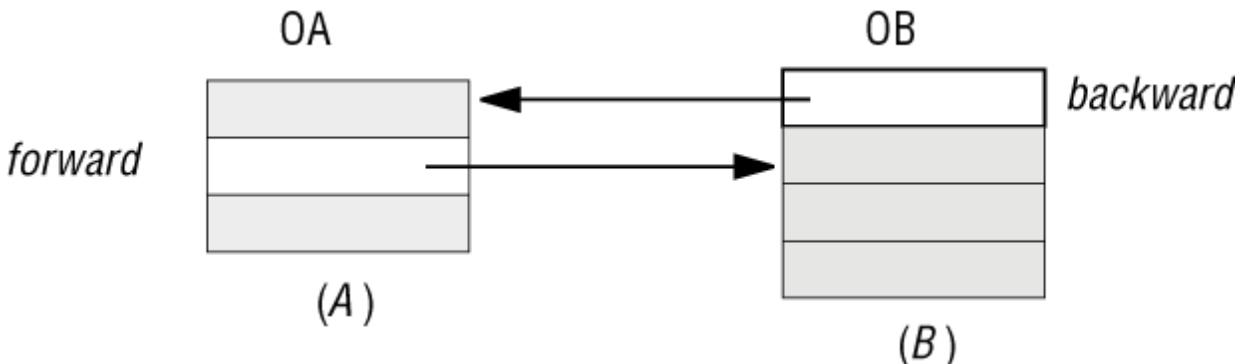


Рис. 11.9. Согласованность ссылок forward и backward

Инвариант round_trip встречается в классах довольно часто. Например, в роли класса A может выступать класс PERSON, характеризующий персону. Ссылка forward может указывать в этом случае на владение персоны - объект класса HOUSE. Ссылка backward в этом классе указывает на владельца дома. Еще одним примером может быть реализация динамической структуры - дерева, узел которого содержит ссылки на старшего сына и на родителя. Для этого класса можно ввести инвариант в стиле round_trip:

Предположим, что инвариант класса B, если он есть, ничего не говорит об атрибуте backward. Следующая версия класса A по-прежнему имеет инвариант:

```
class A feature
  forward: B
  attach (b1: B) is
    -- Ссылка b1 на текущий объект.
  do
```

```

        forward := b1
        -- Обновить ссылку backward объекта b1 для согласованности:
    if b1 /= Void then
        b1.attach (Current)
    end
end
invariant
round_trip: (forward /= Void) implies (forward.backward = Current)
end

```

Вызов `b1.attach` восстанавливает инвариант после обновления `forward`. Класс В должен обеспечить свою собственную процедуру `attach`:

```

class B feature
    backward: B
    attach (a1: A) is
        -- Ссылка a1 на текущий объект.
    do
        backward := a1
    end
end

```

Класс А сделал все для своей корректности: процедура создания по умолчанию гарантирует выполнение инварианта, так как устанавливает `forward` равным `void`, а его единственная процедура гарантирует истинность инварианта. Но рассмотрим выполнение у клиента следующей программы:

```

a1: A; b1: B
...
create a1; create b1
a1.attach (b1)
b1.attach (Void)

```

Вот ситуация после выполнения последней инструкции:

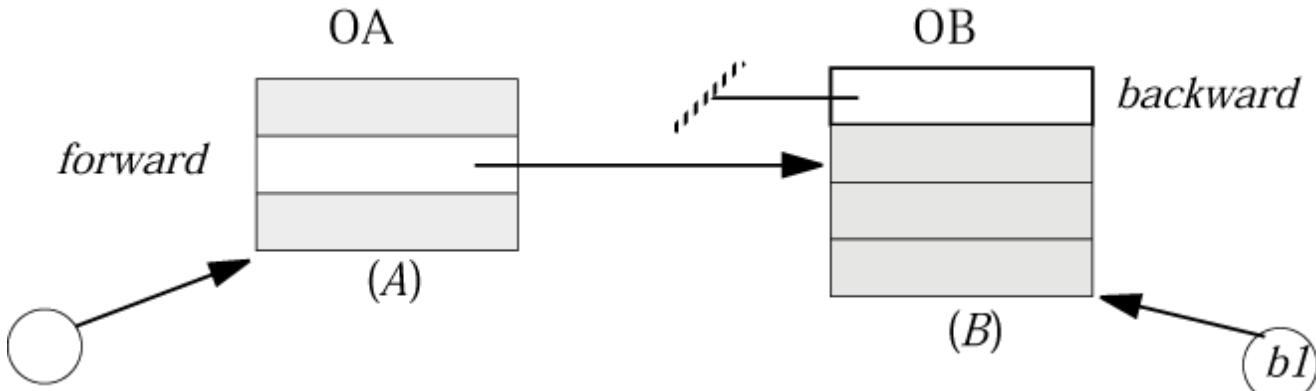


Рис. 11.10. Нарушение инварианта

Инвариант для ОА нарушен. Этот объект теперь указывает на ОВ, но ОВ не указывает на ОА, - `backward` равно `void`. Вызов `b1.attach` мог связать ОВ с любым другим объектом класса А и это тоже было бы некорректно.

Что случилось? Динамические псевдонимы вновь себя проявили. Приведенное доказательство корректности класса А правильно, и единственная процедура этого класса `attach` спроектирована в полном соответствии с замыслом. Но этого недостаточно для сохранения согласованности ОА, так как свойства ОА могут включать экземпляры других классов, а доказательство ничего не говорит об эффекте, производимом свойствами других классов на инвариант из А.

Эта проблема достаточно важна, и заслуживает собственного имени: Непрямой Эффект Инварианта (**Indirect Invariant Effect**). Он может возникать сразу же при допущении динамических псевдонимов, благодаря которому операции могут модифицировать объекты даже без включения любой связанной сущности. Но мы уже видели, как много пользы приносят динамические псевдонимы; и схема `forward` - `backward` далеко не академический пример, это, как отмечалось, полезный образец для практических приложений и библиотек.

Что можно сделать? Промежуточный ответ включает соглашения для мониторинга утверждений в период выполнения. Вы, возможно, удивлялись, почему эффект включения мониторинга утверждений на уровне `assertion` (`invariant`) был описан так:

"Проверка выполнимости инвариантов класса на входе и выходе программы для квалифицированных вызовов".

Почему и на входе и на выходе? Без Непрямого Эффекта Инварианта достаточно было бы проверки на выходе, при условии проверки процедур создания. Но теперь мы должны быть более аккуратными, поскольку между завершением одного вызова и началом вызова другой операции над тем же объектом, могут быть вызовы, задевающие объект, даже если в роли цели выступал совсем другой объект.

Более удовлетворительное решение могло быть получено включением статистического правила, имеющего обязательную силу, гарантирующего, что всякий раз, когда инвариант класса А включает ссылки на экземпляры класса В, инвариант в классе В должен быть зеркальным отображением инварианта из А. В нашем примере можно избежать всех трудностей, включив в класс В инвариант:

```
trip_round: (backward /= Void) implies (backward.forward = Current)
```

Быть может, возможно, обобщить это правило в универсальное правило отображения. Вне зависимости от того, существует ли такое обещающее правило или нет, решение проблемы Непрямого Эффекта Инварианта и избавление необходимости двойной проверки при мониторинге инвариантов требует дальнейших исследований.

Что дальше

Еще не все сделано с Проектированием по контракту. Предстоит изучить два важных следствия рассмотренных принципов:

- Как они приводят к механизму дисциплинированной обработки исключений; это тема следующей лекции.
- Как они комбинируются с наследованием, позволяя нам указать, что любые семантические ограничения, применимые к классу, применимы также и к его потомкам; и что семантические ограничения, применимые к компоненту, применимы и при возможных его переопределениях. Эта тема будет изучаться при рассмотрении наследования.

Обобщая, утверждения и Проектирование по контракту будут сопровождать нас во всей оставшейся части этой книги, позволяя проверить, знаем ли мы, что делают создаваемые нами элементы.

Ключевые концепции

- Утверждения - это булевы выражения, задающие семантические свойства класса и вводящие аксиомы и предусловия соответствующего абстрактного типа данных.
- Утверждения используются в предусловиях (требования, при выполнении которых программы применимы), постусловиях (свойства, гарантируемые на выходе программ), и инвариантах класса (свойства, характеризующие экземпляры класса во время их жизни). Другими конструкциями, включающими утверждения, являются инварианты цикла и инструкции **check**.
- Предусловие и постусловие программы описывают контракт между программой и ее клиентами. Контракт связывает программу, только при условии ее вызова, в состоянии, где предусловие выполняется; в этом случае программа гарантирует выполнимость постусловия на выходе. Понятие заключения контрактов между программами обеспечивает мощную метафору при построении надежного ПО.
- Инвариант класса выражает семантические ограничения экземпляров класса. Инвариант неявно добавляется к предусловиям и постусловиям каждой экспортруемой программы класса.
- Класс описывает одну возможную реализацию АТД; отображение класса в АТД выражается функцией абстракции, обычно частичной. Обратное отношение, обычно, не задается функцией.
- Инвариант реализации, - часть инварианта класса - выражает корректность представления классом соответствующего АТД.
- Цикл может иметь инвариант цикла, позволяющий вывести результат выполнения цикла, и вариант, позволяющий доказать завершаемость цикла.
- Если класс поставляется с утверждениями, то можно формально определить, что означает корректность класса.
- Утверждения служат четырем целям: помогают в конструировании корректных программ; помогают в создании документации, помогают в отладке, являются основой механизма исключений.
- Язык утверждений в нашей нотации не включает логику предикатов первого порядка, но может выражать многие свойства высокого уровня благодаря вызову функций. Функции, включаемые в утверждения должны быть простыми и безупречно корректными.
- Комбинация инвариантов и динамических псевдонимов приводит к Непрямому Эффекту Инварианта, который может стать причиной нарушения инварианта при корректности самого класса.

Библиографические замечания

Из работы Тони Хоара [Hoare 1981]:

Первым защитником использования утверждений в программировании был никто иной, как сам Алан Тьюринг. На конференции в Кембридже 24 июня 1950 г. он представил небольшой доклад "Проверка больших программ", в

которой объяснял эту идею с большой ясностью. "Как можно проверить большую программу, утверждая, что она правильна? Чтобы для проверяющего задача не была слишком трудной, программист обязан написать некоторые утверждения, которые можно проверить индивидуально, и из которых корректность программы следует достаточно просто."

Понятие утверждения, представленное в этой лекции, восходит к работам по корректности программ, пионерами которых были Боб Флойд [Floyd 1967], Тони Хоар [Hoare 1969], Эдсгар Дейкстра [Dijkstra 1976], в дальнейшем описанные в [Gries 1981]. Книга "Введение в теорию языков программирования" (Introduction to the Theory of Programming Languages) [M 1990] представляет обзор этого направления.

Понятие инварианта класса пришло из Хоаровской работы [Hoare 1972a] по инвариантам типов данных. Смотри также приложения к проектированию программ в [Jones 1980], [Jones 1986]. Формальная теория морфизмов между АТД типами может быть найдена у [Goguen 1978].

Библиографические ссылки по формальным языкам спецификаций, включая Z, VDM, OBJ-2, Larch, можно найти в [лекции 6](#). В работе [Lano 1994], содержащей большое число ссылок, описаны ОО-формальные языки спецификаций, включая Object Z, Z++, MooZ, OOZE, SmallVDM, VDM++.

Стандарты по терминологии программных ошибок, дефектов, неисправностей опубликованы IEEE Computer Society [IEEE 1990], [IEEE1993]. Их Web-страница - <http://www.computer.org>

Удивительно, но немногие языки программирования поддерживают синтаксическую поддержку утверждений. Ранним примером (первым, который стал мне известен) был язык AlgolW, созданный Хоаром и Виртом [Hoare 1966], непосредственный предшественник языка Pascal. Другие включают Alphard [Shaw 1981] и Euclid [Lampson 1977], спроектированные специально для разработки корректных программ. Связь с ОО-разработкой и нотация, введенная в этой книге, навеяна утверждениями языка CLU [Liskov 1981], который никогда не был реализован. Другая, базирующаяся на CLU книга Лискова и Гуттага [Liskov 1986] является одной из немногих книг по методологии программирования, в которой глубоко обсуждаются вопросы разработки надежного ПО, предлагая подход на базе "защитного программирования", подвергнутый критике в данной лекции.

Понятие Программирования по контракту, представленное в этой лекции и разрабатываемое в оставшейся части книги, пришло из [M 1987a], продолженное в работах [M 1988], [M1989c], [M 1992a]. В работе [M 1994a] обсуждаются толерантный и требовательный подходы к проектированию предусловий, обращая особое внимание на применение этих подходов к проектированию повторно используемых библиотек, включая политику "требовательной любви". Дальнейший вклад в развитие этих идей был сделан Джеймсом Мак-Кимом [McKim 1995], [McKim 1996], [McKim 1996a], а также [Henderson-Sellers], который занимался исследованием позиции поставщика ПО.

Упражнения

У11.1 Комплексные числа

Напишите спецификацию АТД для класса COMPLEX, описывающую понятие комплексных чисел с арифметическими операциями. Исходите из точной арифметики.

У11.2 Класс и его АТД

Проверьте все предусловия и аксиомы АТД STACK, введенного в предыдущих лекциях, и покажите, отображаются ли они в классе STACK4, а если да, то как.

У11.3 Полные утверждения для стеков

Покажите, что введение закрытой функции body, возвращающей тело стека, сделает возможным утверждениям класса STACK полностью отражать спецификацию соответствующего АТД. Обсудите теоретическую и практическую значимость такого подхода.

У11.4 Экспортирование размера

Почему capacity экспортируется для реализации стеков ограниченных размеров, класс STACK2?

У11.5 Инвариант реализации

Напишите инвариант реализации для класса STACK3.

У11.6 Утверждения и экспорт

Обсудите использование функций в утверждениях, в частности, введение функции correct_index в предусловия программ rut и item. Если добавить эту функцию в класс ARRAY, то какой статус экспорта следует

ей дать?

У11.7 Поиск жучков (bugs)

Покажите, что каждая из четырех попыток бинарного поиска, объявленная как "ошибочная", действительно некорректна. (**Подсказка:** в отличие от доказательства корректности, для доказательства некорректности достаточно предъявить один пример, на котором алгоритм приводит к неверному результату: не завершается, выполняет запрещенную операцию, такую, как выход индекса за допустимые границы, любое другое нарушение предусловия).

У11.8 Нарушение инварианта

В этой лекции было показано, что нарушение предусловия указывает на ошибку клиента, а нарушение постусловия указывает на ошибку поставщика. Объясните, почему нарушение инварианта также указывает на ошибку поставщика.

У11.9 Генерация случайных чисел

Напишите класс, реализующий алгоритм получения псевдослучайных чисел, основанный на последовательности: $n_i = f(n_{i-1})$, где функция f задана, а начальное значение n_0 определяется клиентом класса. Функция не должна иметь побочных эффектов. Определение функции f можно найти в учебниках, таких как [Knuth 1981] и в библиотеках по численным методам.

У11.10 Модуль "очередь"

Напишите класс, реализующий очередь (стратегию доступа "первый пришел - первый ушел", FIFO - "first in - first out"). Задайте подходящие утверждения в стиле класса STACK этой лекции.

У11.11 Модуль "множество"

Напишите класс, реализующий множество элементов произвольного типа со стандартными операциями: проверка принадлежности, добавление нового элемента, объединение, пересечение и другими. Не забудьте включить подходящие утверждения. Приемлема любая корректная реализация, основанная на массивах или связных списках.

Постскриптуm: Катастрофа Ариан 5

Когда первое издание этой книги было опубликовано, Европейское Космическое Агентство опубликовало отчет международного исследования тестирования полета космической ракеты Ариан 5, потерпевшей катастрофу 4 июня 1996 года через 40 секунд после старта, по отчету стоившего 500 миллионов долларов (незастрахованного запуска).

Причина катастрофы: ошибка в бортовой компьютерной системе. Причина этой ошибки: преобразование числа с плавающей точкой, представленного 64 битами, в 16-битовое знаковое целое привело к выбрасыванию исключения. Число задавало горизонтальный наклон (**horizontal bias**) ракеты. Некоторые исключения в системе обрабатывались, используя механизмы языка ADA, описанные в следующей лекции. Но это исключение не обрабатывалось, поскольку ранее проведенный анализ показал, что оно не может встречаться, поэтому решено было не загромождать код обработчиком соответствующего исключения.

Реальная причина: недостаточная спецификация. Проведенный анализ был вполне корректен, - но для траектории полета ракеты Ариан 4. Программный код был повторно использован при полете ракеты Ариан 5, и предположения, хотя и оставленные в маловразумительной документации, были просто забыты. Их просто не применяли к Ариан 5. При Проектировании по контракту было задано предусловие:

```
require
horizontal_bias <= Maximum_horizontal_bias
```

естественно подсказывающие команде, отвечающей за качество, проверить все ли программы выполняют это условие, и своевременно обнаружить возможность его нарушения. Хотя теперь мы уже никогда об этом не узнаем, но, представляется, что почти наверняка эта ошибка была бы обнаружена, вероятно, при статическом анализе, в худшем случае при тестировании с включенным механизмом мониторинга, описанным в этой лекции.

Урок ясен: повторное использование без контрактов безрассудно. Абстрактные модули, определенные нами, как единицы повторного использования, должны поставляться с ясными спецификациями условий их применения - предусловиями, постусловиями, инвариантами. Эти спецификации должны находиться не во внешних документах, а быть частью самих модулей. Эти принципы, которые мы изучили, особенно Проектирование по контракту и Самодокументирование являются необходимым условием любой успешной политики повторного использования. Даже если ошибки будут стоить менее полумиллиарда долларов, всегда помните об этих правилах:

- Повторно используемый модуль должен быть специфицирован.
- Язык программирования должен поддерживать механизм утверждений.
- Спецификации являются частью самого ПО.

1) Общий АТД стек изучался в [лекции 6](#); АТД, задающий стек ограниченной емкости, рассматривался в упражнении У6.9.

2) Эта тема рассматривается в упражнении У11.8. О вычислении предусловий и постусловий см. раздел "Утверждения не являются управляемыми структурами" в этой лекции.

Основы объектно-ориентированного программирования

12. Лекция: Когда контракт нарушается: обработка исключений

Нравится это или нет, но не стоит притворяться, несмотря на все статические предосторожности, некоторые неожиданные и нежелательные события рано или поздно возникнут при одном из выполнений системы. Такие ситуации известны как исключения, и нужно должным образом уметь справляться с ними.

Базисные концепции обработки исключений

Литература по обработке исключений зачастую не очень точно определяет, что вызывает исключение. Как следствие, механизм исключений, представленный в таких языках программирования как PL/I и Ada, часто неправильно используется: вместо того, чтобы резервироваться только для истинно чрезвычайных ситуаций, они заканчивают службу как внутрипрограммные инструкции `goto`, нарушающие принцип Защищенности.

К счастью, теория Проектирования по Контракту, введенная в предыдущей лекции, обеспечивает хорошие рамки для точного определения включаемых концепций.

Отказы

Неформально исключение это аномальное событие, прерывающее выполнение программы. Для получения содержательного определения полезно вначале рассмотреть понятие отказа, непосредственно следующее из идеи контракта.

Программа это не произвольная последовательность инструкций, а реализация некоторой спецификации - контракта программы. Всякий вызов программы должен завершаться в состоянии, удовлетворяющем постусловию и инварианту класса. Неявное следствие контракта - при вызове программы не должны появляться прерывания операционной системы, связанные, например, с обращением к недоступным областям памяти или переполнением при выполнении арифметических операций.

Так должно быть, но в жизни не все происходит так, как должно быть. И мы должны ожидать, что рано или поздно при очередном вызове программы она не сможет выполнить свой контракт. Произойдет системное прерывание, или будет вызвана программа в состоянии, не удовлетворяющем ее предусловию, или в заключительном состоянии будет нарушено постусловие либо инвариант (в двух последних случаях предполагается мониторинг утверждений в период выполнения).

Такие ситуации будем называть **отказом (failure)**.

Определения: успех, отказ

Вызов программы успешен, если он завершается в состоянии, удовлетворяющем контракту. Вызов завершается отказом, если он не успешен.

Будем использовать термины "отказ программы" или просто "отказ", как сокращения более точного термина "вызов программы, завершающийся отказом". Понятно, что сама программа не может быть ни успешной, ни давать отказ. Эти понятия применимы только по отношению к конкретному вызову.

Исключения

Вооружившись понятием отказа, можно теперь определить понятие "исключение". Программа приводит к отказу из-за возникновения некоторых специфических событий (арифметического переполнения, нарушения спецификаций), прерывающих ее выполнение. Такие события и являются исключениями.

Определение: исключение

Исключение - событие периода выполнения, которое может стать причиной отказа программы.

Зачастую исключение будет причиной отказа. Но можно предотвратить отказ, написав программу так, что она будет захватывать возникшее исключение, пытаясь восстановить состояние, допускающее нормальное продолжение вычислений. Вот почему отказ и исключение - это разные понятия: каждый отказ это следствие исключения, но не каждое исключение приводит к отказу.

Изучение программных аномалий в предыдущей лекции привело к появлению терминов **неисправность (fault)** - для событий, приводящих к пагубным последствиям при выполнении программы, **дефект (defect)** - неадекватность программной системы, способная привести к отказам, **ошибка (error)** - неверные решения разработчика или проектировщика, приводящие к дефектам. Отказ - это неисправность; исключение, зачастую, тоже неисправность, но таковым не является, если его возможное появление предвиделось, и программа может справиться с возникшей ситуацией.

Источники исключений

Исключения можно классифицировать, разделив их на категории.

Определение: исключительные ситуации

Исключения могут возникать при выполнении программы в результате следующих ситуаций.

1. Попытка квалифицированного вызова `a.f` и обнаружение, что `a = Void`.
2. Попытка присоединить значение `Void` к развернутой (**expanded**) цели.
3. Выполнение невозможной или запрещенной операции, обнаруживаемое аппаратно или операционной системой.
4. Вызов программы, приводящей к отказу.
5. Предусловие `r` не выполняется на входе.
6. Постусловие `r` не выполняется на выходе.
7. Инвариант класса не выполняется на входе или выходе.
8. Инвариант цикла не выполняется в результате инициализации в предложении `from` или после очередной итерации тела цикла.
9. Итерация тела цикла не уменьшает вариант цикла.
10. Не выполняется утверждение инструкции **check**.
11. Выполнение инструкции, явно включающей исключение.

Случай (1) отражает одно из основных требований к использованию ссылок: вызов `a.f` имеет смысл, когда к `a` присоединен объект, другими словами, когда `a` не `void`. Это обсуждалось в [лекции 8](#) при рассмотрении динамической модели.

Случай (2) также имеет дело с `void` значениями. Напомним, что "присоединение" (**attachment**) покрывает присваивание и передачу аргументов, имеющих одинаковую семантику. В разделе "Гибридное присоединение" [лекции 8](#) отмечалась возможность присваивания ссылки развернутой цели, в результате чего происходит копирование объекта. Но это предполагает существование объекта, но если источник `void`, то присоединение вызовет исключение.

Случай (3) следствие сигналов, посылаемых приложению операционной системой.

Случай (4) возникает при отказе программы, как результат возникновения в ней исключения, с которым она не смогла справиться. Более подробно это будет рассмотрено ниже, но пока обратите внимание на правило, вытекающее из (4):

Отказы и исключения

Отказ программы - причина появления исключения в вызывающей программе.

Случаи (5)-(10) могут встретиться только при мониторинге утверждений, включенных на соответствующем уровне: `assertion` (`require`) для (5), `assertion` (`loop`) для (8) и (9) и так далее.

Случай (11) предполагает вызов процедуры `raise`, выбрасывающей (зажигающей) исключения. Такая процедура будет рассмотрена чуть позднее.

Ситуации отказа

Рассматривая список возможных исключений, полезно определить, когда может встретиться отказ (причина исключения у вызывающей программы):

Определение: случаи отказа

Вызов программы приводит к отказу, если и только если встретилось исключение в процессе выполнения, и программа не смогла с ним справиться.

Определения отказа и исключения взаимно рекурсивны: отказ возникает из-за появления исключений, а одна из причин исключения - отказ при вызове программы (случай (4)).

Обработка исключений

Теперь у нас есть определение того, что может случиться, - исключение - и того, с чем мы бы не хотели столкнуться в результате появления исключения, - отказа. Давайте разыскивать способыправляться с исключениями так, чтобы не возникли отказы. Что может сделать программа, когда ее выполнение прервано из-за нежелательного поведения?

Помощь в нахождении разумного ответа могут дать примеры того, как не следует поступать в подобных ситуациях. Ими мы обязаны механизму сигналов языка C, пришедшему из Unix, и одному учебнику по языку Ada.

Как не следует делать это - C-Unix пример

Первым контрпримером механизма (наиболее полно представленным в Unix, но доступным и на других платформах, реализующих C) является процедура `signal`, вызываемая в следующей форме:

```
signal (signal_code, your_routine)
```

с эффектом вызова обработчика исключения - программы `your_routine`, когда выполнение текущей программы прерывается, выдавая соответствующий код сигнала (`signal_code`). Код сигнала - целочисленная константа, например, `SIGILL` (неверная инструкция - **illegal instruction**) или `SIGFPE` (переполнение с плавающей точкой - **floating-point exception**). В программу можно включить сколь угодно много вызовов процедуры `signal`, что позволяет обрабатывать различные, возможные ошибки.

Теперь предположим, что при выполнении некоторой инструкции произошло прерывание и выработан соответствующий код сигнала. Будет или нет вызвана процедура `signal`, но выполнение программы завершается в не нормальном состоянии. Предположим, что вызывается обработчик события - `your_routine`, пытающийся исправить ситуацию. Беда в том, что, завершив работу, он возвращает управление непосредственно в точку, где произошло прерывание (в не нормальное состояние). Это опасно, вероятнее всего, из этой точки невозможно нормально продолжить работу.

Что необходимо в большинстве подобных случаев - исправить ситуацию и продолжить выполнение, начиная с некоторой особой точки, но не точки прерывания. Мы увидим, что есть простой механизм, реализующий эту схему. Заметьте, он может быть реализован и на C, на большинстве платформ. Достаточно комбинировать процедуру `signal` с двумя другими библиотечными процедурами: `setjmp`, вставляющую маркер в точку, допускающую продолжение вычислений, и `longjmp` для возврата к маркеру. С механизмом `setjmp-longjmp` следует обращаться весьма аккуратно. Поэтому он не ориентирован на обычных программистов, но может использоваться разработчиками компиляторов для реализации высокогоуровневого механизма ОО-исключений, который будет описан в этой лекции.

Как не следует делать это - Ada пример

Приведу пример программы, взятый из одного учебника¹¹ по языку Ada.

```
sqrt (x: REAL) return REAL is
begin
    if x < 0.0 then
        raise Negative;
    else
        normal_square_root_computation;
    end;
exception
    when Negative =>
        put ("Negative argument");
        return;
    when others => ...
end -- sqrt;
```

Этот пример, вероятно, предназначался для синтаксической иллюстрации механизма Ada, и был написан быстро (он, например, отказывается возвращать значение в случае возникновения исключения). Поэтому было бы непорядочно критиковать его, как если бы это был настоящий пример хорошего программирования. Вместе с тем, он ясно показывает нежелательный способ обработки исключений. Поскольку Ada ориентирована на военные и космические приложения, то остается надеяться, что ни одна из реальных программ не следует буквально этой модели.

Целью программы является получение вещественного квадратного корня из вещественного числа. Но что если число отрицательно? В языке Ada нет утверждений, так что в программе проводится проверка, возбуждающая исключение для отрицательных чисел.

Инструкция `raise` прерывает выполнение текущей программы и включает исключение с кодом `Exc`. Это исключение может быть захвачено и обработано при наличии предложений `exception`, имеющих вид:

```
exception
    when code_a1, code_a2, ... => Instructions_a;
    when code_b1, ... => Instructions_b;
    ...
```

Если код исключения совпадает с одним из кодов, указанных в части `when`, то выполняются соответствующие

инструкции. Если, как в примере, есть предложение `when others`, то его инструкции выполняются, когда код исключения не совпадает ни с одним из кодов предыдущих частей `when`. Если нет универсального обработчика `when others`, и код исключения не совпадает ни с одним кодом, то поиск обработчика будет вестись у вызывающей программы, если вызывающей программы нет, то достигнута программа `main` и программа завершается отказом.

В примере нет необходимости переходить к вызывающей программе, поскольку выброшенное исключение с кодом `Negative` захватывается обработчиком с таким же кодом.

Но что делают соответствующие инструкции? Посмотрите еще раз:

```
put ("Negative argument")
return
```

Напечатается сообщение - довольно глубокомысленное, а затем управление перейдет к вызывающей программе, которая, не будучи уведомлена о событии, продолжит свое выполнение, как если бы ничего не случилось. Вспоминая снова о типичных приложениях Ada, можно лишь надеяться, что этой схеме не следуют артиллерийское приложение, в результате которой снаряды могут упасть на головы совсем не тех солдат, для которых вряд ли может служить утешением посланное сообщение об ошибке.

Эта техника, вероятно, хуже, чем C-Unix сигнальный механизм, позволяющий, по крайней мере, возобновить вычисление в точке, где оно остановилось. Обработчик исключения `when`, заканчивающийся инструкцией `return`, даже не продолжает текущую программу; он возвращает управление вызывающей программе, будто бы все прекрасно, в то время как все далеко не прекрасно.

Этот контрпример дает хороший урок Ada-программистам: почти ни при каких обстоятельствах обработчик `when` не должен заканчиваться `return`. Слово "почти" употреблено для полноты картины, поскольку есть особый допустимый случай ложной тревоги (**false alarm**), достаточно редкий, который мы обсудим чуть позже. Опасно и неприемлемо не уведомлять вызывающую программу о возникшей ошибке. Если невозможно исправить ситуацию и выполнить контракт, то программа должна выработать отказ. Язык Ada позволяет сделать это: предложение `exception` может заканчиваться инструкцией `raise` без параметров, повторно выбрасывая исходное исключение, передавая его вызывающей программе. Это и есть подходящий способ завершения выполнения, когда невозможно выполнить свой контракт.

Правило исключений языка Ada

Выполнение любого обработчика исключений должно заканчиваться либо выполнением инструкции `raise`, либо повторением объемлющего программного блока.

Принципы обработки исключений

Контрпримеры помогли указать дорогу к дисциплинированному использованию исключений. Следующие принципы послужат основой обсуждения.

Принципы дисциплинированной обработки исключений

Есть только два легитимных отклика на исключение, возникшее при выполнении программы:

1. **Повторение (Retrying)** - попытка изменить условия, приведшие к исключению, и выполнить программу повторно, начиная все сначала.
2. **Отказ (Failure)** - известный также как "**организованная паника**" (**organized panic**): чистка стека и других ресурсов, завершение вызова и отчет об отказе перед вызывающей программой.

В дополнение, некоторые сигналы операционной системы (случай (3) в классификации исключений) в редких случаях являются откликом на "**ложную тревогу**". Определив, что исключение безвредно, можно возобновить выполнение в точке прерывания.

Давайте начнем рассмотрение с третьего случая - ложной тревоги, обработка которого соответствует основному механизму C-Unix. Вот пример. Некоторые оконные системы будут вызывать исключения, если пользователь перестраивает размеры окна во время выполнения процесса в этом окне. Предположим, что процесс не выполняет никакого вывода в это окно, тогда исключение будет безвредным, и можно возобновить выполнение процесса в прерванной точке. Но даже в этом случае есть лучшие пути, такие как полная блокировка сигналов на время выполнения процесса, чтобы исключение вообще не встретилось. Именно так мы будем поступать с ложными тревогами в механизме, рассматриваемом в следующем разделе.

Ложные тревоги возможны лишь для одного вида сигналов операционной системы - благоприятных сигналов, но нельзя игнорировать арифметическое переполнение или невозможность выделения запрашиваемой память. Исключения всех других категорий также указывают на трудности, не допускающие игнорирования. Было бы абсурдно, например, запускать программу при ложном предусловии.

Повторение - более обнадеживающая стратегия: мы потерпели поражение в битве, но не проиграли войну. Хотя наш первоначальный план выполнения контракта потерпел неудачу, мы можем постараться удовлетворить клиента, применив другую тактику. Если она будет успешной, то исключение не оказывает никакого влияния на клиента. После одной или нескольких попыток, приведших к неудаче, в очередной попытке нам, возможно, удастся полностью выполнить контракт ("Миссия завершена, сэр. Обычные, небольшие проблемы, сэр. Теперь все хорошо, сэр").

Что значит "другая тактика", испытываемая при следующей попытке? Это может быть другой алгоритм; или тот же алгоритм, выполняемый после некоторых произведенных изменений в начальном состоянии (атрибуты, локальные переменные). В некоторых случаях это может быть просто повторный запуск той же программы в надежде, что изменились внешние условия - освободились временно занятые устройства, линии связи и так далее.

При отказе приходится признавать не только поражение в битве, но и невозможность выиграть войну. Мы сдаемся, но прежде следует выполнить два условия, объясняющие использование термина "организованная паника", как более точного синонима понятия "отказ":

- Обеспечить появление исключения у вызывающей программы. В этом и состоит аспект "паники" - программа отказывается жить в соответствии с ее контрактом.
- Восстановить согласованное состояние выполнения - "организованный" аспект.

Что является согласованным состоянием? Корректность класса позволяет дать ответ: состояние, удовлетворяющее инварианту. Мы уже говорили, что программа во время ее выполнения может нарушать инвариант, восстанавливая его в конце работы. Если возникло исключение, то инвариант может быть нарушен. Программа должна восстановить его до возвращения управления вызывающей программе.

Цепочка вызовов

Обсуждая механизм обработки исключений, полезно иметь ясную картину последовательности вызовов, приведших в итоге к исключению. Это понятие уже появлялось при рассмотрении механизма языка Ada.

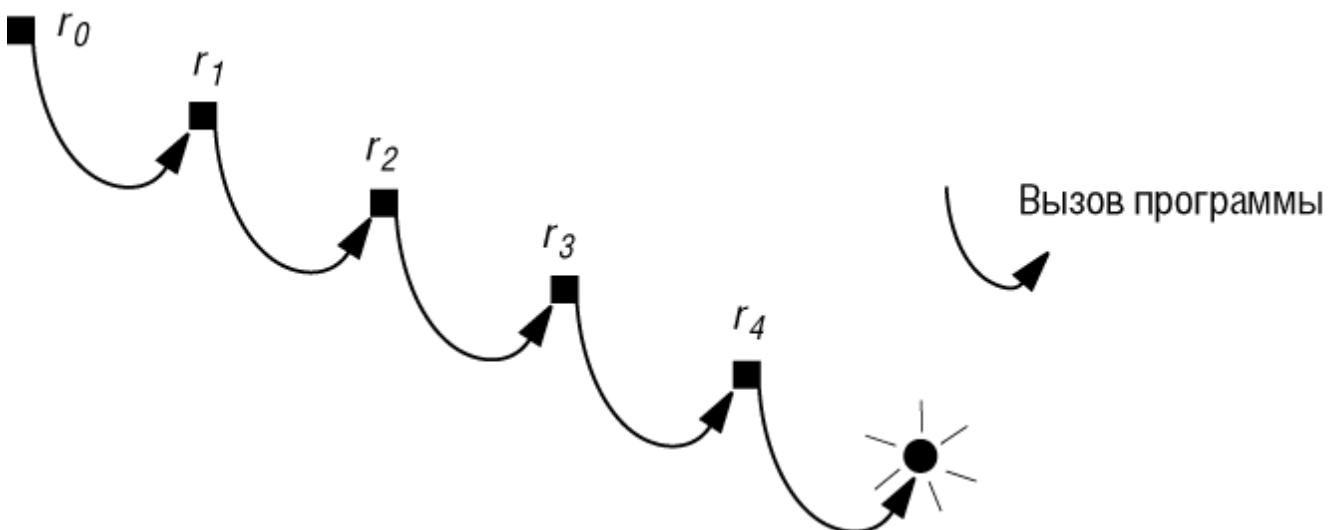


Рис. 12.1. Цепочка вызовов

Пусть r_0 будет корневой процедурой некоторой системы (в Ada это программа `main`). В каждый момент выполнения есть **текущая** программа, вызванная последней и ставшая причиной исключения. Пройдем по цепочке в обратном порядке, начиная с текущей программы, от вызываемой к вызывающей программе. Реверсная цепочка (r_0 , последняя вызванная r_0 программа r_1 , последняя вызванная r_1 программа r_2 и так далее до текущей программы) называется **цепочкой вызовов**.

Если возникает исключение, то для его обработки, возможно, придется подняться по цепочке, пока не будет достигнута программа, способная справиться с исправлением ситуации. Этот процесс заканчивается, когда достигнута программа r_0 и не найден нужный обработчик исключения.

Механизм исключений

Из предшествующего анализа следует механизм исключений, наилучшим образом соответствующий ОО-подходу и идеям Проектирования по Контракту.

Для обеспечения основных свойств введем в язык два новых ключевых слова. Для случаев, в которых необходим точно отрегулированный механизм, будет доступен библиотечный класс `EXCEPTIONS`.

Спаси и Повтори (Rescue и Retry)

Прежде всего, в тексте программы должна быть возможность указания действий, выполняемых при возникновении исключения. Для этой цели и вводится новое ключевое слово `rescue`, задающее предложение с описанием действий, предпринимаемых для восстановления ситуации. Поскольку предложение `rescue` описывает действия, предпринимаемые при нарушении контракта, то разумно поместить его в конце программы после всех других предложений:

```
routine is
  require
    precondition
  local
    ... Объявление локальных сущностей ...
  do
    body
  ensure
    postcondition
  rescue
    rescue_clause
  end
```

Предложение `rescue_clause` является последовательностью инструкций. При возникновении исключения в теле программы вычисление прерывается, и управление передается предложению `rescue`. Хотя есть только одно такое предложение на программу, но в нем можно проанализировать причину исключения и нужным образом реагировать на различные события.

Другой новой конструкцией является инструкция `retry`, записываемая просто как `retry`. Эта инструкция может появляться только в предложении `rescue`. Ее выполнение состоит в том, что она повторно запускает тело программы с самого начала. Инициализация, конечно, не повторяется.

Эти конструкции являются прямой реализацией принципа Дисциплинированной Обработки Исключений. Инструкция `retry` обеспечивает механизм повторения; предложение `rescue`, не заканчивающееся `retry` приводит к отказу.

Как отказаться сразу

Последнее высказывание достойно возведения в ранг принципа.

Принцип отказа

Завершение выполнения предложения `rescue`, не включающее инструкции `retry`, приводит к тому, что вызов программы завершается отказом.

Так что, если и были вопросы, как на практике возникает отказ (ситуация (4) в классификации исключений), то это делается именно так, - при завершении предложения `rescue`.

В качестве специального случая рассмотрим программу, не имеющую предложения `rescue`. На практике именно этот случай характерен для огромного большинства программ. В разрабатываемом подходе к обработке исключений лишь избранные из поставляемых программ должны иметь такое предложение. Игнорируя объявления и другие части программы, можно полагать, что программа без предложения `rescue` имеет вид:

```
routine is
  do
    body
  end
```

Тогда, приняв, как временное соглашение, что отсутствие предложения `rescue` эквивалентно существованию пустого предложения `rescue`, наша программа эквивалента программе:

```
routine is
  do
    body
  rescue
    -- Здесь ничего (пустой список инструкций)
  end
```

Из принципа Отказа вытекают следующие следствия: если исключение встретилось в программе, не имеющей предложения `rescue`, то эта программа вырабатывает отказ, включая исключение у вызывающей программы.

Рассмотрение отсутствующего предложения **rescue**, как присутствующего пустого предложения, является подходящей аппроксимацией на данном этапе рассмотрения. Но нам придется слегка подправить это правило, когда начнем рассматривать эффект исключений на инвариант класса.

Таблица истории исключений

Если в программе произошел отказ, то ли из-за отсутствия предложения **rescue**, то ли потому, что это предложение закончилось без **retry**, она прервёт выполнение вызывающей программы, вызвав в ней исключение типа (4) - отказ в вызываемой программе. Вызывающая программа столкнется с теми же самыми двумя возможностями: либо в ней есть предложение **rescue**, способное исправить ситуацию, либо она выработает отказ и передаст управление вверх по цепочке вызовов. Если на всем пути не найдется программы, способной справиться с исключением, то выполнение всей системы закончится отказом. В этом случае окружение должно сформировать и вывести ясную картину произошедшего - таблицу истории исключений. Вот пример такой таблицы:

Таблица 12.1. Пример таблицы истории исключений

Объект	Класс	Программа	Природа исключения	Эффект
O4	Z_Function	split (from E_FUNCTION)	Feature interpolate: Вызывалась ссылкой void	Повторение
O3	INTERVAL	integrate	interval_big_enough: Нарушено предусловие	Отказ
O2	EQUATION	solve (from GENERAL_EQUATION)	Отказ программы	Отказ
O2	EQUATION	filter	Отказ программы	Повторение
O2	MATH	new_matrix (from BASIC_MATH)	enough_memory: Check Нарушение	Отказ
O1(root)	INTERFACE	make	Отказ программы	Отказ

Эта таблица содержит историю не только тех исключений, которые привели, в конечном счете, к отказу системы, но и исключений, эффект которых был преодолен в результате выполнения **rescue - retry**. Число исключений в таблице может быть ограничено, например, числом 100 по умолчанию. Порядок в таблице сверху вниз является обратным порядку, в котором вызываются программы. Корневая процедура создания записана в последней строке таблицы.

Столбец **Программа** идентифицирует для каждого исключения программу, чей вызов был прерван исключением. Столбец **Объект** идентифицирует цель этого вызова; используемые здесь имена O1 и так далее, но в реальной трассировке они будут внутренними идентификаторами, позволяющие определить, являются ли объекты совпадающими. Столбец **Класс** указывает класс, генерирующий объект.

Столбец **Природа Исключения** указывает, что случилось. Здесь, как показано во второй сверху строке таблицы, могут использоваться метки утверждений, например, **interval_big_enough**, что позволяет точно идентифицировать нарушающее предложение в программе.

Последний столбец указывает, как обрабатывалось исключение, то ли используя Повторение, то ли Отказ. Таблица состоит из последовательности секций, отделяемых толстой линией. Каждая секция, за исключением последней, приводила к Повторению, что указывает на восстановление ситуации. Понятно, что между двумя вызовами, отделенными толстыми линиями, может быть произвольное число вызовов.

Игнорируя такие промежуточные вызовы, - успешные и потому неинтересные для цели нашего обсуждения - здесь приведена цепочка вызовов и возвратов, соответствующая выше приведенной истории исключений. Для реконструкции действий следует следовать по стрелкам, обходя их против часовой стрелки, начиная от программы **make**, изображенной слева вверху.

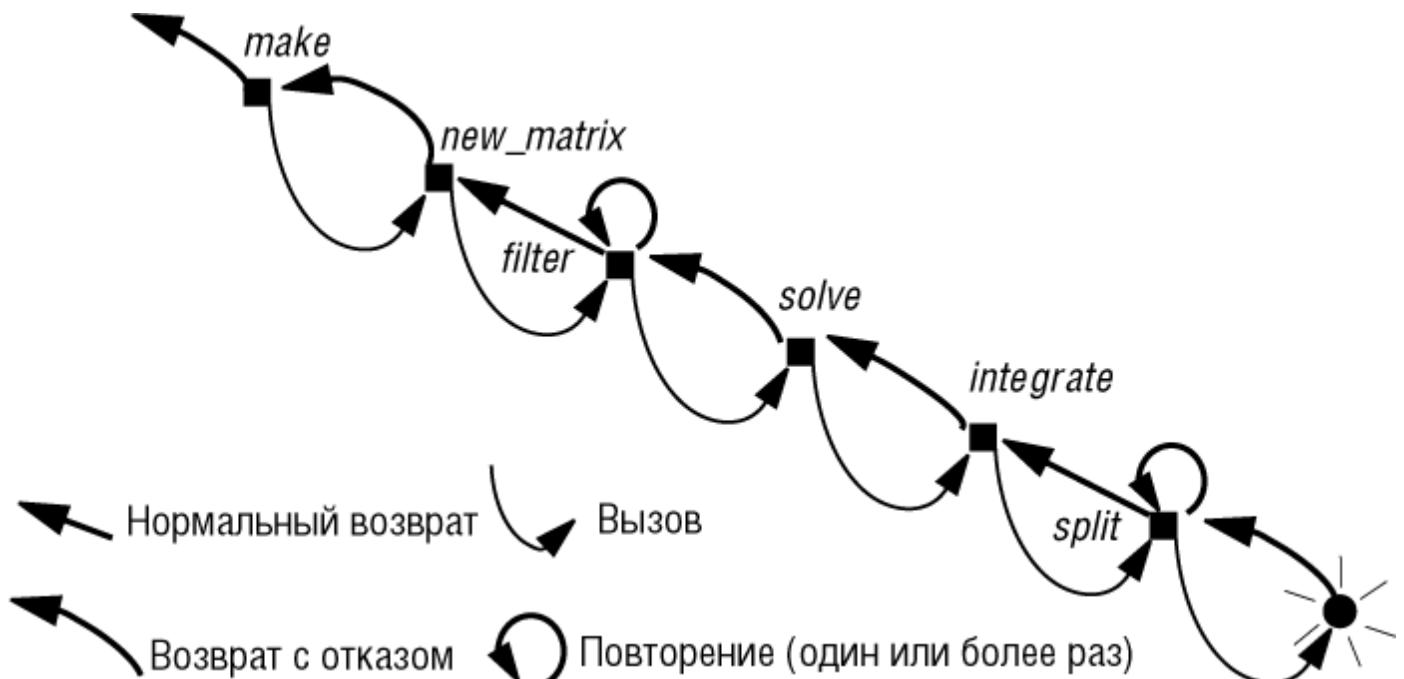


Рис. 12.2. Выполнение, приведшее к отказу

Примеры обработки исключений

Теперь, когда у нас есть базисный механизм, давайте посмотрим, как он применяется в общих ситуациях.

Поломки при вводе

Предположим, что в интерактивной системе необходимо выдать подсказку пользователю, от которого требуется ввести целое. Пусть только одна процедура занимается вводом целых - `read_one_integer`, которая результат ввода присваивает атрибуту `last_integer_read`. Эта процедура работает неустойчиво, - если на ее входе будет нечто, отличное от целого, она может привести к отказу, выбрасывая исключение. Конечно, вы не хотите, чтобы это событие приводило к отказу всей системы. Но поскольку вы не управляете программой ввода, то следует ее использовать и организовать восстановление ситуации, при возникновении исключений. Вот возможная схема:

```
get_integer is
    -- Получить целое от пользователя и сделать его доступным в
    -- last_integer_read.
    -- Если ввод некорректен, запросить повторения, столько раз,
    -- сколько необходимо.
do
    print ("Пожалуйста, введите целое: ")
    read_one_integer
rescue
    retry
end
```

Эта версия программы иллюстрирует стратегию повторения.

Очевидный недостаток - пользователь упорно вводит ошибочное значение, программа упорно запрашивает значение. Это не очень хорошее решение. Можно ввести верхнюю границу, скажем 5, числа попыток. Вот пересмотренная версия:

```
Maximum_attempts: INTEGER is 5
    -- Число попыток, допустимых при вводе целого.
get_integer is
    -- Попытка чтения целого, делая максимум Maximum_attempts попыток.
    -- Установить значение integer_was_read в true или false
    -- в зависимости от успеха чтения.
    -- При успехе сделать целое доступным в last_integer_read.
local
    attempts: INTEGER
do
    if attempts < Maximum_attempts then
        print ("Пожалуйста, введите целое: ")
```

```

    read_one_integer
    integer_was_read := True
else
    integer_was_read := False
end
rescue
    attempts := attempts + 1
    retry
end

```

Предполагается, что включающий класс имеет булев атрибут `integer_was_read`.

Вызывающая программа должна использовать эту программу следующим образом, пытаясь введенное целое присвоить сущности `n`:

```

get_integer
if integer_was_read then
    n := last_integer_read
else
    "Иметь дело со случаем, в котором невозможно получить целое"
end

```

Восстановление при исключениях, сгенерированных операционной системой

Среди событий, включающих исключения, есть сигналы, посылаемые операционной системой, некоторые из которых являются следствием аппаратных прерываний. Примеры: арифметическое переполнение сверху и снизу, невозможные операции ввода-вывода, запрещенные команды, обращение к недоступной памяти, прерывания от пользователя (например, нажата клавиша `break`).

Теоретически можно рассматривать такие условия, как нарушение утверждений. Если $a+b$ приводит к переполнению, то это означает, что вызов не удовлетворяет неявному предусловию функции `+` для целых или вещественных чисел, устанавливающее, что сумма двух чисел должна быть представима в компьютере. Подобное неявное предусловие задается при создании новых объектов (создание копии) - память должно быть достаточно. Отказы встречаются из-за того, что окружение - файлы, устройства, пользователи - не отвечают условиям применимости. Но в таких случаях непрактично или невозможно задавать утверждения, допуская их независимую проверку. Единственное решение - пытаться выполнить операцию, и, если аппаратура или операционная система выдает сигнал о ненормальном состоянии, рассматривать его как исключение.

Рассмотрим проблему написания функции `quasi_inverse`, возвращающей для каждого вещественного x обратную величину $1/x$ или 0 , если x слишком мало.

Подобные задачи по существу нельзя реализовать, не используя механизм исключений. Единственный практический способ узнать, можно ли для данного x получить обратную величину, это выполнить деление. Но деление может спровоцировать переполнение, и если нет механизма управления исключениями, то программа завершится отказом, и будет слишком поздно возвращать 0 в качестве результата.

На некоторых платформах можно написать функцию `invertible`, такую что `invertible(x)` равна `true`, если и только если обратная величина может быть вычислена. Тогда можно написать и `quasi_inverse`. Но это решение не будет переносимым, и может приводить к потере производительности при интенсивном использовании этой функции.

Механизм **rescue-retry** позволяет просто решить эту проблему, по крайней мере, на платформе, включающей сигнал при арифметическом переполнении:

```

quasi_inverse (x: REAL): REAL is
    -- 1/x, если возможно, иначе 0
local
    division Tried: BOOLEAN
do
    if not division Tried then
        Result := 1/x
    end
rescue
    division Tried := True
    retry
end

```

Правила инициализации устанавливают значение `false` для `division_tried` в начале каждого вызова. В теле не нужно предложение `else`, поскольку инициализация установит `Result` равным 0.

Повторение программы, толерантной к неисправностям

Предположим, вы написали текстовый редактор, и к вашему стыду нет уверенности, что он полностью свободен от жучков. Но вам хочется передать эту версию некоторым пользователям для получения обратной связи.

Нашлись смельчаки, готовые принять систему с оставшимися ошибками, понимая, что могут возникать ситуации, когда их запросы не будут выполнены. Но они не будут тестировать ваш редактор на серьезных текстах, (а именно это вам и требуется), если будут бояться, что отказы могут привести к катастрофе, например грубый выход с потерей текста, над которым шла работа последние полчаса. Используя механизм повторения, можно обеспечить защиту от такого поведения.

Предположим, что редактор, как это обычно бывает для подобных систем, содержит основной цикл, выполняющий команды редактора:

```
from ... until exit loop
  execute_one_command
end
```

где тело программы `execute_one_command` имеет вид:

```
"Декодировать запрос пользователя"
"Выполнить команду, реализующую запрос"
```

Инструкция "Выполнить . . ." выбирает нужную программу (например, удалить строку, заменить слово и так далее). Мы увидим в последующих лекциях, как техника наследования и динамическое связывание дает простые, элегантные структуры для подобных ветвящихся решений.

Будем исходить из того, что не все эти программы являются безопасными. Некоторые из них могут отказать в непредсказуемое время. Вы можете обеспечить примитивную, но эффективную защиту против таких событий, написав программу следующим образом:

```
execute_one_command is
  -- Получить запрос от пользователя и, если возможно,
  -- выполнить соответствующую команду.
do
  "Декодировать запрос пользователя"
  "Выполнить подходящую команду в ответ на запрос"
rescue
  message ("Извините, эта команда отказалась")
  message ("Пожалуйста, попробуйте использовать другую команду")
  message ("Пожалуйста, сообщите об отказе автору")
  "Команды, латающие состояние редактора"
  retry
end
```

Эта схема предполагает на практике, что поддерживаемые запросы пользователя включают: "сохранить текущее состояние работы", "завершить работу". Оба последних запроса должны работать корректно. Пользователь, получивший сообщение "Извините . . .", несомненно, захочет сохранить работу и выйти как можно скорее. Некоторые из программ, реализующих команды редактора, могут иметь собственные предложения `rescue`, хотя и приводящие к отказу, но предварительно выдающие более информативные сообщения.

N-версионное программирование

Другим примером повторения программы, толерантной к неисправностям, является реализация N-версионного программирования - подхода, улучшающего надежность ПО.

В основе N-версионного программирования лежит идея избыточности, доказавшая свою полезность в аппаратуре. В критически важных областях зачастую применяется дублирование аппаратуры, например, несколько компьютеров выполняют одни и те же вычисления, и есть компьютер-арбитр, сравнивающий результаты, и принимающий окончательное решение, если большинство компьютеров дало одинаковый результат. Этот подход хорошо защищает от случайных отказов в аппаратуре отдельного устройства. Он широко применяется в аэрокосмической области. (Известен случай, когда при запуске космического челнока сбой произошел в компьютере-арбитре). N-версионное программирование переносит этот подход на разработку ПО в критически важных областях. В этом случае создаются несколько программистских команд, каждая из которых независимо разрабатывает свою версию системы (программы). Предполагается, что ошибки, если они есть, будут у каждой

команды свои.

Это спорная идея; возможно, лучше вложить средства в одну версию, добиваясь ее корректности, чем финансировать две или три несовершенных реализации. Проигнорируем, однако, эти возражения, пусть о полезности идеи судят другие. Нас будет интересовать возможность использования механизма `retry` в ситуации, где есть несколько реализаций, и используется первая из них, не заканчивающаяся отказом:

```
do_task is
    -- Решить проблему, применяя одну из нескольких возможных реализаций.
    require
        ...
    local
        attempts: INTEGER
    do
        if attempts = 0 then
            implementation_1
        elseif attempts = 1 then
            implementation_2
        end
    ensure
        ...
    rescue
        attempts := attempts + 1
        if attempts < 2 then
            "Инструкции, восстанавливающие стабильное состояние"
            retry
        end
    end
end
```

Обобщение на большее, чем две, число реализаций очевидно.

Этот пример демонстрирует типичное использование `retry`. Предложение `rescue` никогда не пытается достигнуть исходной цели, запуская, например, очередную реализацию. Достижение цели - привилегия нормального тела программы.

Заметьте, после двух попыток (в общем случае n попыток) предложение `rescue` достигает конца, не вызывая `retry`, следовательно, приводит к отказу.

Давайте рассмотрим более тщательно, что случается, когда включается исключение во время выполнения `r`. Нормальное выполнение (тела) останавливается; вместо этого начинает выполняться предложение `rescue`. После чего могут встретиться два случая:

- Предложение `rescue` выполнит в конечном итоге `retry`. В этом случае начнется повторное выполнение тела программы. Эта новая попытка может быть успешной, тогда программа нормально завершится и управление вернется к клиенту. Вызов успешен, контракт выполнен. За исключением того, что вызов мог занять больше времени, никакого другого влияния появление исключения не оказывает. Если, однако, повторная попытка снова приводит к исключению, то вновь начнет работать предложение `rescue`.
- Если предложение `rescue` не выполняет `retry`, оно завершится естественным образом, достигнув `end`. (В последнем примере это происходит, когда `attempts >= 2`.) В этом случае программа завершается отказом; она возвращает управление клиенту, сигнализируя о неудаче выбрасыванием исключения. Поскольку клиент должен обработать возникшее исключение, то снова возникают два рассмотренных случая, теперь уже на уровне клиента.

Этот механизм строго соответствует принципу Дисциплинированной Обработки Исключения. Программа завершается либо успехом, либо отказом. В случае успеха ее тело выполняется до конца и гарантирует выполнение постусловия и инварианта. Когда выполнение прерывается исключением, то можно либо уведомить об отказе, либо попытаться повторно выполнить нормальное тело. Но нет никакой возможности выхода из предложения `rescue`, уведомив клиента, что все завершилось нормально.

Задача предложения `rescue`

Последний комментарий позволяет нам продвинуться в лучшем понимании механизма исключений, обосновав теоретическую роль предложения `rescue`. Формальные рассуждения помогут получить полную картину.

Корректность предложения `rescue`

Формальное определение корректности класса выдвигает два требования к компонентам класса. Первое (1) требует, чтобы процедуры создания гарантировали корректную инициализацию - выполнение инварианта класса. Второе (2) напрямую относится к нашему обсуждению, требуя от каждой программы, запущенной при условии

выполнения предусловия и инварианта класса, выполнения в завершающем состоянии постусловия и инварианта класса. Диаграмма, описывающая жизненный цикл объекта, отражает эти требования:

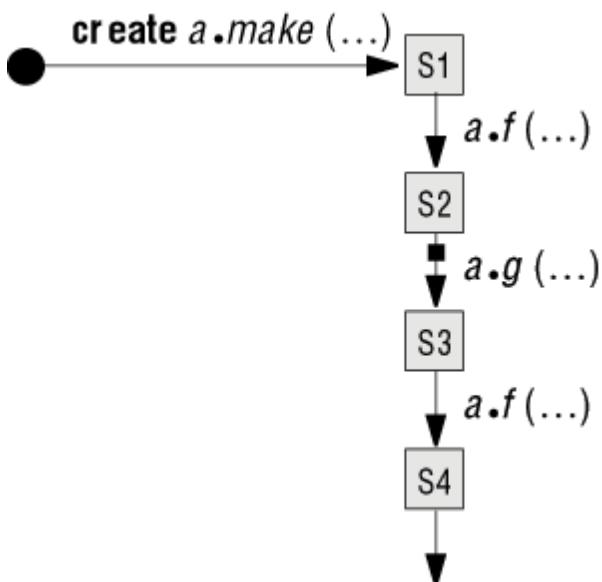


Рис. 12.3. Жизнь объекта

Формально правило (2) говорит:

2. Для каждой экспортируемой программы r и любого множества правильных аргументов x_r

```
{prer (xr) and INV} Bodyr {postr (xr) and INV}
```

Для простоты позвольте в дальнейшем рассмотрении игнорировать аргументы x_r .

Пусть Rescue_r обозначает ту часть предложения rescue , в которой игнорируются все ветви, ведущие к retry , другими словами в этой части сохраняются все ветви, доходящие до конца предложения rescue . Правило (2) задает спецификацию для программ тела - Body_r . Можно ли получить такую же спецификацию для Rescue_r ? Она должна иметь вид:

```
{ ? } Rescuer { ? }
```

с заменой знаков вопроса соответствующими утверждениями. (Полезно, перед дальнейшим чтением постараться самостоятельно задать эти утверждения.)

Рассмотрим, прежде всего, предусловие для Rescue_r . Любая попытка написать нечто **не тривиальное будет ошибкой!** Напомним, чем сильнее предусловие, тем проще работа программы. Любое предусловие для Rescue_r ограничит число случаев, которыми должна управлять эта программа. Но она должна работать во всех ситуациях! Когда возникает исключение, ничего нельзя предполагать, - такова природа исключений. Нам не дано предугадать, когда компьютер даст сбой, или пользователю вздумается нажать клавишу "break".

Поэтому остается единственная возможность - предусловие для Rescue_r равно True . Это самое слабое предусловие, удовлетворяющее всем состояниям и означающее, что Rescue_r должна работать во всех ситуациях.

Для ленивого создателя Rescue_r это "плохая новость", - тот случай, когда "заказчик всегда прав"!

Что можно сказать о постусловии Rescue_r ? Напомню, эта часть предложения rescue ведет к отказу, но, прежде чем передать управление клиенту, необходимо восстановить стабильное состояние. Это означает необходимость восстановления инварианта класса.

Отсюда следует правило, в котором уже больше нет знаков вопросов:

Правило корректности для включающего отказ предложения rescue

3. {True} Rescue_r {INV}

Похожие рассуждения дают правило для Retry_r - части предложения rescue , включающей ветви, приводящие

к инструкции `retry`:

Правило корректности для включающего повтор предложения `rescue`

4. `{True} Retryr {INV and prer }`

Четкое разделение ролей

Интересно сравнить формальные роли тела и предложения `Rescuer`:

```
{prer and INV} Bodyr {postr (xr) INV}  
{True} Rescuer {INV}
```

Входное утверждение сильнее для `Bodyr` - в то время, когда `Rescuer` не накладывает никаких требований, перед началом выполнение тела программы (предложения `do`) должно выполняться предусловие и инвариант. Это упрощает работу `Bodyr`.

Выходное утверждение также сильнее для `Bodyr` - в то время, когда `Rescuer` обязана восстановить только инвариант класса, `Bodyr` обязана сыграть свою роль и обеспечить истинность выполнения постусловия. Это делает ее работу более трудной.

Эти правила отражают разделение ролей между предложением `do` и предложением `rescue`. Задача тела обеспечить выполнение контракта программы, не управляя непосредственно исключениями. Задача `rescue` - управлять обработкой исключениями, возвращая управление либо телу программы, либо вызывающей программе. Но в обязанности `rescue` не входит обеспечение контракта.

Когда нет предложения `rescue`

Формализовав роль предложения `rescue`, вернемся к рассмотрению ситуации, когда это предложение отсутствует в программе. Правило для этого случая было введено ранее, но с обязательством его уточнения. Ранее полагалось, что отсутствующее предложение `rescue` эквивалентно присутствию пустого предложения (`rescue end`). В свете наших формальных правил это не всегда является приемлемым решением. Правило (3) требует:

```
{True} Rescuer {INV}
```

Если `Rescuer` является пустой инструкцией, а инвариант не тождественен `True`, то правило не выполняется.

Зададим точное правило. Класс `Any` является корневым классом - прародителем всех классов. В состав этого класса включена процедура `default_rescue`, наследуемая всеми классами - потомками `Any`:

```
default_rescue is  
    -- Обрабатывает исключение, если нет предложения rescue.  
    -- (По умолчанию: ничего не делает)  
do  
end
```

Программа, не имеющая предложения `rescue`, рассматривается теперь как эквивалентная программе с предложением `rescue` в следующей форме:

```
rescue  
  default_rescue
```

Каждый класс может переопределить `default_rescue`, для выполнения специфических действий, гарантирующих восстановление инварианта класса, вместо эффекта пустого действия, заданного по умолчанию в `GENERAL`. Механизм переопределения компонент класса будет изучаться в последующих лекциях, посвященных наследованию.

Вы, конечно, помните, что одна из ролей процедуры создания состоит в производстве состояния, удовлетворяющего инварианту класса `INV`. Отсюда понятно, что во многих случаях переопределение `default_rescue` может основываться на использовании процедур создания.

Продвинутая обработка исключений

Чрезвычайно простой механизм, разработанный до сих пор, удовлетворяет большинству потребностей обработки исключений. Но некоторые приложения могут требовать более тонкой настройки:

- Возможно, требуется определить природу последнего исключения, чтобы разными исключениями управлять по-разному.
- Возможно, требуется запретить включение исключений для некоторых сигналов.
- Возможно, вы захотите включать собственные исключения.

Можно было бы соответствующим образом расширить механизм, встроенный в язык, но это не кажется правильным подходом. Вот, по меньшей мере, три причины. Первая - свойства нужны только от случая к случаю, так что они будут загромождать язык. Вторая - все, что касается сигналов, может зависеть от платформы, а язык должен быть переносимым. Наконец, третья, - когда выбирается множество подобных свойств, никогда нет полной уверенности, что позже вам не захочется добавить новое свойство, что требовало бы модификации языка - не очень приятная перспектива.

В таких ситуациях следует обращаться не к языку, но к поддерживающим библиотекам. Мы введем библиотечный класс EXCEPTIONS, обеспечивающий необходимые возможности тонкой настройки. Классы, нуждающиеся в таких свойствах, будут наследниками EXCEPTIONS. Некоторые разработчики могут предпочесть отношение встраивания вместо наследования.

Запросы при работе с классом EXCEPTIONS

Класс EXCEPTIONS обеспечивает несколько запросов для получения требуемой информации о последнем исключении. Прежде всего, можно получить целочисленный код этого исключения:

```
exception: INTEGER
    -- Код последнего встретившегося исключения
original_exception: INTEGER
    -- Код последнего исключения - первопричины текущего исключения
```

Разница между exception и original_exception важна в случае "организованной паники". Если программа получила исключение с кодом ос, указывающим на арифметическое переполнение, но не имеет предложения **rescue**, то вызывающая программа получит исключение, код которого, заданный значением exception, будет указывать на "отказ в вызванной программе". Но на этом этапе или выше по цепи вызовов может понадобиться выяснить оригинальное исключение - первопричину появления исключений - код ос, который и будет значением original_exception.

Коды исключений являются целыми. Значения для предопределенных исключений задаются целочисленными константами, обеспечиваемыми классом EXCEPTIONS (который наследует их от класса EXCEPTIONS_CONSTANTS). Вот несколько примеров:

```
Check_instruction: INTEGER is 7
    -- Код исключения при нарушении утверждения check
Class_invariant: INTEGER is ...
    -- Код исключения при нарушении инварианта класса
Incorrect_inspect_value: INTEGER is ...
    -- Код исключения, когда проверяемое значение не является ни одной
    -- ожидаемых констант, если отсутствует часть Else
Loop_invariant: INTEGER is ...
    -- Код исключения при нарушении инварианта цикла
Loop_variant: INTEGER is ...
    -- Код исключения при нарушении убывания варианта цикла
No_more_memory: INTEGER is ...
    -- Код исключения при отказе в распределении памяти
Postcondition: INTEGER is ...
    -- Код исключения при нарушении постусловия
Precondition: INTEGER is ...
    -- Код исключения при нарушении предусловия
Routine_failure: INTEGER is ...
    -- Код исключения при отказе вызванной программы
Void_assigned_to_expanded: INTEGER is ...
```

Так как значения констант не играют здесь роли, то показано только первое из них.

Приведу несколько других запросов, обеспечивающих при необходимости дополнительной информацией. Смысл запросов понятен из их описания:

```
meaning (except: INTEGER)
```

```

-- Сообщение, описывающее природу исключения с кодом except
is_assertion_violation: BOOLEAN
    -- Является ли последнее исключение нарушением утверждения
    -- или нарушением убывания варианта цикла
ensure
    Result = (exception = Precondition) or (exception = Postcondition) or
        (exception = Class_invariant) or
        (exception = Loop_invariant) or (exception = Loop_variant)
is_system_exception: BOOLEAN
    -- Является ли последнее исключение внешним событием
    -- (ошибкой операционной системы)?
is_signal: BOOLEAN
    -- Является ли последнее исключение сигналом операционной системы?
tag_name: STRING
    -- Метка утверждения, нарушение которого привело к исключению
original_tag_name: STRING
    -- Метка последнего нарушенного утверждения оригинальным исключением.
recipient_name: STRING
    -- Имя программы, чье выполнение было прервано последним исключением
class_name: STRING
    -- Имя класса, включающего получателя последнего исключения
original_recipient_name: STRING
    -- Имя программы, чье выполнение было прервано
    -- последним оригинальным исключением
original_class_name: STRING
    -- Имя класса, включающего получателя последнего оригинального исключения

```

Имея эти свойства, предложение `rescue` может управлять каждым исключением особым способом. Например, в классе, наследуемом от `EXCEPTIONS`, предложение `rescue` можно написать так:

```

rescue
    if is_assertion_violation then
        "Случай, обрабатывающий нарушение утверждений"
    else if is_signal then
        "Случай, обрабатывающий сигналы операционной системы"
    else
        ...
end

```

Используя класс `EXCEPTIONS`, можно модифицировать пример `quasi_inverse`, чтобы он выполнял `retry` только при переполнении. Другие исключения, например, нажатие пользователем клавиши "break" не должны приводить к `retry`. Инструкция в предложении `rescue` теперь может иметь вид:

```

if exception = Numerical_error then
    division_tried := True; retry
end

```

Так как здесь нет `else` ветви, то исключения, отличные от `Numerical_error`, будут причиной отказа - корректное следствие, поскольку программа не имеет рецепта восстановления в подобных случаях. Иногда предложение `rescue` пишется специально для того, чтобы обработать определенный вид возможных исключений. Этот стиль позволяет избежать анализа других неожиданных видов исключений.

Какой должна быть степень контроля?

Могут возникнуть замечания по поводу уровня обработки специфических исключений, иллюстрируемых двумя последними примерами. В этой лекции проводилась та точка зрения, что исключение - нежелательное событие; когда оно возникает, то естественная реакция ПО и его разработчика - "я не хочу быть здесь! Выпустите меня отсюда, как можно скорее!". Это, кажется, несовместимым с проведением в предложении `rescue` глубокого анализа источника исключений.

По этой причине я пытался в моей собственной работе избегать детального разбора случаев причины исключений, стараясь показать, что обработка исключений лишь фиксирует ситуацию, если может, а затем либо `fail`, либо `retry`.

Этот стиль, вероятно, слишком строг, и некоторые разработчики предпочитают менее ограниченную схему, используя в полной мере механизм запросов класса `EXCEPTIONS`, позволяющий в тоже время оставаться дисциплинированным. Если вы хотите придерживаться такой схемы, то в классе `EXCEPTIONS` найдете все, что

для этого нужно. Но всегда помните о следующем принципе:

Принцип Простоты Исключений

Вся обработка, выполняемая в предложении **rescue**, должна оставаться простой и фокусироваться на единственной цели - возвратить объект получателя в стабильное состояние, допуская повторение, если это возможно.

Исключения разработчика

Все исключения, изучаемые до сих пор, были результатом событий внешних по отношению к ПО (сигналы операционной системы) или принудительных следствий его работы (нарушение утверждений). В некоторых приложениях полезно, чтобы исключения возникали по воле разработчика в определенных ситуациях.

Такие исключения называются исключениями разработчика. Они характеризуются как целочисленным кодом, отличающимся от системных кодов, так и именем (строкой), которые могут быть использованы, например, в сообщениях об ошибке. Можно использовать следующие свойства для возбуждения исключения разработчика и для анализа его свойств в предложении **rescue**.

```
trigger (code: INTEGER; message: STRING)
    -- Прерывает выполнение текущей программы, выбрасывая исключение с кодом
    -- code и связанным текстовым сообщением.
developer_exception_code: INTEGER
    -- Код последнего исключения разработчика
developer_exception_name: STRING
    -- Имя, ассоциированное с последним исключением разработчика
is_developer_exception: BOOLEAN
    -- Было ли последнее исключение исключением разработчика?
is_developer_exception_of_name (name: STRING): BOOLEAN
    -- Имеет ли последнее исключение разработчика имя name?
ensure
    Result := is_developer_exception and then
        equal (name, developer_exception_name)
```

Иногда полезно связать с исключением разработчика контекст - произвольный объект, структура которого может быть полезной при обработке исключения разработчика:

```
set_developer_exception_context (c: ANY)
    -- Определить с как контекст, связанный с последовательностью
    -- исключений разработчика (причина вызова компонента trigger).
require
    context_exists: c /= Void
developer_exception_context: ANY
    -- Контекст, установленный последним вызовом
set_developer_exception_context
    -- void, если нет такого вызова.
```

Эти свойства позволяют использовать стиль программирования, в котором обработка исключений представляет часть общего процесса работы с программными элементами. Авторы одного из трансляторов при разборе текстов предпочитали выбрасывать исключения при появлении особых случаев, после чего вызывать для их анализа специальные программы. Это не мой стиль работы, но по сути, ничего ошибочного в нем нет, так что механизм исключений разработчика для тех, кому нравится так работать.

Обсуждение

Мы закончили проектирование механизма исключений, совместимого с применяемым ОО-подходом, и следующего идеям Проектирования по Контракту. Благодаря инструкции **retry**, механизм получился более мощным, чем во многих языках программирования. В то же время он может казаться более строгим из-за акцента надержанность при определении точных причин исключения.

Давайте рассмотрим несколько альтернативных идей проектирования, которым можно было бы следовать, и обсудим, почему они были опущены.

Дисциплинированные исключения

Исключения, как они были введены, дают способ справиться с аномалиями, возникающими в процессе выполнения: нарушениями утверждений, сигналами аппаратуры, попытками получить доступ к void ссылкам.

Исследуемый нами подход основан на метафоре контракта, - ни при каких обстоятельствах программа не должна претендовать на успешность, когда фактически имеет место отказ в достижении цели. Программа может быть либо успешной (возможно, после исправления ситуации и нескольких попыток **retry**), либо приводить к отказу.

Исключения в языках Ada, CLU, PL/1 не следуют этой модели. В языке Ada ее инструкция

```
Raise exc
```

прервает выполнение программы и возвратит управление вызывающей программе, которая может обработать исключение в специальном обработчике, или вернет управление на уровень выше. Но здесь нет правила, ограничивающего действия обработчика. Следовательно, полностью возможно игнорировать исключение или вернуть альтернативный результат. Это объясняет, почему некоторые разработчики смотрят на механизм исключений просто как на средство обработки специальных случаев, не включенных в основной алгоритм. Такие приложения исключения рассматривают фактически *raise* как *goto*, что, очевидно, опасно, так как позволяет передавать управление за границы программы. По моему мнению, они злоупотребляют механизмом.

Традиционно есть две точки зрения на исключения. Первая признает исключения необходимым свойством. Она присуща большинству практикующих программистов, знающих как важно сохранить управление во время выполнения программы при возникновении ненормальных условий - аппаратных или программных ошибок. Вторая точка зрения присуща ученым, озабоченным корректностью и систематическим конструированием программ. Они с подозрением относятся к исключениям, рассматривая их как нечто нечистое, старающееся обойти стандартные правила управления программными структурами. Надеюсь, выше разработанный механизм способен примирить обе стороны.

Должны ли исключения быть объектами?

Фанатики объектной ориентации (многие ли из тех, кто открыл красоту этого подхода, не рискуют стать его фанатиками?) могут критиковать представленный механизм за то, что исключения не являются гражданами первого сорта в программном сообществе. Почему исключения не являются объектами?

В ОО-расширении Pascal в среде Delphi исключения действительно представлены объектами.

Не очень понятны преимущества такого решения. Некоторое обоснование можно будет найти в [лекции 4](#) курса "Основы объектно-ориентированного проектирования", посвященной ответу на вопрос, каким должен быть класс. Объект является экземпляром абстрактно определенного типа данных, характеризуемого его компонентами. Исключение, конечно, как мы видели в классе EXCEPTIONS, имеет компоненты, заданные целочисленным кодом, текстовым сообщением. Но эти компоненты являются **запросами**, в то время, как **истинные** объекты имеют **команды**, изменяющие состояние объекта. Исключения не находятся под управлением программной системы; они результат событий, находящихся вне пределов ее достижимости.

Доступность их свойств через запросы и команды класса EXCEPTIONS достаточна для удовлетворения потребностей разработчиков, которые хотят обрабатывать исключения конкретного вида.

Методологическая перспектива

Финальное замечание и обзор. Обработка исключений, имеющая дело со специальными и нежелательными случаями, - не единственный ответ на общую проблему устойчивости. Мы уже приобрели некоторую методологическую интуицию, но более полный ответ появится в лекции, обсуждающей проектирование интерфейсов модулей, позволяя нам понять место обработки исключений в широком арсенале методов устойчивости и расширения.

Ключевые концепции

- Обработка исключений - это механизм, позволяющий справиться с неожиданными условиями, возникшими в период выполнения.
- Отказ - это невозможность во время выполнения программы выполнить свой контракт.
- Программа получает исключение в результате: отказа вызванной ею программы, нарушения утверждений, сигналов аппаратуры или операционной системы об аномалиях, возникших в ходе их работы.
- Программная система может включать также исключения, спроектированные разработчиком.
- Программа имеет два способа справиться с исключениями - Повторение вычислений (**Retry**) и Организованная Паника. При Повторении тело программы выполняется заново. Организованная Паника означает отказ и формирование исключения у вызывающей программы.
- Формальная роль обработчика исключений, не заканчивающегося **retry**, состоит в восстановлении инварианта, но не в обеспечении контракта программы. Последнее всегда является делом тела программы (предложения **do**). Формальная роль ветви, заканчивающейся **retry**, состоит в восстановлении инварианта и предусловия, так чтобы тело программы могло попытаться в новой попытке выполнить контракт.
- Базисный механизм обработки исключений, включаемый в язык, должен оставаться простым, если только поощрять прямую цель обработки исключений - Организованную Панику или Повторение. Для приложений, нуждающихся в более тонком контроле над исключениями, доступен класс EXCEPTIONS, позволяющий

добраться до свойств каждого вида исключений и провести их обработку. Этот класс позволяет создавать и обрабатывать исключения разработчика.

Библиографические замечания

[Liskov 1979] и [Cristian 1985] предлагали другие точки зрения на исключения. Многие из работ по ПО, толерантному к отказам, ведут начало от понятия "восстанавливающий блок" [Randell 1975]. Такой блок используется в задаче, когда основной алгоритм отказывается выдавать решение. Этим "восстанавливающий блок" отличается от предложения `rescue`, которое никогда не пытается достичь основной цели, хотя и может повторно запустить выполнение, предварительно "залатав" все повреждения.

[Hoare 1981] содержит критику механизма исключений Ada.

Подход к обработке исключений, разработанный в этой лекции, был впервые представлен в [М 1988e] и [М 1988].

Упражнения

У12.1 Наибольшее целое

Предположим, компьютер генерирует исключение, когда сложение целых дает переполнение. Используя обработку исключений, напишите приемлемую по эффективности функцию, возвращающую наибольшее положительное целое, представимое на этой машине.

У12.2 Объект Exception

Несмотря на скептицизм, высказанный в разделе "Обсуждение" этой лекции по поводу рассматривания исключений как объектов, зайдитесь развитием этой идеи и обсудите, как мог бы выглядеть класс EXCEPTION, полагая, что экземпляры этого класса обозначают исключения, появившиеся при выполнении. Не путайте его с классом EXCEPTIONS, который доступен благодаря наследованию и обеспечивает общие свойства исключений. Попытайтесь, в частности, наряду с запросами, включить команды в разрабатываемый вами класс.

¹⁾ Sommerville, Morrison "Software Development with Ada", Addison-Wesley, 1987. Синтаксис и некоторые идентификаторы изменены для приведения в соответствие со стилем данной книги.

Основы объектно-ориентированного программирования

13. Лекция: Поддерживающие механизмы

Выше рассмотрены все основные методы создания ОО-программного продукта, кроме одного важнейшего набора механизмов. Недостающий раздел - наследование и все, что к нему относится. Перед тем как перейти к этой последней составляющей подхода, опишем несколько механизмов, важных для создания систем: внешние программы и инкапсуляцию не ОО-программного продукта; передача аргументов; структуры управления; выражения; действия со строками; ввод и вывод. Эти технические аспекты существенны для понимания дальнейших примеров. Они хорошо сочетаются с основными концепциями.

Взаимодействие с не объектным ПО

До сих пор, элементы ПО выражались полностью в ОО-нотации. Но программы появились задолго до распространения ОО-технологии. Часто возникает необходимость соединить объектное ПО с элементами, написанными, например, на языках C, Fortran или Pascal. Нотация должна поддерживать этот процесс.

Сначала следует рассмотреть языковой механизм, а затем поразмышлять над его более широким значением как части процесса разработки ОО-продукта.

Внешние программы

ОО-системы состоят из классов, образованных компонентами (features), в частности, подпрограммами, содержащими инструкции. Что же является правильным уровнем модульности (granularity) для интегрирования внешнего программного продукта?

Конструкция должна быть общей - это исключает классы, существующие только в ОО-языках. Инструкции - слишком низкий уровень. Последовательность, в которой две ОО-инструкции окаймляют инструкцию на языке C:

```
-- только в целях иллюстрации
create x l make (clone (a))
(struct A) *x = &y; /* A piece of C */
x.display
```

трудно было бы понять, проверить, сопровождать.

Остается уровень компонентов. Он разумен и допустим, поскольку инкапсуляция компонентов совместима с ОО-принципами. Класс является реализацией типа данных, защищенных скрытием информации. Компоненты - единицы взаимодействия класса с остальной частью ПО. Поскольку клиенты полагаются на официальную спецификацию компонентов (краткую форму) независящую от их реализации, внешнему миру неважно, как написан компонент - в ОО-нотации или нет.

Отсюда вытекает понятие внешней программы. Внешняя программа имеет большинство признаков нормальной программы: имя, список аргументов, тип результата, если это функция, предусловие и постусловие, если они уместны. Вместо предложения **do** она имеет предложение **external**, определяющее язык реализации. Следующий пример взят из класса, описывающего символьные файлы:

```
put (c: CHARACTER) is
    -- Добавить с в конец файла.
    require
        write_open: open_for_write
    external
        "C" alias "_char_write";
    ensure
        one_more: count = old count + 1
    end
```

Предложение **alias** факультативно и используется, только если оригинальное имя внешней программы отличается от имени, данного в классе. Это случается, когда внешнее имя недопустимо в ОО-нотации, например, имя, начинающееся с символа подчеркивания (используемое в языке C).

Улучшенные варианты

Описанный механизм включает большинство случаев и достаточен для целей описания нашей книги. На практике полезны некоторые уточнения:

- Некоторые внешние программные элементы могут быть **макросами**. Они имеют вид подпрограмм в ОО-мире, но любой их вызов предполагает вставку тела макроса в точке вызова. Этого можно достичь вариацией имени языка (как, например, "C:[macro]...").
- Необходимо также разрешить вызовы программ из "динамически присоединяемых библиотек" (DLL), доступных в Windows и других платформах. Программа DLL загружается динамически во время первого вызова. Имя программы и библиотеки разрешается также задавать динамически в период выполнения. Поддержка DLL должна включать как способ статической спецификации имени, так и полностью динамический подход с использованием библиотечных классов DYNAMIC_LIBRARY и DYNAMIC_ROUTINE. Эти классы можно инициализировать во время выполнения, создавая объекты, представляющие динамически определенные библиотеки и подпрограммы.
- Необходима и связь в обратном направлении, позволяющая не объектному ПО создавать объекты и вызывать компоненты. Например, графической системе может понадобиться механизм **обратного вызова (callback mechanism)**,зывающий определенные компоненты класса.

Все эти возможности присутствуют в ОО-среде, описанной в последней лекции. Однако их подробное обсуждение - это отдельный разговор.

Использование внешних программ

Внешние программы являются частью ОО-метода, помогая сочетать старое ПО с новым. Любой метод проектирования ПО, допускающий возможность повторного использования, должен допускать программный код, написанный на других языках. Трудно было бы убедить потенциального пользователя, что надо отказаться от всего существующего ПО, поскольку с этой минуты начинается повторное использование.

Открытость остальному миру - требование большинства программных продуктов. Это можно назвать **принципом скромности**: авторы новых инструментов должны дать возможность пользователям иметь доступ к ранее имевшимся возможностям.

Внешние программы также необходимы для обеспечения доступа к аппаратуре и возможностям операционной системы. Типичный пример - класс файлов. Другой пример - класс ARRAY, чей интерфейс рассматривался в предыдущих лекциях, и чья реализация основана на внешних программах: процедура создания make использует программу распределения памяти, функция доступа item использует внешний механизм для быстрого доступа к элементам массива, и т.д.

Эта техника обеспечивает ясный интерфейс между ОО-миром и другими подходами. Для клиентов внешняя программа - это просто программа. В примере, программа на C _char_write обрела статус компонента (feature) класса, дополнена предусловием и постусловием и получила стандартное имя rwt. Возможности, внутренне опирающиеся на не ОО-механизмы, получают новую упаковку абстрактных данных, так что участники ОО-мира начинают рассматривать их как законных граждан сообщества, и их низкое происхождение никогда не упоминается в "изысканном обществе". ("Изысканное общество" не означает бесклассовое.)

ОО-изменение архитектуры (re-architecturing)

Понятие внешней программы хорошо соответствует остальной части подхода. Основной вклад метода - архитектурный: объектная технология говорит, как разработать структуру систем, чтобы обеспечить расширяемость, надежность и повторное использование. Она также говорит, как заполнить эту структуру. Но что по-настоящему определяет, является ли система объектной, - так это ее модульная организация. Для использования ОО-архитектуры часто разумно использовать прием, называемый обертыванием (wrap), одевая в одежды класса внутренние элементы.

Крайний, но не совсем абсурдный, способ использования нотации - построить систему полностью на внешних программах. Объектная технология тогда служит просто инструментом упаковки, использующим мощные механизмы инкапсуляции: классы, утверждения, скрытие информации, клиент, наследственность.

Но обычно нет причины заходить так далеко. ОО-нотация адекватна вычислениям любого рода и столь же эффективна, как и вычисления на языках Fortran или С. В каких случаях полезна ОО-инкапсуляция внешнего ПО? Один из них мы видели: обеспечение доступа к операциям, зависящим от платформы. Другой - проблема многих организаций - управление старым ПО, доставшимся в наследство и продолжающим широко использоваться. Объектная технология предлагает возможность обновления таких систем, изменяя их архитектуру, но не переписывая их полностью.

Эта техника, которую можно назвать **ОО-перестройкой (object-oriented re-architecturing)** дает интересное решение сохранения ценных свойств существующего ПО, готовя его к будущему расширению и эволюции.

Однако для этого необходимы определенные условия:

- Необходимо суметь подобрать хорошие абстракции для старого ПО, которое, не будучи объектным, как

правило, имеет дело с абстракциями функций, а не данных. Но в этом и состоит задача - обернуть старые функции в новые классы. Если с выделением абстракций не удастся справиться, то никакая ОО-перестройка не поможет.

- Наследуемое ПО должно быть хорошего качества. Перестроенное старье остается старьем - возможно хуже первоначального, поскольку оно будет скрыто под слоями абстракции.

Эти два требования частично сходны, поскольку качество любого ПО в значительной степени определяется качеством его структуры.

Когда они выполнены, можно использовать **внешний** механизм для построения интересного ОО-программного продукта, основанного на прежних разработках. Приведем два примера, являющихся частью среды, описанной в последней лекции.

- Библиотека Vision (библиотеки описываются в [лекции 14](#) курса "Основы объектно-ориентированного проектирования") дает переносимую графику и механизмы пользовательского интерфейса, позволяющие разработчикам создавать графические приложения для многих различных платформ с ощущением обычной перекомпиляции. Внутренне, она основана на "родных" механизмах, используемых во внешних программах. Точнее, ее нижний уровень инкапсулирует механизмы соответствующих платформ.
- Другая библиотека, Math, обеспечивает широкий набор возможностей численных вычислений в таких областях как теория вероятностей, статистика, численное интегрирование, линейные и нелинейные уравнения, дифференциальные уравнения, оптимизация, быстрое преобразование Фурье, анализ временных рядов. Внутренне она основана на коммерческой библиотеке подпрограмм, библиотеке NAG от Nag Ltd., Oxford, но обеспечивает пользователям ОО-интерфейс. Библиотека скрывает используемые ею программы и предлагает абстрактные объекты, понятные математику, физику или экономисту, представленные классами: INTEGRATOR, BASIC_MATRIX, DISCRETE_FUNCTION, EXPONENTIAL_DISTRIBUTION. Прекрасные результаты достигаются благодаря качеству внешних программ - NAG аккумулирует сотни человеко-лет разработки и реализации численных алгоритмов. К нему добавлены ОО-преимущества: классы, скрытие информации, множественное наследование, утверждения, систематическая обработка ошибок через исключительные ситуации, согласованное именование.

Эти примеры типичны для сочетания лучших традиционных программных продуктов и объектной технологии.

Вопрос совместимости: гибридный программный продукт или гибридные языки?

Теоретически, мало кто не согласится с принципом скромности или будет отрицать необходимость механизма интеграции между ОО-разработками и старым ПО. Противоречия возникают, когда выбирается уровень интеграции.

Многие языки - самыми известными являются Objective-C, C++, Java, Object Pascal и Ada 95 - пошли по пути добавления ОО-конструкций в существовавший не ОО-язык. Они известны как **гибридные языки (hybrid languages)** - см. [лекцию 17](#) курса "Основы объектно-ориентированного проектирования".

Техника интеграции, описанная выше, основывалась на внешних программах и ОО-перестройке. Это другой принцип: необходимость в совместимости **ПО** не означает перегрузку **языка** механизмами,ющими расходиться с принципами объектной технологии.

- Гибрид добавляет новый языковой уровень к существующему языку, например С. В результате сложность может ограничить привлекательность объектной технологии - простоту идей.
- Начинающие часто с трудом осваивают гибридный язык, поскольку для них неясно, что именно является ОО, а что досталось из прошлого.
- Старые механизмы могут быть несовместимыми, по крайней мере, с некоторыми аспектами ОО-идей. Есть много примеров несоответствий между системой типов языков С или Pascal и ОО-подходом.
- Не объектные механизмы часто конкурируют со своими аналогами. Например, C++ предлагает, наряду с динамическим связыванием, возможность динамического выбора, используя аппарат указателей функций. Это смущает неспециалиста, не понимающего, какой подход выбрать в данном случае. В результате, программный продукт, хотя и создан ОО-средой, по сути является реализацией на языке С, и не дает ожидаемого качества и производительности, дискредитируя объектную технологию.

Если целью является получение лучших программных продуктов и процесса их разработки, то компромисс на уровне языка кажется неправильным подходом. **Взаимодействие (Interfacing)** ОО-инструментария и приемов с достижениями прошлого и **смешивание (mixing)** различных уровней технологии - не одно и то же.

Можно привести пример из электроники. Конечно, полезно сочетать различные уровни технологии в одной системе, например, звуковой усилитель включает несколько диодов наряду с транзисторами и интегральными схемами. Но мало проку от компонента, который является полудиодом, полуторанзистором.

ОО-разработка должна обеспечивать совместимость с ПО, построенным на других подходах, но не за счет преимуществ и целостности метода. Этого и достигает **внешний** механизм: отдельные миры, каждый из которых состоятелен и имеет свои достоинства, и четкий интерфейс, обеспечивающий взаимодействие между ними.

Передача аргументов

Один из аспектов нотации требует разъяснений: что происходит со значениями, переданными в качестве аргументов подпрограмме?

Рассмотрим вызов в форме

$r(a_1, a_2, \dots, a_n)$

соответствующий программе

$r(x_1: T_1, x_2: T_2, \dots, x_n: T_n) \text{ is } \dots$

где r может быть как функцией, так и процедурой, и вызов может быть квалифицированным, как в $b.r(\dots)$. Выражения a_1, a_2, \dots, a_n называются фактическими аргументами, а x_1 - формальными. (Помните, что для родовых параметров типа остается термин "параметр".)

Встают важные вопросы: каково соответствие между фактическими и формальными аргументами? Какие операции допустимы над формальными аргументами? Каково их влияние на соответствующие фактические аргументы?

Ответ на первый вопрос: эффект связывания фактических - формальных аргументов таков же как соответствующего присваивания. Обе операции называются **присоединением (attachment)**. В предыдущем вызове можно считать, что запуск программы начинается с выполнения команд, неформально эквивалентных присваиванием:

$x_1 := a_1; x_2 := a_2; \dots x_n := a_n$

Ответ на второй вопрос: внутри тела программы любой формальный аргумент x защищен. Программа не может применять к нему прямых модификаций, таких как:

- Присваивание x значения в форме $x := \dots$
- Процедуры создания, где x является целью: **create x.make (...)**

Читатели, знакомые с механизмом передачи, известным как вызов по значению, поймут, что здесь ограничения более строгое: при вызове по значению формальные аргументы инициализируются значениями фактических, но затем могут быть целью любых операций.

Ответ на третий вопрос - что может программа делать с фактическими аргументами? - вытекает из того, что присоединение используется для задания семантики связывания формальных и фактических аргументов. Присоединение (см. [лекцию 8](#)) означает копирование либо ссылки, либо объекта. Это зависит от того, являются ли соответствующие типы развернутыми:

- Для ссылок (обычный случай) при передаче аргументов копируется ссылка, - **Void**, либо присоединенная к объекту.
- Для развернутых типов (включающих основные типы INTEGER, REAL и т.п.), при передаче аргументов копируется объект.

В первом случае, запрет операций прямой модификации означает, что нельзя модифицировать **ссылку (reference)** через повторное присоединение или создание. Но если ссылка не пустая, то разрешается модифицировать присоединенный **объект**.

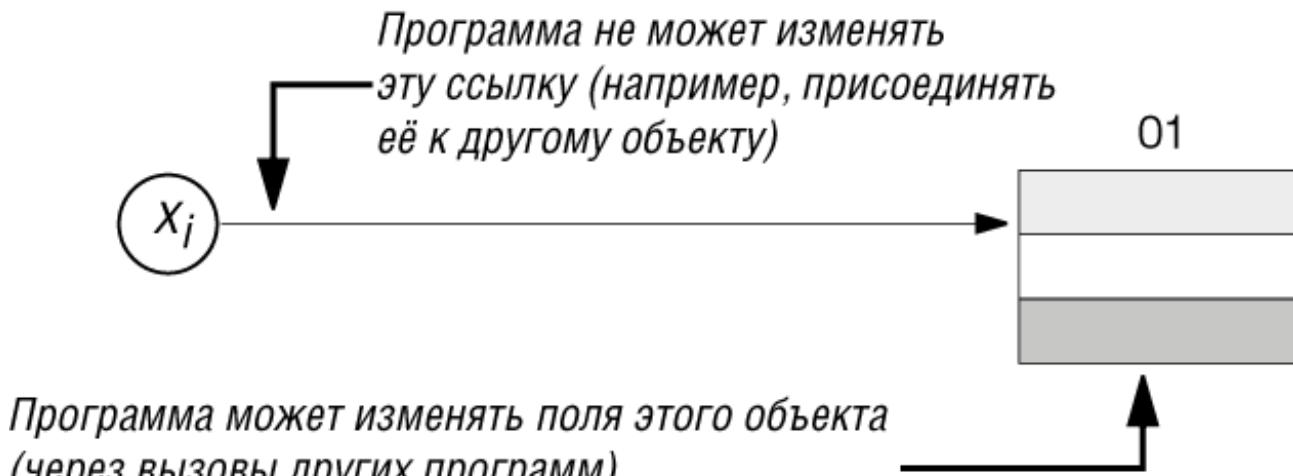


Рис. 13.1. Допустимые операции на аргументе ссылки

Если x_i - один из формальных аргументов r , то тело программы может содержать вызов:

$x_i.p(\dots)$

где p - процедура, применимая к x_i , (объявлена в базовом классе типа T_i аргумента x_i). Процедура может модифицировать поля объекта, присоединенного к x_i во время выполнения, то есть объекта, присоединенного к соответствующему фактическому аргументу a_i .

Вызов $q(a)$ никогда не может изменить значение a , если a развернутого типа и является объектом. Если же a является ссылкой, то ссылка не меняется, но объект, присоединенный к ней, может измениться в результате вызова.

Существует много причин, по которым не следует позволять программам прямую модификацию их аргументов. Одна из самых убедительных - **Конфликтующие присваивания**. Предположим, что язык допускает присваивания аргументам, и процедура¹⁾

```
don't_look_innocuous (a, b: INTEGER) is -- я выгляжу
  -- безвредной, но не стоит мне доверять.
  do
    a := 0; b := 1
  end
```

Теперь рассмотрим вызов $don't_look_innocuous(x, x)$. Каково значение x после возвращения: 0 или 1? Ответ зависит от того, как компилятор реализует изменения формальных - фактических аргументов при выходе программы. Это ставит в тупик не только программистов, использующих язык Fortran.

Разрешение программе изменять аргументы приводит к ограничениям на фактические аргументы. В этом случае он должен быть элементом, способным изменять свое значение, что допустимо для переменных, но не постоянных атрибутов (см. [лекцию 18](#)). Недопустимым фактическим аргументом становится сущность *Current*, выражения, такие как $a + b$. Устранение модификации аргументов позволяет избежать подобных ограничений и использовать любые выражения в качестве фактических аргументов.

Следствием этих правил является признание того, что только три способа допускают модификацию значения ссылки x : процедура создания **create** $x \dots$; присваивание $x := y$; и попытка присваивания $x := y$, обсуждаемая ниже. Передача x как фактического аргумента никогда не модифицирует x .

Это также означает, что процедура не возвращает ни одного результата, функция - официальный результат, представленный сущностью *Result*. Для получения нескольких результатов необходимо одно из двух:

- Использовать функцию, возвращающую объект с несколькими полями (обычно, возвращается ссылка на такой объект).
- Использовать процедуру, изменяющую поля объектов соответствующих атрибутов. Затем клиент может выполнять запросы к этим полям.

Первый прием уместен, когда речь идет о составном результате. Например, функция не может возвращать два значения, соответствующих заглавию и году публикации книги, но может возвращать одно значение типа *BOOK*, с атрибутами *title* и *publication_year*. В более общих ситуациях применяются процедуры. Эта техника будет обсуждаться вместе с вопросом побочных эффектов в разделе принципов модульного

проектирования²⁾.

Инструкции

ОО-нотация, разработанная в этой книге, императивна: вычисления специфицируются через команды (commands), также называемые инструкциями (instructions). (Мы избегаем обычно применимого термина оператор (предложение) (statement), поскольку в слове есть оттенок выражения, описывающего факты, а хотелось подчеркнуть императивный характер команды.)

Для имеющих опыт работы с современными языками инструкции выглядят как хорошие знакомые. Исключение составляют некоторые специальные свойства циклов, облегчающие их верификацию. Вот список инструкций: Вызов процедуры, Присваивание, Условие, Множественный выбор, Цикл, Проверка, Отладка, Повторное выполнение, Попытка присваивания.

Вызов процедуры

При вызове указывается имя подпрограммы, возможно, с фактическими аргументами. В инструкции вызова подпрограмма должна быть процедурой. Вызов функции является выражением. Хотя сейчас нас интересуют инструкции, следующие правила применимы в обоих случаях.

Вызов может быть квалифицированным или неквалифицированным. Для неквалифицированного вызова подпрограммы из включающего класса в качестве цели используется текущий экземпляр класса. Этот вызов имеет вид:

r (без аргументов), или

$r(x, y, \dots)$ (с аргументами)

Квалифицированный вызов явно называет свою цель, заданную некоторым выражением. Если a - выражение некоторого типа, C - базовый класс этого типа, $a \cdot q$ одна из программ C , то квалифицированный вызов имеет форму $a \cdot q$. Опять же, за q может следовать список фактических аргументов; a может быть неквалифицированным вызовом функции с аргументами, как в $r(m) \cdot q(n)$, где $r(m)$ - это цель. В качестве цели можно также использовать более сложное выражение при условии заключения его в скобки, как в $(vector1 + vector2).count$.

Также разрешаются квалифицированные вызовы с многоточием в форме: $a \cdot q_1 q_2 \dots q_n$, где a , так же, как и q_i , может включать список фактических аргументов.

Экспорт управляет применением квалифицированных вызовов. Напомним, что компонент f , объявленный в классе B , доступен в классе A (**экспортирован классу**), если предложение **feature**, объявляющее f , начинается с **feature** (без дальнейшего уточнения) или **feature {X, Y, ...}**, где один из элементов списка $\{X, Y, \dots\}$ является A или предком A . Имеет место:

Правило Квалифицированного Вызова

Квалифицированный вызов вида $b \cdot q_1 \cdot q_2 \dots q_n$, появляющийся в классе C корректен, только если он удовлетворяет следующим условиям:

1. Компонент, стоящий после первой точки, q_1 , должно быть доступен в классе C .
2. В вызове с многоточием, каждый компонент после второй точки, то есть, каждое q_i для $i > 1$, должен быть доступен в классе C .

Чтобы понять причину существования второго правила, отметим, что $a \cdot q \cdot r \cdot s$ - краткая запись для

$b := a \cdot q; c := b \cdot r; c \cdot s$

которая верна только, если q , r и s доступны классу C , в котором появляется этот фрагмент. Не имеет значения, доступно ли r базовому классу типа q , и доступно ли s базовому классу типа r .

Вызовы могут иметь инфиксную или префиксную форму. Выражение $a + b$, записанное в инфиксной форме, может быть переписано в префиксной форме: $a.plus(b)$. Для обеих форм действуют одинаковые правила применимости.

Присваивание (Assignment)

Инструкция присваивания записывается в виде:

```
x := e
```

где x - сущность, допускающая запись (writable), а e - выражение совместимого типа. Такая сущность может быть:

- неконстантным атрибутом включающего класса;
- локальной сущностью включающей подпрограммы. Для функции допустима сущность Result.

Сущности, не допускающие запись, включают константные атрибуты и формальные аргументы программы - которым, как мы видели, подпрограмма не может присваивать новое значение.

Создание (Creation)

Инструкция создания изучалась в предыдущих лекциях³ в двух ее формах: без процедуры создания, как в **create x**, и с процедурой создания, как в **create x.p (...)**. В обоих случаях x должна быть сущностью, допускающей запись.

Условная Инструкция (Conditional)

Эта инструкция задает различные формы обработки в зависимости от выполнения определенных условий.
Основная форма:

```
if boolean_expression then
    instruction; instruction; ...
else
    instruction; instruction; ...
end
```

где каждая ветвь может иметь произвольное число инструкций (а возможно и не иметь их).

Будут выполняться инструкции первой ветви, если boolean_expression верно, а иначе - второй ветви.
Можно опустить часть **else**, если второй список инструкций пуст, что дает:

```
if boolean_expression then
    instruction; instruction; ...
end
```

Когда есть более двух возможных случаев, можно избежать вложения (nesting) условных команд в частях **else**, используя одну или более ветвей **elseif**, как в:

```
if c1 then
    instruction; instruction; ...
elseif c2 then
    instruction; instruction; ...
elseif c3 then
    instruction; instruction; ...
...
else
    instruction; instruction; ...
end
```

где часть **else** остается факультативной. Это дает возможность избежать вложения

```
if c1 then
    instruction; instruction; ...
else
    if c2 then
        instruction; instruction; ...
    else
        if c3 then
            instruction; instruction; ...
    ...

```

```

else
    instruction; instruction; ...
end
end

```

Когда необходим множественный разбор случаев, более удобна инструкция множественного выбора **inspect**, обсуждаемая ниже.

ОО-метод, благодаря полиморфизму и динамическому связыванию, уменьшает необходимость явных условных инструкций и множественного выбора, поддерживая неявную форму выбора. Когда объект применяет некоторый компонент, имеющий несколько вариантов, то во время выполнения нужный вариант выбирается автоматически в соответствии с типом объекта. Этот неявный стиль выбора обычно предпочтительнее, но, конечно, инструкции явного выбора остаются необходимыми.

Множественный выбор

Инструкция множественного выбора (также известная, как инструкция Case) производит разбор вариантов, имеющих форму: $e = v_i$, где e - выражение, а v_i - константы того же типа. Хотя условная инструкция (`if e = v1 then ... elseif e = v2 then...`) работает, есть две причины, оправдывающие применение специальной инструкции, что является исключением из обычного правила: "если нотация дает хороший способ сделать что-то, нет необходимости вводить другой способ". Вот эти причины:

- Разбор случаев настолько распространен, что заслуживает особого синтаксиса, увеличивающего ясность, позволяя избежать бесполезного повторения "`e =`".
- Компиляторы могут использовать особенно эффективную технику реализации, - таблицу переходов (**jump table**), - неприменимую к общим условным инструкциям и избегающую явных проверок.

Что касается типа анализируемых величин (тип e и v_i), то инструкции множественного выбора достаточно поддерживать только целые и булевые значения. Согласно правилу, они фактически должны объявляться либо все как INTEGER, либо как CHARACTER. Общая форма инструкции такова:

```

inspect
  e
when v1 then
  instruction; instruction; ...
when v2 then
  instruction; instruction; ...
...
else
  instruction; instruction; ...
end

```

Все значения v_i должны быть различными; часть **else** facultativna; каждая из ветвей может иметь произвольное число инструкций или не иметь их.

Инструкция действует так: если значение e равно значению v_i (это может быть только для одного из них), выполняются инструкции соответствующей ветви; иначе, выполняются инструкции в ветви **else**, если они есть.

Если отсутствует **else**, и значение e не соответствует ни одному v_i , то возникает исключительная ситуация ("Некорректно проверяемое значение"). Это решение может вызвать удивление, поскольку соответствующая условная инструкция в этом случае ничего не делает. Но оно характеризует специфику инструкции множественного выбора. Когда вы пишете **inspect** с набором значений v_i , нужно включить ветвь **else**, даже пустую, если вы понимаете, что во время выполнения значения e могут не соответствовать никаким v_i . Если вы не включаете **else**, то это эквивалентно явному утверждению: "значение e всегда является одним из v_i ". Проверяя это утверждение и создавая исключительную ситуацию при его нарушении, реализация оказывает нам услугу. Бездействие в данной ситуации - означает ошибку - в любом случае, ее необходимо устраниć как можно раньше.

Одно из частых приложений инструкции множественного выбора - анализ символа, введенного пользователем⁴:

```

inspect
  first_input_letter

```

```

when 'D' then
    "Удалить строку"
when 'I' then
    "Вставить строку"
...
else
    message ("Неопознанная команда; введите H для получения справки")
end

```

Когда значения v_i целые, то они могут быть определены как уникальные (unique values), концепция которых рассмотрена в следующей лекции. Это делает возможным в объявлении определить несколько абстрактных констант, например, Do, Re, Mi, Fa, Sol, La, Si: INTEGER is unique, и затем анализировать их в инструкции: inspect note when Do then...when Re then...end.

Как и условные инструкции, инструкции множественного выбора не должны использоваться для замены неявного выбора, основанного на динамическом связывании.

Циклы

Синтаксис циклов описан при обсуждении Проектирования по Контракту ([лекция 11](#)):

```

from
    initialization_instructions
invariant
    invariant
variant
    variant
until
    exit_condition
loop
    loop_instructions
end

```

Предложения **invariant** и **variant** факультативны. Предложение **from** требуется, хотя и может быть пустым. Оно задает инициализацию параметров цикла. Не рассматривая сейчас факультативные предложения, выполнение цикла можно описать следующим образом. Вначале происходит инициализация, и выполняются **initialization_instructions**. Затем следует "циклический процесс", определяемый так: если **exit_condition** верно, то циклический процесс - пустая инструкция (**null instruction**); если условие неверно, то циклический процесс - это выполнение **loop_instructions**, затем следует (рекурсивно) повторение циклического процесса.

Проверка

Инструкция проверки рассматривалась при обсуждении утверждений ([лекция 11](#)). Она говорит, что определенные утверждения должны удовлетворяться в определенных точках:

```

check
    assertion -- Одно или больше предложений
end

```

Отладка

Инструкция отладки является средством условной компиляции. Она записывается так:

```
debug instruction; instruction; ... end
```

В файле управления (Асе-файле) для каждого класса можно включить или отключить параметр **debug**. При его включении все инструкции отладки данного класса выполняются, при отключении - они не влияют на выполнение.

Эту инструкцию можно использовать для включения специальных действий, выполняющихся только в режиме отладки, например, печати некоторых величин.

Повторение вычислений

Инструкция повторного выполнения рассматривалась при обсуждении исключительных ситуаций ([лекция 12](#)). Она появляется только в предложении `rescue`, повторно запуская тело подпрограммы, работа которой была прервана.

Выражения

Выражение задает вычисление, вырабатывающее значение, - объект или ссылку на объект. Выражениями являются:

- неименованные (манифестные) константы;
- сущности (атрибуты, локальные сущности, формальные аргументы, `Result`);
- вызовы функций;
- выражения с операторами (технически - это специальный случай вызова функций);
- `Current`.

Манифестные константы

Неименованная или манифестная константа задается значением, синтаксис которого позволяет определить и тип этого значения, например, целое 0. Этим она отличается от символьной константы, чье имя не зависит от значения.

Булевых констант две, - `True` и `False`. Целые константы имеют обычную форму, например:

`453 -678 +66623`

В записи вещественных (real) констант присутствует десятичная точка. Целая, либо дробная часть может отсутствовать. Может присутствовать знак и экспонента, например:

`52.5 -54.44 +45.01 .983 -897. 999.e12`

Символьные константы состоят из одного символа в одинарных кавычках, например, `'A'`. Для цепочек из нескольких символов используется библиотечный класс `STRING`, описанный ниже.

Вызовы функций

Вызовы функций имеют такой же синтаксис, как и вызовы процедур. Они могут быть квалифицированные и неквалифицированные: в первом случае используется нотация с многоточием. При соответствующих объявлениях класса и функций, они, например, таковы:

`b.f`
`b.g(x, y, ...)`
`b.h(u, v).i.j(x, y, ...)`

Правило квалифицированного вызова, приведенное для процедур, применимо также к вызовам функций.

Текущий объект

Зарезервированное слово `Current` означает текущий экземпляр класса и может использоваться в выражении. Само `Current` - тоже выражение, а не сущность, допускающая запись. Значит присваивание `Current`, например, `Current := some_value` будет синтаксически неверным.

При ссылке на компонент (атрибут или программу) текущего экземпляра нет необходимости писать `Current.f`, достаточно написать `f`. Поэтому `Current` используется реже, чем в ОО-языках, где каждая ссылка на компонент должна быть явно квалифицированной. (Например, в Smalltalk компонент всегда квалифицирован, даже когда он применим к текущему экземпляру.) Случай, когда надо явно называть `Current` включают:

- Передачу текущего экземпляра в качестве аргумента в программу, как в `a.f (Current)`. Обычное применение - создание копии (`duplicate`) текущего экземпляра, как в `x := clone (Current)`.
- Проверку, - присоединена ли ссылка к текущему экземпляру, как в проверке `x = Current`.
- Использование `Current` в качестве опорного элемента в "закрепленном объявлении" в форме `like Current` ([лекция 16](#)).

Выражения с операторами

Выражения могут включать знаки операций или операторы.

Унарные операторы + и - применяются к целым и вещественным выражениям и **не** применяются к булевым выражениям.

Бинарные операторы, имеющие точно два операнда, включают операторы отношения:

= /= < > <= >=

где /= означает "не равно". Значение отношения имеет булев тип.

Выражения могут включать один или несколько operandов, соединенных операторами. Численные operandы могут соединяться следующими операторами:

+ - . / ^ // \\
где // целочисленное деление, \\ целый остаток, а ^ степень (возведение в степень).

Булевые operandы могут соединяться операторами: **and**, **or**, **xor**, **and then**, **or else**, **implies**. Последние три объясняются в следующем разделе; **xor** - исключающее или.

Предшествование операторов, основанное на соглашениях обычной математики, строится по "Принципу Наименьшей Неожиданности". Во избежание неопределенности и путаницы, в книге используются скобки, даже там, где они не очень нужны.

Нестрогие булевые операторы

Операторы **and then** и **or else** (названия заимствованы из языка Ada), а также **implies** не коммутативны и называются **нестрогими** (non-strict) булевыми операторами. Их семантика следующая:

Нестрогие булевые операторы

- **a and then b** ложно, если a ложно, иначе имеет значение b.
- **a or else b** истинно, если a истинно, иначе имеет значение b.
- **a implies b** имеет то же значение, что и: **(not a) or else b**.

Первые два определения, как может показаться, дают ту же семантику, что и **and** и **or**. Но разница выявляется, когда b не определено. В этом случае выражения, использующие стандартные булевые операторы, математически не определены, но данные выше определения дают результат: если a ложно, то **a and then b** ложно независимо от b; а если a истинно, то **a and then b** истинно независимо от b. Аналогично, **a implies b** истинно, если a ложно, даже если b не определено.

Итак, нестрогие операторы могут давать результат, когда стандартные не дают его. Типичный пример:

$(i /= 0) \text{ and then } (j // i = k)$

которое, согласно определению, ложно, если i равно 0. Если бы в выражении использовался **and**, а не **and then**, то из-за неопределенности второго операнда при i равном 0 статус выражения неясен. Эта неопределенность скажется во время выполнения:

1. Если компилятор создает код, вычисляющий оба операнда, то во время выполнения произойдет деление на ноль, и возникнет исключительная ситуация.
2. Если же генерируется код, вычисляющий второй operand только тогда, когда первый истинен, то при i равном 0 возвратится значение ложь.

Для гарантии интерпретации (2), используйте **and then**. Аналогично,

$(i = 0) \text{ or else } (j // i = k)$

истинно, если i равно 0, а вариант **or** может дать ошибку во время выполнения.

Можно недоумевать, почему необходимы два новых оператора - не проще и не надежнее ли просто

поддерживать стандарт операторов **and** и **or** и принимать, что они означают **and then** и **or else**? Это не изменило бы значение булева выражения, когда оба оператора определены, но расширило бы круг случаев, где выражения могут получать непротиворечивое значение. Именно так некоторые языки программирования, в частности, ALGOL, W и C, интерпретируют булевые операторы. Однако есть теоретические и практические причины сохранять два набора различных операторов.

- С точки зрения теории, стандартные математические булевые операторы коммутативны: **a and b** всегда имеет значение такое же, как **b and a**, в то время как **a and then b** может быть определенным, когда **b and then a** не определено. Когда порядок operandов не имеет значения, предпочтительно использовать коммутативный оператор.
- С точки зрения практики, некоторые оптимизации компилятора становятся невозможными, если требуется, чтобы компилятор вычислял operandы в заданном выражением порядке, как в случае с некоммутативными операторами. Поэтому лучше использовать стандартные операторы, если известно, что оба operandы определены.

Отметим, что можно смоделировать нестрогие операторы посредством условных команд на языке, не включающем такие операторы. Например, вместо

```
b := ((i /= 0) and then (j // i = k))
```

можно написать

```
if i = 0 then b := false else b := (j // i = k) end
```

Нестрогая форма, конечно, проще. Это особенно ясно, когда она используется как условие выхода из цикла:

```
from
  i := a.lower
invariant
  -- Для всех элементов из интервала [a.lower .. i - 1], (a @ i) /= x
variant
  a.upper - i
until
  i > a.upper or else (a @ i = x)
loop
  i := i + 1
end;
Result := (i <= a.upper)
```

Цель - сделать **Result** верным, если и только если значение **x** находится в массиве **a**. Использование **or** здесь будет неверным. В этом случае всегда могут вычисляться два operandы, так что при истинности первого операнда (**i > a.upper**) произойдет попытка доступа к несуществующему элементу массива **a(@aupper+1)**, что приведет к ошибке во время выполнения (нарушение предусловия при включенной проверке утверждений).

Решение без нестрогих операторов будет неэлегантным.

Другой пример - утверждение, например, инварианта класса, выражающее, что первое значение списка **l** целых неотрицательно, при условии, что список непустой:

```
l.empty or else l.first >= 0
```

При использовании **or** инвариант был бы некорректен. Здесь нет способа написать условие без нестрогих операторов (кроме написания специальной функции и вызова ее в утверждении). Базовые библиотеки алгоритмов и структур данных содержат много таких случаев.

Оператор **implies**, описывающий включения, также нестрогий. Форма **implies** менее привычна, но часто более ясна, например, последний пример выглядит лучше в записи:

```
(not l.empty) implies (l.first >= 0)
```

Строки

Класс **STRING** описывает символьные строки. Он имеет специальный статус, поскольку нотация допускает

манифестные строковые константы, обозначающие экземпляры STRING.

Строковая константа записывается в двойных кавычках, например,

```
"ABcd Ef ~*_ 01"
```

Символ двойных кавычек должны предваряться знаком %, если он появляется как один из символов строки.

Неконстантные строки также являются экземплярами класса STRING, чья процедура создания make принимает в качестве аргумента ожидаемую начальную длину строки, так что

```
text1, text2: STRING; n: INTEGER;  
...  
create text1.make (n)
```

динамически размещает строку text1, резервируя пространство для n символов. Заметим, что n - только исходный размер, не максимальный. Любая строка может увеличиваться и сжиматься до произвольного размера.

На экземплярах STRING доступны многочисленные операции: сцепление, выделение символов и подстрок, сравнение и т.д. (Они могут изменять размер строки, автоматически запуская повторное размещение, если размер строки становится больше текущего.)

Присваивание строк означает разделение (sharing): после text2 := text1, любая модификация text1 модифицирует text2, и наоборот. Для получения копии строки, а не копии ссылки, используется клонирование text2 := clone (text1).

Константную строку можно объявить как атрибут:

```
message: STRING is "Your message here"
```

Ввод и вывод

Два класса библиотеки KERNEL обеспечивают основные средства ввода и вывода: FILE и STD_FILES.

Среди операций, определенных для объекта f типа FILE, есть следующие:

```
create f.make ("name") -- Связывает f с файлом по имени name.  
f.open_write -- Открытие f для записи  
f.open_read -- Открытие f для чтения  
f.put_string ("A_STRING") -- Запись данной строки в файл f
```

Операции ввода-вывода стандартных файлов ввода, вывода и ошибок, можно наследовать из класса STD_FILES, определяющего компоненты input, output и error. В качестве альтернативы можно использовать предопределенное значение io, как в io.put_string ("ABC"), обходя наследование.

Лексические соглашения

Идентификатор - это последовательность из символа подчеркивания, буквенных и цифровых символов, начинающаяся с буквы. Нет ограничений на длину идентификатора, что позволяет сделать ясными имена компонентов и классов.

Регистр в идентификаторах не учитывается, так что Hi, hi, HI и hI - все означают один и тот же идентификатор. Было бы опасным позволять двум идентификаторам, различающимся только одним символом, скажем Structure и structure, обозначать различные элементы. Лучше попросить разработчиков включить воображение, чем рисковать возникновением ошибок.

Нотация включает набор точных стандартных соглашений по стилю (см. [лекцию 26](#) курса "Основы объектно-ориентированного проектирования"): имена классов (INTEGER, POINT ...) и формальные родовые параметры (G в LIST [G]) записываются в верхнем регистре; предопределенные сущности и выражения (Result, Current...) и константные атрибуты (Pi) начинаются с буквы верхнего регистра и продолжаются в нижнем

регистре. Все другие идентификаторы (неконстантные атрибуты, формальные аргументы программ, локальные сущности) - в нижнем регистре. Хотя компиляторы не проверяют эти соглашения, не являющиеся частью спецификации, они важны для удобочитаемости текстов программных продуктов и последовательно применяются в библиотеках и текстах этой книги.

Ключевые концепции

- Внешние программы доступны через хорошо определенный интерфейс.
- Объектная технология может служить в качестве механизма упаковки наследуемого ПО.
- Подпрограммы не могут модифицировать свои аргументы, хотя они могут изменять **объекты**, связанные с этими аргументами.
- Нотация включает небольшой набор инструкций: присваивания, выбора, цикла, вызова, отладки и проверки.
- Выражения следуют общепринятому стилю. `Current` - выражение, обозначающее текущий экземпляр. Не будучи сущностью, `Current` не может быть целью присваивания.
- Нестрогие булевы операторы эквивалентны стандартным булевым оператором, когда определены оба операнда, но могут быть определенными в случаях, когда стандартные операторы не определены.
- Строки, ввод и вывод определяются простыми библиотечными классами.
- Регистр незначим в идентификаторах, хотя правила стиля включают рекомендуемые соглашения по записи имен.

Упражнения

У13.1 Внешние классы

При обсуждении интеграции внешнего не объектного ПО с объектной системой отмечалось, что компоненты являются тем уровнем, на котором нужно осуществлять интеграцию. Когда же речь идет об интеграции с ПО, созданным на другом объектном языке, уровнем интеграции могут быть классы. Рассмотрите понятие "внешнего класса" как дополнение к нотации книги.

У13.2 Избегая нестрогих операторов

Напишите цикл для поиска элемента `x` в массиве `a`, подобный алгоритму в этой лекции, но не использующий нестрогих операторов.

-
- ¹⁾ ПРЕДУПРЕЖДЕНИЕ: некорректный текст программы. Только для целей иллюстрации.
- ²⁾ См. [лекцию 5](#) курса "Основы объектно-ориентированного проектирования", особенно "Схема, основанная на опыте".
- ³⁾ См. "Инструкция создания" и "Процедуры создания", [лекцию 8](#). Один из вариантов рассмотрен в "Полиморфное создание", [лекция 14](#).
- ⁴⁾ Это элементарная схема. О более сложных технических приемах обработки пользовательских команд см. [лекцию 3](#) курса "Основы объектно-ориентированного проектирования".

Основы объектно-ориентированного программирования

14. Лекция: Введение в наследование

Интересные системы редко рождаются на пустом месте. Почти всегда новые программы являются расширениями предыдущих разработок, лучший способ создания нового - это подражание старым образцам, их уточнение и комбинирование. Традиционные методы проектирования по большей части не уделяли внимания этому аспекту разработки. В ОО-технологии он является весьма существенным. Ранее изученные приемы явно недостаточны. Классы, конечно, дают способ хорошей декомпозиции на модули и обладают многими качествами, ожидаемыми от повторно используемых компонентов: они являются однородными, согласованными модулями; в соответствии с принципом Скрытия информации можно легко отделять интерфейсы от реализаций; универсальность придает им определенную гибкость, а благодаря утверждению, можно точно задавать их семантику. Но для достижения повторного использования и расширяемости нужно нечто большее. Всякий комплексный подход, обеспечивающий повторное использование, должен столкнуться с проблемой повторяемости (*repetition*) и изменчивости (*variation*), проанализированной в одной из предыдущих лекций (см. [лекцию 4](#)). Для устранения многократного переписывания одного и того же кода, ведущего к потерям в времени, появлению противоречий и ошибок, нужны методы, улавливающие поразительную общность, присущую многим группам однотипных структур - в сем текстовым редакторам, во всем таблицам, во всем программам обработки файлов, - учитывая при этом многие различия в характеристиках конкретных случаев. При обеспечении расширяемости (*extendibility*) преимущество описанной выше системы типов состоит в гарантированной совместности во время компиляции, но она запрещает многие вполне законные комбинации элементов. Например, нельзя объявить массив, содержащий геометрические объекты различных совместных типов, таких как `POINT` (ТОЧКА) и `SEGMENT`(ОТРЕЗОК). Чтобы достичь прогресса в повторном использовании или в расширяемости требуется в оспользоваться преимуществами концептуальных отношений между классами: один класс может быть расширением, специализацией или комбинацией других классов. Метод и язык должны поддерживать запись и использование этих отношений. Эта поддержку выполняет наследование. Центральная и восхитительная составляющая объектной технологии - отношение наследования - потребует для полного освоения нескольких лекций. В данной лекции рассматриваются фундаментальные понятия. В трех следующих описываются более специальные аспекты: множественное наследование, переименование, субконтракты, влияние на систему типов. Лекция 6 курса "Основы объектно-ориентированного проектирования" дополнит эти технические рассмотрения, рассмотрев методологическую перспективу: как использовать наследование и как избежать его неверного применения.

Многоугольники и прямоугольники

Для объяснения основных понятий рассмотрим простой пример. Здесь приведен скорее набросок этого примера, а не полный его вариант, но он хорошо показывает все существенные идеи.

Многоугольники

Предположим, что требуется построить графическую библиотеку. Ее классы будут описывать геометрические абстракции: точки, отрезки, векторы, круги, эллипсы, многоугольники, треугольники, прямоугольники, квадраты и т. п.

Рассмотрим вначале класс, описывающий многоугольники. Операции будут включать вычисление периметра, параллельный перенос и вращение. Этот класс может выглядеть так:

```
indexing
  description: "Многоугольники с произвольным числом вершин"
class POLYGON creation
  ...
feature -- Доступ
  count: INTEGER
    -- Число вершин
  perimeter: REAL is
    -- Длина периметра
    do ... end
feature -- Преобразование
  display is
    -- Вывод многоугольника на экран.
    do ... end
  rotate (center: POINT; angle: REAL) is
    -- Поворот на угол angle вокруг точки center.
    do
      ... См. далее ...
    end
  translate (a, b: REAL) is
    -- Сдвиг на a по горизонтали, на b по вертикали.
    do ... end
  ... Объявления других компонентов ...
feature {NONE} -- Реализация
  vertices: LINKED_LIST [POINT]
    -- Список вершин многоугольника
invariant
  same_count_as_implementation: count = vertices.count
  at_least_three: count >= 3
    -- У многоугольника не менее трех вершин (см. упражнение У14.2)
end
```

Атрибут `vertices` задает список вершин, выбор линейного списка - это лишь одно из возможных представлений (массив мог бы оказаться лучше).

Приведем реализацию типичной процедуры `rotate`. Эта процедура осуществляет поворот на заданный угол вокруг заданного центра поворота. Для поворота многоугольника достаточно повернуть по очереди каждую его вершину.

```
rotate (center: POINT; angle: REAL) is
    -- Поворот вокруг точки center на угол angle.
    do
        from
            vertices.start
        until
            vertices.after
        loop
            vertices.item.rotate (center, angle)
            vertices.forth
        end
    end
```

Чтобы понять эту процедуру заметим, что компонент `item` из `LINKED_LIST` возвращает значение текущего элемента списка. Поскольку `vertices` имеют тип `LINKED_LIST [POINT]`, то `vertices.item` обозначает точку, к которой можно применить процедуру поворота `rotate`, определенную для класса `POINT` в предыдущей лекции. Это вполне корректно и достаточно общепринято - давать одно и то же имя (в данном случае `rotate`), компонентам разных классов, поскольку результирующее множество каждого из них имеет свой явно определенный тип. (Это ОО-форма перегрузки.)

Более важна для наших целей процедура вычисления периметра многоугольника. Единственный способ вычислить периметр многоугольника - это в цикле пройти по всем его вершинам и просуммировать длины всех ребер. Вот возможная реализация процедуры `perimeter`:

```
perimeter: REAL is
    -- Сумма длин ребер
    local
        this, previous: POINT
    do
        from
            vertices.start; this := vertices.item
            check not vertices.after end -- Следствие условия at_least_three
        until
            vertices.is_last
        loop
            previous := this
            vertices.forth
            this := vertices.item
            Result := Result + this.distance (previous)
        end
        Result := Result + this.distance (vertices.first)
    end
```

В этом цикле просто последовательно складываются расстояния между соседними вершинами. Функция `distance` была определена в классе `POINT`. Значение `Result`, возвращаемое этой функцией, при инициализации получает значение 0. Из класса `LINKED_LIST` используются следующие компоненты: `first` дает первый элемент списка, `start` сдвигает курсор, на этот первый элемент, `forth` передвигает его на следующий, `item` выдает значение элемента под курсором, `is_last` определяет, является ли текущий элемент последним, `after` узнает, что курсор оказался за последним элементом. Как указано в команде `check` инвариант `at_least_three` обеспечивает правильное начало и завершение цикла. Он стартует в состоянии `not after`, в котором элемент `vertices.item` определен. Допустимо применение `forth` один или более раз, что, в конце концов, приведет в состояние, удовлетворяющее условию выхода из цикла `is_last`.

Прямоугольники

Предположим теперь, что нам требуется новый класс, представляющий прямоугольники. Можно было бы начать его проектировать заново. Но прямоугольники это специальный вид многоугольников и у них много общих компонент: их также можно сдвигать, поворачивать и выводить на экран. С другой стороны, у них есть ряд специфических компонентов (например, диагонали), специальные свойства (число вершин равно четырем, а углы являются прямыми) и возможны специальные варианты некоторых операций (вычисление периметра можно устроить проще, чем в приведенном выше алгоритме).

Преимущества такой смеси общих и специфических компонентов можно использовать, определив класс `RECTANGLE`

как **наследника (heir)** класса POLYGON. При этом все компоненты класса POLYGON, называемого **родителем (parent)** класса RECTANGLE, по умолчанию будут применимы и к классу-наследнику. Для этого достаточно включить в RECTANGLE **предложение наследования (inheritance clause)**:

```
class RECTANGLE inherit
    POLYGON
feature
    ... Компоненты, специфичные для прямоугольников ...
end
```

В предложении **feature** класса-наследника компоненты родителя не повторяются: они автоматически доступны благодаря предложению о наследовании. В нем будут указаны лишь компоненты, специфичные для наследника. Это могут быть новые компоненты, такие как `diagonal`, а также переопределяемые наследуемые компоненты.

Вторая возможность полезна для такого компонента, который уже имелся у родителя, но у наследника должен быть описан в другом виде. Рассмотрим периметр `perimeter`. Для прямоугольников его можно вычислить более эффективно: не нужно вычислять четыре длины сторон, достаточно удвоить сумму длин двух сторон. Наследник, переопределяющий некоторый компонент родителя, должен объявить об этом в предложении наследования, включив предложение **redefine**:

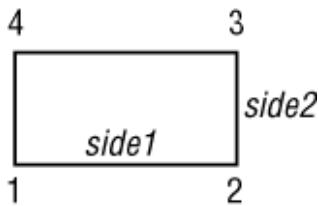
```
class RECTANGLE inherit
    POLYGON
        redefine perimeter end
feature
    ...
end
```

Это позволяет включить в предложение **feature** класса RECTANGLE новую версию компонента `perimeter`, которая заменит его версию из класса POLYGON. Если не включить объявление **redefine**, то новое объявление компонента `perimeter` среди других компонентов класса RECTANGLE приведет к ошибке, поскольку у RECTANGLE уже есть компонент `perimeter`, унаследованный от POLYGON, т.е. у некоторого компонента окажется два определения.

Класс RECTANGLE выглядит следующим образом:

```
indexing
    description: "Прямоугольники, - специальный случай многоугольников"
class RECTANGLE inherit
    POLYGON
        redefine perimeter end
creation
    make
feature -- Инициализация
    make (center: POINT; s1, s2, angle: REAL) is
        -- Установить центр прямоугольника в center, длины сторон
        -- s1 и s2 и ориентацию angle.
        do ... end
feature -- Access
    side1, side2: REAL
        -- Длины двух сторон
    diagonal: REAL
        -- Длина диагонали
    perimeter: REAL is
        -- Сумма длин сторон
        -- (Переопределение версии из POLYGON)
        do
            Result := 2 S (side1 + side2)
        end
invariant
    four_sides: count = 4

first_side: (vertices.i_th (1)).distance (vertices.i_th (2)) = side1
    second_side: (vertices.i_th (2)).distance (vertices.i_th (3)) = side2
    third_side: (vertices.i_th (3)).distance (vertices.i_th (4)) = side1
    fourth_side: (vertices.i_th (4)).distance (vertices.i_th (1)) = side2
end
```



Для списка `i_th(i)` дает элемент в позиции `i` (`i`-й элемент, следовательно это имя запроса).

Так как RECTANGLE является наследником класса POLYGON, то все компоненты родительского класса применимы и к новому классу: `vertices`, `rotate`, `translate`, `perimeter` (в переопределенном виде) и все остальные. Их не нужно повторять в определении нового класса.

Этот процесс транзитивен: всякий класс, будучи наследником RECTANGLE, например, SQUARE, также обладает всеми компонентами класса POLYGON.

Основные соглашения и терминология

Кроме терминов "наследник" и "родитель" будут полезны следующие термины:

Терминология наследования

Потомок класса C - это любой класс, который наследует C явно или неявно, включая и сам класс C. (Формально, это либо C, либо, по рекурсии, потомок некоторого наследника C).

Собственный потомок класса C - это потомок, отличный от самого C.

Предок C - это такой класс A, для которого C является потомком. **Собственный предок C** - это такой класс A, для которого C является собственным потомком.

В литературе также встречаются термины "подкласс" и "суперкласс", но мы не будем их использовать из-за неоднозначности.

Имеется также терминология для компонентов класса: компонент либо является **наследуемым** (перешедшим от некоторого собственного предка), либо **непосредственным** (введенным в данном классе).

При графическом представлении структур ОО-ПО, в котором классы изображаются эллипсами, связи по отношению наследования показываются в виде одинарных стрелок. Тем самым они отличаются от связей по отношению "быть клиентом", которые представляются двойными стрелками.

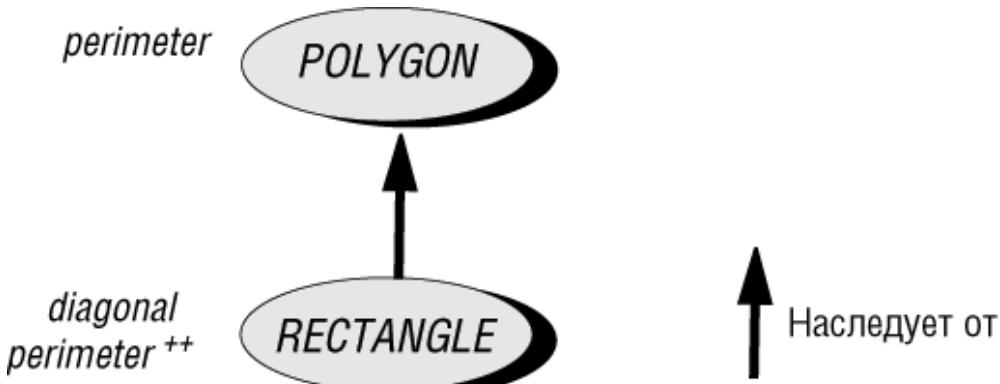


Рис. 14.1. Связь по наследованию

Переопределяемый компонент отмечается `++` (это соглашение принято в Business Object Notation (B.O.N.)).

Стрелка указывает вверх от наследника к родителю. Это соглашение легко запомнить - оно представляет отношение "наследовать от". В литературе встречается и обратное направление таких стрелок. Хотя обычно выбор графического представления является делом вкуса, в данном случае, одно из них явно лучше другого, поскольку одно наводит на мысль о правильном отношении, а другое может привести к путанице. Стрелка - это не просто произвольная пиктограмма, она указывает на одностороннюю связь между своими двумя концами. В данном случае:

- Всякий экземпляр наследника можно рассматривать как экземпляр родителя, а обратное неверно.
- В тексте наследника всегда упоминается его родитель, но не наоборот. Это, на самом деле, является важным свойством ОО-метода, вытекающим из принципа Открыт-Закрыт, согласно которому класс не "знает" списка своих наследников и других собственных потомков.

Хотя у нас нет жесткого правила, определяющего для достаточно сложных систем размещение классов на диаграммах наследования, мы будем, по возможности, помещать класс выше его наследника.

Наследование инварианта

Хотелось бы указать инвариант класса RECTANGLE, который говорил бы, что число сторон прямоугольника равно четырем и что длины сторон последовательно равны side1, side2, side1 и side2.

У класса POLYGON также имеется инвариант, который применим и к его наследнику:

Правило наследования инварианта

Инвариант класса является конъюнкцией утверждений из его раздела **invariant** и свойств инвариантов его родителей (если таковые имеются).

Поскольку у родителей класса могут быть свои родители, то это правило рекурсивно: в результате полный инвариант класса получается как конъюнкция собственного инварианта и инвариантов классов всех его предков.

Это правило отражает одну из важных характеристик наследования: сказать, что В наследует А - это утверждать, что каждый экземпляр В является также экземпляром А. Вследствие этого всякое выраженное инвариантом ограничение целостности, применимое к экземплярам А, будет также применимо и к экземплярам В.

В нашем примере второе предложение (`at_least_three`) инварианта POLYGON утверждает, что число сторон должно быть не менее трех, оно является следствием предложения `four_sides` из инварианта класса RECTANGLE, которое требует, чтобы сторон было ровно четыре.

Наследование и конструкторы

Ранее не показанная процедура создания (конструктор) для класса POLYGON может иметь вид

```
make_polygon (vl: LINKED_LIST [POINT]) is
    -- Создание по вершинам из vl.
    require
        vl.count >= 3
    do
        ...Инициализация представления многоугольника по элементам из vl ...
    ensure
        -- vertices и vl состоят из одинаковых элементов (это можно выразить
формально)
    end
```

Эта процедура берет список точек, содержащий по крайней мере три элемента, и использует его для создания многоугольника.

Ей дано собственное имя `make_polygon`, чтобы избежать конфликта имен при ее наследовании классом RECTANGLE, у которого имеется собственная процедура создания `make`. Мы не рекомендуем так делать в общем случае, в следующей лекции будет показано, как давать процедуре создания класса POLYGON стандартное имя `make`, а затем использовать переименование в предложении о наследовании класса RECTANGLE, чтобы предотвратить коллизию имен.

Приведенная выше процедура создания класса RECTANGLE имеет четыре аргумента: точку, служащую центром, длины двух сторон и ориентацию. Отметим, что компонент `vertices` применим к прямоугольникам, поэтому процедура создания для RECTANGLE создает список вершин `vertices` (четыре угла вычисляются по центру, длинам сторон и ориентации).

Общая процедура создания для многоугольников не удобна прямоугольникам, так как приемлемы только списки из четырех элементов, удовлетворяющих инварианту класса RECTANGLE. Процедура создания для прямоугольников, в свою очередь, не годится для произвольных многоугольников. Это обычное дело: процедура создания родителя не подходит для наследника. Нельзя гарантировать, что она будет удовлетворять его новому инварианту.

Например, если у наследника имеются новые атрибуты, то процедуре создания нужно будет их инициализировать, для чего потребуются дополнительные аргументы. Отсюда общее правило:

Правило наследования конструктора

При наследовании свойство процедуры быть конструктором не сохраняется.

Наследуемая процедура создания все еще доступна в наследнике, как и любой другой компонент родителя, но она не сохраняет статус конструктора. Этим статусом обладают только процедуры, перечисленные в предложении `creation` наследника.

В некоторых случаях родительский конструктор подходит и для наследника. Тогда его просто нужно указать в предложении `creation`:

```
class B inherit
  A
creation
  make
feature
  ...
```

где процедура `make` наследуется без изменений от класса `A`, у которого она также указана в предложении `creation`.

Пример иерархии

В конце обсуждения полезно рассмотреть пример `POLYGON-RECTANGLE` в контексте более общей иерархии типов геометрических фигур.

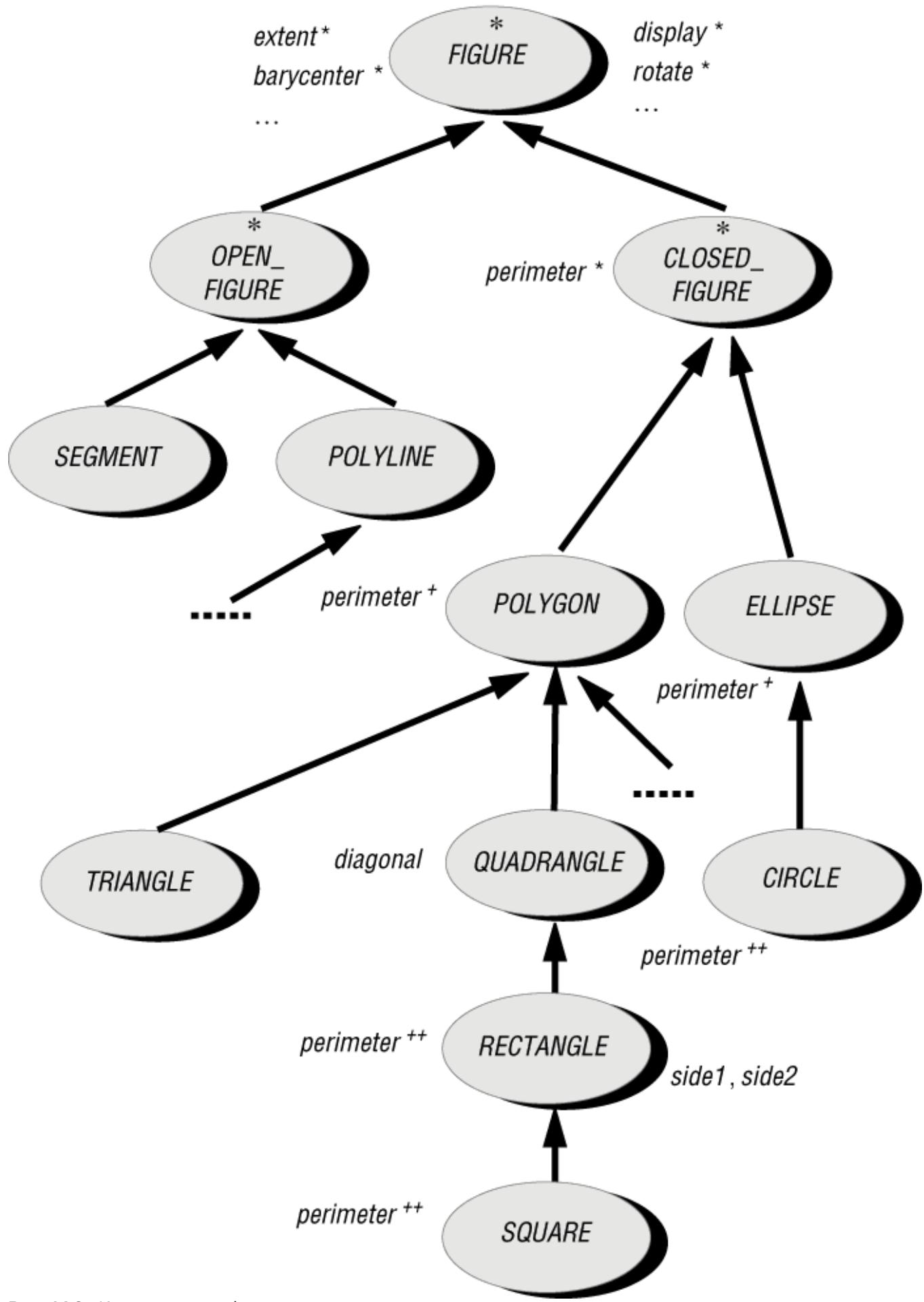


Рис. 14.2. Иерархия типов фигур

Фигуры разбиты на замкнутые и незамкнутые. Примером замкнутой фигуры кроме многоугольника является также эллипс, а частным случаем эллипса является круг.

Рядом с классами указаны их разные компоненты. Символ "++" означает "переопределено", а символы "+" и "*"

будут объяснены далее.

Ранее для простоты RECTANGLE был наследником класса POLYGON. Поскольку указанная классификация основана на числе вершин, то представляется разумным ввести промежуточный класс QUADRANGLE для четырехугольников на том же уровне, что и классы TRIANGLE, PENTAGON и т. п. Тогда компонент diagonal (диагональ) можно переместить на уровень класса QUADRANGLE.

Отметим, что класс SQUARE, наследник класса RECTANGLE, характеризуется инвариантом side1 = side2. Аналогично, у эллипса имеются два фокуса, а у круга они сливаются в один, что определяет инвариант класса CIRCLE: equal (focus1 = focus2).

Полиморфизм

Иерархии наследования позволяют достаточно гибко работать с объектами, сохраняя надежность статической типизации. Поддерживающие их методы: полиморфизм и динамическое связывание - одни из самых фундаментальных аспектов архитектуры ПО, обсуждаемой в этой книге. Начнем с полиморфизма.

Полиморфное присоединение

"Полиморфизм" означает способность обладать несколькими формами. В ОО-разработкеическими формами обладают сущности (элементы структур данных), способные во время выполнения присоединяться к объектам разных типов, что контролируется статическими объявлениями.

Предположим, что для структуры наследования на рисунке вверху объявлены следующие сущности:

```
p: POLYGON; r: RECTANGLE; t: TRIANGLE
```

Тогда допустимы следующие присваивания:

```
p := r  
p := t
```

Эти команды присваивают в качестве значения сущности, обозначающей многоугольник, сущность, обозначающую прямоугольник в первом случае, и сущность, обозначающую треугольник - во втором.

Такие присваивания, в которых тип источника (правой части) отличен от типа цели (левой части), называются **полиморфными присваиваниями**. Сущность, входящая в полиморфное присваивание слева (в примере это p) является **полиморфной сущностью**.

До введения наследования все присваивания были мономорфными (не полиморфными): можно было присваивать точку точке, книгу книге, счет счету. С появлением полиморфизма возможных действий становится больше.

Приведенные в примере полиморфные присваивания легитимны, поскольку структура наследования позволяет рассматривать экземпляр класса RECTANGLE или TRIANGLE как экземпляр класса POLYGON. Мы говорим, что в таком случае тип источника **согласован с** типом цели. В обратном направлении присваивание недопустимо, т.е. некорректно писать r := p. Вскоре это важное правило будет рассмотрено более подробно.

Кроме присваивания, полиморфизм имеет место и при передаче аргументов, например в вызовах вида f (r) или f (t) при условии объявлении компонента f в виде:

```
f (p: POLYGON) is do ... end
```

Напомним, что присваивание и передача аргументов имеют одинаковую семантику, и оба называются **присоединением (attachment)**. Когда источник и цель имеют разные типы, можно говорить о **полиморфном (polymorphic) присоединении**.

Что на самом деле происходит при полиморфном присоединении?

Все сущности, встречающиеся в предыдущих примерах полиморфных присваиваний, имеют тип ссылок: возможными значениями p, r и t являются не объекты, а ссылки на объекты. Поэтому результатом присваивания p := r является просто новое присоединение ссылки.

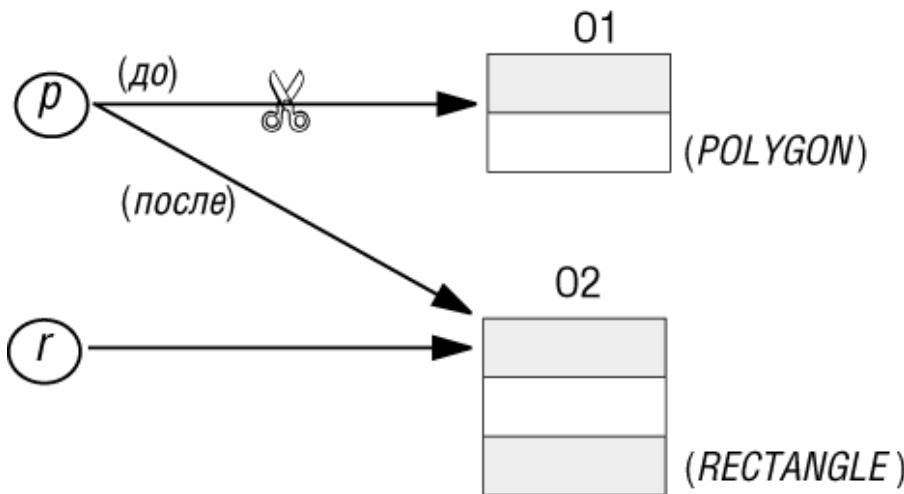


Рис. 14.3. Полиморфное присоединение ссылки

Несмотря на название, не следует представлять полиморфизм как некоторую трансмутацию объектов во время выполнения программы. Будучи один раз создан, объект никогда не изменяет свой тип. Так могут поступать только ссылки, которые могут указывать на объекты разных типов. Отсюда также следует, что за полиморфизм не нужно платить потерей эффективности, перенаправление ссылки - очень быстрая операция, ее стоимость не зависит от включенных в эту операцию объектов.

Полиморфные присоединения допускаются только для целей типа ссылки, но, ни в коем случае, для расширенных типов. Поскольку у класса-потомка могут быть новые атрибуты, то соответствующие ему экземпляры могут иметь больше полей. На [рис. 14.3](#) видно, что объект класса RECTANGLE больше, чем объект класса POLYGON. Такая разница в размерах объектов не приводит к проблемам, если все, что заново присоединяется, имеет тип ссылки. Но если r - не ссылка, а имеет развернутый тип (например, объявлена как expanded POLYGON), то значением r является непосредственно некоторый объект, и всякое присваивание r будет менять содержимое этого объекта. В этом случае никакой полиморфизм невозможен.

Полиморфные структуры данных

Рассмотрим массив многоугольников:

```
poly_arr: ARRAY [POLYGON]
```

Когда некоторое значение x присваивается элементу этого массива, как в вызове

```
poly_arr.put (x, some_index)
```

(для некоторого допустимого значения индекса $some_index$), то спецификация класса ARRAY указывает, что тип присваиваемого значения должен быть согласован с типом фактического родового параметра:

```
class ARRAY [G] creation
  ...
  feature - Изменение элемента
    put (v: G; i: INTEGER) is
      -- Присвоить v элементу с индексом i
  ...
end
```

Так как тип формального аргумента v , соответствующего x , в классе определен как G , а фактический родовой параметр, соответствующий G в вызове $poly_arr$, - это POLYGON, то тип x должен быть согласован с ним. Как мы видели, для этого x не обязан иметь тип POLYGON, подойдет любой потомок типа POLYGON.

Поэтому, если границы массива равны 1 и 4, то можно объявить некоторые сущности:

```
p: POLYGON; r: RECTANGLE; s: SQUARE; t: TRIANGLE
```

и, создав соответствующие объекты, можно выполнить операции

```
poly_arr.put (p, 1)
poly_arr.put (r, 2)
poly_arr.put (s, 3)
poly_arr.put (t, 4)
```

которые присвают элементам массива ссылки на объекты различных типов.

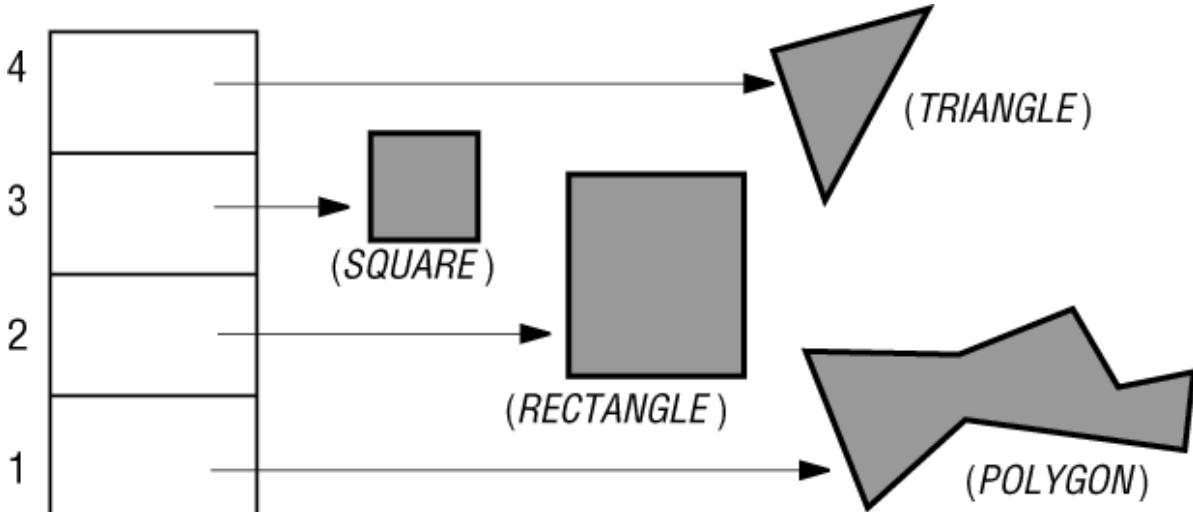


Рис. 14.4. Полиморфный массив

На этом рисунке графические объекты представлены соответствующими геометрическими фигурами, а не обычными диаграммами объектов с набором их полей.

Такие структуры данных, содержащие объекты разных типов, имеющих общего предка, называются **полиморфными структурами данных**. Далее будут рассмотрены многочисленные примеры таких структур. Массивы - это только одна из возможностей, полиморфными могут быть любые структуры контейнеров: списки, стеки и т.п.

Полиморфные структуры данных реализуют цель, сформулированную в начале лекции: объединение порождения и наследования для достижения максимальной гибкости и надежности. Имеет смысл напомнить [рис. 10.1](#), иллюстрирующий эту мысль:

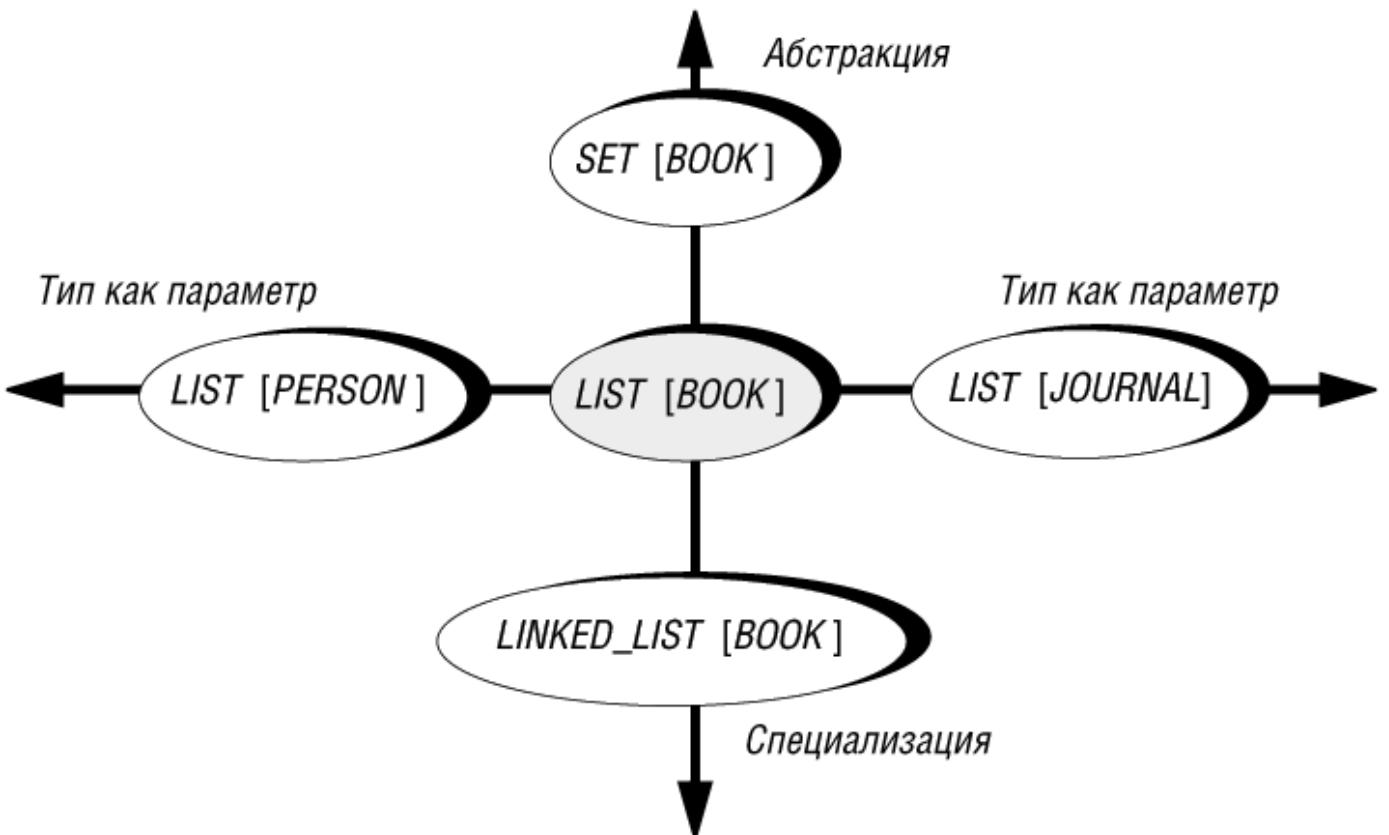


Рис. 14.5. Измерения обобщения

Типы, которые на [рис. 10.1](#) неформально назывались SET_OF_BOOKS и т. п., заменены типами, выведенными из родового универсального типа, - SET [BOOK].

Такая комбинация универсальности и наследования является весьма сильным средством. Оно позволяет описывать структуру объектов с нужной степенью общности. Например,

LIST [RECTANGLE]: может содержать квадраты, но не треугольники.

LIST [POLYGON]: может содержать квадраты, прямоугольники, треугольники, но не круги.

LIST [FIGURE]: может содержать экземпляры любого типа из иерархии FIGURE, но не книги или банковские счета.

LIST [ANY]: может содержать объекты любого типа.

В последнем случае использован класс ANY, который условимся считать предком любого класса (он будет подробнее рассмотрен далее).

Варьируя место класса, выбираемого в качестве фактического родового параметра, в иерархии, можно точно установить границы типов объектов, допустимых в определяемом контейнере.

Типизация при наследовании

Замечательная гибкость, обеспечиваемая наследованием, не связана с потерей надежности, поскольку используется **статическая проверка типов**, гарантирующая во время компиляции отсутствие некорректных комбинаций типов во время выполнения.

Согласованность типов

Наследование согласовано с системой типов. Основные правила легко объяснить на приведенном выше примере. Предположим, что имеются следующие объявления:

p: POLYGON

r: RECTANGLE

Выделим в приведенной выше иерархии нужный фрагмент ([рис. 14.6](#)).

Тогда законны следующие выражения:

- p.perimeter: никаких проблем, поскольку perimeter определен для многоугольников;
- p.vertices, p.translate (...), p.rotate (...) с корректными аргументами;
- r.diagonal, r.side1, r.side2: эти три компонента объявлены на уровне RECTANGLE или QUADRANGLE;
- r.vertices, r.translate (...), r.rotate (...): эти компоненты объявлены на уровне POLYGON или еще выше и поэтому применимы к прямоугольникам, наследующим все компоненты многоугольников;
- r.perimeter: то же, что и в предыдущем случае. Но у вызываемой здесь функции имеется новое определение в классе RECTANGLE, так что она отличается от функции с тем же именем из класса POLYGON.

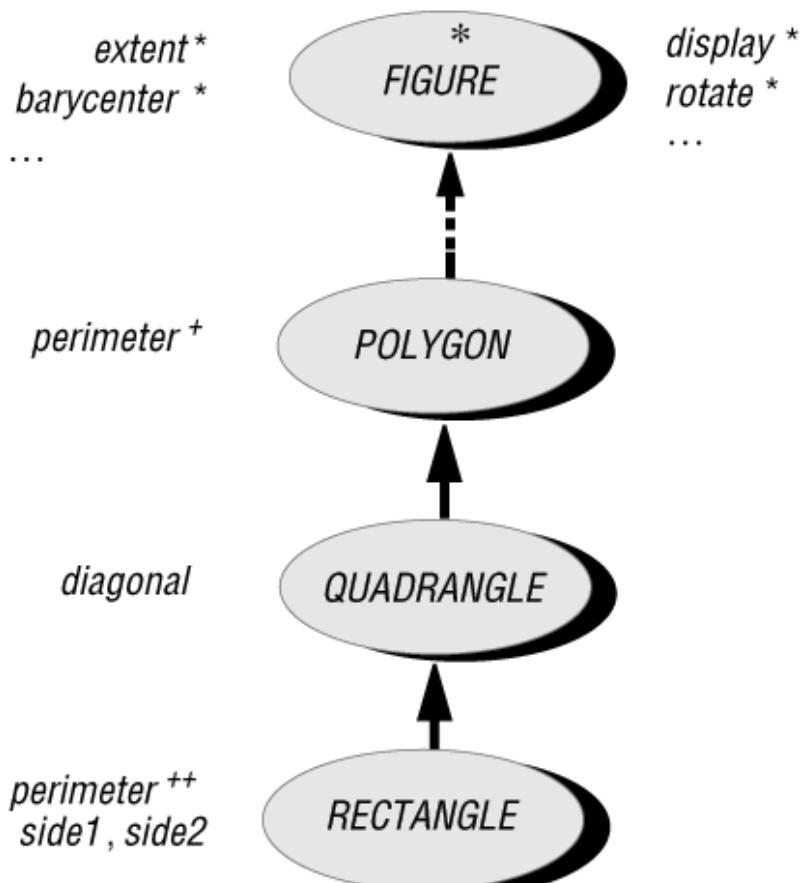


Рис. 14.6. Фрагмент иерархии геометрических фигур

А следующие вызовы компонентов незаконны, так как эти компоненты недоступны на уровне многоугольника:

```
p.side1  
p.side2  
p.diagonal
```

Это рассмотрение основано на первом фундаментальном правиле типизации:

Правило Вызова Компонентов

Если тип сущности x основан на классе C , то в вызове компонента $x.f$ сам компонент f должен быть определен в одном из предков C .

Напомним, что класс C является собственным предком. Фраза "тип сущности x основан на классе C " напоминает, что для классов, порожденных из родовых, тип может включать не только имя класса: `LINKED_LIST [INTEGER]`. Но базовый класс для типа - это `LINKED_LIST`, так что родовой параметр никак не участвует в нашем правиле.

Как и все другие правила корректности, рассматриваемые в этой книге, правило Вызова Компонентов является статическим, - его можно проверять на основе текста системы, а не по ходу ее выполнения. Компилятор (который, как правило, выполняет такую проверку) будет отвергать классы, содержащие некорректные вызовы компонентов. Если успешно реализовать проверку правил типизации, то не возникнет риска того, что скомпилированная система когда-либо во время выполнения применит некоторый компонент к объекту неподходящего типа.

Статическая типизация - это один из главных ресурсов ОО-технологии для достижения объявленной в 1-ой лекции цели - надежности ПО.

Уже отмечалось, что не все подходы к построению ОО-ПО имеют статическую типизацию. Наиболее известным представителем языков с динамической типизацией является Smalltalk, в котором не действует статическое правило вызова, но допускается, чтобы вычисление аварийно завершалось в случае возникновения ошибки: "сообщение не понятно". В лекции, посвященной типизации, будет приведено сравнение разных подходов.

Пределы полиморфизма

Неограниченный полиморфизм был бы несовместим со статическим понятием типа. Допустимость полиморфных операций определяется наследственностью.

Все примеры полиморфных присваиваний, такие, как $r := g$ и $r := t$, в качестве типа источника используют потомков класса-цели. Скажем, что в таком случае тип источника согласован с классом цели. Например, `SQUARE` согласован с `RECTANGLE` и с `POLYGON`, но не с `TRIANGLE`. Чтобы уточнить это понятие, дадим формальное определение:

Определение: согласованность

Тип U согласован с типом T , только если базовый класс для U является потомком базового класса для T ; при этом для универсально порожденных типов каждый фактический параметр U должен (по рекурсии) быть согласован с соответствующим формальным параметром T .

Почему недостаточно понятия потомка в этом определении? Причина снова в том, что допускается порождение из родовых классов, поэтому приходится различать типы и классы. Для каждого типа имеется **базовый класс**, который при отсутствии порождения совпадает с самим типом (например, `POLYGON` является базовым для себя). При этом для универсально порожденного класса базовым является универсальный класс с опущенными родовыми параметрами. Например, для класса `LIST [POLYGON]` базовым будет класс `LIST`. Вторая часть определения говорит о том, что $V[Y]$ будет согласован с $A[X]$, если V является потомком A , а Y - потомком X .

Заметим, что поскольку каждый класс является собственным потомком, то каждый тип согласован сам с собой.

При таком обобщении понятия потомка получаем второе важное правило типизации:

Правило согласования типов

Присоединение к источнику у цели x (т. е. присваивание $x := y$ или использование y в качестве фактического параметра в вызове процедуры с соответствующим формальным параметром x) допустимо только тогда, когда тип U согласован с типом x .

Правило согласования типов выражает тот факт, что специальное можно присваивать общему, но не наоборот. Поэтому присваивание $r := g$ допустимо, а $g := r$ нет.

Это правило можно проиллюстрировать следующим образом. Предположим, что я настолько ненормален, что послал в компанию Любимцы-По-Почте заказ на "Animal" ("Животное"). В этом случае, что бы я ни получил: собаку, божью коровку или дельфина-касатку, у меня не будет права пожаловаться. (Предполагается, что `DOG` и все прочие являются потомками класса `ANIMAL`). Но если я заказал собаку, а почтальон принес мне утром коробку с надписью

ANIMAL, или, например, MAMMAL (млекопитающее), то я имею право вернуть ее отправителю, даже если из нее доносится недвусмысленный лай и тявканье. Поскольку мой заказ не был исполнен в соответствии со спецификацией, я ничего не должен фирмке Любимцы-По-Почте.

Экземпляры

С введением полиморфизма нам требуется уточнить терминологию, связанную с экземплярами. Содержательно, экземпляры класса - это объекты времени выполнения, построенные в соответствии с определением класса. Но сейчас в этом качестве нужно также рассматривать объекты, построенные для собственных потомков класса. Вот более точное определение:

Определение: прямой экземпляр, экземпляр

Прямой экземпляр класса C - это объект, созданный в соответствии с точным определением C с помощью команды создания `create x ...`, в которой цель x имеет тип C (или, рекурсивно, путем клонирования прямого экземпляра C).

Экземпляр C - это прямой экземпляр потомка C.

Из последней части этого определения следует, что прямой экземпляр класса C является также экземпляром C, так как класс входит во множество своих потомков.

Таким образом, выполнение фрагмента:

```
p1, p2: POLYGON; r: RECTANGLE  
...  
create p1 ...; create r ...; p2 := r
```

создаст два экземпляра класса POLYGON, но лишь один прямой экземпляр (тот, который присоединен к p1). Другой объект, на который указывают p2 и r, является прямым экземпляром класса RECTANGLE, а следовательно, экземпляром обоих классов POLYGON и RECTANGLE.

Хотя понятия прямого экземпляра и экземпляра определены выше для классов, они естественно распространяются на любые типы (с базовым классом и возможными родовыми параметрами).

Полиморфизм означает, что элемент некоторого типа может присоединяться не только к прямым экземплярам этого типа, но и к другим его экземплярам. Можно считать, что роль правила согласования типов состоит в обеспечении следующего свойства:

Статико-динамическая согласованность типов

Сущность типа T может во время исполнения прикрепляться только к экземплярам класса T.

Статический тип, динамический тип

Название последнего свойства предполагает различие "статического типа" и "динамического типа". Тип, который используется при объявлении некоторого элемента, является **статическим типом** соответствующей ссылки. Если во время выполнения эта ссылка присоединяется к объекту некоторого типа, то этот тип становится **динамическим типом** этой ссылки.

Таким образом, при объявлении `r: POLYGON` статический тип ссылки, обозначенной `r`, есть `POLYGON`, после выполнения `create r` динамическим типом этой ссылки также является `POLYGON`, а после присваивания `r := r`, где `r` имеет тип `RECTANGLE` и не пусто, динамическим типом становится `RECTANGLE`.

Правило согласования типов утверждает, что динамический тип всегда должен соответствовать статическому типу.

Чтобы избежать путаницы напомним, что мы имеем дело с тремя уровнями: **сущность** - это некоторый идентификатор в тексте класса, во время выполнения ее значение является **ссылкой** (за исключением развернутого случая), ссылка может быть присоединена к **объекту**.

У объекта имеется только динамический тип, который он получил в момент создания. Этот тип во время жизни объекта не изменяется.

В каждый момент во время выполнения у ссылки имеется динамический тип, тип того объекта, к которому она сейчас присоединена (или специальный тип `NONE`, если эта ссылка пуста). Динамический тип может изменяться в результате операций переприсоединения.

Только у сущности имеются и статический, и динамический типы. Ее статический тип - это тип, с которым она была объявлена: если объявление имеет вид `x: T`, то этим типом будет `T`. Ее динамический тип в каждый момент выполнения - это тип значения этой ссылки, т.е. того объекта, к которому она присоединена.

В развернутом случае нет ссылки, значением *x* является объект типа *T*, и *T* является и статическим типом и единственным возможным динамическим типом для *x*.

Обоснованы ли ограничения?

Приведенные выше правила типизации могут иногда показаться слишком строгими. Например, второй оператор в обоих случаях статически отвергается:

1. *p* := *r*; *r* := *p*
2. *p* := *r*; *x* := *p.diagonal*

В (1) запрещается присваивать многоугольник сущности-прямоугольнику, хотя во время выполнения так получилось, что этот многоугольник является прямоугольником (аналогично тому, как можно отказаться принять собаку из-за того, что на клетке написано "животное"). В (2) компонент *diagonal* оказался не применим к *r* несмотря на то, что во время выполнения он, фактически, присутствует.

Но более аккуратный анализ показывает, что наши правила вполне обоснованы. Если ссылка присоединяется к объекту, то лучше избежать будущих проблем, убедившись в том, что их типы согласованы. А если хочется применить некоторую операцию прямоугольника, то почему бы сразу не объявить цель прямоугольником?

На практике, случаи вида (1) и (2) маловероятны. Присваивания типа *p* := *r* обычно встречаются внутри некоторых управляющих структур, которые зависят от условий, определяемых во время выполнения, например, от ввода данных пользователем. Более реалистичная полиморфная схема может выглядеть так:

```
create r.make (...); ...
screen.display_icons           -- Вывод значков для разных многоугольников
screen.wait_for_mouse_click    -- Ожидание щелчка кнопкой мыши
x := screen.mouse_position     -- Определение места нажатия кнопки
chosen_icon := screen.icon_where_is (x)   -- Определение значка,
                                              -- на котором находится указатель мыши
if chosen_icon = rectangle_icon then
  p := r
elseif ...
  p := "Многоугольник другого типа" ...
end
... Использование p, например, p.display, p.rotate, ...
```

В последней строке *p* может обозначать любой многоугольник, поэтому можно к нему применять только общие компоненты из класса POLYGON. Понятно, что операции, подходящие для прямоугольников, такие как *diagonal*, должны применяться только к *r* (например, в первом предложении *if*). Если придется использовать *p* в операторах, следующих за оператором *if*, то к нему могут применяться лишь операции, применимые ко всем видам многоугольников.

В другом типичном случае *p* просто является формальным параметром процедуры:

```
some_routine (p: POLYGON) is ...
```

и можно выполнять вызов *some_routine (r)*, корректный в соответствии с правилом согласования типов. Но при написании процедуры об этом вызове еще ничего не известно. На самом деле, вызов *some_routine (t)* для *t* типа TRIANGLE или любого другого потомка класса POLYGON будет также корректен, таким образом, можно считать, что *p* представляет некоторый вид многоугольников - любой из их видов. Тогда вполне разумно, что к *p* применимы только компоненты класса POLYGON.

Таким образом, в случае, когда невозможно предсказать точный тип присоединяемого объекта, полиморфные сущности (такие как *p*) весьма полезны.

Может ли быть польза от неведения?

Поскольку введенные только что понятия играют важную роль в последующем, стоит еще раз повторить несколько последних положений. (На самом деле, в этом коротком пункте не будет ничего нового, но он поможет лучше понять основные концепции и подготовит к введению новых понятий).

Если вы все еще испытываете неудобство от невозможности написать *p.diagonal* после присваивания *p* := *r* (в случае (2)), то вы не одиноки. Это шокирует многих людей, когда они впервые сталкиваются с этими понятиями. Мы знаем, что *p* - это прямоугольник, почему же у нас нет доступа к его диагонали? По той причине, что это было бы бесполезно. После полиморфного присваивания, как показано на следующем фрагменте из предыдущего рисунка, один и тот же объект типа RECTANGLE имеет два имени: имя многоугольника *p* и прямоугольника *r*.

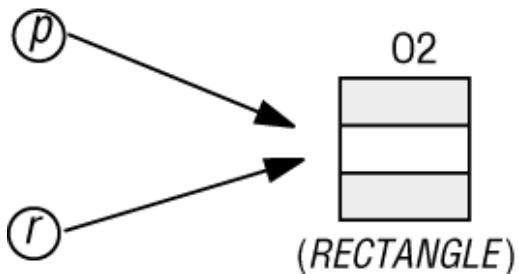


Рис. 14.7. После полиморфного присваивания

В таком случае, поскольку известно, что объект O2 является прямоугольником и доступен через имя прямоугольника *r*, зачем пытаться использовать доступ к его диагонали посредством операции *p.diagonal?* Это не имеет смысла, так как можно просто написать *r.diagonal*, используя официальное имя прямоугольника и сняв все сомнения в правомерности применения его операций. Использование имени многоугольника *p*, которое может с тем же успехом обозначать треугольник, ничего не дает и приводит к неопределенности.

Действительно, полиморфизм **теряет** информацию: когда в результате присваивания *p := r* появляется возможность ссылаться на прямоугольник O2 через имя многоугольника *r*, то теряется нечто важное - возможность использовать специфические компоненты прямоугольника. В чем тогда польза? В данном случае - ни в чем. Как уже отмечалось, интерес возникает, когда заранее неизвестно, каков будет вид многоугольника *p* после выполнения команды *if some_condition then p := r else p := something_else ...* или когда *p* является формальным аргументом процедуры и неизвестно, каков будет тип фактического аргумента. Но в этих случаях было бы некорректно и опасно применять к *p* что-либо кроме компонентов класса POLYGON.

Продолжая тему животных, представим, что некто спрашивает: "У вас есть домашний любимец?" и вы отвечаете: "Да, кот!". Это похоже на полиморфное присваивание - один объект известен под двумя именами разных типов: "мой_домашний_любимец" и "мой_кот" обозначают сейчас одно животное. Но они не служат одной цели, первое имя является менее информативным, чем второе. Можно одинаково успешно использовать оба имени при звонке в отдел отсутствующих хозяев компании Любимцы-По-Почте ("Я собираюсь в отпуск, сколько будет стоить наблюдение за моим_домашним_любимцем (или: моим_котом) в течение двух недель?") Но при звонке в другой отдел с вопросом: "Могу ли я привезти во вторник моего домашнего любимца, чтобы отстричь когти?", вы не запишитесь на прием, пока не уточните, что имели в виду своего кота.

Когда хочется задать тип принудительно

В некоторых случаях нужно выполнить присваивание, не соответствующее структуре наследования, и допустить, что при этом в качестве результата не обязательно будет получен объект. Такого, обычно, не бывает, когда ОО-метод применяется к объектам, внутренним для некоторой программы. Но можно, например, поучить по сети объект с его объявленным типом, и поскольку нет возможности контролировать источник происхождения этого объекта, то объявления статических типов ничего не гарантируют и прежде, чем использовать объект, необходимо проверить его тип.

При получении коробки с надписью "Животное" вместо ожидаемой надписи "Собака", можно соблазниться и все же ее открыть, зная, что, если внутри будет не собака, то потеряется право на возврат посылки и, в зависимости от того, что из нее появится, можно лишиться даже возможности рассказать эту историю.

В таких случаях требуется новый механизм - **попытка присваивания**, который позволит писать команду вида *r ?= p* (где *?=* обозначает символ попытки присваивания, в отличие от *:=* для обычного присваивания), означающую "выполнить присваивание, если тип объекта соответствует *r*, а иначе сделать *r* пустым". Но мы пока не готовы понять, как такая команда сочетается с ОО-методом, поэтому вернемся к этому вопросу в следующих лекциях. (А до того, считайте, что вы ничего об этом не читали).

Полиморфное создание

Введение наследования и полиморфизма приводит к небольшому расширению механизма создания объектов, который позволяет непосредственно создавать объекты типов-потомков.

Напомним, что команды создания (процедуры-конструкторы) имеют один из следующих видов:

```
create x
create x.make ( . . . )
```

где вторая форма подразумевает и требует, чтобы базовый класс для типа *T*, приписанного *x*, содержал предложение **creation**, в котором *make* указана как одна из процедур-конструкторов. (Разумеется, процедура создания может иметь любое имя, - *make* рекомендуется по умолчанию). Результатом выполнения первой команды является создание нового объекта типа *T*, его инициализация значениями, заданными по умолчанию, и его присоединение к *x*. А при выполнении второй инструкции для создания и инициализации объекта будет вызываться *make* с заданными

аргументами.

Предположим, что у Т имеется собственный потомок U. Мы можем захотеть использовать x полиморфно и присоединить сразу к прямому экземпляру U, а не к экземпляру Т. Возможное решение использует локальную сущность типа U.

```
some_routine (...) is
  local
    u_temp: U
  do
    ...; create u_temp.make (...); x := u_temp; ...
  end
```

Это работает, но чересчур громоздко, особенно в контексте многозначного выбора, когда захочется присоединить x к экземпляру одного из нескольких возможных типов наследников. Локальные сущности (u_temp в нашем примере) играют только временную роль, их объявления и присваивания загромождают текст программы. Поэтому нужны специальные варианты конструкторов:

```
create {U} x
create {U} x.make (...)
```

Результат должен быть тот же, что и у конструкторов **create**, приведенных выше, но создаваемый объект должен являться прямым экземпляром U, а не Т. Этот вариант должен удовлетворять очевидному ограничению: тип U должен быть согласован с типом Т, а во второй форме make должна быть определена как процедура создания в классе, базовом для U, и если этот класс имеет одну или несколько процедур создания, то применима лишь вторая форма. Заметим, что здесь не важно, имеет ли сам класс Т процедуры создания, - все зависит только от U.

Типичное применение связано с созданием экземпляра одного из нескольких возможных типов:

```
f: FIGURE
...
"Вывести значки фигур"
if chosen_icon = rectangle_icon then
  create {RECTANGLE} f
elseif chosen_icon = circle_icon then
  create {CIRCLE} f
else
  ...
end
```

Этот новый вид конструкторов объектов приводит к введению понятия **тип при создании**, обозначающего тип создаваемого объекта в момент его создания конструктором:

Для формы с неявным типом `create x ...` тип при создании есть тип x.

Для формы с явным типом `create {U} x ...` тип при создании есть U.

Динамическое связывание

Динамическое связывание дополнит переопределение, полиморфизм и статическую типизацию, создавая базисную тетраполию наследования.

Использование правильного варианта

Операции, определенные для всех вариантов многоугольников, могут реализовываться по-разному. Например, `perimeter` (**периметр**) имеет разные версии для общих многоугольников и для прямоугольников, назовем эти версии `perimeterPOL` и `perimeterRECT`. У класса `SQUARE` также будет свой вариант (умноженная на 4 длина стороны). При этом естественно возникает важный вопрос: что случится, если программа, имеющая разные версии, будет применена к полиморфной сущности?

Во фрагменте

```
create p.make (...); x := p.perimeter
```

ясно, что будет использована версия `perimeterPOL`. Точно так же во фрагменте

```
create r.make (...); x := r.perimeter
```

будет использована версия `perimeterRECT`. Но что, если полиморфная сущность `r` статически объявлена как многоугольник, а динамически ссылается на прямоугольник? Предположим, что нужно выполнить фрагмент:

```
create r.make (...)  
p := r  
x := p.perimeter
```

Правило динамического связывания утверждает, что версию применяемой операции определяет **динамическая форма объекта**. В данном случае это будет `perimeterRECT`.

Конечно, более интересный случай возникает, когда из текста программы нельзя заключить, какой динамический тип будет иметь `r` во время выполнения. Например, что будет во фрагменте

```
-- Вычислить периметр фигуры выбранной пользователем  
p: POLYGON  
...  
if chosen_icon = rectangle_icon then  
  create {RECTANGLE} p.make (...)  
elseif chosen_icon = triangle_icon then  
  create {TRIANGLE} p.make (...)  
elseif  
  ...  
end  
...  
x := p.perimeter
```

или после условного полиморфного присваивания `if ... then p := r elseif ... then p := t ... ;` или если `r` является элементом полиморфного массива многоугольников, или если `r` является формальным аргументом с объявленным типом `POLYGON` некоторой процедуры, которой вызвавшая ее процедура передала фактический аргумент согласованного типа?

Тогда в зависимости от хода вычисления динамическим типом `p` будет `RECTANGLE`, или `TRIANGLE`, или т.п. У нас нет никакого способа узнать, какой из этих случаев будет иметь место. Но, благодаря динамическому связыванию, этого и не нужно знать: что бы ни случилось с `p`, при вызове будет выполнен правильный вариант компонента `perimeter`.

Эта способность операций автоматически приспосабливаться к тем объектам, к которым они применяются, является одной из главных особенностей ОО-систем, непосредственно относящейся к обсуждаемым в начале книги вопросам качества ПО. Ее последствия будут подробней рассмотрены далее в этой лекции.

Динамическое связывание позволяет завершить начатое выше обсуждение аспектов, связанных с потерей информации при полиморфизме. Сейчас стало понятно, почему не страшно потерять информацию об объекте: после присваивания `p := q` или вызова `some_routine (q)`, в котором `p` является формальным аргументом, теряется специфическая информация о типе `q`, но если применяется операция `p.polygon_feature`, для которой `polygon_feature` имеет специальную версию, применимую к `q`, то будет выполняться именно эта версия.

Вполне допустимо посыпать ваших любимцев в отдел отсутствующих хозяев, который обслуживает все виды, если наверняка известно, что, когда придет время еды, ваш кот получит кошачью еду, а пес - собачью.

Переопределение и утверждения

Если клиент класса `POLYGON` вызывает `p.perimeter`, то он ожидает получить значение периметра `p`, определенное спецификацией функции `perimeter` в определении этого класса. Но теперь, благодаря динамическому связыванию, клиент может вызвать другую программу, переопределенную в некотором классе-потомке. В классе `RECTANGLE` переопределение улучшает эффективность и не изменяет результат, но что помешало бы переопределить периметр так, чтобы новая версия вычисляла бы, скажем, площадь?

Это противоречит духу переопределения. Переопределение должно изменять реализацию процедуры, а не ее семантику. К счастью, утверждения позволяют ограничить семантику процедур. Неформально, основное правило контроля за переопределением и динамическим связыванием можно сформулировать просто: предусловие и постусловие программы должны быть применимы к любому ее переопределению, и, как мы уже видели, инвариант класса автоматически должен распространяться на всех его потомков.

Точные правила будут приведены ниже. Но уже сейчас можно заметить, что переопределение не является произвольным: допускаются только переопределения, сохраняющие семантику. Это дело автора программы - выразить ее семантику достаточно точно, но оставить при этом свободу для будущих реализаций.

О реализации динамического связывания

Может возникнуть опасение, что динамическое связывание - это дорогой механизм, требующий во время выполнения поиска по графу наследования и поэтому накладных расходов, растущих с увеличением глубины этого графа.

К счастью, это не так в случае хорошо спроектированного (и статически типизированного) ОО-языка. Более детально это будет обсуждаться в конце лекции, но мы можем уже сейчас успокоить себя тем, что последствия динамического связывания не будут существенными для эффективности при работе в подходящем окружении.

Отложенные компоненты и классы

Полиморфизм и динамическое связывание означают, что в процессе проектирования ПО можно рассчитывать на абстракции и быть уверенными в том, что при выполнении будет выбрана подходящая реализация. Но перед выполнением все должно быть полностью реализовано.

Однако полная реализация не всегда нужна. Частично реализованные или не реализованные абстрактные элементы ПО помогают при решении многих задач: анализе проблемы и проектировании архитектуры системы (в этом случае можно их сохранить в заключительном продукте, чтобы запомнить ход анализа и проектирования), при фиксации соглашений между реализаторами, при описании промежуточных точек в классификации.

Отложенные компоненты и классы обеспечивают необходимый механизм абстракции.

Движения произвольных фигур

Чтобы понять необходимость в отложенных процедурах и классах, снова рассмотрим иерархию фигур FIGURE.

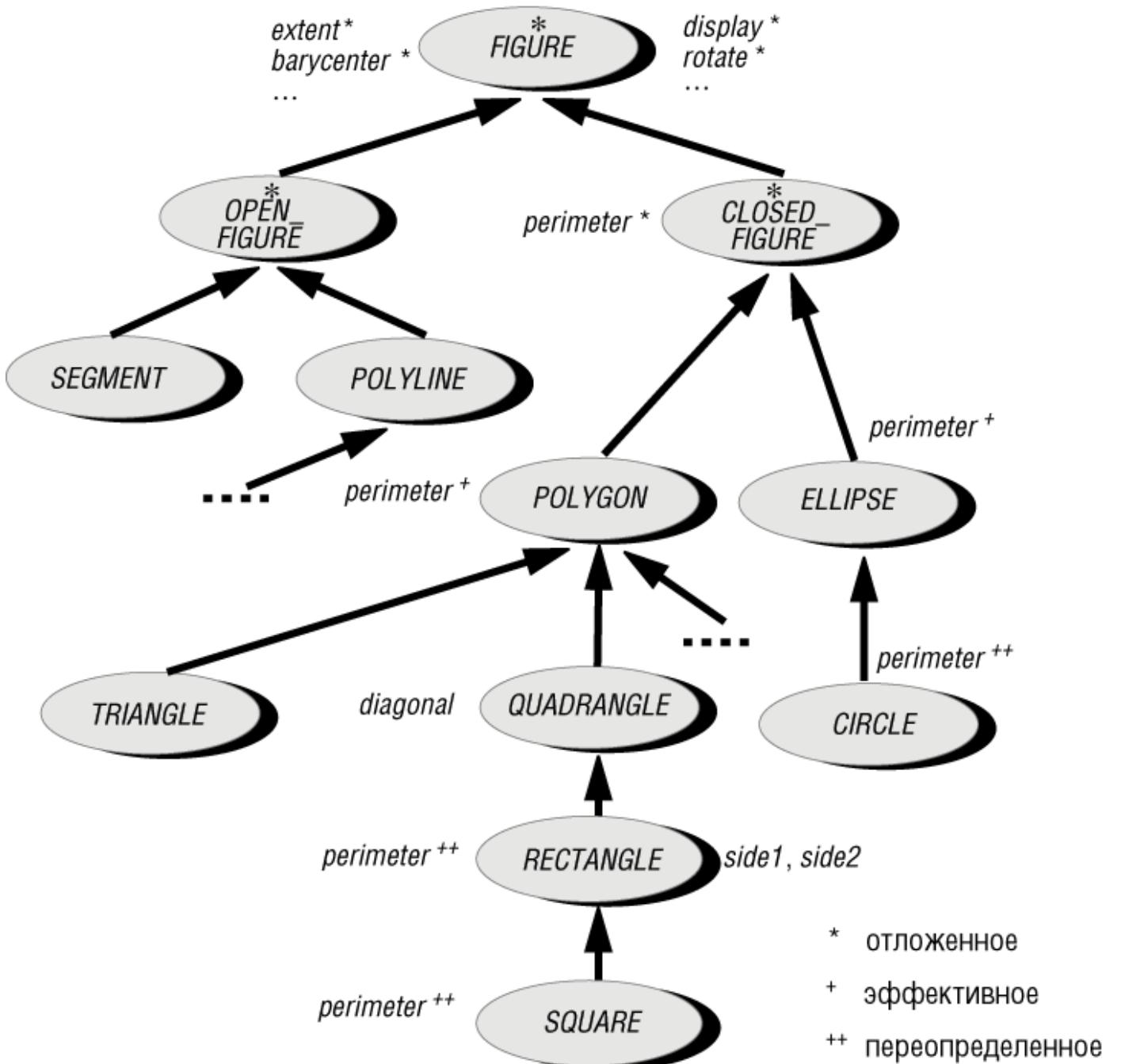


Рис. 14.8. Снова иерархия FIGURE

Наиболее общим понятием здесь является FIGURE. Основываясь на механизмах полиморфизма и динамического

связывания, можно попытаться применить описанную ранее общую схему:

```
transform (f: FIGURE) is
    -- Применить специфическое преобразование к f.
    do
        f.rotate (... )
        f.translate (... )
    end
```

с соответствующими значениями опущенных аргументов. Тогда все следующие вызовы корректны:

```
transform (r)           -- для r: RECTANGLE
transform (c)           -- для c: CIRCLE
transform (figarray.item (i)) -- для массива фигур: ARRAY [POLYGON]
```

Иными словами, требуется применить преобразования `rotate` и `translate` к фигуре `f` и предоставить механизму динамического связывания выбор подходящей версии (различной для классов `RECTANGLE` и `CIRCLE`), зависящей от текущего вида фигуры `f`, который выяснится во время выполнения.

Это действительно работает и является типичным примером элегантного стиля, ставшего возможным благодаря полиморфизму и динамическому связыванию, стиля, основанного на принципе Единственного выбора. Требуется только переопределить `rotate` и `translate` для различных вовлеченных в вычисление классов.

Но переопределять-то нечего! Класс `FIGURE` - это очень общее понятие, покрывающее все виды двумерных фигур. Ясно, что невозможно написать версию процедур `rotate` и `translate`, подходящую для всех фигур "вообще", не уточнив информацию об их виде.

Таким образом, мы имеем ситуацию, в которой процедура `transform` будет выполняться корректно, благодаря динамическому связыванию, но статически она незаконна, поскольку `rotate` и `translate` не являются компонентами класса `FIGURE`. Проверка типов выявит в вызовах `f.rotate` и `f.translate` ошибки.

Можно, конечно, ввести на уровне класса `FIGURE` процедуру `rotate`, которая ничего не будет делать. Но это опасный путь, компоненты `rotate (center, angle)` имеют интуитивно хорошо понятную семантику и "ничего не делать" не является их разумной реализацией.

Отложенный компонент

Таким образом, нужен способ спецификации компонентов `rotate` и `translate` на уровне класса `FIGURE`, который возлагал бы обязанность по их фактической реализации на потомков этого класса. Это достигается объявлением этих компонентов как "отложенных". При этом вся часть тела процедуры с командами заменяется ключевым словом **deferred**. В классе `FIGURE` будет объявление:

```
rotate (center: POINT; angle: REAL) is
    -- Повернуть на угол angle вокруг точки center.
    deferred
end
```

и аналогично будет объявлен компонент `translate`. Это означает, что этот компонент известен в том классе, где появилось такое объявление, но его реализации находятся в классах - собственных потомках. В таком случае вызов вида `f.rotate` в процедуре `transform` становится законным.

Объявленный таким образом компонент называется **отложенным** компонентом. Компонент, не являющийся отложенным, - имеющий реализацию (например, любой из ранее встретившихся нам компонентов), называется **эффективным**.

Эффективизация компонента

В некоторых собственных потомках класса `FIGURE` потребуется заменить отложенную версию эффективной. Например,

```
class POLYGON inherit
    CLOSED FIGURE
feature
    rotate (center: POINT; angle: REAL) is
        -- Повернуть на угол angle вокруг точки center.
        do
            ... Команды для поворота всех вершин ...
        end
    ...
end
```

end

Заметим, что POLYGON наследует компоненты класса FIGURE не непосредственно, а через класс CLOSED_PICTURE, в котором процедура rotate остается отложенной.

Этот процесс обеспечения реализацией отложенного компонента называется **эффективизацией (effecting)**. (Эффективный компонент - это компонент, снабженный реализацией.)

Не нужно в предложении **redefine** некоторого класса описывать отложенные компоненты, получающие реализацию, поскольку у них не было настоящего определения в месте объявления. В этом классе просто помещаются определения таких компонентов, совместимые по типам с их первоначальными объявлениями как, например, в случае компонента rotate.

Задание реализации компонента, конечно, близко к его переопределению и, за исключением включения в предложении **redefine**, подчиняется тем же правилам. Поэтому нужен общий термин.

Определение: повторное объявление

Повторное объявление компонента - означает определение или переопределение его реализации.

Разница между этими двумя формами повторного объявления хорошо иллюстрируется примерами, приведенными при их определении:

- При переходе от POLYGON к RECTANGLE компонент perimeter уже реализован у родителя, и мы хотим предложить новую его реализацию в классе RECTANGLE. Это переопределение. Заметим, что этот компонент еще раз переопределяется в классе SQUARE.
- При переходе от FIGURE к POLYGON у родителя нет реализации компонента rotate, и мы хотим реализовать его в классе POLYGON. Это эффективизация. Собственные потомки POLYGON могут, конечно, переопределить эту эффективную версию.

Может появиться нужда в некотором изменении параметров наследуемого отложенного компонента, после которого оно все так же останется отложенным. Эти изменения могут затрагивать сигнатуру компонента - типы ее аргументов и результата - и его утверждения (точные ограничения будут указаны в следующей лекции). В отличие от перехода от отложенного компонента к эффективному, такой переход от отложенного к отложенному рассматривается как переопределение и требует предложения **redefine**. Приведем резюме четырех возможных случаев нового объявления:

Таблица 14.1. Эффекты повторного объявления

Повторное объявление компонента к	Повторное объявление компонента от	
	Отложенный	Эффективный
Отложенный	Переопределение	Отмена определения
Эффективный	Эффективизация	Переопределение

В этой таблице имеется один еще не рассмотренный случай: **отмена определения** - переход от эффективного компонента к отложенному. При этом отменяется исходная реализация и начинается новая жизнь.

Отложенные классы

Как мы видели, компонент может быть отложенным или эффективным. То же относится и к классам.

Определение: отложенный класс, эффективный класс

Класс является отложенным, если у него имеется отложенный компонент.

В противном случае, класс является эффективным.

Таким образом, чтобы класс был эффективным, должны быть эффективными все его компоненты. Один или несколько отложенных компонентов делают класс отложенным. В этом случае класс должен содержать специальную метку:

Правило объявления отложенного класса

Объявление отложенного класса должно включать подряд идущие ключевые слова **deferred class** (в отличие от одного слова **class** для эффективных классов).

Поэтому класс FIGURE будет объявлен следующим образом:

```
deferred class FIGURE feature
    rotate (...) is
        ... Объявления отложенных компонентов ...
        ... Объявления других компонентов ...
end
```

Обратно, если класс отмечен как отложенный, то у него должен быть хотя бы один отложенный компонент. При этом класс может быть отложенным, даже если в нем самом не объявлен ни один отложенный компонент, так как у него может быть отложенный родитель, от которого он унаследовал отложенный компонент, не ставший у него эффективным. В нашем примере в классе OPEN_PICTURE, скорее всего, останутся отложенными компоненты display, rotate и многие другие, унаследованные от класса FIGURE, поскольку понятие незамкнутой фигуры не настолько конкретизировано, чтобы поддерживать стандартные реализации этих операций. Поэтому этот класс является отложенным и будет объявлен как

```
deferred class OPEN_PICTURE inherit  
    FIGURE  
    ...
```

даже если в нем самом не вводится ни один отложенный компонент.

Потомок отложенного класса является эффективным классом, если все отложенные компоненты его родителей имеют в нем эффективные определения и в нем не вводятся никакие собственные отложенные компоненты. Эффективные классы, такие как POLYGON и ELLIPSE, должны обеспечить реализацию отложенных компонентов display, rotate.

Для удобства мы будем называть тип отложенным, если его базовый класс является отложенным. Таким образом, класс FIGURE, рассматриваемый как тип, является отложенным. Если родовой класс LIST является отложенным (как это и должно быть, если он представляет понятие списка, не зависящее от реализации), то тип LIST [INTEGER] является отложенным. Учитывается только базовый класс: С [X] будет эффективным, если класс С эффективный, и отложенным, если С является отложенным, независимо от статуса X.

Соглашения о графических обозначениях

Сейчас можно полностью объяснить графические символы, использованные на [рис. 14.8](#). Звездочкой отмечаются отложенные компоненты или классы:

```
FIGURE*  
display*  
perimeter* -- На уровне класса OPEN_PICTURE на рис. 14.8
```

Знак плюс означает "эффективный" и им отмечается эффективизация компонента:

```
perimeter+ -- На уровне POLYGON на рис. 14.8
```

Чтобы указать, что класс эффективный, можно отметить его знаком +. По умолчанию, неотмеченный класс считается эффективным, так же как в текстовом виде объявление **class C** без ключевого слова **deferred** означает, что класс эффективный.

Можно присоединять одиночный плюс к компоненту для указания того, что он стал эффективным. Например, компонент perimeter появляется как отложенный и, следовательно, имеет вид perimeter* в классе CLOSED_PICTURE. Затем на уровне POLYGON для этого компонента дается реализация и он отмечается в этом классе как perimeter+.

Наконец, два знака плюс отмечают переопределение:

```
perimeter++ -- На уровне RECTANGLE и SQUARE на рис. 14.8
```

Что делать с отложенными классами?

Присутствие отложенных элементов в системе вызывает вопрос: "что случится, если компонент rotate применить к объекту типа FIGURE?" или в общем виде - "можно ли применить отложенный компонент к прямому экземпляру отложенного класса?" Ответ может обескуражить: такой вещи как объект типа FIGURE не существует - прямых экземпляров отложенных классов не бывает.

Правило отсутствия экземпляров отложенных классов

Тип создания в процедуре создания не может быть отложенным.

Напомним, что тип создания - это тип x, для формы **create x**, и U для формы **create {U} x**. Тип считается отложенным, если таков его базовый класс.

Поэтому вызов конструктора **create f** некорректен и будет отвергнут компилятором, если типом f будет один из отложенных классов: FIGURE, OPEN_PICTURE, CLOSED_PICTURE. Это правило устраняет опасность ошибочных

вызовов компонентов.

Отметим однако, что даже, если тип сущности *f* отложенный, то допустима явная форма процедуры создания - **create{RECTANGLE} f**, поскольку здесь типом создания является эффективный потомок FIGURE - класс RECTANGLE. Мы уже видели, как этот прием используется в многовариантной процедуре создания для объектов класса FIGURE, которые, в зависимости от контекста, будут экземплярами эффективных классов RECTANGLE, CIRCLE и др.

Может показаться, что это правило ограничивает полезность отложенных классов, делая их просто синтаксической уловкой для обмана системы статических типов. Это было бы верно, если бы не полиморфизм и динамическое связывание. Нельзя создать объект типа FIGURE, но можно объявить полиморфную сущность этого типа, а затем использовать ее, не зная точно, к объекту какого типа она присоединена в конкретном вычислении:

```
f: FIGURE
...
f := "Некоторое выражение эффективного типа, такого как CIRCLE или POLYGON"
...
f.rotate (some_point, some_angle)
f.display
...
```

Такие примеры являются комбинацией и кульминацией уникальных средств абстракции ОО-метода таких, как классы, скрытие информации, единственный выбор, наследование, полиморфизм, динамическое связывание, отложенные классы (и, как будет видно дальше, утверждения). Вы манипулируете объектами, не зная точно их типов, задавая только минимум информации, необходимой для требуемых операций. Имея надежный штамп контролера типов, удостоверяющий согласованность вызовов этих операций с их объявлением, можно рассчитывать на большую силу - динамическое связывание, которая позволяет применять корректную версию каждой операции, не зная точно, что это за версия.

Задание семантики отложенных компонентов и классов

Хотя у отложенного компонента нет реализации, а у отложенного класса либо нет реализации, либо он реализован частично, часто требуется задать их абстрактные семантические свойства. Для этой цели можно использовать утверждения.

Как и другие классы, отложенный класс может иметь инвариант, а у отложенного компонента может быть предусловие, постусловие или оба эти утверждения.

Рассмотрим пример линейных списков, описанных независимо от конкретной реализации. Как и для многих других структур такого рода, удобно связать с каждым списком курсор, указывающий на текущий активный элемент.

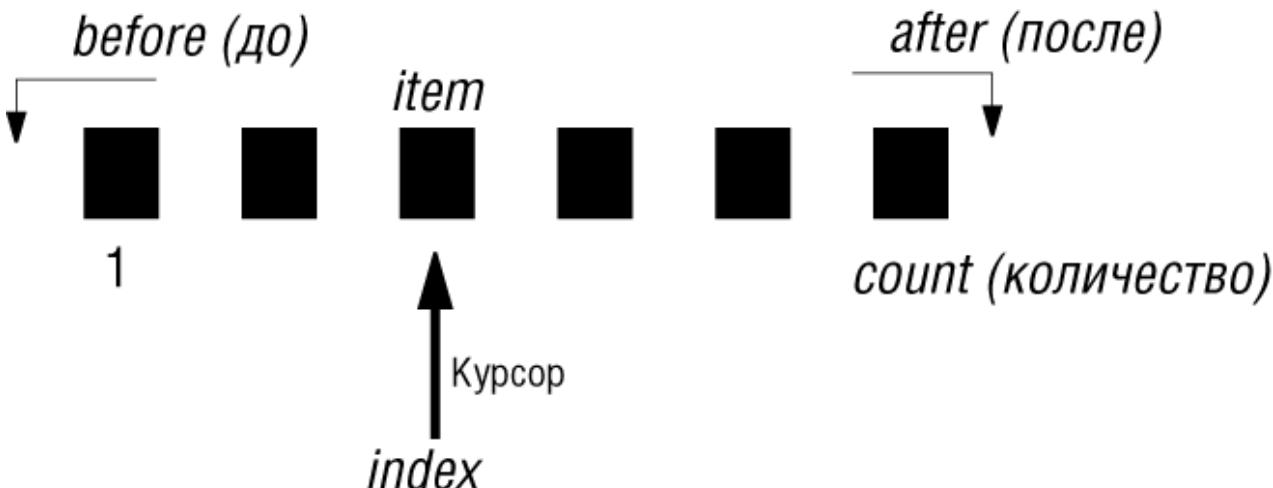


Рис. 14.9. Список с курсором

Этот класс является отложенным:

```
indexing
  description: "Линейные списки"
deferred class
  LIST [G]
feature -- Access
  count: INTEGER is
    -- Число элементов
  deferred
  end
  index: INTEGER is
```

```

    -- Положение курсора
    deferred
    end
item: G is
    -- Элемент в позиции курсора
    deferred
    end
feature - Отчет о статусе
    after: BOOLEAN is
        -- Курсор за последним элементом?
        deferred
        end
before: BOOLEAN is
    -- Курсор перед первым элементом?
    deferred
    end
feature - Сдвиг курсора
    forth is
        -- Передвинуть курсор на одну позицию вперед.
        require
            not after
        deferred
        ensure
            index = old index + 1
        end
    ... Другие компоненты ...
invariant
    non_negative_count: count >= 0
    offleft_by_at_most_one: index >= 0
    offright_by_at_most_one: index <= count + 1
    after_definition: after = (index = count + 1)
    before_definition: before = (index = 0)
end

```

Здесь инвариант выражает соотношения между разными запросами. Первые два предложения утверждают, что курсор может выйти за границы множества элементов не более чем на одну позицию слева или справа.



Рис. 14.10. Позиции курсора

Два последних предложения инварианта можно также представить в виде постусловий: `ensure Result = (index = count + 1)` для `after` и `ensure Result = (index = 0)` для `before`. Такой выбор всегда возникает при выражении свойств, включающих только запросы без аргументов. Я предпочитаю использовать предложения инварианта, рассматривая такие свойства как глобальные свойства класса, а не прикреплять их к конкретному компоненту.

Утверждения о `forth` точно выражают то, что должна делать эта процедура: передвигать курсор на одну позицию. Поскольку курсор должен оставаться в пределах списка элементов плюс две позиции "меток" слева и справа, то применение `forth` требует выполнения условия `not after`, а результатом будет, как сказано в постусловии, увеличение `index` на один.

Вот другой пример - наш старый друг стек. Нашей библиотеке потребуется общий класс `STACK [G]`, который будет отложенным, так как он должен покрывать всевозможные реализации. Его собственные потомки, такие как `FIXED_STACK` и `LINKED_STACK`, будут описывать конкретные реализации. Одной из отложенных процедур класса `STACK` является `put`:

```

put (x: G) is
    -- Поместить x на вершину.
    require
        not full
    deferred

```

```

ensure
    not_empty: not empty
    pushed_is_top: item = x
    one_more: count = old count + 1
end

```

Булевские функции `empty` и `full` (также отложенные на уровне STACK) выражают свойство стека быть пустым и заполненным.

Только с помощью утверждений отложенные классы достигают своей полной силы. Как уже отмечалось (хотя детали появятся через две лекции), предусловия и постусловия применимы ко всем переопределениям процедуры. Это особенно важно в отложенном случае: в нем такие утверждения будут ограничивать все допустимые реализации. Таким образом, приведенная спецификация ограничивает все варианты `put` в потомках класса STACK.

Благодаря использованию утверждений, можно сделать отложенные классы достаточно информативными и семантически богатыми, несмотря на отсутствие у них реализаций.

В конце этой лекции мы вновь обратимся к отложенным классам и исследуем глубже их роль в процессе ОО-анализа, проектирования и реализации.

Способы изменения объявлений

Возможность изменить объявление компонента - переопределить или дать его реализацию - обеспечивает гибкость и последовательное проведение разработки. Имеется еще два метода, усиливающих эти качества:

- Возможность изменить объявление функции на атрибут.
- Простой способ сослаться на первоначальную версию в теле нового определения.

Повторное объявление функции как атрибута

Повторные объявления позволяют активно применять один из центральных принципов модульности - принцип Унифицированного Доступа (Uniform Access).

Напомним (см. [лекцию 3](#)), что этот принцип утверждает (первоначально в менее технических терминах, но сейчас мы можем позволить себе быть более точными), что с точки зрения клиента не должно быть никакой существенной разницы между атрибутом и функцией без аргументов. В обоих случаях компонент является запросом и все, что их отличает, - это их внутреннее представление.

Первым примером этого был класс, описывающий банковские счета, в котором компонент `balance` мог быть реализован как функция, которая добавляет вклады и вычитает снимаемые суммы, или как атрибут, изменяемый по мере необходимости так, чтобы отражать текущий баланс. Для клиента это было все равно (за исключением, возможно, эффективности).

С появлением наследования можно пойти дальше и позволить, чтобы в классе наследуемая функция была переопределена как атрибут.

Наш прежний пример хорошо подходит для иллюстрации. Пусть имеется класс ACCOUNT1:

```

class ACCOUNT1 feature
    balance: INTEGER is
        -- Текущий баланс
    do
        Result := list_of_deposits.total - list_of_withdrawals.total
    end
    ...
End

```

Тогда в потомке может быть выбрана вторая реализация из нашего первоначального примера, переопределяющая `balance` как атрибут:

```

class ACCOUNT2 inherit
    ACCOUNT1
        redefine balance end
feature
    balance: INTEGER
        -- Текущий баланс
    ...
end

```

По-видимому, в классе ACCOUNT2 нужно будет переопределить некоторые процедуры, такие как `withdraw` и

`deposit`, чтобы, кроме других своих обязанностей они еще модифицировали нужным образом `balance`, сохраняя в качестве инварианта свойство: `balance = list_of_deposits.total - list_of_withdrawals.total`.

В этом примере новое объявление является переопределением. Его результатом может также оказаться превращение отложенного компонента в атрибут. Например, пусть в отложенном классе `LIST` имеется компонент

```
count: INTEGER is
    -- Число вставленных элементов
    deferred
end
```

Тогда в реализации списка этот компонент может быть реализован как атрибут:

```
count: INTEGER
```

Если нас попросят применить эту классификацию, чтобы разбить компоненты на атрибуты и подпрограммы, то мы условимся рассматривать отложенный компонент как подпрограмму, несмотря на то, что для отложенного компонента с результатом и без аргументов само понятие отложенности означает, что мы еще не сделали выбор, как его реализовать - функцией или атрибутом. Фраза "отложенный компонент" передает эту неопределенность и предпочтительней фразы "отложенная подпрограмма".

Переобъявление функции как атрибута, объединенное с полиморфизмом и динамическим связыванием, приводят к полной реализации принципа Унифицированного Доступа. Сейчас можно не только реализовать запрос клиента вида `a.service` либо через память, либо посредством вычисления, но один и тот же запрос в процессе одного вычисления может в одних случаях запустить доступ к некоторому полю, а в других - вызвать некоторую функцию. Это может, в частности, случиться при выполнении одного и того же вызова `a.balance`, если по ходу вычисления `a` будет полиморфно присоединяться к объектам разных классов.

Обратного пути нет

Можно было бы ожидать, что допустимо и обратное переопределение атрибута в функцию без аргументов. Но нет. Присваивание - операция применимая к атрибутам, - становится бессмысленной для функций. Предположим, что `a` - это атрибут класса `C`, и некоторая подпрограмма содержит команду

```
a := some_expression
```

Если потомок `C` переопределит `a` как функцию, то эта функция будет не применима, поскольку нельзя использовать функцию в левой части присваивания.

Отсутствие симметрии (допустимо изменять объявление функции на объявление атрибута, но не наоборот) неприятно, но неизбежно и не является на практике серьезным препятствием. Оно означает, что объявление некоторого компонента атрибутом является окончательным и необратимым выбором, в то время как объявление его функцией все еще оставляет место для последующих реализаций через память, а не через вычисление.

Использование исходной версии при переопределении

Рассмотрим некоторый класс, который переопределяет подпрограмму, унаследованную от родителя. Обычная схема переопределения состоит в том, чтобы выполнить все, что делает исходная версия, предпослав ей или поместив за ней некоторые специальные действия.

Например, класс `BUTTON`, наследник класса `WINDOW`, может переопределить компонент `display`, рисующий кнопку, так чтобы вначале рисовалось окно, а затем появлялась рамка:

```
class BUTTON inherit
    WINDOW
    redefine display end
feature -- Вывод
    display is
        -- Изобразить как кнопку.
        do
            "Изобразить как нормальное окно"; -- См. ниже
            draw_border
        end
    ... Другие компоненты ...
end
```

где `draw_border` - это процедура нового класса. Для того чтобы "Изобразить как нормальное окно", нужно вызвать исходную версию `display`, технически известную как **precursor** (предшественник) процедуры `draw_border`.

Это достаточно общий случай, и желательно ввести для него выбрать специальное обозначение. Конструкцию

Precursor

можно использовать в качестве имени компонента, но только в теле переопределяемой подпрограммы. Вызов этого компонента, если нужно с аргументами, является вызовом родительской версии этой процедуры (предшественника).

Поэтому в последнем примере часть "Изобразить как нормальное окно" можно записать просто как

Precursor

Это будет означать вызов исходной версии этой процедуры из класса WINDOW, допустимый при переопределении процедуры классом-наследником WINDOW. Precursor - это зарезервированное имя сущности такое же, как Result или Current, и оно так же пишется курсивом с заглавной первой буквой.

В данном примере переопределяемый компонент является процедурой и поэтому вызов конструкции Precursor - это команда. Этот же вызов может участвовать при переопределении функции в выражении:

```
some_query (n: INTEGER): INTEGER is
-- Значение, возвращаемое версией родителя, если оно
-- положительно, иначе ноль
do
    Result := (Precursor (n)).max (0)
end
```

В случае множественного наследования, рассматриваемого в следующей лекции, у процедуры может быть несколько предшественников, что позволяет объединить несколько наследуемых процедур в одну. Тогда для устранения неоднозначности нужно будет указывать родителя, например, Precursor {WINDOW}.

Заметим, что использование конструкции Precursor не делает компонент-предшественник компонентом данного класса, компонентом является только его переопределенная версия. (В частности, предшествующая версия может не удовлетворять новому инварианту.) Целью конструкции является облегчение переопределения в случае, когда новая версия включает старую.

В более сложном случае, когда, в частности, требуется использовать и предшествующую и новую версии в качестве компонентов класса, можно воспользоваться дублируемым наследованием, при котором родительский компонент, фактически, дублируется, и у наследника создаются два законченных компонента. Это будет подробно обсуждаться при рассмотрении дублируемого наследования.

Смысл наследования

Мы уже рассмотрели основные способы наследования. Многое еще предстоит изучить, в частности, множественное наследование и детали того, что происходит с утверждениями в контексте наследования (понятие субконтрактов).

Но вначале следует поразмышлять над этими фундаментальными понятиями и выяснить их значение для вопроса о качестве ПО и для процесса разработки ПО.

Двойственная перспектива

По-видимому, нигде двойственная роль классов как модулей, с одной стороны, и типов - с другой, не проявляется так отчетливо, как при изучении наследования. При взгляде на класс, как на модуль, наследник описывает расширение модуля-родителя, а при взгляде на него, как на тип, он описывает подтип типа родителя.

Хотя некоторые аспекты наследования больше относятся к взгляду на класс, как на тип, большая часть полезна для обоих подходов, о чем свидетельствует приведенная примерная классификация (на которой отражены также несколько еще не изученных аспектов: переименование, скрытие потомков, множественное и повторное наследование). Ни один из рассматриваемых аспектов не относится исключительно к взгляду на класс, как на модуль.



Рис. 14.11. Механизмы наследования и их роль

Эти два взгляда дополняют друг друга, придавая наследованию силу и гибкость. Эта сила может даже показаться пугающей, что побуждает предложить разделить механизм на два: на возможность расширять модули и на механизм выделения подтипов. Но когда мы вникнем в проблему глубже (в лекции о методологии наследования), то обнаружим, что у такого разделения имеется множество недостатков, и нет явных преимуществ. Наследование - это объединяющий принцип, как и многие другие объединяющие идеи в науке, он соединяет вместе явления, рассматриваемые ранее как различные.

Взгляд на класс как на модуль

С этой точки зрения наследование особенно эффективно в качестве метода повторного использования.

Модуль это множество служб, предлагаемых внешнему миру. Без наследования каждому новому модулю пришлось бы самому определять все предоставляемые им службы. Конечно, реализации этих служб могут основываться на службах, предоставляемых другими модулями: это и есть цель отношения "быть клиентом". Но единственным способом определить новый модуль является добавление новых служб к ранее определенным модулям.

Наследование предоставляет эту возможность. Если В является наследником А, то все службы (компоненты) А автоматически доступны в В, и их не нужно в нем явно определять. В соответствии со своими целями В может добавить новые компоненты. Дополнительная гибкость обеспечивается переопределением, позволяющим В по-разному использовать реализации, предлагаемые А: некоторые из них не меняются, а другие переделываются в более подходящие для данного класса версии.

Это приводит к такому стилю разработки ПО, при котором вместо попытки решать каждую новую задачу с нуля поощряется ее решение, основанное на предыдущих достижениях и на расширении их результатов. Его смысл состоит в экономии - зачем повторять то, что уже однажды было сделано? - и в скромности, в духе известного замечания Ньютона, что он смог достичь таких высот только потому, что стоял на плечах гигантов.

Полное преимущество этого подхода лучше всего понимается в терминах принципа Открыт-Закрыт, введенного в одной из предыдущих лекций. (Стоило бы перечитать этот раздел в свете только что введенных понятий.) Этот принцип утверждает, что хорошая структура модуля должна быть и закрытой, и открытой.

- Закрытой, поскольку клиентам для выполнения их собственной разработки нужны службы модуля и, будучи один раз зафиксированы в некоторой его версии, они не должны изменяться при введении новых служб, в которых клиент не нуждается.
- Открытой, так как нет никакой гарантии, что с самого начала в модуль были включены все службы, потенциально необходимые некоторому клиенту.

Эти два требования представляют дилемму, и классическая структура модулей не дает ключа к ее разгадке. Но наследование эту проблему решает. Класс закрыт, так как он может компилироваться, заноситься в библиотеку и использоваться классами-клиентами. Но он также открыт, поскольку любой новый класс может его использовать в качестве родителя, добавляя новые компоненты и меняя объявления некоторых унаследованных компонентов, при этом совершенно не нужно изменять исходный класс и беспокоить его клиентов. Это фундаментальное свойство при применении наследования к построению повторно используемого расширяемого ПО.

Если бы довести эту идею до предела, то каждый класс просто добавлял бы один компонент к его родителям! Конечно, это не рекомендуется. Решение завершить класс не следует принимать легковесно, оно должно основываться на осознанном заключении о том, что класс в его нынешнем состоянии уже обеспечивает логически последовательный набор служб - стройную абстракцию данных - для потенциальных клиентов.

Следует помнить, что принцип Открыт-Закрыт не отменяет последующей переделки неадекватных служб. Если плохой результат явился следствием неверной спецификации компонента, то мы не сможем модифицировать класс так, чтобы это не отразилось на его клиентах. Однако, благодаря переопределению, принцип Открыт-Закрыт все еще

применим, если вводимое изменение согласовано с объявленной спецификацией.

Одним из самых трудных вопросов, связанных с проектированием повторно используемых структур модулей, была необходимость использовать преимущества большой общности, которая может существовать у разных однотипных групп абстракций данных - у всех хеш-таблиц, всех последовательных таблиц и т. п. Используя структуры классов, связанных наследованием, можно получить выигрыш, зная логические соотношения между разными реализациями. Внизу на диаграмме представлен грубый и частичный набросок возможной структуры библиотеки для работы с таблицами. В этой схеме естественно используется множественное наследование, которое будет детально обсуждаться в следующей лекции.

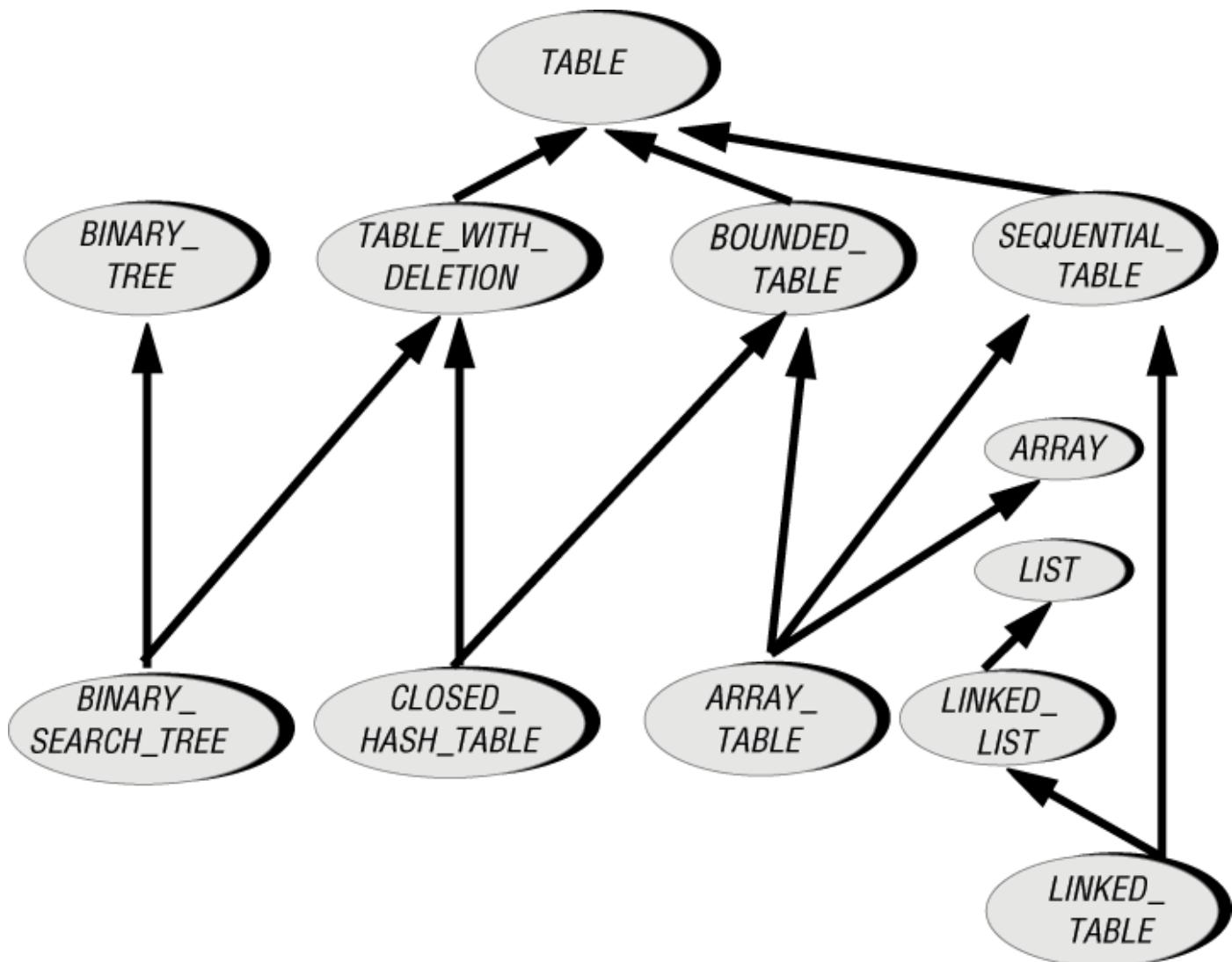


Рис. 14.12. Набросок структуры библиотеки таблиц

Эта диаграмма наследования представляет только набросок, хотя на ней показаны типичные для этих структур связи по наследованию. Систематическую классификацию таблиц и других контейнеров, основанную на наследовании, см. в [М 1994а].

При таком взгляде требование повторного использования можно выразить весьма точно: идея состоит в том, чтобы передвинуть определение каждого компонента как можно выше в иерархии наследования так, чтобы он мог наследоваться максимально возможным числом классов-потомков. Можно представлять этот процесс как **игру переиспользования**, в которую играют на доске, представляющей иерархии наследования (такие, как на [рис. 14.12](#)), фигурами, представляющими компоненты. Выигрывает тот, кто сможет в результате открытия абстракций более высокого уровня передвинуть как можно больше компонентов как можно выше, и по пути, благодаря обнаружению общих свойств, сможет слить наибольшее число фигур.

Взгляд на класс как на тип

С точки зрения типов наследование адресуется и к повторному использованию, и к расширяемости, в частности, к тому, что в предыдущем обсуждении называлось непрерывностью. Здесь ключом является динамическое связывание.

Тип - это множество объектов, характеризуемых (как мы знаем из теории АТД) определенными операциями. INTEGER описывают множество целых чисел с арифметическими операциями, POLYGON - это множество объектов с операциями vertices, perimeter и другими.

Для типов наследование представляет отношение "является", например, во фразах "каждая собака является млекопитающим", "каждое млекопитающее является животным". Аналогично, прямоугольник является многоугольником.

Что означает это отношение?

- Если рассматривать значения каждого типа, то это отношение является просто отношением включения множеств: собаки образуют подмножество множества животных, экземпляры класса RECTANGLE образуют подмножество экземпляров класса POLYGON. (Это следует из определения "экземпляра" в начале этой лекции, заметим, что прямой экземпляр класса RECTANGLE не является прямым экземпляром класса POLYGON).
- Если рассматривать операции, применимые к каждому типу, то сказать, что В есть А, означает, что каждая операция, применимая к А применима также и к экземплярам В. (Однако при переопределении В может создать свою собственную реализацию, которая для экземпляров В заменит реализацию, предоставленную А.)

Используя это отношение можно описывать схемы отношения "является", представляющие многие варианты типов, например, все варианты класса FIGURE. Каждая новая версия таких подпрограмм как *rotate* и *display* определяется в классе, задающем соответствующий вариант типа. В случае таблиц, например, каждый класс на графике обеспечивает свою собственную реализацию операций *search*, *insert*, *delete*, разумеется, за исключением тех случаев, когда для него подходит реализация родителя.

Предостережение об использовании отношения "является" ("is a"). Начинающие - но я полагаю, ни один из читателей, добрившийся до этого места даже с минимумом внимания, - иногда путают наследование с отношением "экземпляр - образец", считая класс SAN_FRANCISCO наследником класса CITY. Это, как правило, ошибка: CITY - это класс, у которого может быть экземпляр, представляющий Сан Франциско. Чтобы избежать таких ошибок, достаточно помнить, что термин "является" означает не "x является одним из A" (например, "Сан Франциско является городом (CITY)), т.е. отношением между экземпляром и категорией, а выражает "всякий B является A" (например, "всякий ГОРОД является ГЕОГРАФИЧЕСКОЙ_ЕДИНИЦЕЙ"), т.е. отношение между двумя категориями, в программировании - двумя классами. Некоторые авторы предпочитают называть это отношение "является разновидностью" или "может действовать как" [Gore 1996]. Отчасти это дело вкуса (и частично этот предмет будет обсуждаться в лекции о методологии наследования), но поскольку мы уже знаем, как избежать тривиальной ошибки, то будем и далее использовать наиболее распространенное название "является", не забывая при этом, что оно относится к отношению между категориями.

Наследование и децентрализация

Имея динамическое связывание, можно создавать **децентрализованные архитектуры ПО**, необходимые для достижения целей повторного использования и расширяемости. Сравним ОО-подход, при котором самодостаточные классы предоставляют свои множества вариантов операций, с классическими подходами. В Паскале или Аде можно использовать тип записи с вариантами

```
type FIGURE =
  record
    "Общие поля"
    case figtype: (polygon, rectangle, triangle, circle,...) of
      polygon: (vertices: LIST_OF_POINTS; count: INTEGER);
      rectangle: (side1, side2: REAL;...);
    ...
  end
```

чтобы определить различные виды фигур. Но это означает, что всякая программа, которая должна работать с фигурами (поворачивать и т.п.) должна проводить разбор возможных случаев:

```
case f.figure_type of
  polygon: ...
  circle: ...
...
end
```

В случае таблиц процедура *search* должна была бы использовать ту же структуру. Неприятность состоит в том, что эти процедуры должны обладать чересчур большими знаниями о будущем всей системы: они должны точно знать, какие типы фигур в ней допускаются. Любое добавление нового типа или изменение существующего будет затрагивать каждую процедуру.

Ne sutor ultra crepidam, (для сапожника ничего сверх сандалий) - это принцип разработки ПО: процедуре поворота не требуется знать полный список типов фигур. Ей должно хватать информации необходимой для выполнения своей работы: поворота некоторых видов фигур.

Распределение информации среди чересчур большого количества процедур является главным источником негибкости классических подходов к разработке ПО. Основные трудности модификации ПО можно проследить, анализируя эту проблему. Она также частично объясняет, почему так трудно управлять программными проектами, когда совсем

небольшие изменения имеют далеко идущие последствия, заставляя разработчиков переделывать модули, которые, казалось бы, были успешно завершены.

ОО-методы также сталкиваются с этой проблемой. Изменение реализации операции затрагивает только тот класс, в котором применяется эта реализация. Добавление нового варианта некоторого типа в большинстве случаев не затронет другие классы. Причиной является децентрализация: классы заведуют своими собственными реализациями и не вмешиваются в дела друг друга. В применении к людям это звучало бы как Вольтеровское **Cultivez votre jardin**, - ухаживайте за своим собственным садом. В применении к модулям существенным является требование получения децентрализованных структур, которые изящно поддаются расширению, модификации, комбинированию и повторному использованию.

Независимость от представления

Динамическое связывание связано с одним из принципиальных аспектов повторного использования: независимостью от представления, т.е. возможностью запрашивать выполнение некоторой операции, имеющей несколько вариантов, не уточняя, какой из них будет применен. В предыдущей лекции при обсуждении этого понятия использовался пример вызова

```
present := has (x, t)
```

который должен применить подходящий алгоритм поиска, зависящий от вида *t* во время выполнения. Если *t* объявлена как таблица, но может присоединяться к экземпляру бинарного дерева поиска, хеш-таблице и т. п. (в предположении, что все необходимые классы доступны), то при динамическом связывании вызов

```
present := t.has (x)
```

найдет во время выполнения подходящую версию процедуры *has*. С помощью динамического связывания достигается то, что было невозможно получить с помощью перегрузки и универсальности: клиент может запросить некоторую операцию, а поддерживающая язык система автоматически найдет ее соответствующую реализацию.

Таким образом, объединение классов, наследования, переопределения, полиморфизма и динамического связывания дают прекрасные ответы на вопросы, поставленные в начале этой книги: требования повторного использования, критерии, принципы и правила модульности.

Парадокс расширения-специализации

Наследование иногда рассматривается как расширение, а иногда как специализация. Хотя эти два толкования как будто противоречат друг другу, оба они истинны - но с разных точек зрения.

Все снова зависит от того, смотрим ли мы на класс как на тип или как на модуль. В первом случае наследование, представляющее отношение "является", - это специализация: "собака" более специальное понятие, чем "животное", а "прямоугольник" - чем "многоугольник". Как уже отмечалось, это соответствует отношению включения подмножества во множество: если в наследник *A*, то множество объектов, представляющих во время выполнения *B* является подмножеством соответствующего множества для *A*.

Но с точки зрения модуля, при которой класс рассматривается как поставщик служб, в реализует службы *A* и свои собственные. Малому числу объектов часто позволяют иметь больше компонентов, так как это приводит к увеличению информации. Переходя от произвольных животных к собакам, мы можем добавить специфическое для них свойство "лаять", а при переходе от многоугольников к прямоугольникам можно добавить компонент "диагональ". Поэтому по отношению к реализованным компонентам отношение включения направлено в другую сторону: компоненты, применимые к экземплярам *A*, являются подмножеством компонент, применимых к экземплярам *B*.

>Здесь мы говорим о реализуемых компонентах, а не о предлагаемых (клиентам) службах, потому что при соединении скрытия информации с наследованием, как мы увидим, в может скрыть от клиентов некоторые из компонентов, в то время как *A* их экспортовал своим клиентам.

Таким образом, наследование является специализацией с точки зрения типов и расширением с точки зрения модулей. Это и есть парадокс расширения-специализации: чем больше применяемых компонентов, тем меньше объектов, к которым они применяются.

Парадокс расширения-специализации - это одна из причин для устранения термина "подкласс", предполагающего понятие "подмножество". Другой, уже отмеченной, является встречающееся в литературе сбывающее с толку использование термина "подкласс" для обозначения как прямого, так и непрямого наследования. Эти проблемы не возникают при использовании точно определенных терминов: **наследник, потомок и собственный потомок** и двойственных к ним терминов: **родитель, предок и собственный предок**.

Роль отложенных классов

Отложенные классы являются одним из важнейших связанных с наследованием механизмов, предназначенных для решения описанных в начале книги проблем конструирования ПО.

Назад к абстрактным типам данных

Насыщенные утверждениями отложенные классы хорошо подходят для представления АТД. Прекрасный пример - отложенный класс для стеков. Мы уже описывали процедуру `put`, сейчас приведем возможную версию полного описания этого класса.

```
indexing
description:
    "Стеки (распределительные структуры с дисциплиной Last-in, First-Out), %
     %не зависящие от выбора представления"
deferred class
    STACK [G]
feature -- Доступ
    count: INTEGER is
        -- Число элементов.
        deferred
        end
    item: G is
        -- Последний вставленный элемент.
        require
            not_empty: not empty
        deferred
        end
feature - Отчет о статусе
    empty: BOOLEAN is
        -- Стек пустой?
        do
            Result := (count = 0)
        end
    full: BOOLEAN is
        -- Стек заполнен?
        deferred
        end
feature - Изменение элемента
    put (x: G) is
        -- Втолкнуть x на вершину.
        require
            not full
        deferred
        ensure
            not_empty: not empty
            pushed_is_top: item = x
            one_more: count = old count + 1
        end
    remove is
        -- Вытолкнуть верхний элемент.
        require
            not empty
        deferred
        ensure
            not_full: not full
            one_less: count = old count - 1
        end
    change_top (x: T) is
        -- Заменить верхний элемент на x
        require
            not_empty: not empty
        do
            remove; put (x)
        ensure
            not_empty: not empty
            new_top: item = x
            same_number_of_items: count = old count
        end
    wipe_out is
        -- Удалить все элементы.
        deferred
        ensure
            no_more_elements: empty
        end
invariant
```

```

non_negative_count: count >= 0
empty_count: empty = (count = 0)
end

```

Этот класс показывает, как можно реализовать эффективную процедуру, используя отложенные: например, процедура `change_top` реализована в виде последовательных вызовов процедур `get_top` и `put`. (Такая реализация для некоторых представлений, например, для массивов, может оказаться не самой лучшей, но эффективные потомки класса `STACK` могут ее переопределить.)

Если сравнить класс `STACK` со спецификацией соответствующего АТД, приведенной в [лекции 6](#), то обнаружится удивительное сходство. Подчеркнем, в частности, соответствие между функциями АТД и компонентами класса, и между пунктом PRECONDITIONS и предусловиями процедур. Аксиомы представлены в постусловиях процедур и в инварианте класса.

Добавление операций `change_top`, `count` и `wipe_out` в данном случае несущественно, так как они легко могут быть включены в спецификацию АТД (см. упражнение У6.8). Отсутствие явного эквивалента функции `new` из АТД также несущественно, так как созданием объектов будут заниматься процедуры-конструкторы в эффективных потомках этого класса. Остаются три существенных отличия.

Первое из них - это введение функции `full`, рассчитанной на реализации с ограниченным числом элементов стека, например, на реализацию массивами. Это типичный пример ограничения, которое несущественно на уровне спецификации, но необходимо для разработки практических систем. Отметим однако, что это отличие между АТД и отложенным классом можно легко устранить, включив в спецификацию АТД средства для охвата ограниченных стеков. При этом общность не будет потеряна, так как некоторые реализации (например, с помощью списков) могут реализовывать `full` тривиальными процедурами, всегда возвращающими ложь.

Второе отличие, отмеченное при обсуждении разработки по контракту, состоит в том, что спецификация АТД полностью аппликативна (функциональна), она включает функции без побочных эффектов. А отложенный класс, несмотря на его абстрактность, является императивным (процедурным), например `put` определена как процедура, изменяющая стек, а не как функция, которая берет в качестве аргумента один стек и возвращает другой.

Наконец, как тоже уже отмечалось, механизм утверждений недостаточно выразителен для некоторых аксиом АТД. Из четырех аксиом стеков

Для всех $x: G, s: \text{STACK } [G]$,

1. $\text{item}(\text{put}(s, x)) = x$
2. $\text{remove}(\text{put}(s, x)) = s$
3. $\text{empty}(\text{new})$
4. $\text{not empty}(\text{put}(s, x))$

все, кроме (2), имеют прямые эквиваленты среди утверждений. (Мы предполагаем, что для (3) процедуры-конструкторы у потомков обеспечивают выполнение условия `empty`). Причины таких ограничений уже были объяснены и были намечены возможные пути их преодоления - языки формальных спецификаций IFL.

Отложенные классы как частичные интерпретации: классы поведения

Не все отложенные классы так близки к АТД как `STACK`. В промежутке между полностью абстрактным классом, таким как `STACK`, в котором все существенные компоненты отложены, и эффективным классом, таким как `FIXED_STACK`, описывающим единственную реализацию АТД, имеется место для реализаций АТД с различной степенью завершенности.

Типичным примером является иерархия реализаций таблиц, которая помогла нам понять роль частичной общности при изучении повторного использования. Первоначальный рисунок, показывающий отношения между вариантами, можно сейчас перерисовать в виде диаграммы наследования.

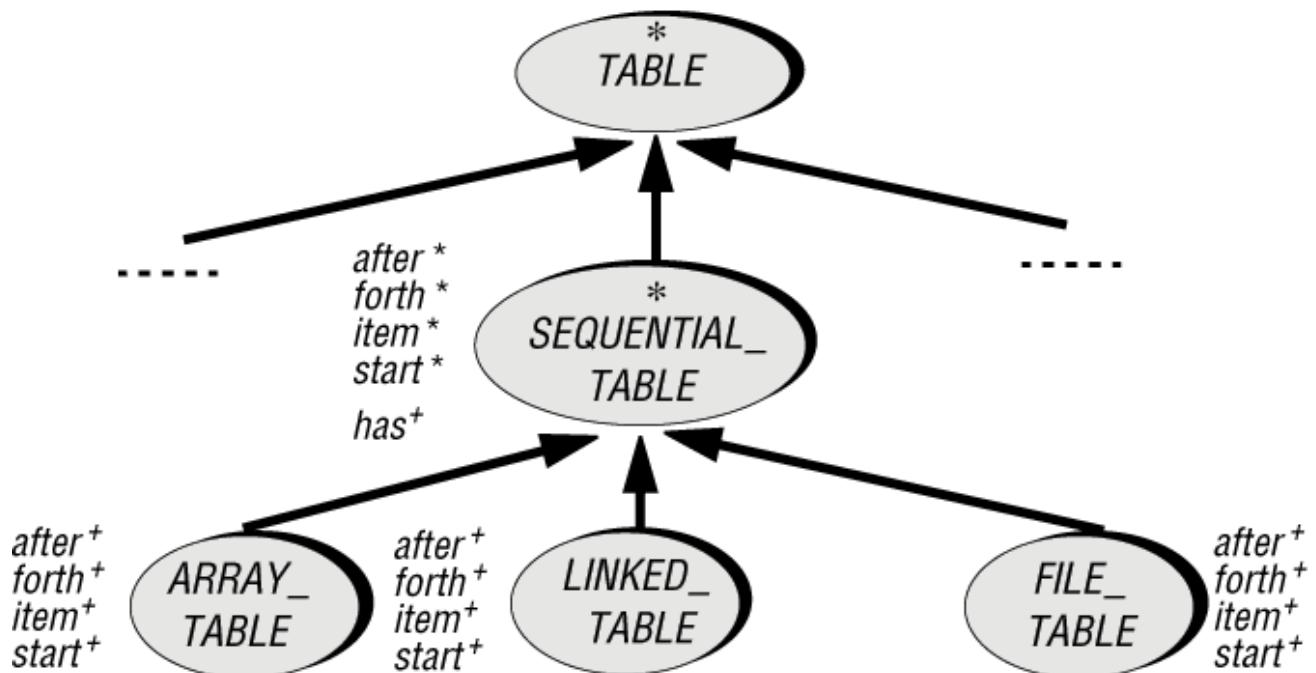


Рис. 14.13. Варианты понятия "таблица"

Наиболее общий класс TABLE является полностью или почти полностью отложенным, так как на этом уровне мы можем объявить несколько компонентов, но не можем предложить никакой существенной их реализации. Среди вариантов имеется класс SEQUENTIAL_TABLE, представляющий таблицы, в которые элементы вставляются последовательно. Примерами таких таблиц являются массивы, связанные списки и последовательные файлы. Соответствующие им классы в нижней части рисунка являются эффективными.

Особый интерес представляют такие классы как SEQUENTIAL_TABLE. Этот класс все еще отложенный, но его статус находится посредине между полностью отложенным статусом как у класса TABLE и полностью эффективным как у ARRAY_TABLE. У него достаточно информации, чтобы позволить себе реализацию некоторых специфических алгоритмов, например, в нем можно полностью реализовать последовательный поиск:

```

has (x: G): BOOLEAN is
    -- x имеется в таблице?
do
    from start until after or else equal (item, x) loop
        forth
    end
    Result := not after
end
    
```

Эта функция эффективна, хотя ее алгоритм использует отложенные компоненты. Компоненты **start** (поместить курсор в первую позицию), **forth** (сдвинуть курсор на одну позицию), **item** (значение элемента в позиции курсора), **after** (находится ли курсор за последним элементом?) являются отложенными в классе SEQUENTIAL_TABLE и в каждом из показанных на рисунке потомков этого класса они реализуются по-разному.

Эти реализации были приведены при обсуждении повторного использования. Например класс ARRAY_TABLE может представлять курсор числом **i**, так что процедура **start** реализуется как **i := 1**, а **item** как **t @ i** и т.д.

Отметим важность включения предусловия и постусловия компонента **forth**, а также инварианта объемлющего класса для гарантирования того, что все будущие реализации будут удовлетворять одной и той же базовой спецификации. Эти утверждения приводились ранее в этой лекции (в несколько ином контексте для класса LIST, но непосредственно применимы и здесь).

Это обсуждение в полной степени показывает соответствие между классами и АТД:

- Полностью отложенный класс, такой как TABLE, соответствует АТД.
- Полностью эффективный класс, такой как ARRAY_TABLE, соответствует реализации АТД.
- Частично отложенный класс, такой как SEQUENTIAL_TABLE, соответствует семейству реализаций (или, что эквивалентно, частичной реализации) АТД.

Такой класс как SEQUENTIAL_TABLE, аккумулирующий черты, свойственные некоторым вариантам АТД, можно назвать **классом поведения (behavior class)**. Классы поведения предоставляют важные образцы для конструирования ОО-ПО.

Класс SEQUENTIAL_TABLE дает представление о том, как ОО-технология, используя понятие класса поведения, отвечает на последний оставшийся открытый в [лекции 4](#) вопрос о "Факторизации общих поведений".

Особенно интересна возможность определения такой эффективной процедуры в классе поведения, которая использует в своей реализации отложенные процедуры. Эта возможность проиллюстрирована выше процедурой has. Она показывает, как можно использовать частично отложенные классы для того, чтобы зафиксировать общее поведение нескольких вариантов. В отложенном классе описывается только то общее, что у всех них имеется, а описание вариаций остается потомкам.

Ряд примеров в последующих лекциях будет базироваться на этом методе, который играет важную роль в применении ОО-методов к построению повторно используемого ПО. Он особенно полезен при создании библиотек для конкретных предметных областей и реально применяется во многих контекстах. Типичным примером, описанным в [М 1994а], является разработка библиотек Lex и Parse, предназначенных для анализа языков. В частности, Parse определяет общую схему разбора, по которой будет обрабатываться любой текст (формат данных для языка программирования и т.п.), структура которого соответствует некоторой грамматике. Классы поведения высокого уровня содержат небольшое число отложенных компонентов, таких как post_action, описывающих семантические действия, которые должны выполняться после разбора некоторой конструкции. Для определения собственной семантической обработки пользователю достаточно реализовать эти компоненты.

Такая схема широко распространена. В частности, бизнес-приложения часто следуют стандартным образцам - обработать полученные за день счета, выполнить соответствующую проверку требований на платежи, ввести новых заказчиков и так далее, - индивидуальные компоненты которых могут варьироваться.

В таких случаях можно предоставить набор классов поведения со смесью эффективных компонент, описывающих известную часть, и отложенных компонент, задающих изменяемые элементы. Как правило, эффективные компоненты будут вызывать в своих телах отложенные. При таком подходе потомки могут создавать реализации, удовлетворяющие их потребностям.

Не все изменяемые элементы следует откладывать. Если доступна реализация по умолчанию, то ее следует включить в качестве эффективного компонента, который при необходимости можно переопределить на уровне потомка. Это упростит разработку потомков, так как в них нужно будет реализовывать новые версии лишь тех компонент, которые отличаются от реализаций по умолчанию. Разумеется, такой метод следует применять лишь при наличии подходящей реализации по умолчанию, в противном случае соответствующий компонент следует объявить отложенным (как, например, display в классе FIGURE).

Этот метод является частью более общего подхода, который можно окрестить "Не вызывайте нас, мы вызовем вас": не прикладная система вызывает повторно используемые примитивы, а универсальная схема позволяет разработчикам приложений размещать их собственные варианты в стратегических местах.

Эта идея не является абсолютно новой. Древняя и весьма почтенная СУБД IMS фирмы IBM уже использовала нечто в этом роде. Структура управления графических систем (таких как система X для Unix) включает "цикл по событиям", в котором на каждой итерации вызываются специфические функции, поставляемые разработчиками приложений. Этот подход известен как схема обратного вызова (**callback scheme**).

То, что предлагает ОО-метод, благодаря классам поведения, представляет систематическую, обеспечивающую безопасность поддержку этой техники разработки. Эта поддержка включает классы, наследование, проверку типов, отложенные классы и компоненты, а также утверждения, позволяющие разработчику сразу зафиксировать, каким условиям должны всегда удовлетворять изменяемые элементы.

Программы с дырами

Только что обсужденные методы являются центральным вкладом ОО-подхода в повторное использование: они предлагают не замороженные навсегда компоненты (которые можно обнаружить в библиотеках подпрограмм), а гибкие решения, которые предоставляют базисные схемы и могут быть адаптированы к нуждам многих разнообразных приложений.

Одной из центральных тем при обсуждении повторного использования была необходимость соединить эту цель с адаптивностью во избежание дилеммы: **переиспользовать** или **переделывать**. Этому в точности соответствует только что описанная схема, для которой можно предложить название "программы с дырами". В отличие от библиотек подпрограмм, в которых все, кроме значений фактических параметров, жестко фиксировано, у программ с дырами, использующих классы, образцом для которых служит модель SEQUENTIAL_TABLE, имеется место для частей, создаваемых пользователем.

Эти наблюдения помогают понять образ "блока Лего", часто используемый при обсуждении повторного использования. В наборе Лего компоненты фиксированы, детская фантазия направлена на составление из них интересной структуры. Тот же подход свойственен и программированию, - истоки его в традиционных библиотеках подпрограмм. Часто при разработке ПО требуется в точности обратное: сохранять структуру, но заменять компоненты. На самом деле, этих компонентов может еще и не быть, на их места помещаются "заглушки" (отложенные компоненты), вместо которых затем нужно вставить эффективные варианты.

По аналогии с детскими игрушками можно вернуться в детство и представить себе игровую доску с отверстиями разной формы, в которые ребенок должен вставлять соответствующие фигуры. Он должен понять, что квадратный

блок подходит для квадратного отверстия, а круглый блок - для круглого отверстия.

Можно также представлять частично отложенный класс поведения (или набор таких классов, называемый "библиотекой"), как устройство с несколькими электрическими розетками - отложенными классами - в которые разработчик приложения будет вставлять совместимые с ними устройства. Эту метафору можно продолжить: для устройства важны меры предосторожности - утверждения, выражающие требования к допустимым съемным устройствам, например, спецификация розетки определяет допустимое напряжение, силу тока и другие электрические параметры.

Роль отложенных классов при анализе и глобальном проектировании

Отложенные классы играют также ключевую роль при использовании ОО-метода не только на уровне реализации, но и на самых ранних и верхних уровнях построения системы - анализе и глобальном проектировании. Целью является создание спецификации системы и ее архитектуры, для проекта требуется также абстрактное описание каждого модуля без деталей его реализации.

Обычно даваемая в этом случае рекомендация состоит в использовании отдельных обозначений: некоторого "метода" анализа (за этим термином во многих случаях стоит просто некоторая графическая нотация) и некоторого ЯПП (PDL) (языка проектирования программ, зачастую тоже графического). Но у этого подхода много недостатков:

- Разрыв между последовательными шагами процесса разработки представляет серьезную угрозу для качества ПО. Необходимость трансляции из одного формализма в другой может привести к ошибкам и подвергает опасности целостность системы. ОО-технология, напротив, предлагает перспективу непрерывного процесса разработки ПО.
- Многоярусный подход является особенно губительным для этапов сопровождения и эволюции системы. Крайне сложно гарантировать согласованность проекта и реализации на этих этапах.
- Наконец, большинство существующих подходов к анализу и проектированию не предлагают никакой поддержки формальной спецификации функциональных свойств модулей, не зависящей от их реализации, например в форме утверждений.

Последний комментарий приводит к **парадоксу уровней**: точная нотация, подобная языку, используемому в этой книге, иногда отклоняется как "низкоуровневая" или "ориентированная на реализацию", поскольку внешне выглядит как язык программирования. На самом же деле, благодаря утверждениям и такому механизму абстракции как отложенные классы, их уровень существенно выше уровня большинства имеющихся подходов к анализу и проектированию. Многим требуется время, чтобы осознать это, поскольку раньше их учили тому, что высокий уровень абстракции означает неопределенность и что абстракция всегда должна быть неточной.

Использование отложенных классов для анализа и проектирования позволяет нам одновременно быть абстрактными и точными, и применять один и тот же язык на протяжении всего процесса разработки. При этом устраняются разрывы в концепциях, переход от описания модуля на высоком уровне к реализациям может происходить плавно внутри одного формализма. Даже нереализованные операции проектируемых модулей, представленные отложенными процедурами, можно достаточно точно охарактеризовать с помощью предусловий, постуловий и инвариантов.

Система обозначений, которая к этому моменту развернута почти до конца, покрывает этапы анализа и проектирования, а также и реализации. Одни и те же понятия и конструкции применяются на всех стадиях, различаются только уровни абстракции и детализации.

Обсуждение

В этой лекции введены основные понятия, связанные с наследованием. Оценим сейчас достоинства некоторых введенных соглашений. Дальнейшие комментарии о механизме наследования (в частности, о множественном наследовании) появятся в следующей лекции.

Явное переопределение

Роль предложения **redefine** состоит в улучшении читаемости и надежности. Компиляторам, на самом деле, оно не нужно, так как в классе может быть лишь один компонент с данным именем, то объявленный в данном классе компонент, имеющий то же имя, что и компонент некоторого предка, может быть только переопределением этого компонента (или ошибкой).

Не следует пренебрегать возможностью ошибки, так как программист может наследовать некоторый класс, не зная всех компонентов, объявленных в его предках. Для избежания этой опасности требуется явно указать каждое переопределение. В этом и состоит основная роль предложения **redefine**, которое также полезно при чтении класса.

Доступ к предшественнику процедуры

Напомним правило использования конструкции **Precursor** (. . .): она может появляться только в переопределяемой версии процедуры.

Этим обеспечивается цель введения этой конструкции: позволить новому определению использовать первоначальную реализацию. При этом возможность явного указания родителя устраниет всякую неопределенность (в частности, при множественном наследовании). Если бы допускался доступ любой процедуры к любому компоненту предков, то текст

класса было бы трудно понять, читателю все время приходилось бы обращаться к текстам многих других классов.

Динамическое связывание и эффективность

Можно подумать, что сила механизма динамического связывания приведет во время выполнения к недопустимым накладным расходам. Такая опасность существует, но аккуратное проектирование языка и хорошие методы его реализации могут ее предотвратить.

Дело в том, что динамическое связывание требует несколько большего объема действий во время выполнения. Сравним вызов обычной процедуры в традиционном языке программирования (Pascal, Ada, C, ...)

1. `f (x, a, b, c...)`

с ОО-формой

2. `x.f (a, b, c...)`

Разница между этими двумя формами уже была разъяснена при введении понятия класса, для идентификации типа модуля. Но сейчас мы понимаем, что это связано не только со стилем, имеется также различие и в семантике. В форме (1), какой именно компонент обозначает имя `f` известно статически во время компиляции или, в худшем случае, во время компоновки, если для объединения раздельно откомпилированных модулей используется компоновщик. Однако при динамическом связывании такая информация недоступна статически: для `f` в форме (2) выбор компонента зависит от объекта, к которому присоединен `x` во время конкретного выполнения. Каким будет этот тип нельзя (в общем случае) определить по тексту программы, это служит источником гибкости этого ранее разрекламированного механизма.

Предположим вначале, что динамическое связывание реализовано наивно. Во время выполнения хранится копия иерархии классов. Каждый объект содержит информацию о своем типе - вершине в этой иерархии. Чтобы интерпретировать во время выполнения `x.f`, окружение ищет соответствующую вершину и проверяет, содержит ли этот класс компонент `f`. Если да, то прекрасно, мы нашли то, что требовалось. Если нет, то переходим к вершине-родителю и повторяем всю операцию. Может потребоваться проделать путь до самого верхнего класса (или нескольких таких классов в случае множественного наследования).

В типизированном языке нахождение подходящего компонента гарантировано, но в нетипизированном языке, таком как Smalltalk, поиск может быть неудачным, и придется завершить выполнение диагнозом "сообщение не понято".

Такая схема все еще применяется с различными оптимизациями во многих реализациях не статически типизированных языков. Она приводит к существенным затратам, снижающим эффективность. Хуже того, эти затраты не прогнозируемые и **растут с увеличением глубины структуры наследования**, так как алгоритм может постоянно проходить путь до корня иерархии наследования. Это приводит к конфликту между повторным использованием и эффективностью, поскольку упорная работа над повторное использование может приводить к введению дополнительных уровней наследования. Представьте состояние бедного разработчика, который перед добавлением нового уровня наследования должен оценить, как это ударит по эффективности. Нельзя ставить разработчиков ПО перед таким выбором.

Такой подход является одним из главных источников неэффективности реализаций языка Smalltalk. Это также объясняет, почему он (по крайней мере, в коммерческих реализациях) не поддерживает множественного наследования. Причина - в том, что из-за необходимости обходить весь граф, а не одну ветвь, накладные расходы оказываются чрезмерными.

К счастью, использование статической типизации устраняет эти неприятности. При правильно построенной системе типов и алгоритмах компиляции нет никакой нужды перемещаться по структуре наследования во время выполнения. Для ОО-языка со статической типизацией возможные типы `x` не произвольны, а ограничены потомками исходного типа `x`, поэтому компилятор может упростить работу системы выполнения, построив массив структурных данных, содержащих всю необходимую информацию. При наличии этих структур данных накладные расходы на динамическое связывание сильно уменьшаются: они сводятся к **вычислению индекса и доступу к массиву**. Важно не только то, что такие затраты невелики, но и то, что они ограничены **константой**, и поэтому можно не беспокоиться о рассмотренной выше проблеме соотношения между переиспользованием и эффективностью. Будет ли структура наследования в вашей системе иметь глубину 2 или 20, будет ли в ней 100 классов или 10000, максимальные накладные расходы всегда одни и те же. Они не зависят и от того, является ли наследование единичным или множественным.

Открытие в 1985г. этого свойства, т.е. того, что даже при множественном наследовании можно реализовать вызов динамически связываемого компонента за константное время, было главным побудительным толчком к разработке проекта (среди прочего приведшего к появлению первого и настоящего изданий этой книги), направленного на построение современного окружения для разработки ПО, отталкивающегося от идей великолепно введенных языками Simula 67, распространенных затем на множественное наследование (длительный опыт применения Симулы показал, что непозволительно ограничиваться единичным наследованием), приведенных в соответствие с принципами современной программной инженерии и объединяющих эти идеи с самыми полезными результатами формальных подходов к спецификации, построению и верификации ПО. Создание эффективного механизма динамического

связывания за константное время, которое на первый взгляд может показаться второстепенным среди указанного набора целей, на самом деле было настоятельно необходимым.

Эти наблюдения могут показаться странными тому, кто познакомился с ОО-технологией через линзу представлений об ОО-анализе и проектировании, в которых реализация и эффективность рассматриваются как приземленные предметы, которыми следует заниматься после того, как решено все остальное. Эффективность является одним из ключевых факторов, который должен рассматриваться на всех шагах, когда речь идет о реальной разработке промышленного ПО, о реальной увязке инженерных решений. Как отмечалось в одной из предыдущих лекций, если вы откажетесь от эффективности, то эффективность откажется от вас. Конечно, ОО-технология это нечто большее, чем динамическое связывание за константное время, но без этого не могло бы быть никакой успешной ОО-технологии.

Оценка накладных расходов

Оказывается, можно грубо оценить потери на накладные расходы для описанных выше методов динамического связывания. Следующие цифры взяты из опытов ISE по использованию динамического связывания (даные получены при отключении объясняемой ниже оптимизации статического связывания).

Для процедуры, которая ничего не делает, т. е. описана как **p1 is do end**, превышение времени динамического связывания над временем статического связывания (например, над эквивалентной процедурой на C) составляет около 30%.

Это, конечно, оценка сверху, поскольку реальные процедуры что-нибудь да делают. Цена динамического связывания одинакова для всех процедур независимо от времени их выполнения, поэтому, чем больший объем вычислений выполняет процедура, тем меньше относительная доля накладных расходов. Если вместо p1 использовать процедуру, которая выполняет некоторые типичные операции, такую как

```
p2 (a, b, c: INTEGER) is
  local
    x, y
  do
    x := a; y := b + c + 1; x := x * y; p2
    if x > y then x := x + 1 else x := x - 1 end
  end
```

то накладные расходы падают до 15%. Для программы, выполняющей нечто более существенное (например, некоторый цикл) их доля совсем мала.

Статическое связывание как оптимизация

В некоторых случаях главным требованием является эффективность, и даже указанные выше небольшие накладные расходы нежелательны. В этом случае можно заметить, что они не всегда обоснованы. Вызов **x.f (a, b, c...)** не нуждается в динамическом связывании в следующих случаях:

1. **f** нигде в системе не переопределяется (имеет только одно объявление);
2. **x** не является полиморфной, иначе говоря, не является целью никакого присоединения, источник которого имеет другой тип.

В любом из таких случаев, выявляемых хорошим компилятором, сгенерированный для **x.f (a, b, c...)** код может быть таким же, как и код, генерируемый компиляторами C, Pascal, Ada или Fortran для вызова **f (x, a, b, c...)**. Никакие накладные расходы не потребуются.

Компилятор ISE, являющийся частью окружения, описанного в последней лекции, сейчас выполняет оптимизацию (1), планируется добавить и (2) (анализ (2) является, фактически, следствием механизмов анализа типов, описанных в лекции о типизации).

Хотя (1) интересно и само по себе, непосредственная его польза ограничивается сравнительно низкой стоимостью динамического связывания (см. приведенную выше статистику). Настоящий выигрыш от него непрямой, поскольку (1) дает возможность третьей оптимизации:

3. При любой возможности применять **автоматическую подстановку кода процедуры**.

Такая подстановка означает расширение тела программы текстом вызываемой процедуры в месте ее вызова. Например, для процедуры

```
set_a (x: SOME_TYPE) is
  -- Сделать x новым значением атрибута a.
  do
    a := x
  end
```

компилятор может сгенерировать для вызова `s.set_a(some_value)` такой же код, какой компилятор Pascal генерирует для присваивания `s.a := some_value` (недопустимое для нас обозначение, поскольку оно нарушает скрытие информации). В этом случае вообще нет накладных расходов, поскольку сгенерированный код не содержит вызова процедуры.

Подстановка кода традиционно рассматривается как оптимизация, которую должны задавать **программисты**. Ada включает прагму (указание транслятору) `inline`, С и C++ предлагают аналогичные механизмы. Но этому подходу присущи внутренние ограничения. Хотя для небольшой, статичной программы компетентный программист может сам определить, какие процедуры можно подставлять, для больших развивающихся проектов это сделать невозможно. В этом случае компилятор с приличным алгоритмом определения подстановок будет намного превосходить догадки программистов.

Для каждого вызова, к которому применимо автоматическое статическое связывание (1), ОО-компилятор может определить, основываясь на анализе соотношения между временем и памятью, стоит ли применять автоматическую подстановку кода процедуры (3). Это одна из самых поразительных оптимизаций - одна из причин, по которой можно достичь эффективности произведенного вручную кода Си или Фортрана, а иногда, на больших системах и превзойти ее.

К улучшению эффективности, растущему с увеличением размера и сложности программ, автоматическая подстановка кода добавляет преимущество большей надежности и гибкости. Как уже отмечалось, подстановка кода семантически корректна только для процедуры, которую можно статически ограничить, например, как в случаях (1) и (2). Это не только допустимо, но также вполне согласуется с ОО-методом, в частности, с принципом Открыт-Закрыт, если разработчик на полпути разработки большой системы добавит переопределение некоторого компонента, имевшего к этому моменту только одну реализацию. Если же код процедуры вставляется вручную, то в результате может получиться программа с ошибочной семантикой (поскольку в данном случае требуется динамическое связывание, а вставка кода, конечно, означает статическое связывание). Разработчики должны сосредотачиваться на построении корректных программ, не занимаясь утомительными оптимизациями, которые при выполнении вручную приводят к ошибкам, а на деле могут быть автоматизированы.

Имеются и некоторые другие требования для того, чтобы подстановка кода была корректной, в частности, она применима только к нерекурсивным вызовам. Даже корректную подстановку следует применять при разумном соотношении между временем и памятью: подставляемая процедура должна быть небольшой и должна вызываться небольшое число раз.

Последнее замечание об эффективности. Опубликованная статистика для ОО-языков показывает, что где-то от 30% до 60% вызовов на самом деле используют динамическое связывание. Это зависит от того, насколько интенсивно разработчики используют специфические свойства методов. В системе ISE это соотношение близко к 60%. С использованием только что описанных оптимизаций платить придется только за динамическое связывание только тех вызовов, которые действительно в нем нуждаются. Для оставшихся динамических вызовов накладные расходы не только малы (ограничены константой), но и логически необходимы, - в большинстве случаев для достижения результата, эквивалентного динамическому связыванию, придется использовать условные операторы (`if ... then ...` или `case ... of ...`), которые могут оказаться дороже приведенного выше простого механизма, основанного на доступе к массивам. Поэтому неудивительно, что ОО-программы, откомпилированные хорошим компилятором, могут соревноваться с написанным вручную кодом на С.

Кнопка под другим именем: когда статическое связывание ошибочно

К этому моменту должен стать понятным главный вывод из изложенных в этой лекции принципов наследования:

Принцип динамического связывания

Если результат статического связывания не совпадает с результатом динамического связывания, то такое статическое связывание семантически некорректно.

Рассмотрим вызов `x.r`. Если `x` объявлена типа А, но в процессе вычисления была присоединена к объекту типа В, а в классе В компонент `r` переопределен, то использование в этом вызове исходной версии `r` из класса А - это не вопрос выбора, это просто ошибка!

Безусловно, имелись причины для переопределения `r`. Одной из них могла быть оптимизация, как в случае с компонентом `perimeter` в классе `RECTANGLE`, но могло также оказаться, что исходная версия `r` просто некорректно работает для объектов из В. Рассмотрим, например, эскизно описанный класс `BUTTON` (КОНОПКА), являющийся наследником класса `WINDOW` (ОКНО) в некоторой оконной системе (кнопки являются специальным видом окон). В этом классе переопределена процедура `display`, так как изображение кнопки немного отличается от изображения обычного окна (например, нужно показать ее рамку). В этом случае, если `w` имеет объявленный тип `WINDOW`, но динамически связана, благодаря полиморфизму, с объектом типа `BUTTON`, то вызов `w.display` **должен** исполняться для "кнопочной" версии! Использование `display` из класса `WINDOW` приведет к искажению изображения на экране.

Мы не должны позволить, чтобы нас обманула гибкость системы типов, основанная на наследовании, особенно ее правило совместимости типов, позволяющее объявлять сущность на уровне абстракции более высоком, чем уровень типа присоединенного объекта во время конкретного выполнения. Во время выполнения программы единственное, что имеет значение, - это те объекты, к которым применяются компоненты, а сущности - имена в тексте программы - уже

давно забыты. Кнопка под любым именем остается кнопкой, независимо от того, названа ли она в программе кнопкой или присоединена к сущности типа окно.

Это рассуждение можно подкрепить некоторым математическим анализом. Напомним условие корректности процедуры из [лекции 11](#) об утверждениях:

$\{ \text{pre}_r(x_r) \text{ and } \text{INV} \} \text{ Body}_r \{ \text{post}_r(x_r) \text{ and } \text{INV} \}$.

Для целей нашего обсуждения его можно немного упростить, оставив только часть, относящуюся к инвариантам классов, опустив аргументы и используя в качестве индекса имя класса A:

[A-CORRECT]

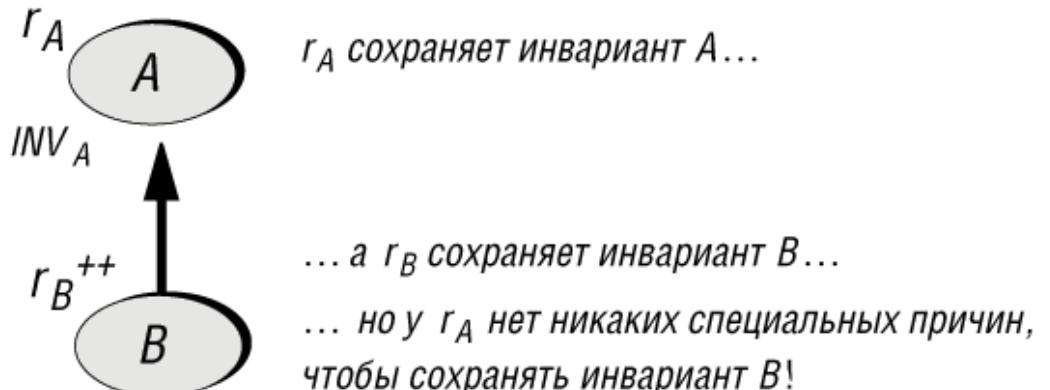
$\{\text{INV}_A\} r_A \{\text{INV}_A\}$

Содержательно это означает, что всякое выполнение процедуры r из класса A сохраняет инвариант этого класса. Предположим теперь, что мы переопределили r в некотором собственном потомке B. Соответствующее свойство будет выполняться, если новый класс корректен:

[B-CORRECT]

$\{\text{INV}_B\} r_B \{\text{INV}_B\}$

Напомним, что инварианты накапливаются при движении вниз по структуре наследования, так что INV_B влечет INV_A , но, как правило, не наоборот.



$\text{INV}_B = \text{INV}_A \text{ and }$ другие_предложения

Рис. 14.14. Версия родителя может не удовлетворять новому инварианту

Напомним, например, как RECTANGLE добавляет собственные условия к инварианту класса POLYGON. Другой пример, рассмотренный при изучении инвариантов в [лекции 11](#), это класс ACCOUNT1 с компонентами withdrawals_list и deposits_list; его собственный потомок ACCOUNT2 добавляет к нему, возможно, по соображениям эффективности, новый атрибут balance для постоянного запоминания текущего баланса счета. К инварианту добавляется новое предложение:

```
consistent_balance: deposits_list.total - withdrawals_list.total = current_balance
```

Из-за этого, возможно, придется переопределить некоторые из процедур класса ACCOUNT1; например, процедура deposit, которая использовалась просто для добавления элемента в список deposits_list, сейчас должна будет модифицировать также balance. Иначе класс просто станет ошибочным. Это аналогично тому, что версия процедуры display из класса WINDOW не является корректной для экземпляра класса BUTTON.

Предположим теперь, что к объекту типа B, достижимому через сущность типа A, применяется статическое связывание. При этом из-за того, что соответствующая версия процедуры r_A , как правило, не будет поддерживать необходимый инвариант (как, например, deposit_{ACCOUNT1} для объектов типа ACCOUNT2 или display_{WINDOW} для объектов типа BUTTON), будет получаться неверный объект (например, объект класса ACCOUNT2 с неправильным полем balance или объект класса BUTTON, неправильно показанный на экране).

Такой результат - объект, не удовлетворяющий инварианту своего класса, т.е. основным, универсальным ограничениям на все объекты такого вида - является одним из самых страшных событий, которые могут случиться во время выполнения программы. Если такая ситуация может возникнуть, то нечего надеяться на верный результат вычисления.

Суммируем: **статическое связывание является либо оптимизацией, либо ошибкой**. Если его семантика совпадает с семантикой динамического связывания (как в случаях (1) и (2)), то оно является оптимизацией, которую может выполнить компилятор. Если у него другая семантика, то это ошибка.

Подход языка C++ к связыванию

Учитывая широкое распространение и влияние языка C++ на другие языки, нужно разъяснить, как в нем решаются некоторые из обсуждаемых здесь вопросов.

Соглашения, принятые в C++, кажутся странными. По умолчанию связывание является статическим. Чтобы процедура (в терминах C++ - функция или метод) связывалась динамически, она должна быть специально объявлена как виртуальная (**virtual**).

Это означает, что приняты два решения:

1. Сделать программиста ответственным за выбор статического или динамического связывания.
2. Использовать статическое связывание в качестве предопределенного.

Оба нарушают ОО-разработку ПО, но в различной степени: (1) можно попробовать объяснить, а (2) защищать трудно.

По сравнению с подходом этой книги (1) ведет к другому пониманию того, какие задачи должны выполняться людьми (разработчиками ПО), а какие - компьютерами (более точно, компиляторами). Это та же проблема, с которой мы столкнулись при обсуждении автоматического распределения памяти. Подход C++ продолжает традиции C и дает программисту полный контроль над тем, что случится во время выполнения, будь то размещение объекта или вызов процедуры. В отличие от этого, в духе ОО-технологии стремление переложить на плечи компилятора все утомительные задачи, выполнение которых вручную приводит к ошибкам, и для которых имеются подходящие алгоритмы. В крупном масштабе и на большом промежутке времени компиляторы всегда справляются с работой лучше.

Конечно, разработчики отвечают за эффективность их программ, но они должны сосредотачивать свои усилия на том, что может действительно существенно повлиять на результат: на выборе подходящих структур данных и алгоритмов. За все остальное несут ответственность разработчики языков и компиляторов.

Отсюда и несогласие с решением (1): C++ считает, что статическое связывание, как и подстановка кода, должно определяться разработчиками, а развиваемый в этой книге ОО-подход полагает, что за это отвечает компилятор, который будет сам оптимизировать вызовы. Статическое связывание - это оптимизация, а не выбор семантики.

Для ОО-метода имеется еще одно негативное последствие (1). Всегда при определении процедуры требуется указать политику связывания: является она виртуальной или нет, т.е. будет связываться динамически или статически. Такая политика противоречит принципу Открыт-Закрыт, так как заставляет разработчика с самого начала угадывать, что будет переопределяться, а что - нет. Это не соответствует тому, как работает наследование: на практике может потребоваться переопределить некоторый компонент в далеком потомке класса, при проектировании которого нельзя было это предвидеть. При подходе C++, если разработчик исходного класса такого не предусмотрел, то придется снова вернуться к этому классу, чтобы изменить объявление компонента на **virtual**. При этом предполагается, что исходный текст доступен для модификации. А если его нет, или у разработчика нет права его менять, то вас ожидает горькая участь.

По этим причинам решение (1), требующее, чтобы программисты сами задавали политику связывания, мешает эффективному применению ОО-метода.

Решение (2) - использовать статическое связывание в качестве предопределенного - еще хуже. Очень трудно подобрать доводы в его пользу с точки зрения проектирования языка. Как мы видели, выбор статического связывания всегда приводит к ошибкам, если его семантика отличается от динамического. Поэтому не может быть никаких причин для его выбора в качестве предопределенного.

Одно дело - сделать программистов, а не компиляторы ответственными за оптимизацию в безопасных случаях (т.е. попросить их явно указывать статическое связывание, если они считают, что это корректно), но заставлять их писать нечто специальное, чтобы получить корректную семантику - это совсем другое. Если верно или неверно понятые соображения эффективности начинают брать верх над основополагающим требованием корректности ПО, то что-то не в порядке.

Даже в языке, заставляющем программиста отвечать за выбор политики связывания (такое решение принято в C), предопределенное значение должно быть противоположным. Вместо того, чтобы требовать объявлять динамически связываемые функции виртуальными (**virtual**), язык должен был бы использовать динамическое связывание по умолчанию и разрешить программистам выделять словом **static** (или каким-нибудь другим) компоненты, для которых они хотели бы запросить оптимизацию, доверив им самим (в традиции C и C++) удостоверяться в том, что она допустима.

Это различие особенно важно для начинающих, которые, естественно, имеют тенденцию доверять значениям по умолчанию. Даже для языка, менее страшного, чем C++, нельзя предполагать, что кто-либо сразу справится со всеми деталями наследования. Ответственный подход к этому должен гарантировать корректную семантику для новичков (и вообще, для разработчиков, начинающих новый проект, которые "хотят чтобы прежде всего он был правильным, а уж затем быстрым"), а затем предоставить возможности оптимизации для тех, кому это требуется и кто хорошо разбирается в предмете.

Имея в виду широко распространенный интерес к "совместимости снизу - вверх", создание комитета для изменения политики связывания в C++, особенно пункта (2), будет тяжелым делом, но стоит попытаться пролить свет на опасность нынешних соглашений.

Прискорбно, но подход C++ влияет и на другие языки, например, политика динамического связывания в языке Borland Delphi, продолжающем прежние расширения Паскаля, по сути, та же, что и в C++. Отметим все же, что вышедший из недр C++ язык Java в качестве базового использует динамическое связывание.

Эти наблюдения позволяют дать некоторый практический совет. Что разработчик может сделать при использовании C++ или иного языка с той же политикой связывания? Самым лучшим для разработчиков, не имеющих возможности переключиться на другие средства или ждать улучшений в этом языке, было бы объявлять **все** функции как виртуальные и тем самым разрешить их любые переопределения в духе ОО-разработки ПО. (К сожалению, некоторые компиляторы C++ ограничивают число виртуальных функций в системе, но можно надеяться, что эти ограничения будут сняты).

Парадокс этого совета в том, что он возвращает нас назад к ситуации, в которой все вызовы реализуются через динамическое связывание и требуют несколько большего времени выполнения. Иными словами, соглашения (1) и (2) языка C++, предназначенные для улучшения эффективности, в конце концов, если следовать правилу: "корректность прежде всего", срабатывают против этого!

Неудивительно, что эксперты по C++ не советуют использовать "чересчур много" объектной ориентированности. Уолтер Брайт (Walter Bright), автор одного из самых популярных компиляторов C++, пишет в [Bright 1995]:

Хорошо известно, что чем больше C++ [механизмов] вы используете в некотором классе, тем медленнее его код. К счастью, есть несколько вещей, позволяющих склонить чашу весов в вашу пользу. Во-первых, не используйте без большой необходимости виртуальные функции [т. е. динамическое связывание], виртуальные базовые классы [отложенные классы], деструкторы и т.п. Другой источник разбухания - это множественное наследование [...]. Если у вас сложная иерархия классов с одной или двумя виртуальными функциями, то попробуйте устраниТЬ виртуальный аспект и, быть может, сделать то же самое, используя проверки и ветвления.

Иными словами: не прибегайте к использованию ОО-методов. (В том же тексте отстаивается и "группировка всех кодов инициализации" для локализации ссылки - приглашение нарушить элементарные принципы модульного проектирования, которые, как мы видели, предполагают, что каждый класс должен сам отвечать за все, связанное с его инициализацией.)

В этой лекции предложен другой подход: в первую очередь разработчик ОО-ПО должен быть уверен в том, что семантика вызова всегда будет правильной, а это гарантируется динамическим связыванием. Затем можно использовать достаточно изощренные методы компиляции, чтобы порождать статическое связывание или подстановку кода для тех вызовов, которые, как установлено на основе строгого алгоритмического анализа, не требуют динамического связывания.

Ключевые концепции

- С помощью наследования можно определять новые классы как расширение, специализацию и комбинацию ранее определенных классов.
- Класс, наследующий другому классу, называется его наследником, а исходный класс - его родителем. Распространенные на произвольное число уровней (включая ноль) эти понятия становятся понятиями потомка и предка.
- Наследование является ключевым методом как для повторного использования, так и для расширяемости.
- Плодотворное применение наследования требует переопределения (предоставления классу возможности переписать реализацию некоторых компонентов его собственного предка), полиморфизма (возможности связывать ссылку во время выполнения с экземплярами разных классов), динамического связывания (динамического выбора подходящего варианта переопределенного компонента), совместности типов (требования, чтобы всякая сущность могла присоединяться только к экземплярам типов-наследников).
- С точки зрения модулей наследник расширяет набор служб, предоставляемых его родителями. В частности, это полезно для повторного использования.
- С точки зрения типов отношение между наследником и его родителем - это отношение "**является**". Оно полезно как для повторного использования, так и для расширяемости.
- Функцию без аргументов можно переопределить как атрибут, но не наоборот.
- Методы наследования, в особенности, динамическое связывание, позволяют разрабатывать децентрализованную архитектуру, в которой каждый вариант операции определяется в том же модуле, где описан соответствующий вариант структуры данных.
- Для типизированных языков динамическое связывание можно реализовать с малыми накладными расходами. Связанные с ним оптимизации, в частности, применяемое компилятором статическое связывание и подстановка кода, помогают ОО-программам достичь или превзойти эффективность выполнения традиционных программ.
- Отложенные классы содержат один или более отложенный (не реализованный) компонент. Они описывают частичные реализации абстрактных типов данных.
- Способность эффективных подпрограмм вызывать отложенные позволяет примирить с помощью "классов поведения" повторное использование с расширяемостью.
- Отложенные классы являются основным средством, используемым ОО-методами на стадиях анализа и проектирования.
- Утверждения, применяемые к отложенным компонентам, позволяют точно специфицировать отложенные

классы.

- Если семантики динамического и статического связывания различны, то всегда нужно выбирать динамическое связывание. Если же они действуют одинаково, то статическое связывание следует рассматривать как оптимизацию, которую лучше возложить на компилятор. Компилятор может проверить и безопасно применить как эту оптимизацию, так и оптимизацию, связанную с подстановкой кода подпрограммы в точках вызова.

Библиографические замечания

Понятия (единичного) наследования и динамического связывания были введены в языке Симула 67, на который можно найти ссылки в [лекции 17](#) курса "Основы объектно-ориентированного проектирования". Отложенные процедуры - это тоже изобретение Симулы (под другим именем (виртуальные процедуры) и при других соглашениях).

Отношение "является" изучалось, в основном, с точки зрения приложений искусственного интеллекта в [Brachman 1983].

Формальное изучение наследования и его семантики проведено в [Cardelli 1984].

Соглашение об использовании для переопределения двойного плюса пришло из системы обозначений Business Object Notation, предложенной Nerson'ом и Walden'ом (ссылки в [лекции 9](#) курса "Основы объектно-ориентированного проектирования").

Конструкция `Precursor` (аналогичная конструкции `super` в языке Smalltalk, но с важным отличием, разрешающим ее использовать только для переопределения процедур) является результатом неопубликованной совместной работы с Roger Browne, James McKim, Kim Walden и Steve Tynor.

Упражнения

У14.1 Многоугольники и прямоугольники

Дополните версии классов `POLYGON` и `RECTANGLE`, наброски которых приведены в начале лекции. Включите в них подходящие процедуры создания.

У14.2 Многоугольник с малым числом вершин

Инвариант класса `POLYGON` требует, чтобы у каждого многоугольника было, по крайней мере, три вершины; отметим, что функция `perimeter` не будет работать для пустого многоугольника. Измените определение этого класса так, чтобы он покрывал и случаи вырожденных многоугольников с числом вершин меньше трех.

У14.3 Геометрические объекты с двумя координатами

Опишите класс `TWO_COORD`, задающий объекты с двумя вещественными координатами, среди наследников которого были бы классы `POINT` (ТОЧКА), `COMPLEX` (КОМПЛЕКСНОЕ_ЧИСЛО) и `VECTOR` (ВЕКТОР). Будьте внимательны при помещении каждого компонента на подходящий для него уровень иерархии.

У14.4 Наследование без классов

В этой лекции были представлены два взгляда на наследование: будучи модулем, класс-наследник предлагает службы своего родителя плюс еще некоторые, будучи типом, он реализует отношение "является" (каждый экземпляр наследника является также экземпляром каждого из родителей). "Пакетами" модульных, но не ОО-языков (таких как Ada (Ada) или Modula-2 (Modula-2)) являются модули, но не типы. При первой интерпретации к ним можно было бы применить наследование. Обсудите, в каком виде наследование может быть введено в модульные языки. Не забудьте рассмотреть при этом принцип Открыт-Закрыт.

У14.5 Классы без объектов

Не разрешается создавать объекты отложенных классов. В одной из предыдущих лекций был указан другой способ создания класса без объектов: включить в него пустую процедуру создания. Эквивалентны ли эти два механизма? Можно ли выделить случаи, когда использование одного из них предпочтительнее, чем другого? (Указание: в отложенном классе должен быть хоть один отложенный компонент.)

У14.6 Отложенные классы и прототип

Отложенные классы нельзя инициализировать. С другой стороны, были приведены аргументы в пользу того, чтобы в первой версии класса в проекте все компоненты оставались отложенными. Может появиться желание "выполнить" такой проект: при проектировании ПО иногда хочется вступить в игру как можно раньше, исполнить неполные реализации, чтобы получить практический опыт и проверить некоторые аспекты системы даже при неполностью реализованных других аспектах. Обсудите доводы за и против того, чтобы иметь в компиляторе специальную параметр "прототип", позволяющий инициализировать отложенный класс и выполнить отложенный компонент (как пустую операцию). Обсудите детали.

У14.7 Библиотека поиска в таблицах (семестровый проект)

Основываясь на обсуждении таблиц в этой лекции и в лекции о повторном использовании, спроектируйте библиотеку классов таблиц, включающую различные категории представлений таблиц: хеш-таблицы, последовательные (линейные) таблицы, древообразные таблицы и др.

У14.8 Виды отложенных компонентов

Может ли атрибут быть отложенным?

У14.9 Комплексные числа

(Это упражнение предполагает знакомство со всеми лекциями вплоть до 5-й курса "Основы объектно-ориентированного проектирования".) В примере, рассмотренном при обсуждении интерфейса модулей, использовались комплексные числа с двумя разными представлениями, при этом соответствующие изменения в представлениях остались "за кадром". Определите можно ли получить эквивалентный результат с помощью наследования, а именно, создать класс COMPLEX (КОМПЛЕКСНЫЕ) и его наследников CARTESIAN_COMPLEX (КОМПЛЕКСНЫЕ_В_ДЕКАРТОВЫХ_КООРДИНАТАХ) и POLAR_COMPLEX (КОМПЛЕКСНЫЕ_В_ПОЛЯРНЫХ_КООРДИНАТАХ).

Основы объектно-ориентированного программирования

15. Лекция: Множественное наследование

Полноценное применение наследования требует важного расширения этого механизма. Изучая его основы, мы столкнулись с необходимостью порождать новые классы от нескольких классов-родителей. Эта возможность, известная как множественное (multiple) наследование (именуемое так в противовес единичному (single) наследованию), действительно нужна для построения надежных ОО-решений. Множественное наследование это, по сути, прямое приложение уже рассмотренных принципов наследования, - класс вправе иметь произвольное число родителей. Однако, изучая этот вопрос более внимательно, можно обнаружить две интересные проблемы: потребность в смене имен компонентов, которая может оказаться полезной и при единичном наследовании; дублируемое (repeated) наследование, при котором два класса связаны отношением предок-потомок более чем одним способом.

Примеры множественного наследования

Выясним, прежде всего, в каких ситуациях множественное наследование и в самом деле уместно. Для этого рассмотрим ряд типичных примеров, заимствованных из разных предметных областей.

Такой краткий экскурс тем более необходим, что несмотря на элегантность, простоту множественного наследования и реальную потребность в нем, демонстрация этого механизма подчас создает впечатление чего-то сложного и таинственного. И хотя эту точка зрения не подтверждает ни практика, ни теория, она распространилась достаточно широко, и теперь мы просто обязаны потратить немного времени на изучение случаев, в которых множественное наследование действительно совершенно необходимо.

Пример, неподходящий для введения

Сначала покончим с одним бытующим заблуждением. Для этого рассмотрим пример, приводимый (в том или ином виде) во многих статьях, книгах и лекциях, но зачастую порождающий недоверие к множественному наследованию. И дело не в том, что этот пример неверен; просто при первом знакомстве с проблемой он не может служить иллюстрацией, поскольку является собой образец нетипичного применения этого механизма.

В стандартной формулировке примера речь заходит о классах TEACHER и STUDENT, и вам тут же предлагают отметить тот факт, что отдельные студенты тоже преподают, и советуют ввести класс TEACHING_ASSISTANT, порожденный от TEACHER и STUDENT.

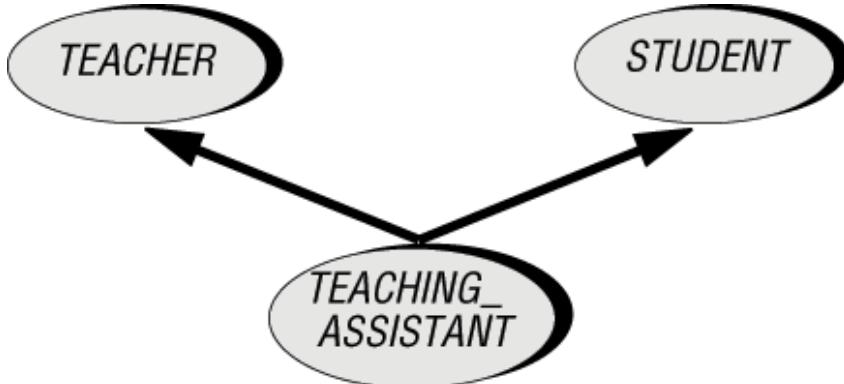


Рис. 15.1. Пример множественного наследования

Выходит, в этой схеме что-то не так? Не обязательно. Но как начальный пример он весьма неудачен. Все дело в том, что STUDENT и TEACHER - не отдельные абстрактные понятия, а вариации на одну тему UNIVERSITY_PERSON.

Поэтому, увидев картину в целом, мы обнаружим пример не просто множественного, но **дублируемого (repeated)** наследования - схемы, изучаемой позже в этой лекции, в которой класс является правильным наследником другого класса двумя или более различными путями:

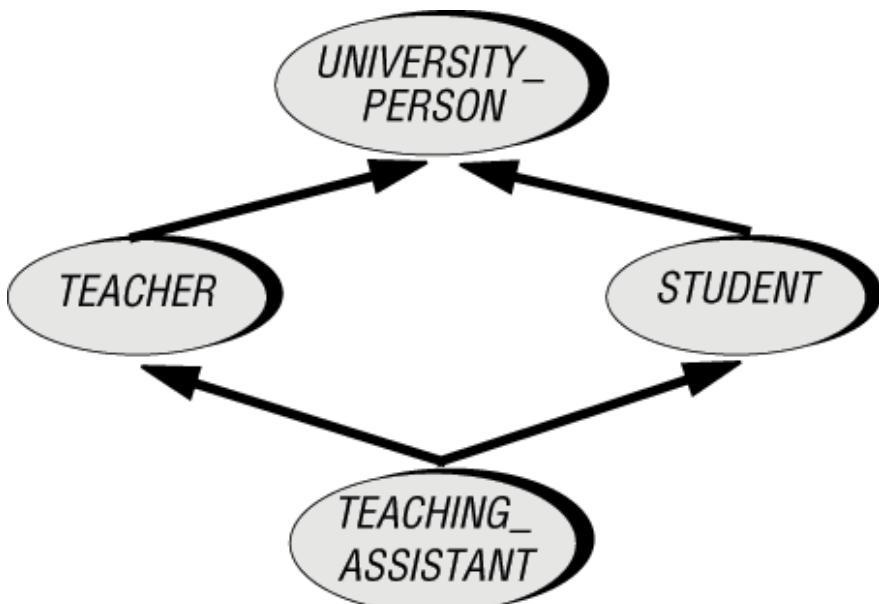


Рис. 15.2. А это пример дублируемого наследования

Дублируемое наследование - это особый случай. Его применение требует большого опыта в использовании более простых форм порождения классов. Этот пример нельзя обсуждать с начинающими просто потому, что он создает впечатление конфликтов между отдельными компонентами, наследуемых от обоих родителей, в то время как речь идет о свойстве, приходящем от общего предка. При правильном подходе исправить эту проблему не составит труда. Но было бы серьезной ошибкой начинать разговор с таких исключительных и непростых случаев, делая вид, будто они характерны для всего множественного наследования.

По-настоящему распространенные случаи множественного наследования не вызывают таких проблем. В их основе - не варианты одной, а сочетание **различных абстракций**. Именно это чаще всего и требуется при построении структур наследования, именно это и следует обсуждать при первом знакомстве с предметом. Дальнейшие примеры - из этой серии.

Может ли самолет быть имуществом?

Наш первый подходящий пример относится скорее к моделированию систем, чем к проектированию программных продуктов. Однако он наглядно иллюстрирует ситуацию, в которой множественное наследование необходимо.

Пусть класс AIRPLANE описывает самолет. Среди запросов к нему могут быть число пассажиров (passenger_count), высота (altitude), положение (position), скорость (speed); среди команд - взлететь (take_off), приземлиться (land), набирать скорость (set_speed).

Независимо от него может иметься класс ASSET, описывающий понятие имущества. К его компонентам можно отнести такие атрибуты и методы, как цена покупки (purchase_price), цена продажи (resale_value), уменьшить в цене (depreciate), перепродать (resell), внести очередной платеж (pay_installment).

Наверное, вы догадались, к чему мы клоним: компания ведь может владеть самолетом! И для пилота самолет компании это просто машина, способная взлетать, садиться, набирать скорость. Для финансиста это имущество, имеющее (очень высокую) цену покупки, (слишком низкую) цену продажи, и вынуждающее компанию ежемесячно платить по кредиту.

Для моделирования понятия "самолет компании" прибегнем к множественному наследованию:

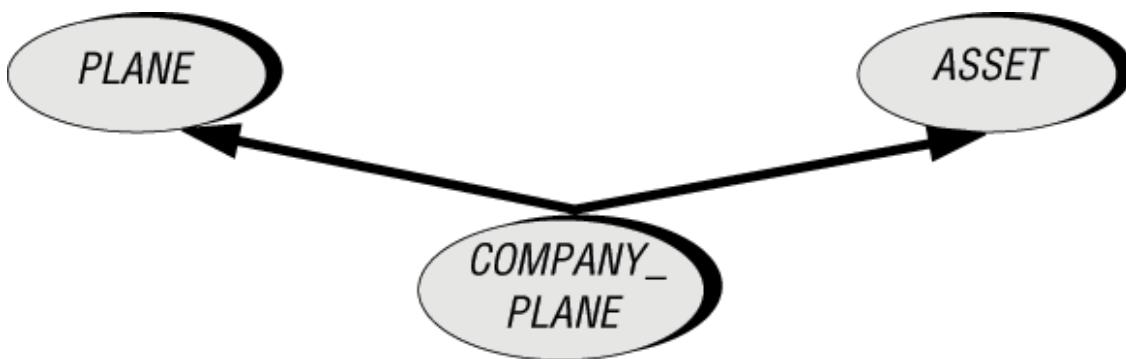


Рис. 15.3. Самолет компании

```

class COMPANY_PLANE inherit
  PLANE
  ASSET
  
```

```

feature
    ... Любой компонент, характерный для самолетов компании,
    (отличающийся от наследуемых компонентов родителей) ...
end

```

Родителей класса достаточно перечислить в предложении **inherit**. (Как обычно, можно разделять их имена точкой с запятой, хотя это не обязательно.) Порядок перечисления классов не играет никакой роли.

В моделировании систем найдется еще немало примеров, подобных COMPANY_PLANE.

- Наручные часы-калькулятор моделируются с применением множественного наследования. Один родитель позволяет устанавливать время и отвечать на такие запросы, как текущее время и текущая дата. Другой - электронный калькулятор - поддерживает арифметические операции.
- Наследником классов судно и грузовик является амфибия (AMPHIBIOUS_VEHICLE). Наследник классов: судно, самолет - гидросамолет (HYDROPLANE). (Как и с TEACHING_ASSISTANT, здесь также возможно дублируемое наследование, поскольку каждый из классов-родителей является потомком средства передвижения VEHICLE.)
- Ужин в ресторане; поездка в вагоне поезда - вагон-ресторан (EATING_CAR). Вариант: спальный вагон (SLEEPING_CAR).
- Диван-кровать (SOFA_BED), на котором можно не только читать, но и спать.
- "Дом на колесах" (MOBILE_HOME) - вид транспорта (VEHICLE) и жилище (HOUSE) одновременно; и так далее.

С точки зрения программиста эти примеры представляют академический интерес - нам платят за построение систем, а не за построение модели мира. Впрочем, во многих практических приложениях с аналогичными комбинациями абстрактных понятий вы обязательно столкнетесь. Более подробный пример из графической среды разработки ISE мы изложим чуть ниже.

Числовые и сравнимые значения

Следующий пример напрямую относится к повседневной практике ОО-разработки и неразрывно связан с построением библиотеки Kernel.

Ряд классов Kernel, потенциально необходимых всем приложениям, требуют поддержки таких операций арифметики, как **infix "+", infix "-", infix "**", prefix "-"**, а также специальных значений **zero** (единичный элемент группы с операцией "+") и **one** (единичный элемент группы с операцией "*"). Эти компоненты используют отдельные классы библиотеки Kernel: INTEGER, REAL и DOUBLE. Впрочем, они нужны и другим, заранее не определенным классам, например, классу MATRIX, который описывает матрицы определенного вида. Приведенные абстракции уместно объединить в отложенном классе NUMERIC, являющемся частью библиотеки Kernel:

```

deferred class NUMERIC feature
    ... infix "+", infix "-", infix "**", prefix "-", zero, one...
end

```

NUMERIC имеет строгое математическое определение. Его экземпляры служат для представления элементов кольца (множества с двумя операциями, каждая из которых индуцирует на нем группу, причем одна из операций коммутативна, а вторая дистрибутивна относительно первой).

Многим классам необходимо отношение порядка с операциями сравнения элементов. **Такая возможность полезна для классов Kernel, таких как STRING, и для многих других классов. Поэтому в состав библиотеки входит отложенный класс COMPARABLE:**

```

deferred class COMPARABLE feature
    ... infix "<", infix "<=", infix ">", infix ">="...
end

```

Математически его экземпляры - это полностью упорядоченные множества с заданным отношением порядком.

Не все потомки COMPARABLE должны быть потомками NUMERIC. В классе STRING арифметика не нужна, однако нужен порядок. Обратно, не все потомки NUMERIC должны быть потомками COMPARABLE. Так, на множестве матриц с действительными коэффициентами есть сложение, умножение, единица, нуль, что придает ей свойства кольца, но нет отношения порядка. Поэтому COMPARABLE и NUMERIC должны оставаться различными классами, и ни один из них не должен быть потомком другого.

Объекты некоторых типов, однако, имеют числовую природу и одновременно допускают сравнение. (Такие классы моделируют вполне упорядоченные кольца.) Примеры таких классов - REAL и INTEGER. Целые и действительные числа сравнивают, складывают и умножают. Их описание можно построить на множественном наследовании:

```

expanded class REAL inherit
    NUMERIC

```

```

COMPARABLE
feature
...
end

```

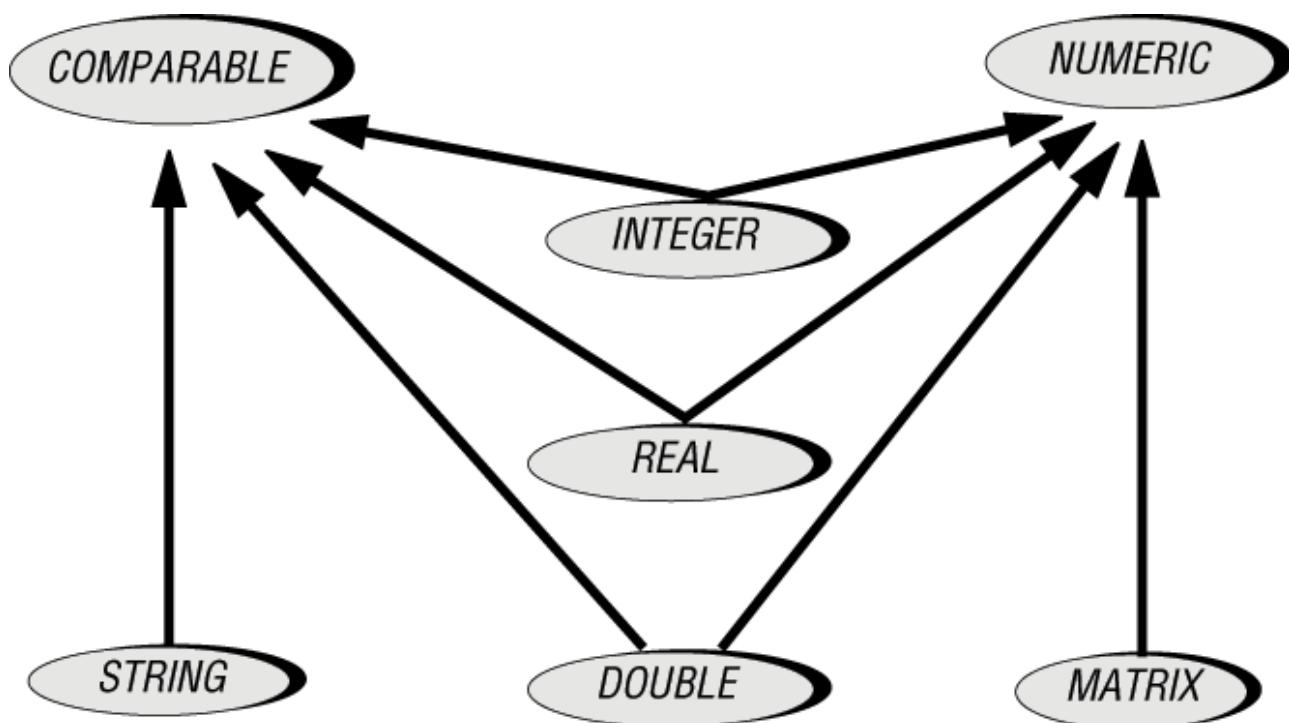


Рис. 15.4. Структура множественного и единичного наследования

Окна - это деревья и прямоугольники

Рассмотрим оконную систему с произвольной глубиной вложения окон:

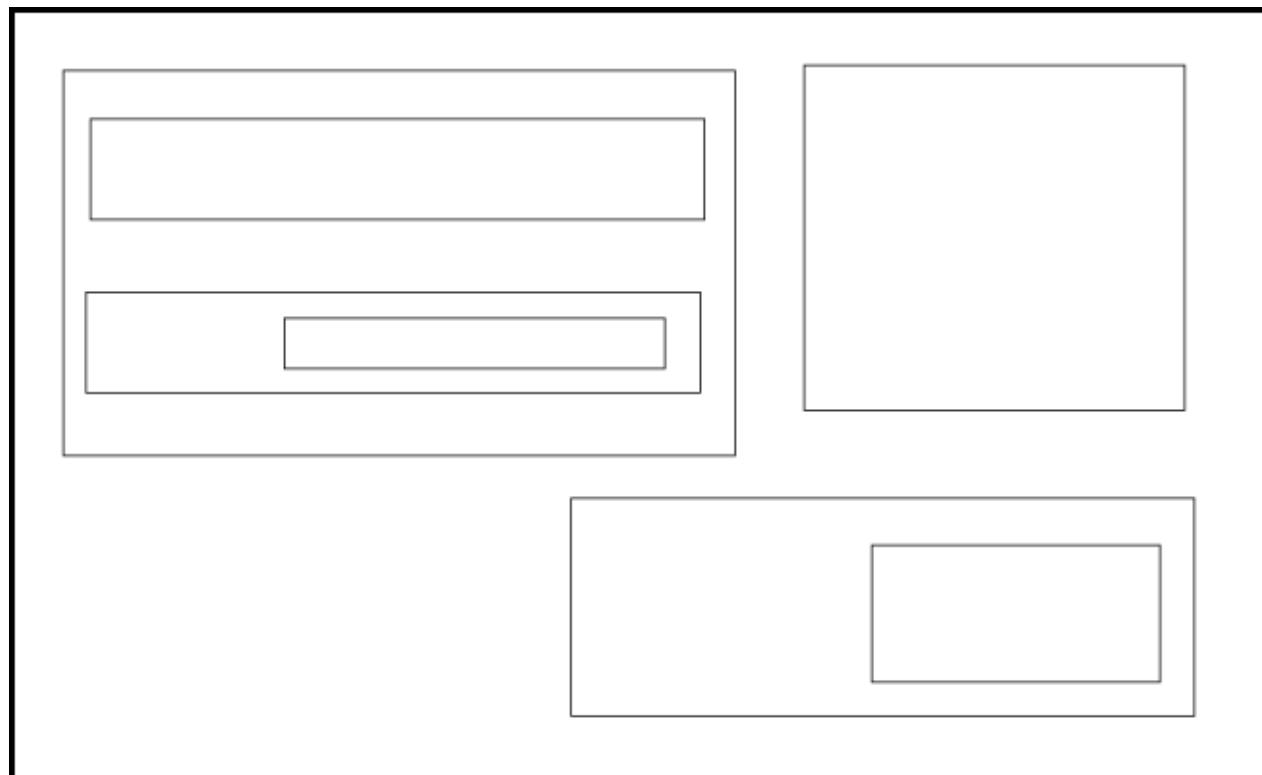


Рис. 15.5. Окна и подокна

В соответствующем классе WINDOW мы найдем компоненты двух основных видов:

- те, что рассматривают окно как иерархическую структуру (список подокон, родительское окно, число подокон, добавить, удалить подокно);
- те, что рассматривают окно как графический объект (высота, ширина, отобразить, спрятать, переместить окно).

Этот класс можно написать как единое целое, смешав все компоненты. Однако такой проект будет не самым

удачным. Класс WINDOW следует рассматривать как сочетание двух абстракций:

- иерархической структуры, представленной классом TREE;
- прямоугольного экранного объекта, представленного классом RECTANGLE.

На практике класс будет описан так:

```
class WINDOW inherit
    TREE [WINDOW]
    RECTANGLE
feature
    ... Характерные компоненты окна ...
end
```

Обратите внимание, класс TREE является родовым (generic) классом, а потому требует указания фактического родового параметра, здесь - самого класса WINDOW. Рекурсивная природа определения отражает рекурсию, присущую моделируемой ситуации, - окно является одновременно деревом окон.

Далее, можно подметить, что отдельные окна не содержат ничего, кроме текста. Этую особенность окон можно реализовать вложением, представив класс TEXT_WINDOW как клиента класса STRING, **введя** атрибут

```
text: STRING
```

Предпочтем, однако, вариант, в котором текстовое окно **является** одновременно строкой. В этом случае используем множественное наследование с родителями WINDOW и STRING. (Если же все наши окна содержат лишь текст, их можно сделать прямыми потомками TREE, RECTANGLE и STRING, однако и здесь решение "в два хода" возможно будет более предпочтительным.)

Деревья - это списки и их элементы

Класс дерева TREE - еще один яркий пример множественного наследования.

Деревом называется иерархическая структура, составленная из узлов с данными. Обычно ее определяют так: "Дерево либо пусто, либо содержит объект, именуемый его корнем, с присоединенным списком деревьев (рекурсивно определяемых) - потомков корневого узла". К этому добавляют определение **узла**: "Пустое дерево не содержит узлов; узлами непустого дерева являются его корень и по рекурсии узлы потомков". Эти определения, хотя и отражают рекурсивную сущность дерева, не способны показать его внутренней простоты.

Мы же заметим, что между понятиями дерева и узла нет серьезных различий. Узел можно определить как поддерево, корнем которого он является. В итоге приходим к классу TREE [G], который описывает как узлы, так и деревья. Формальный родовой параметр G отражает тип данных в каждом узле. Следующее дерево, является, например, экземпляром TREE [INTEGER]:

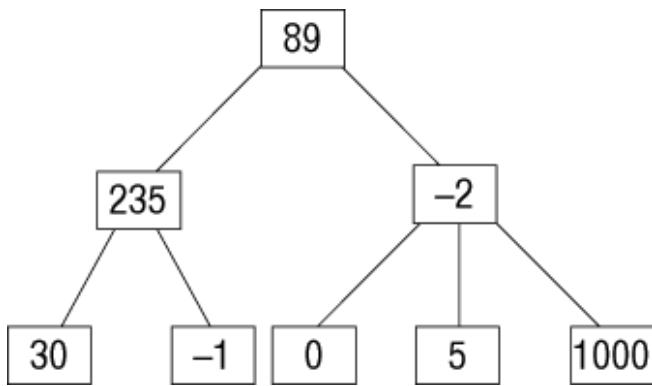


Рис. 15.6. Дерево целых чисел

Вспомним также о понятии списка, чей класс LIST рассмотрен в предыдущих лекциях. В общем случае его реализация требует введения класса CELL для представления его элементов структуры.

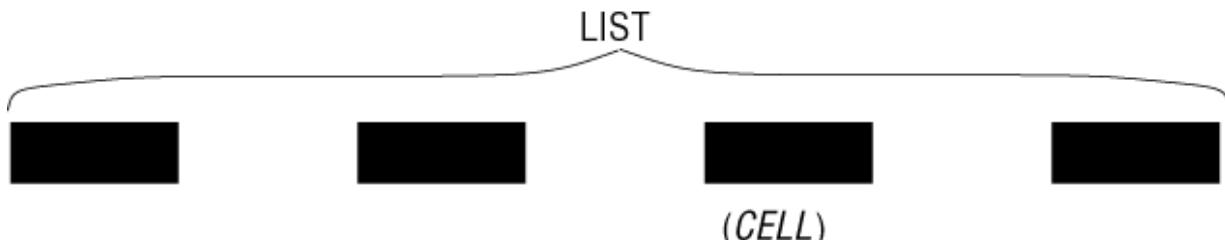


Рис. 15.7. Представление списка

Эти понятия позволяют прийти к простому определению дерева: дерево (или его узел) есть список, - список его потомков, но является также потенциальным элементом списка, поскольку может представлять поддерево другого дерева.

Определение: дерево

Дерево - это список и элемент списка одновременно.

Это определение еще потребует доработки, однако, уже сейчас позволяет описать класс:

```
deferred class TREE [G] inherit
    LIST [G]
    CELL [G]
feature
    ...
end
```

От класса LIST наследуются такие компоненты как количество узлов (`count`), добавление, удаление узлов и т. д.

От класса CELL наследуются компоненты, позволяющие работать с узлами, задающими родителя или братьев: следующий брат, добавить брата, присоединить к другому родителю.

Этот пример характерен тем, что иллюстрирует преимущества повторного использования при множественном наследовании. Создание специальных компонентов вставки или удаления поддеревьев означало бы повторение того, что уже сделано для списка элементов. Нам же остаются лишь косметические доработки.

Кроме того, следует позаботиться о добавлении в предложение `feature` специфических компонентов, присущих только деревьям, и компонентов, являющихся результатом взаимных компромиссов, неизбежных при любой свадьбе, и обеспечивающих взаимную гармонию родительских классов. Их текст невелик и займет в классе TREE чуть больше страницы, поскольку наш класс вполне законный плод союза списков и элементов списка.

Этот процесс подобен процессу, применяемому математиками при комбинировании теорий: топологическое векторное пространство является одновременно топологическим пространством и векторным пространством. Здесь тоже необходимы некоторые связующие аксиомы.

Составные фигуры

Следующий пример больше чем пример, - он послужит нам образцом проектирования классов в самых различных ситуациях.

Рассмотрим структуру, введенную в предыдущей лекции для изучения наследования и содержащую классы графических фигур: `FIGURE`, `OPEN FIGURE`, `POLYGON`, `RECTANGLE`, `ELLIPSE` и т.д. До сих пор в этой структуре использовалось лишь единичное наследование.

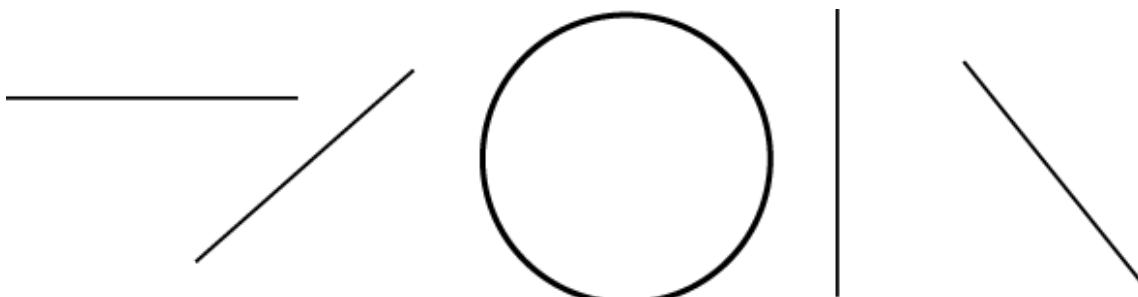


Рис. 15.8. Элементарные фигуры

Пусть в этой иерархии представлены все нужные нам базовые фигуры. Однако в библиотеку классов хотелось бы включить и не базовые фигуры, имеющие широкое распространение. Конечно, любое изображение каждый раз можно строить из примитивов, но это неудобно. Поэтому мы создадим библиотеку фигур, часть которых будут базовыми, а часть - построена на их основе. Так, из экземпляров базисных классов: отрезка и окружности можно собрать колесо:

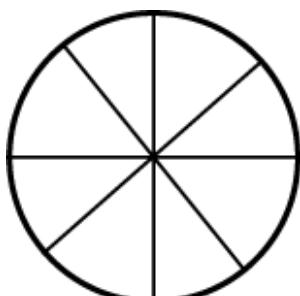


Рис. 15.9. Составная фигура

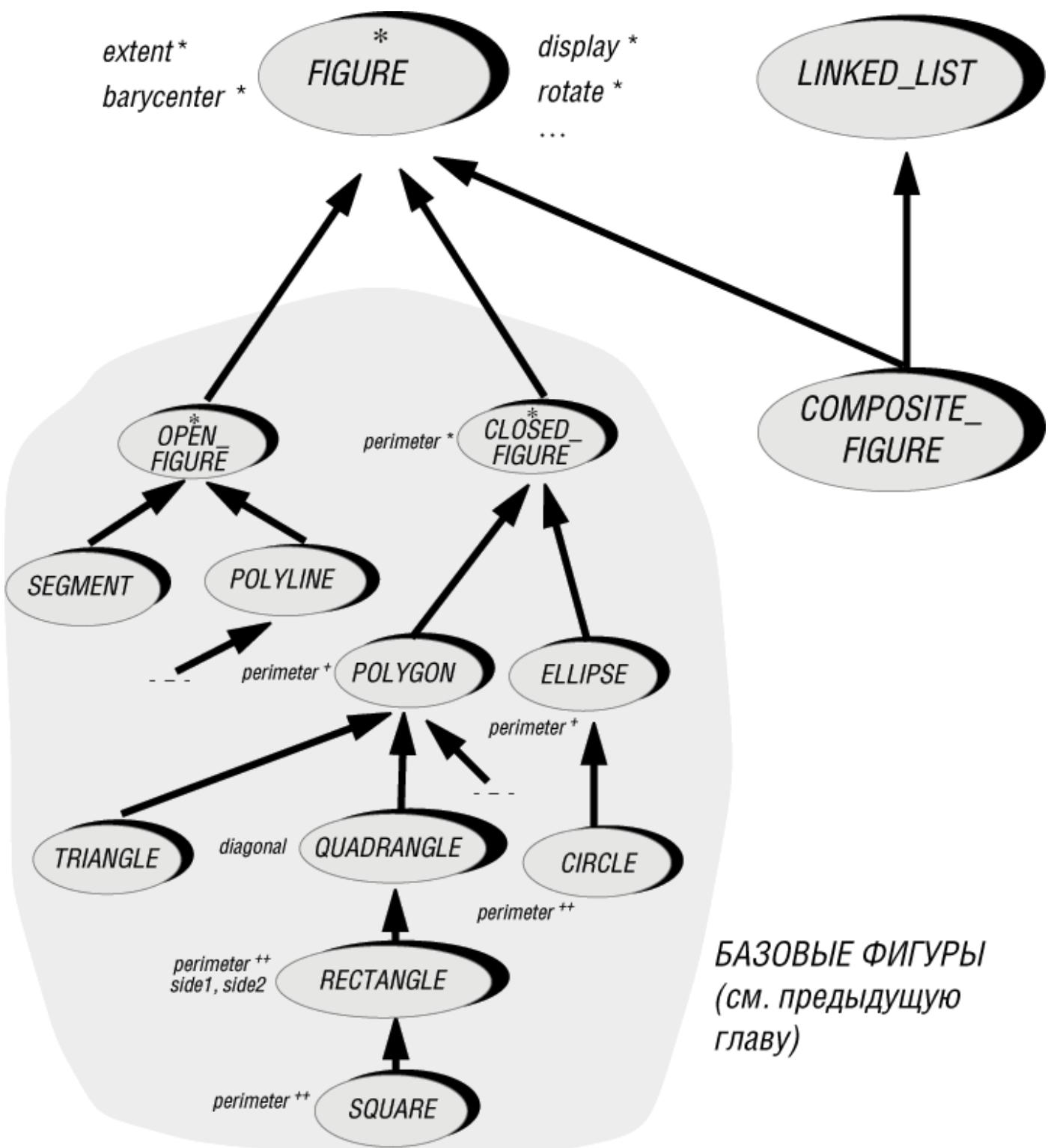
Колесо, в свою очередь, может пригодиться при рисовании велосипеда, и т. д.

Итак, нам необходим универсальный механизм создания новых фигур, построенных на основе существующих, но, будучи построенными, используемыми наравне с базовыми.

Назовем новые фигуры составными (COMPOSITE FIGURE). Каждую такую фигуру, безусловно, надо порождать от FIGURE, что позволит ей быть "на равных" с базовыми примитивами. Составная фигура - это еще и список фигур, ее образующих, каждая из которых может быть базовой или составной. Воспользуемся множественным наследованием (рис. 15.10).

Для получения эффективного класса COMPOSITE FIGURE выберем одну из возможных реализаций списка, например связный список - LINKED_LIST. Объявление класса будет выглядеть так:

```
class COMPOSITE FIGURE inherit
    FIGURE
    LINKED_LIST [FIGURE]
feature
    ...
end
```



БАЗОВЫЕ ФИГУРЫ
(см. предыдущую главу)

Рис. 15.10. Составная фигура - это фигура и список фигур одновременно

Предложение **feature** записывать приятно вдвойне. Работа с составными фигурами во многом сводится к работе со всеми их составляющими. Например, процедура **display** может быть реализована так:

```

display is
-- Отображает фигуру, последовательно отображая все ее компоненты.
do
  from
  start
  until
  loop
  item.display
  forth
end
end
  
```

Как и в предыдущих рассмотрениях, мы предполагаем, что класс список предлагает механизм обхода элементов, основанный на понятии курсора. Команда `start` устанавливает курсор на первый элемент, если он есть (иначе `after` сразу же равно `True`), `after` указывает, обошел ли курсор все элементы, `item` дает значение элемента, на который указывает курсор, `forth` передвигает курсор к следующему элементу.

Я нахожу эту схему прекрасной и, надеюсь, вы тоже пленитесь ее красотой. В ней вы найдете почти весь арсенал средств: классы, множественное наследование, полиморфные структуры данных (`LINKED_LIST [FIGURE]`), динамическое связывание (вызов `item.display` применяет метод `display` **того класса, которому принадлежит текущий элемент списка**), рекурсию (каждый элемент `item` сам может быть составной фигурой без ограничения глубины вложенности). Подумать только: есть люди, которые могут прожить всю жизнь и не увидеть этого великолепия!

Но можно пойти еще дальше. Обратимся к другим компонентам `COMPOSITE FIGURE` - методам вращения (`rotate`) и переноса (`translate`). Они также должны выполнять надлежащие операции над каждым элементом фигуры, и каждый из них может во многом напоминать `display`. Для ОО-проектировщика это может стать причиной тревоги: хотелось бы избежать повторения; потому выполним преобразование - от инкапсуляции к повторному использованию. (Это могло бы стать девизом.) Техника, рассматриваемая здесь, состоит в использовании отложенного класса "итератор", чьи экземпляры способны выполнять цикл по `COMPOSITE FIGURE`. Его эффективным потомком может стать `DISPLAY_ITERATOR`, а также ряд других классов. Реализацию этой схемы мы оставляем читателю (см. упражнение 15.4).

Описание составных структур с применением множественного наследования и списка или иного контейнерного класса, как одного из родителей, - это универсальный **образец проектирования**. Примерами его воплощения являются **подменю** (см. упражнение 15.8), а также **составные команды** в ряде интерактивных систем.

Брак по расчету

В приведенных примерах оба родителя играли симметричные роли, но это не всегда так. Иногда вклад каждого из них различен по своей природе.

Важным приложением множественного наследования является обеспечение реализации абстракции, описанной отложенным классом, используя свойства, обеспечиваемые эффективным классом. Один класс абстрактен, второй - эффективен.

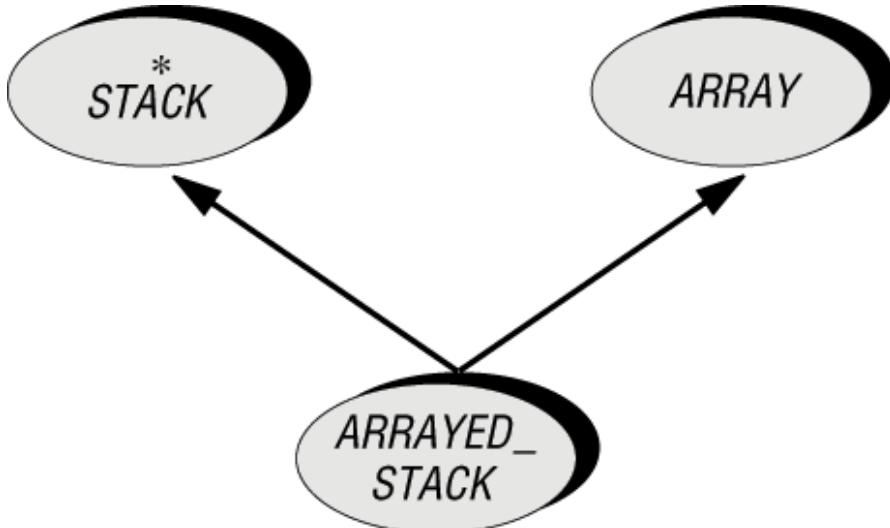


Рис. 15.11. Брак по расчету

Рассмотрим реализацию стека, заданную массивом. У нас уже есть классы для поддержки стеков и массивов в отдельности (абстрактный `STACK` и эффективный `ARRAY`, см. предыдущие лекции). Лучший способ реализации класса `ARRAYED_STACK` (стек, заданный массивом) - описать его как наследника классов `STACK` и `ARRAY`. Это концептуально верно: стек-массив одновременно является стеком (с точки зрения клиента) и массивом (с позиций поставщика). Вот описание класса:

```

indexing
    description: "Стек, реализованный массивом"
class ARRAYED_STACK [G] inherit
    STACK [G]
    ARRAY [G]
        ... Здесь будут добавлены предложения переименования ...
feature
    ... Реализация отложенных подпрограмм класса STACK
    в терминах операций класса ARRAY (см. ниже)...
end

```

ARRAYED_STACK предлагает ту же функциональность, что и STACK, делая эффективными отложенные компоненты: full, put, count ..., реализуя их как операции над массивом.

Вот схема некоторых типичных компонентов: full, count и put. Так, условие, при котором стек полон, имеет вид:

```
full: BOOLEAN is
      -- Является ли стек (его представление) заполненным?
      do
          Result := (count = capacity)
      end
```

Компонент capacity унаследован от класса ARRAY и задает емкость стека, равную числу элементов массива. Для count потребуется ввести атрибут:

```
count: INTEGER
```

Это пример эффективной реализации отложенного компонента как атрибута. Наконец,

```
put (x: G) is
      -- Втолкнуть x на вершину.
      require
          not full
      do
          count := count + 1
          array_put (x, count)
      end
```

Процедура array_put унаследована от класса ARRAY. Ее цель - записать новое значение в указанный элемент массива.

Компоненты capacity и array_put имели в классе ARRAY имена count и put. Смену прежних имен мы поясним позднее.

Класс ARRAYED_STACK типичен как вариант наследования, образно именуемый "**брак по расчету**". Оба класса, - абстрактный и эффективный, - дополняя друг друга, создают достойную пару.

Помимо эффективной реализации методов, отложенных (deferred) в классе STACK, класс ARRAYED_STACK способен переопределять реализованные. Компонент change_top, реализованный в STACK в виде последовательности вызовов remove и put, можно переписать более эффективно:

```
array_put (x, count)
```

Указание на переопределение компонента следует ввести в предложение наследования:

```
class ARRAYED_STACK [G] inherit
    STACK [G]
    redefine change_top end
    ... Остальное, как прежде ...
```

Инвариант этого класса может иметь вид

```
invariant
    non_negative_count: count >= 0
    bounded: count <= capacity
```

Первое утверждение выражает свойство АТД. Фактически оно присутствует в родительском классе STACK и потому является избыточным. Здесь оно приводится в педагогических целях. Из окончательной версии класса его нужно изъять. Второе утверждение включает емкость массива - capacity. Это - **инвариант реализации**.

Сравнив ARRAYED_STACK с представленным ранее классом STACK2, вы увидите, как сильно он упростился благодаря наследованию. Это сравнение мы продолжим при обсуждении методологии наследования, в ходе которого ответим на критику, звучащую иногда в адрес наследования "по расчету" и так называемого **наследования реализаций**.

Структурное наследование

Множественное наследование просто необходимо, когда необходимо задать для класса ряд дополнительных свойств, помимо свойств, заданных базовой абстракцией.

Рассмотрим механизм создания объектов с постоянной структурой (способных сохраняться на долговременных носителях). Поскольку объект является "сохраняемым", то у него должны быть свойства, позволяющие его чтение и запись. В библиотеке Kernel за эти свойства отвечает класс STORABLE, который может быть родителем любого класса. Очевидно, такой класс, помимо STORABLE, должен иметь и других родителей, а значит, схема не сможет работать, не будь множественного наследования. Примером может служить изученное выше наследование с родителями COMPARABLE и NUMERIC. Форма наследования при которой родитель задает общее структурное свойство, и, чаще всего, имеет имя, заканчивающееся на -ABLE, называется схемой наследования **структурного** вида.

Без множественного наследования нет способа указать, что некоторая абстракция обладает двумя структурными свойствами - числовыми и сохранения, сравнения и хеширования. Выбор только **одного** из родителей подобен выбору между отцом и матерью.

Наследование функциональных возможностей

Вот еще одна типичная ситуация. Многие программные инструменты должны сохранять "историю", что позволяет пользователям:

- просмотреть список последних команд;
- вторично выполнить последнюю команду;
- выполнить новую команду, отредактировав для этого предыдущую;
- аннулировать действие последней команды, которая не сумела закончить свою работу.

Такой механизм привлекателен для любой интерактивной среды, однако его создание требует больших усилий. Поэтому историю поддерживают лишь немногие инструменты (к примеру, ряд "командных оболочек" Unix и Windows), да и те нередко частично. Универсальные же решения не зависят от конкретного инструмента. Их можно инкапсулировать в класс, а от него - породить другой класс для управления рабочей сессией любого инструмента. (Решение с применением классов-клиентов допустимо, но не так привлекательно.) И снова без множественного наследования не обойтись, так как недостаточно иметь родителя, знающего только историю.

Набор полезных возможностей предоставляет класс TEST, инкапсулирующий ряд механизмов тестирования класса: прием и хранение данных от пользователя, вывод и хранение результата, сравнение, регрессное тестирование и т.д. Хотя решение с использованием вложения может быть предпочтительным, неплохо иметь возможность при тестировании класса X определять класс X_TEST, порожденный от X и TEST.

Далее мы будем встречать и другие примеры наследования **функциональных возможностей**, при котором один класс F инкапсулирует набор, например констант или методов математической библиотеки, а другой, объявляя себя потомком F, может ими воспользоваться.

Лунка и кнопка

Вот пример, в котором, как и раньше, без множественного наследования не обойтись. Идейно он близок к примеру с корпоративным самолетом, спальным вагоном и другими типами, полученными в результате объединения абстракций. Впрочем, теперь мы будем работать с понятиями из практики программирования.

Среда разработки ISE, описанная в [лекции 19](#) курса "Основы объектно-ориентированного проектирования", подобно другим графическим приложениям, содержит "кнопки" для выполнения определенных действий. В среду встроен механизм "выбрать и перетащить" (pick and throw), аналог традиционного механизма баксировки drag-and-drop. С его помощью можно выбрать объект на экране; при этом курсор мыши превращается в "камешек", форма которого указывает тип выбранного объекта. Камешек можно перетащить и опустить в **лунку**, форма которой соответствует камешку, инициируя тем самым определенное действие. Например, инструментарий Class Tool, позволяющий исследовать свойства класса, имеет "классную лунку", опустив в которую камешек нового класса, вы перенастроите инструмент на показ его свойств.

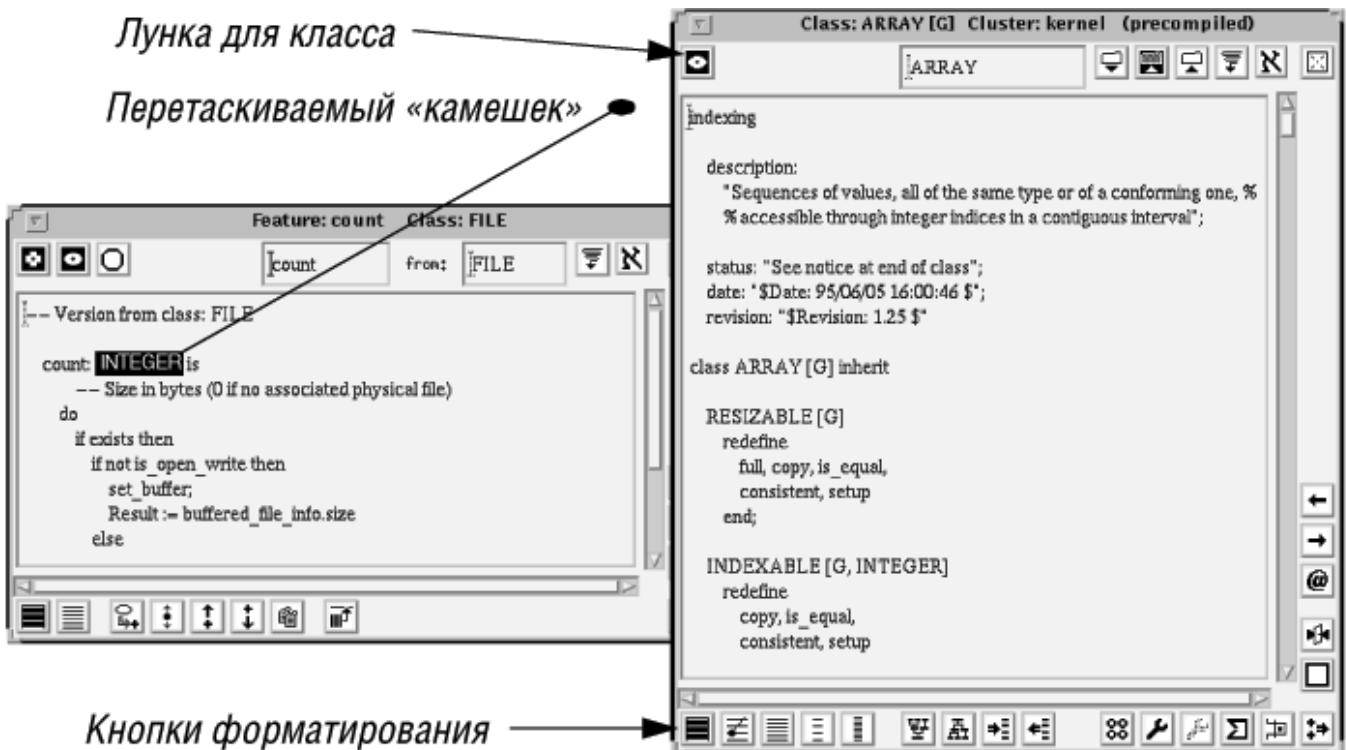


Рис. 15.12. Pick and throw (Выбрать и перетащить)

Обратите внимание на нижнюю строку с кнопками форматирования. Нажатие каждой из них позволяет получить разнообразную информацию о классе ARRAY, **например** краткую форму класса. Как показано на рисунке, пользователь, работая в окне Feature Tool, выбрал щелчком правой кнопки класс INTEGER. **Он передвигает** его в направлении "лунки" класса в окне Class Tool, настроенного сейчас на ARRAY. Перетаскивание завершается щелчком правой кнопки на "лунке" класса, форма которой соответствует форме камешка. Тем самым Class Tool будет перенастроен на работу с выбранным классом INTEGER.

Иногда удобнее, чтобы "лунка" была одновременно и кнопкой, что позволяет не только "загонять" в нее объект, но независимо от этого щелкать по ней левой кнопкой. Таковой является наша "лунка" класса, точка внутри которой указывает на присутствие в ней объекта (сначала ARRAY, а затем INTEGER). Щелчок по ней левой кнопкой перенастроит инструмент на работу с текущим объектом, что полезно, когда дисплей отражает другую информацию. Такая лунка с кнопкой реализуется специальным классом BUTTONHOLE.

Нетрудно догадаться, что класс BUTTONHOLE возникает в результате наследования от классов BUTTON и HOLE. Новый класс сочетает в себе компоненты и свойства обоих родителей, реагирует как кнопка, и допускает операции как над лункой.

Оценка

Приведенные примеры наглядно проиллюстрировали мощь и силу механизма множественного наследования. Необходимость его применения подтверждена опытом построения универсальных библиотек [М 1994а].

Как объединить две абстракции, если множественное наследование недоступно? Видимо, вы должны выбрать одну из них как "официальный" родительский класс, а все компоненты второй просто скопировать, превратив новый класс в ее "нелегального" потомка. В результате на нелегальной части класса теряется полиморфизм, все преимущества повторного использования и многое другое, что неприемлемо.

Переименование компонентов

Иногда при множественном наследовании возникает проблема конфликта имен (name clash). Ее решение - переименование компонентов (feature renaming) - не только снимает саму проблему, но и способствует лучшему пониманию природы классов.

Конфликт имен

Каждый класс обладает доступом ко всем компонентам своих родителей. Он может использовать их, не указывая тот класс, в котором они были описаны. После обработки **inherit** в классе **class C inherit A ...** метод f класса C становится известен как f. То же справедливо и для клиентов: при объявлении сущности x типа C вызов компонента записывается как x. f без каких-либо ссылок на A. Все метафоры "хромают", иначе можно было бы говорить, что наследование - форма усыновления: С усыновляет все компоненты A.

Усыновление не меняет присвоенных имен, и набор имен компонентов данного класса содержит наборы имен компонентов каждого его родителя.

А если родители класса разные компоненты назвали одним именем? Возникает противоречие, поскольку согласно установленному ранее правилу запрещена перегрузка имен: в классе имя компонента обозначает только один компонент. Это правило не должно нарушаться при наличии родителей класса. Рассмотрим пример:

```
class SANTA_BARBARA inherit
    LONDON
    NEW_YORK
feature
    ...
end-- class SANTA_BARBARA
```

Что предпринять, если LONDON и NEW_YORK имеют в своем составе компонент с именем, например, foo (нечто)?

Ни при каких обстоятельствах нельзя нарушить запрет перегрузки имен компонентов. Как следствие, класс SANTA_BARBARA окажется некорректным, что обнаружится при трансляции.

Вспомним класс TREE, порожденный от классов CELL и LIST, каждый из которых имеет компонент с именем item. Кроме того, оба класса имеют метод, названный put. Выбор каждого имени не случаен, и мы не хотим менять их в исходных классах лишь потому, что кому-то пришла идея объединить эти классы в дерево.

Что делать? Исходный код классов LONDON и NEW_YORK может быть недоступен; или на его исправления может быть наложен запрет; а при отсутствии такого запрета, возможно, вам не захочется ничего менять, поскольку LONDON написан не вами, и выход новой версии класса заставит все начинать с нуля. Наконец, самое главное, принцип Открыт-Закрыт не разрешает исправлять модули при их повторном использовании.

Всегда ошибочно обвинять в грехах своих родителей. Проблема конфликта имен возникла в самом классе. В нем должно найтись и решение.

Класс, наследующий от разных родителей разные компоненты с идентичным именем, не будет корректен, пока мы не включим в его декларацию наследования одно или несколько предложений переименования rename. Каждое из них назначает новое локальное имя одному или нескольким унаследованным компонентам. Например:

```
class SANTA_BARBARA inherit
    LONDON
        rename foo as fog end
    NEW_YORK
feature
    ...
end
```

Как внутри SANTA_BARBARA, так и во всех клиентах этого класса компонент LONDON с именем foo будет называться fog, а одноименный компонент NEW_YORK - просто foo. Клиенты LONDON, как и прежде, будут знать этот компонент под именем foo.

Этого достаточно для устранения конфликта (если других совпадений нет, а класс LONDON и класс NEW_YORK не содержат компонента с именем fog). В противном случае можно переименовать компонент класса NEW_YORK:

```
class SANTA_BARBARA inherit
    LONDON
        rename foo as fog end
    NEW_YORK
        rename foo as zoo end
feature
    ...
end
```

Предложение rename следует за указанием имени родителя и предшествует любым выражениям redefine, если такие имеются. Можно переименовать и несколько компонентов, как в случае:

```
class TREE [G] inherit
    CELL [G]
        rename item as node_item, put as put_right end
```

где устраняется конфликт между одноименными компонентами CELL и LIST. Компоненту CELL с именем item дается идентификатор node_item, аналогично и put переименовывается в put_right.

Результат переименования

Убедимся, что нам понятен результат этого действия. Пусть класс SANTA_BARBARA имеет вид (оба унаследованных компонента foo в нем переименованы):

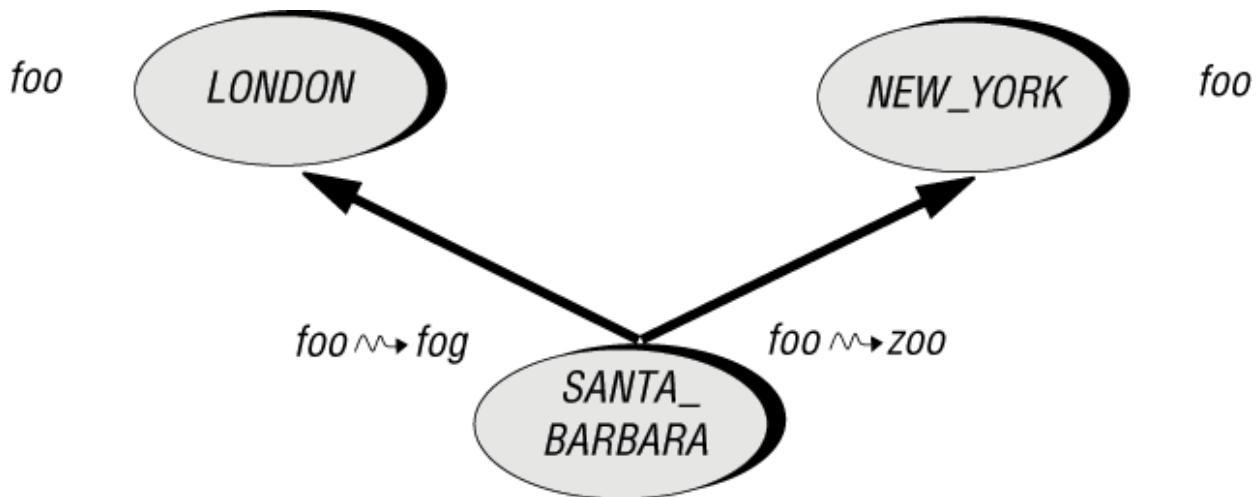


Рис. 15.13. Устранение конфликта имен

(Обратите внимание на графическое обозначение операции смены имен.) Пусть также имеются сущности трех видов:

l: LONDON; n: NEW_YORK; s: SANTA_BARBARA

Вызовы l.foo и s.fog будут являться корректными. После полиморфного присваивания l := s все останется корректным, поскольку имена обозначают один и тот же компонент. Аналогично, корректны вызовы n.foo, s.zoo, которые после n := s также будут давать одинаковый результат.

В то же время, следующие вызовы некорректны:

- l.zoo, l.fog, n.zoo, n.fog, так как ни LONDON, ни NEW_YORK не содержат компонентов с именем fog или zoo;
- s.foo, поскольку после смены имен класс SANTA_BARBARA уже не имеет компонента с именем foo.

При всей искусственности имен пример хорошо иллюстрирует природу конфликта имен. Хотите верьте, хотите нет, но приходилось слышать, что конфликт порождает "глубокую семантическую проблему". Это неправда. Конфликт имен - простая синтаксическая проблема. Если бы автор первого класса сменил имя компонента на fog, или автор второго - на zoo, конфликта бы не было, и в каждом случае - это всего лишь замена буквы. Конфликт имен - это обычная неудача, он не вскрывает никаких глубоких проблем, связанных с классами, и не свидетельствует об их неспособности работать совместно. Возвращаясь к метафоре брака, можно сказать, что конфликт имен - это не драма (обнаруженная несовместимость групп крови), а забавный факт (матери обоих супругов носят имя Татьяна, и это вызывает трудности для будущих внуков, которые можно преодолеть, договорившись, как называть обеих бабушек).

Смена имен и переопределение

В предыдущей лекции мы обсудили переопределение компонентов, полученных по наследству. (Помните, что переопределение эффективного компонента задает его новое определение, а для отложенного компонента задает его реализацию.) Сравнение переименования и переопределения компонентов поможет многое прояснить.

- Переопределение меняет компонент, но сохраняет его имя.
- Переименование меняет имя, но сохраняет компонент.

При помощи переопределения можно добиться того, чтобы **одно и то же** имя компонента ссылалось на фактически **различные** компоненты в зависимости от типа объекта, к которому оно применяется (в этом случае говорят о динамическом типе соответствующей сущности). Это - семантический механизм.

Смена имен - это синтаксический механизм, позволяющий ссылаться на **один и тот же** компонент, фигурирующий в разных классах под **разными** именами.

Иногда то и другое можно совмещать:

```

class SANTA_BARBARA inherit
    LONDON
        rename
            foo as fog
        redefine
            fog
        end
    ...
  
```

Если, как и раньше, `l: LONDON; s: SANTA_BARBARA`, и выполнено присваивание `l := s`, то оба вызова `l.foo`, `s.fog` включают переопределенную версию компонента `fog`, объявление которого должно появиться в предложении **feature** класса.

Заметьте: **redefine** содержит уже новое имя компонента. Это нормально, поскольку под этим именем компонент известен классу. Именно поэтому **rename** должно находиться выше всех остальных предложений наследования (таких, как **redefine** и пока неизвестные читателю **export**, **undefine**, **select**). После выполнения **rename** компонент теряет свой прежний идентификатор и становится известным под новым именем классу, его потомкам и его клиентам.

Подбор локальных имен

Возможность переименования наследуемого компонента небезынтересна и при отсутствии конфликта имен. Она позволяет разработчику класса подбирать подходящие имена для всех компонентов, как описанных в самом классе, так и унаследованных от предков.

Имя, под которым класс наследует компонент предка, может ничего не говорить клиентам класса. Его выбор определялся интересами клиентов предка, в то время как новый класс вписан в новый контекст и представляет иную абстракцию с собственной системой понятий. Смена имен позволяет решить возникающие проблемы, разделяя компоненты и их имена.

Хорошим примером является класс `WINDOW`, порожденный от класса `TREE`. Последний описывает иерархическую структуру, единую для всех деревьев, в том числе и для окон, но имена, понятные в исходном контексте, могут не подходить для интерфейса между `WINDOW` и его клиентами. Смена имен дает возможность привести их в соответствие с местными обычаями:

```
class WINDOW inherit
    TREE [WINDOW]
        rename
            child as subwindow, is_leaf as is_terminal, root as screen,
            arity as child_count, ...
        end
    RECTANGLE
feature
    ... Характерные компоненты window ...
end
```

Аналогично, класс `TREE`, который сам порожден от `CELL`, может сменить имя `right` на `right_sibling` и т.д. Путем смены имен класс может создать удобный набор наименований своих "служб" вне зависимости от истории их создания.

Играем в имена

Смена имен подчеркивает важность именования - как компонентов, так и классов - в практике ОО-разработки ПО. Формально, класс - это отражение имен компонентов в сами компоненты. Компоненты известны остальному миру благодаря именам.

В последней лекции будет дан ряд правил выбора имен компонентов. Заметим, что предпочтение следует отдавать общезвестным именам: `count`, `put`, `item`, `remove`, ... - выбор которых подчеркивает общность абстракций, существующую, несмотря на объективные различия классов. Придерживаясь этого стиля, вы увеличите вероятность конфликта имен при множественном наследовании, но отчасти избавитесь от переименований, имевших место в случае с классом `WINDOW`. Но каким бы правилам не отдавалось предпочтение, должна быть обеспечена гибкость в подборе имен, отвечающих потребностям каждого класса.

Использование родительской процедуры создания

Еще один пример иллюстрирует типичный случай переименования процедуры создания класса. Вспомните класс `ARRAYED_STACK`, полученный порождением от `STACK` и `ARRAY`. Процедура создания `ARRAY` размещает в памяти массив с заданными границами:

```
make (minb, maxb: INTEGER) is
    -- создать массив с границами minb и maxb
    -- (пустой если minb > maxb)
    do ... end
```

Для создания стека необходимо создать массив, позволяющий вместить заданное число элементов. Реализация основана на процедуре создания `ARRAY`:

```
class ARRAYED_STACK [G] inherit
    STACK [G]
```

```

        redefine change_top end
ARRAY [G]
    rename
    count as capacity, put as array_put, make as array_make
end
creation
    make
feature -- Initialization
    make (n: INTEGER) is
        -- Создать стек, допускающий размещение n элементов.
        require
            non_negative_size: n >= 0
        do
            array_make (1, n)
        ensure
            capacity_set: capacity = n
            empty: count = 0
        end
    ... Другие компоненты ...
invariant
    count >= 0; count <= capacity
end

```

Заметим, что выполнение соглашений об именах - выбор `make` как стандартного имени базовой процедуры создания - привело бы к конфликту, который, впрочем, не возникает благодаря переименованию, устраниющему заодно двусмысленность в отношении `count` и `put`. Оба имени встречаются в каждом классе.

Плоские структуры

Смена имен - лишь одно из средств, используемых мастером наследования для построения полноценных классов, удовлетворяющих потребностям своих клиентов. Другим таким средством является переопределение. В этой и следующей лекции мы увидим еще несколько таких механизмов: отмену определений (`undefined`), соединение (`join`), выделение (`select`), скрытие потомков (`descendant hiding`). Мощь этих комбинируемых механизмов делает наследование излишне заметным, поэтому иногда возникает необходимость в существовании версии класса, свободной от наследования, - плоской форме (`flat form`).

Плоская форма класса

Наследование - это скорее инструмент **поставщика класса**, чем клиента; это прежде всего внутренний механизм эффективного построения классов. И действительно, клиенту нужно знать о наследовании и структуре семейства классов ровно столько, чтобы он мог применять полиморфизм и динамическое связывание.

Как следствие, у нас должна быть возможность представить класс в самодостаточном виде независимо от его генеалогии. Это особенно важно, когда наследование служит для разделения различных компонентов сложной абстракции, как в случае концепции окон, частями которой являются деревья и прямоугольники.

Эту задачу решает плоская форма класса. Но вам не придется ее создавать. Ее построит один из инструментов среды разработки, который можно запустить, введя команду сценария (`flat class_name`) или щелкнув по соответствующей пиктограмме.

Плоская форма класса С - это корректная запись класса, имеющая, - с точки зрения клиента, не использующего полиморфизм, - ту же семантику, что и класс С, но лишенная всех предложений наследования. Именно так выглядел бы любой класс, если бы его создатель не мог пользоваться наследованием. Построение плоской формы предполагает:

- устранение предложения `inherit`, если оно есть;
- сохранение в неизменном виде всех определений и переопределений из С;
- введение в класс объявлений всех унаследованных компонентов, скопированных из соответствующих классов-родителей, с учетом всех указанных в `inherit` преобразований: переименования, переопределения, отмены определений, выделения (`select`), объединения компонентов;
- добавление к каждому унаследованному компоненту строки комментария вида: `from ANCESTOR`, где указано имя ближайшего предка, (пере)определенного компонент (а в случае объединения компонентов - победившая сторона);
- восстановление полной формы предусловий и постусловий унаследованных методов (по правилам наследования утверждений, изложенным в следующей лекции);
- восстановление полного инварианта класса как конъюнкции (`and`) всех родительских инвариантов с последующим преобразованием в случае применения переименованных или выделенных компонентов.

Полученный в результате класс содержит все компоненты оригинала, как введенные в самом классе, так и полученные им от предков (вторая категория компонентов от первой отличается лишь комментарием). В случае наличия меток в секциях объявления компонентов, например, `feature` - Access, подобные метки остаются. Секции с

одинаковыми метками объединяются. В каждой секции компоненты выстраиваются по алфавиту.

На рисунке показана часть плоской формы класса LINKED_TREE из библиотеки Base. Результат получен с применением Class Tool в среде разработки ISE. Для повторения результата настройте Class Tool на LINKED_TREE и щелкните по кнопке формата Flat.

The screenshot shows the 'Flat form of class LINKED_TREE [G]' window. The code is as follows:

```
class
LINKED_TREE [G]

creation
make

feature -- Access

child_cursor: CURSOR is
-- Current cursor position
-- (from LINKED_LIST)
do
!LINKED_LIST_CURSOR [G]! Result.make (child, child_after, child_before)
end;

child_index: INTEGER is
-- Index of current position
-- (from LINKED_LIST)
local
p: LINKED_LIST_CURSOR [G]
do
if child_after then
Result := arity + 1
elseif not child_before then
```

Кнопки форматирования: **flat** **flat-short** **short**

Рис. 15.14. Отображение плоской формы

Применение плоской формы

Плоская форма класса - ценный инструмент разработчика. Именно она позволяет увидеть все компоненты класса, собранные в одном месте, игнорируя то, как они были получены в играх с наследованием. При чтении текста класса трудно бывает понять, что стоит за именем каждого из его компонентов. Это один из недостатков наследования. Плоская форма класса решает эту проблему, формируя полную картину происходящего.

Кроме того, она может оказаться полезной при построении автономной версии класса, не обремененной историей порождения. Потеря полиморфизма снижает ценность такого класса.

Краткая плоская форма

Плоская форма класса дает корректное описание класса. Помимо роли, которую она играет в интересах документации, она представляет интерес для разработчиков, имеющих дело с самим классом или его потомками. Клиентам же класса нужна более абстрактная картина с меньшим числом деталей.

В одной из предыдущих лекций мы уже видели, роль краткой формы класса (кнопка **short** на рисунке обеспечивает ее построение).

Объединение двух понятий дает новое понятие краткой плоской формы (flat-short form). Как и краткая форма класса, она содержит лишь общедоступную информацию, в ней не указаны скрытые компоненты, а для экспортимемых компонентов не приводится реализация, в частности, предложения **do**. Как и плоская форма, краткая плоская форма задает все компоненты класса - и унаследованные, и описанные в нем самом.

Краткая плоская форма является основным методом документирования классов, в том числе повторно используемых классов библиотек. В этом виде информация о классе становится доступна его клиентам (и тем, кто занимается сопровождением класса). Краткая плоская форма служит для описания всех классов в библиотеке Base [М 1994а].

Дублируемое наследование

Дядюшка Жак: С кем желаете Вы говорить, сударь, с конюхом или с поваром? Ибо я у Вас и то, и другое.

Мольер, "Скупой"

Дублируемое наследование (repeated inheritance) возникает, когда класс является потомком другого класса более чем на одном пути наследования. При этом возникает потенциальная неоднозначность, которую и следует разрешить.

В явном виде такой вариант наследования возникает только в достаточно серьезных разработках. Если вас интересуют лишь ключевые составляющие объектной методологии, то можно сразу перейти к чтению следующей лекции.

Общие предки

Множественное наследование не запрещает, например, того, чтобы класс D был наследником классов B и C, каждый из которых является потомком класса A. Эту ситуацию называют дублируемым наследованием.

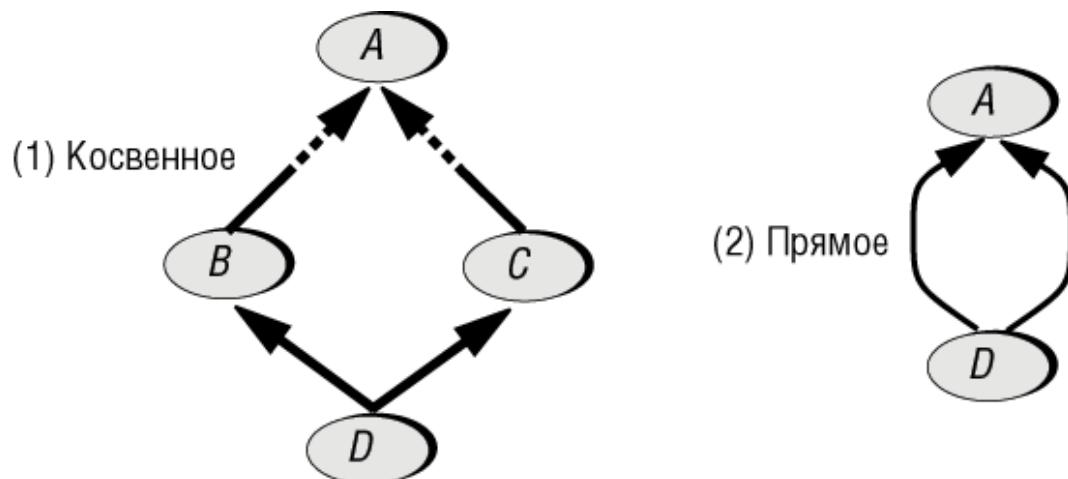


Рис. 15.15. Дублируемое наследование

Если B и C наследники потомков A, (случай 1), то такое наследование называется косвенным. Если A, B и C - это один класс (случай 2), - наследование называется прямым, что может быть записано в виде:

```
class D inherit
    A
    A
    ...
feature
    ...
end
```

По обе стороны океана

Следующий пример позволит нам промоделировать ситуацию дублируемого наследования и изучить возникающие проблемы. Пусть класс DRIVER имеет атрибуты:

```
age: INTEGER
address: STRING
violation_count: INTEGER      -- Число записанных нарушений
```

и методы:

```
pass_birthday is do age := age + 1 end
pay_fee is
    -- Оплата ежегодной лицензии.
    do ... end
```

Класс наследник, US_DRIVER учитывает налоговое законодательство США, другой, FRENCH_DRIVER, - налоговое законодательство Франции.

Рассмотрим категорию людей, которым в течение года приходится водить машину в обеих странах. Нужного класса у нас еще нет, и простым решением этой проблемы кажется множественное наследование. Опишем класс FRENCH_US_DRIVER как порожденный от US_DRIVER и FRENCH_DRIVER. Налицо дублируемое наследование.

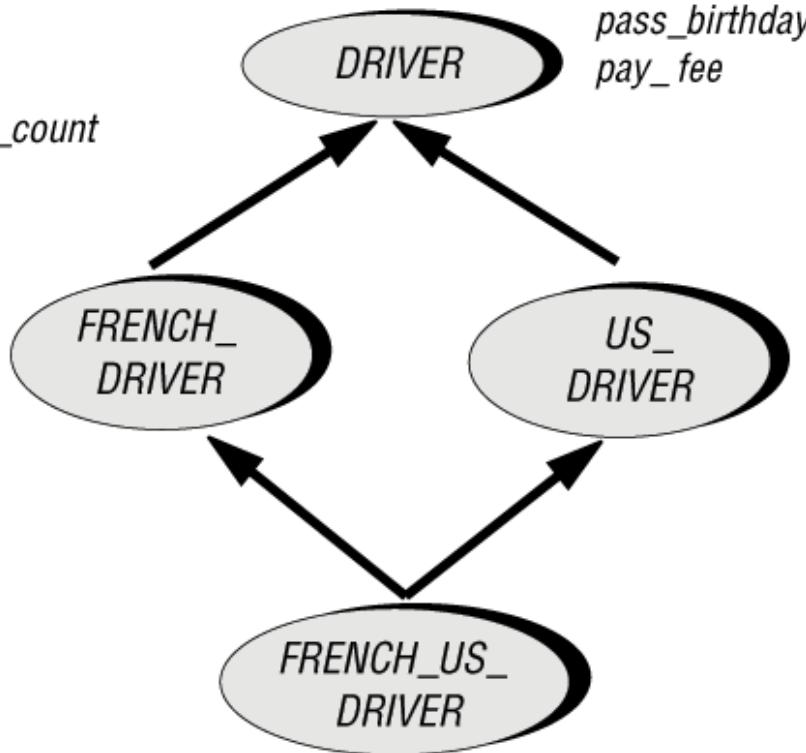


Рис. 15.16. Типы водителей

Совместное использование и репликация

Из приведенного примера вытекает основная проблема дублируемого наследования: каков смысл компонентов дублируемого потомка (**FRENCH_US_DRIVER**), полученных от дублируемого предка (**DRIVER**)?

Рассмотрим компонент *age*. Он наследуется от обоих потомков **DRIVER**, так что, на первый взгляд, возникает конфликт имен, требующий переименования. Однако такое решение было бы неадекватно проблеме, так как реального конфликта здесь нет - атрибут *age* унаследованный от **DRIVER**, задает возраст водителя, и он один и тот же для всех потомков (если только не менять свои данные в зависимости от страны пребывания). То же относится к процедуре *pass_birthday*.

Внимательно перечитайте правило о конфликте имен:

Класс, наследующий от разных родителей различные компоненты с идентичным именем, некорректен.

Компоненты *age* (также как и *pass_birthday*), наследованные классом **FRENCH_US_DRIVER** от обоих родителей, не являются "различными", поэтому реального конфликта не возникает. Заметьте, неоднозначность могла бы возникнуть лишь в случае переопределения компонента в одном из классов. Чуть позже мы покажем, как справиться с этой проблемой, а пока предположим, что переопределений не происходит.

Если компонент дублируемого предка под одним и тем же именем наследуется от двух и более родителей, он становится **одним** компонентом дублируемого потомка. Этот случай будем называть **совместным использованием** компонента (**sharing**).

Всегда ли применяется совместное использование? Нет. Рассмотрим компоненты *address*, *pay_fee*, *violation_count*. Обращаясь в службу регистрации автотранспорта в разных странах, водители скорее всего будут указывать разные адреса и по-разному платить ежегодные сборы. Впрочем, и нарушения правил тоже будут различны. Каждый из таких компонентов, следует представить в дублируемом потомке двумя разными компонентами. Данный случай будем называть **репликацией** (**replication**).

Этот, да и другие примеры, свидетельствует о том, что мы не добьемся желаемого, если все компоненты дублируемого предка будем использовать совместно или наоборот реплицировать. Поэтому необходима возможность настройки **каждого компонента** при дублируемом наследовании.

Чтобы совместно использовать один из компонентов, достаточно под одним именем унаследовать исходную версию этого компонента от обоих родителей. Но как реализовать репликацию? Делая все наоборот: породив один компонент под двумя разными именами.

Эта идея не противоречит общему правилу, согласно которому каждое имя в классе служит обозначением лишь одного компонента. Поэтому репликация компонента означает переименование при наследовании.

Правило дублируемого наследования

У дублируемого потомка версии дублируемого компонента, наследуемые под одним и тем же именем, представляют

один компонент. Версии, наследуемые под разными именами, представляют разные компоненты, являясь репликацией оригинала дублируемого предка.

Это правило, распространяясь как на атрибуты, так и на методы, дает нам мощный механизм репликации: из одного компонента класса его потомки могут получить два или более компонента. Для атрибутов оно означает введение нового поля во всех экземплярах класса, для метода - новую процедуру или функцию, изначально - с тем же алгоритмом работы.

За исключением особых случаев, включающих переопределение, репликация может носить только концептуальный характер: фактического дублирования кода не происходит, но дублируемый потомок имеет доступ к двум компонентам.

Правило придает желаемую гибкость процессу объединения классов. Вот как может выглядеть класс FRENCH_US_DRIVER:

```
class FRENCH_US_DRIVER inherit
    FRENCH_DRIVER
        rename
            address as french_address,
            violation_count as frenchViolationCount,
            pay_fee as pay_french_fee
        end
    US_DRIVER
        rename
            address as us_address,
            violation_count as usViolationCount,
            pay_fee as pay_us_fee
        end
feature
    ...
end
```

В данном случае смена имен происходит на последнем этапе - у дублируемого потомка, но полное или частичное переименование могло быть выполнено и родителями - US_DRIVER и FRENCH_DRIVER. Важно, что будет в конце, - получит ли компонент при дублируемом наследовании одно или разные имена.

Компоненты age и pass_birthday переименованы не были, а потому, как мы и хотели, они используются совместно.

Реплицируемый атрибут, скажем, address, в каждом экземпляре класса FRENCH_US_DRIVER будет представлен несколькими полями данных. Тогда при условии, что эти классы содержат только указанные нами компоненты, их экземпляры будут выглядеть как на [рис. 15.18](#).

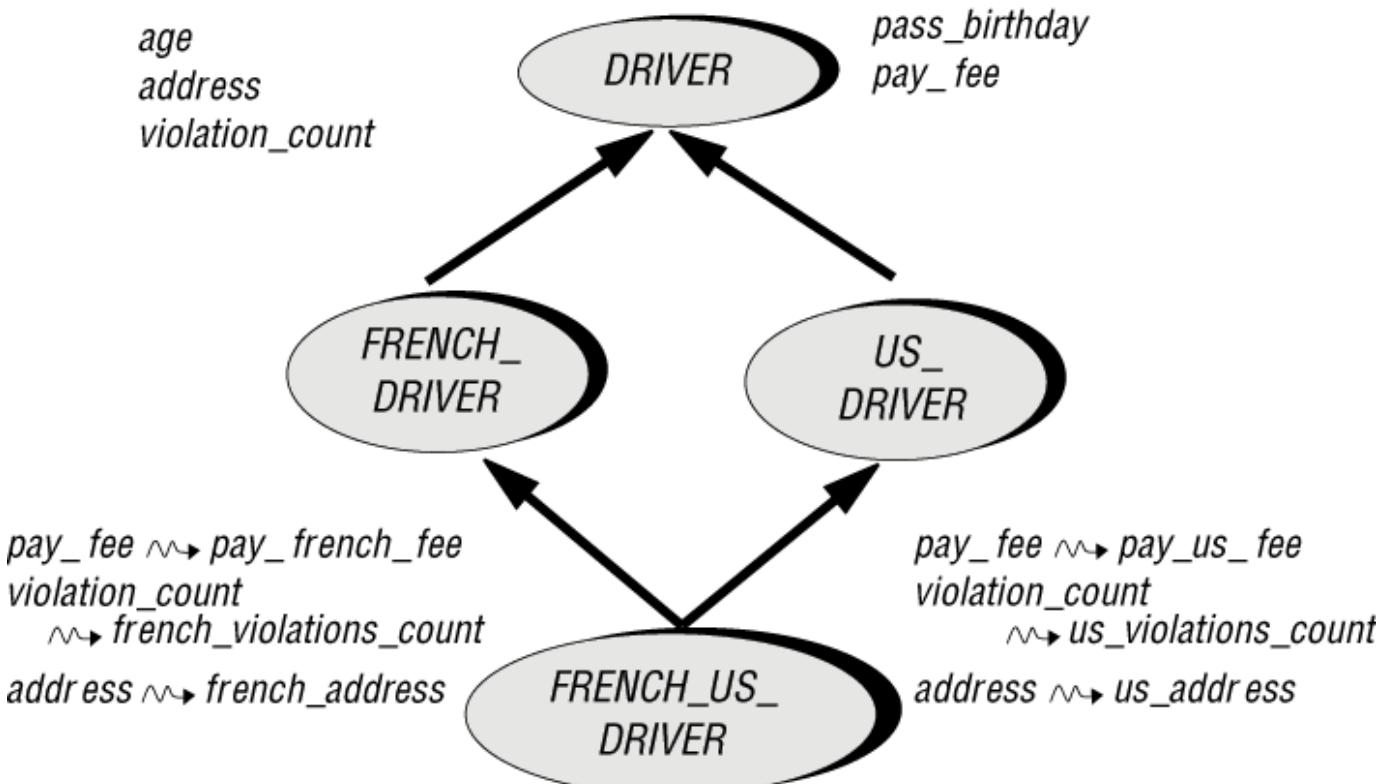


Рис. 15.17. Совместное использование и репликация

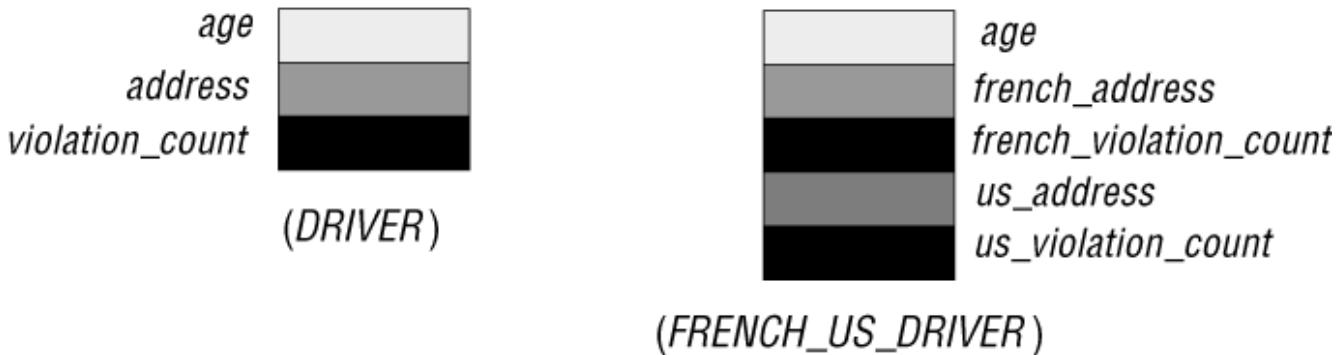


Рис. 15.18. Репликация атрибутов

(Организация FRENCH_DRIVER и US_DRIVER аналогична организации DRIVER, см. рисунок.)

Особенно важным в реализации классов является умение избегать репликации совместно используемых компонентов, например *age* из FRENCH_US_DRIVER. Не имея достаточно опыта, можно легко допустить такую ошибку и реплицировать все поля класса. Тратить память впустую недопустимо, так как по мере спуска по иерархии "мертвое" пространство будет лишь возрастать, что приведет к катастрофически неэффективному расходованию ресурсов. (Помните, что каждый атрибут во время выполнения потенциально представлен во многих экземплярах класса и его потомков.)

Механизм компиляции, описанный в конце этой книги, на деле дает гарантию того, что потеря памяти на атрибуты не будет, - концептуально совместно используемые (shared) атрибуты класса будут располагаться в общей для них (shared) физической памяти. Это - один из сложнейших компонентов реализации наследования и вызовов при динамическом связывании. Ситуация усложняется еще и тем, что подобное дублируемое наследование не должно влиять на производительность, что означает:

- нулевые затраты на поддержку универсальности;
- низкие, ограниченные константой, затраты на динамическое связывание (не зависящие от наличия в системе дублируемого наследования классов).

Поскольку существует реализация, отвечающая этим целям, то и в любой системе техника дублируемого наследования не должна требовать значительных издержек.

Дублируемое наследование в C++ следует другому образцу. Уровень, на котором принимается решение, разделять или дублировать компоненты, - это класс. Поэтому при необходимости дублирования одного компонента, приходится дублировать все. В Java эта проблема исчезает, поскольку запрещено множественное наследование.

Ненавязчивое дублирующее наследование

На практике не столь часто встречаются примеры, подобные "межконтинентальным" водителям, в которых нужны и репликация компонентов, и их совместное применение. Они не для новичков. Следует приобрести опыт, чтобы браться за них.

Иначе в попытке использовать дублирующее наследование "в лоб", можно лишь все усложнить, когда это и не нужно.

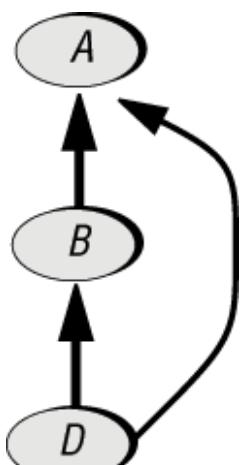


Рис. 15.19. Избыточное наследование

На рисунке показана типичная ошибка начинающих (или рассеянных разработчиков): класс D объявлен наследником B, ему нужны также свойства класса A, но B сам является потомком A. Забыв о транзитивности наследования, разработчик пишет:

```
class D ... inherit
```

В

А

В итоге возникает дублируемое наследование. Его **избыточность** очевидна. Впрочем, при надлежащем соблюдении принятых соглашений все компоненты классов (при сохранении их имен) будут использоваться совместно, новых компонентов не появится, и дополнительных издержек не будет. Даже если в В часть имен атрибутов меняется, единственным следствием этого станет лишь некоторый расход памяти.

Из этого есть только одно исключение: случай, когда В переопределяет один из компонентов А, что приведет к неоднозначности в D. Но тогда, как будет показано ниже, компилятор выдаст сообщение об ошибке, предлагая выбрать в D один из двух вариантов компонента.

Избыточное, хотя и безвредное наследование может произойти, если А - это класс, реализующий универсальные функции, например ввода-вывода, необходимые В и D. В этом случае достаточно объявить D наследником В. Это автоматически делает D потомком А, что позволяет обращаться ко всем нужным функциям. Избыточное наследование не нанесет никакого вреда, оставшись практически без последствий.

Такие случаи "безвредного" наследования могут происходить при порождении от универсальных классов ANY и GENERAL, речь о которых пойдет в следующей лекции.

Правило переименования

В этом разделе мы не введем никаких новых понятий, а лишь точнее сформулируем известные правила и приведем пример, призванный пояснить сказанное.

Начнем с запрета возникновения конфликта имен:

Определение: финальное имя

Финальным именем компонента класса является:

- Для непосредственного компонента (объявленного в самом классе) - имя, под которым оно объявлено.
- Для наследуемого компонента без переименования - финальное имя компонента (рекурсивно) в том родительском классе, от которого оно унаследовано.
- Для переименованного компонента - имя, полученное при переименовании.

Правило одного имени

Разные эффективные компоненты одного класса не могут иметь одно и то же финальное имя.

Конфликт имен происходит в том случае, когда два разных по сути компонента, оба эффективные (реализованные), имеют одно финальное имя. Такой конфликт делает класс некорректным, однако ситуацию легко исправить, добавив надлежащее предложение переименования.

Ключевым в тексте правила является слово "**разные**". Если под одним именем мы наследуем от родителей компонентов их общего предка, действует принцип совместного использования компонентов: наследуется **один компонент**, и конфликта имен не возникает.

Запрет на дублирование имен касается лишь эффективных компонентов. Если один или более компонентов с омонимичными именами являются отложенными, их можно фактически **слиять** воедино, поскольку отсутствует несовместимость реализаций. Подробнее мы поговорим об этом чуть ниже.

Приведенные правила просты и интуитивны. Чтобы в последний раз нам убедиться в их правильном понимании, построим простой пример, демонстрирующий допустимые и недопустимые варианты наследования.

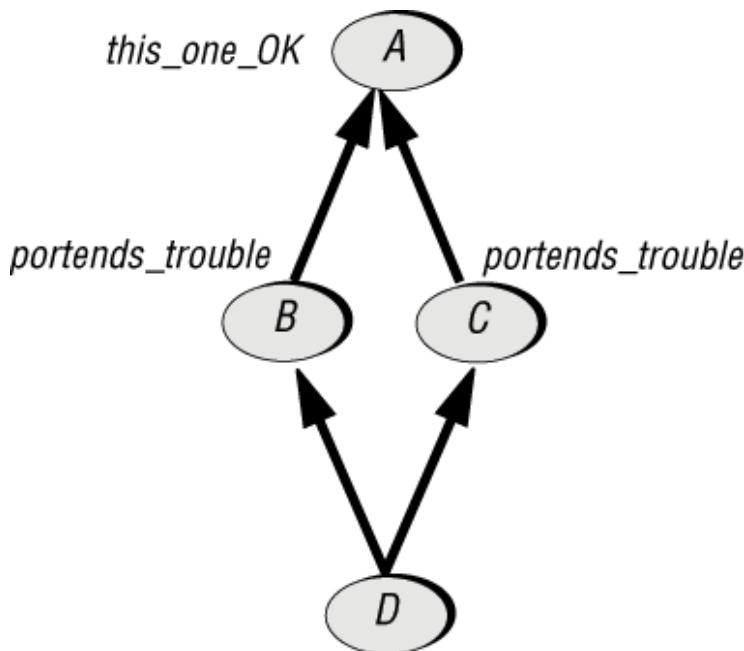


Рис. 15.20. Два варианта наследования

```

class A feature
    this_one_OK: INTEGER
end
class B inherit A feature
    portends_trouble: REAL
end
class C inherit A feature
    portends_trouble: CHARACTER
end
class D inherit
    -- Это неправильный вариант!
    B
    C
end

```

Класс D наследует `this_one_OK` дважды, один раз от B, другой раз - от C. Конфликта имен не возникает, поскольку данный компонент будет использоваться совместно. На самом деле, это - один компонент предка A.

Два компонента `portend_trouble` ("предвещающие беду") заслуженно получили такое имя. Они различны, потому их появление в D ведет к конфликтам имен, делая класс некорректным. (У них разные типы, но и одинаковые типы никак не повлияли бы на ход нашего обсуждения.)

Переименовав один из компонентов, мы с легкостью сделаем D корректным:

```

class D inherit
    -- Этот вариант класса теперь полностью корректен.
    B
    rename portends_trouble as does_not_portend_trouble_anymore end
    C
end

```

Конфликт переопределений

Пока в ходе наследования мы меняли лишь имена. А что, если промежуточный предок, такой, как B или C (см. последний рисунок), переопределит дублируемый компонент? При динамическом связывании это может привести к неоднозначности в D.

Проблему решают два простых механизма: отмена определения (`undefined`) и выделение (`selection`). Как обычно, вы сами примете участие в их разработке и убедитесь в том, что при четкой постановке задачи нужная конструкция языка становится совершенно очевидной.

Пусть дублируемый компонент переопределяется в одной из ветвей:

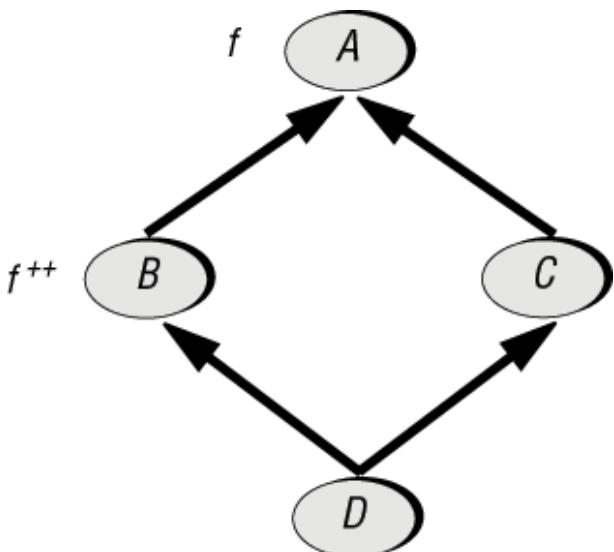


Рис. 15.21. Переопределение - причина потенциальной неоднозначности

Класс В переопределяет *f*. Поэтому в D этот компонент представлен в двух вариантах: результат переопределения в В и исходный вариант из А, полученный через класс С. (Можно предполагать, что и С переопределяет *f*, но это не внесет в наше рассуждение ничего нового.) Такое положение дел отличается от предыдущих случаев, в которых мы имели лишь один вариант компонента, возможно, наследуемый под разными именами.

Что произойдет в результате? Ответ зависит от того, под одним или разными именами класс D наследует варианты компонентов. Подразумевает ли дублируемое наследование репликацию или совместное использование? Рассмотрим эти случаи по порядку.

Конфликт при совместном использовании: отмена определения и соединение компонентов

Предположим вначале, что две версии наследуются под одним и тем же именем. Это случай совместного использования. Одному имени должен в точности соответствовать один компонент. Возможны три ситуации.

- Если одна версия отложена, а другая - эффективна, то сложностей не возникает, будет использован эффективный вариант компонента. Заметим, что этот случай явно предусмотрен правилом одного имени: речь в нем идет лишь о конфликте имен двух эффективных версий.
- Каждая версия эффективна, однако обе они переопределяются в D в предложении **redefine**. Проблемы снова не возникает, поскольку обе версии сливаются в одну, переопределяемую в тексте класса.
- Обе версии эффективны, но обе не переопределяются, тогда действительно возникает конфликт имен. Класс D будет отвергнут, как нарушающий правило одного имени.

Нередко (3) означает ошибку: создана неоднозначность имен, и ее необходимо исправить. Тривиальным решением проблемы является **переименование** одного из вариантов, но тогда мы от рассматриваемого случая совместного использования переходим к репликации, изучаемой ниже.

Есть и другая, более изощренная возможность решения конфликта (3). Она состоит в том, чтобы позволить одному из вариантов "взять верх" над другим. Дальнейшее очевидно - свести эту ситуацию к (1), сделав один из двух вариантов отложенным.

Правила переопределения дают возможность переопределить компонент *f* как отложенный, хотя для этого и потребуется ввести промежуточный класс, скажем С', - наследника С, единственная роль которого - в переопределении отложенного *f*. Затем класс D должен быть порожден не от С, а от С'. Сложно и некрасиво. Вместо этого нам нужен простой языковой механизм: **undefine**. В секции наследования класса он приводит к появлению нового предложения:

```

class D inherit
    B
    C
        undefine f end
feature
    ...
end

```

Синтаксически предложение **undefine** следует за **rename** (всякая отмена определения должна действовать на окончательный вариант имени компонента), но до **redefine** (прежде, чем что-то переопределять, мы должны позаботиться об отмене ненужных определений).

Признаком того, что предлагаемый языковой механизм желателен, почти всегда является его направленность на решение нескольких проблем (соответственно, плохой механизм создает больше проблем, чем решает). Механизм

отмены определений отвечает этому требованию: он позволяет **соединять** компоненты в условиях множественного (не обязательно - дублируемого) наследования. Пусть мы хотим свести воедино две абстракции:

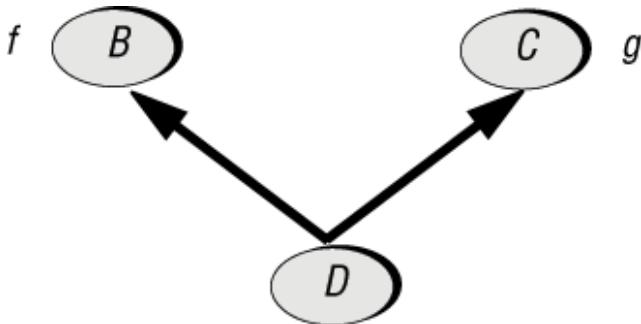


Рис. 15.22. Два родителя и слияние компонентов

Мы хотим, чтобы D трактовал f и g как один компонент. Очевидно, это возможно лишь при условии совместности семантики и сигнатур обоих компонентов (числа и типов аргументов и результата, если он есть). Допустим, что имена компонентов различны, и мы хотели бы сохранить имя f. Добиться желаемого можно, объединив переименование с отменой определения:

```

class D inherit
  B
  C
    rename
      g as f
    undefine
      f
  end
feature
  ...
end
  
```

В получил полное превосходство над C, передавая классу D как сам компонент, так и его имя. Возможны и другие сочетания: компонент можно получить от одного из родителей, имя - от другого; можно переименовать оба компонента, присвоив им новое имя в D.

Еще один, более "симметричный" вариант соединения компонентов, заключается в замене обоих унаследованных вариантов на новый компонент. Достаточно указать оба компонента в предложении **redefine**, убедившись предварительно, что оба компонента имеют одно и то же финальное имя (добавив, если надо, выражение **rename**). В результате конфликта имен не возникнет (случай (2)), а объединение двух вариантов даст новый компонент.

Конфликты при репликации: выделение

Рассмотрим теперь случай конфликтов переопределений, связанных с репликацией. Пусть при дублируемом наследовании происходит переопределение и переименование эффективного компонента, так что имеем два эффективных компонента, наделенных собственными именами.

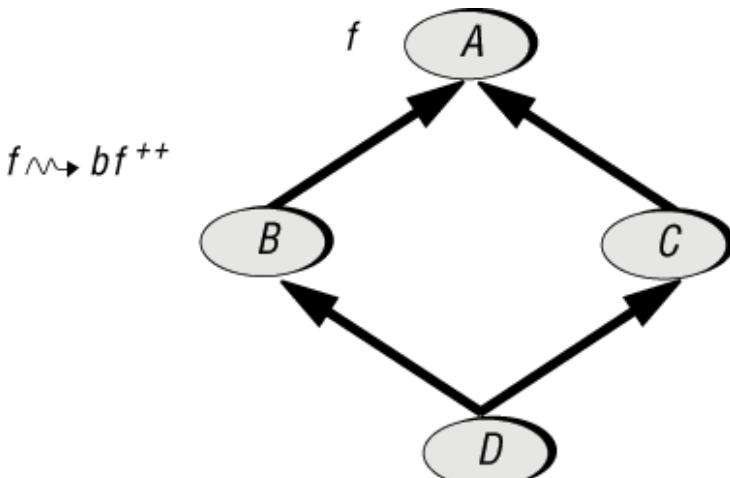


Рис. 15.23. Необходимость выделения

Представленный на рисунке класс B меняет имя f на bf и переопределяет сам компонент. При этом мы опять полагаем, что С никак не меняет f, иное предположение нисколько не повлияет на ход нашего рассуждения. Более того, результат остался бы прежним, если бы B переопределял компонент f без его переименования, которое мы могли отложить до описания D. Допустим также, что речь не идет о соединении компонентов (которое происходит при

переопределении обоих или отмене определения одного).

Поскольку компоненты наследуются под разными именами, то происходит их репликация. Класс D получает пару независимых компонентов, которые, в отличие от предыдущих случаев репликации, не являются копиями одного и того же компонента.

В отличие от случая совместного использования не возникает конфликта имен. Однако возникают другие конфликты, относящиеся к динамическому связыванию. Пусть полиморфная сущность a1 типа A (общий предок) на этапе выполнения связывается с экземпляром типа D (общим потомком). Что тогда означает вызов a1.f?

Правило динамического связывания гласит: вызываемый вариант f выбирается с учетом типа цели - объекта D. Но теперь это впервые нельзя истолковать однозначно: D содержит два равноценных варианта, известных под именами f и bf, соответствующих оригиналу f класса A.

Как и при конфликте имен, нельзя позволять компилятору делать выбор, пользуясь собственными правилами, - это противоречило бы принципам ясности и надежности. Управление ситуацией должно оставаться за автором разработки.

Для устранения неоднозначности необходим простой языковой механизм - предложение **select**. Вот версия класса, в которой предпочтение при динамическом связывании сущности f типа A отдается версии класса C:

```
class D inherit
  B
  C
    select f end
feature
  ...
end
```

В этом варианте предпочтение отдается версии класса B:

```
class D inherit
  B
    select bf end
  C
feature
  ...
end
```

Синтаксически предложение **select** следует за предложениями **rename**, **undefine** и **redefine**, если таковые имеются (выбор осуществляется после переименования и переопределения). Применение этого механизма регламентирует следующее правило:

Правило выделения

Класс, наследовавший две или более различные и эффективные версии компонента дублируемого предка и не переопределивший их, должен включить одну из них в предложение **select**.

Механизм **select** устраниет неоднозначность раз и навсегда. Потомкам класса нет необходимости (и они не должны) повторять выделение.

Выделение всех компонентов

Любой конфликт переопределений должен быть разрешен посредством **select**. Если, объединяя два класса, вы натолкнулись на ряд конфликтов, возможно, вы захотите, чтобы один из классов "одержал верх" (почти) в каждом из них. В частности, так происходит в ситуации, метафорично названной "брак по расчету" (вспомните, ARRAYED_STACK - потомок STACK и ARRAY), в которой классы-родители имеют общего предка. (В библиотеках Base оба класса действительно являются удаленными (distant) потомками общего класса CONTAINER.) В этом случае один из родителей (STACK) служит источником спецификаций, и вам, быть может, захочется, чтобы (почти) все конфликты были разрешены именно в его пользу.

Решение задачи упрощает следующая запись, дающая возможность не перечислять все конфликтующие компоненты. Предложение **inherit** класса может содержать такое описание (не более одного) родителя:

```
SOME_PARENT
  select all end
```

Результат очевиден: все конфликты переопределений, - точнее те из них, что останутся после обработки других **select**, - разрешатся в пользу SOME_PARENT. Последнее уточнение означает, что вы по-прежнему вправе отдать предпочтение другим родителям в отношении некоторых компонентов.

Сохранение исходной версии при переопределении

(Этот раздел посвящен весьма специальному вопросу, и при первом чтении книги его можно пропустить.)

Приступая к изучению наследования, мы познакомились с простой конструкцией `Precursor`, позволяющей переопределяемому компоненту вызывать его исходную версию. Механизм дублируемого наследования дает возможность обратиться к более универсальному (хотя и более "тяжеловесному") решению, пригодному в тех редких случаях, когда базовых средств не хватает.

Вернемся к известному нам классу `BUTTON` - потомку `WINDOW`, переопределяющему `display`:

```
display is
    -- Показ кнопки на экране.
do
    window_display
    special_button_actions
end
```

где `window_display` выводит кнопку как обычное окно, а `special_button_actions` добавляет элементы, специфические для кнопки, отображая, например, ее границы. Компонент `window_display` в точности совпадает с `WINDOW`-вариантом `display`.

Мы уже знаем, как написать `window_display`, используя механизм `Precursor`. Если метод `display` переопределен в нескольких родительских классах, то желаемый класс можно указать в фигурных скобках: `Precursor {WINDOW}`. Того же результата можно достичь, прибегнув к дублируемому наследованию, заставив класс `Button` быть потомком двух классов `Window`:

```
indexing
    WARNING: "Это первая попытка - данная версия некорректна!"
class BUTTON inherit
    WINDOW
        redefine display end
    WINDOW
        rename display as window_display end
feature
    ...
end
```

Одна из ветвей наследования меняет имя `display`, а потому, по правилу дублируемого наследования `BUTTON`, будет иметь два варианта компонента. Один из них переопределен, но имеет прежнее имя; второй переопределен не был, но именуется теперь `window_display`.

Этот вариант кода почти корректен, однако в нем не хватает подвыражения `select`. Если, как это обычно бывает, мы хотим выбрать переопределенную версию, то запишем:

```
indexing
note: "Это (корректная!) схема дублируемого наследования, %
      % использующая оригинальную версию переопределяемого компонента"
class BUTTON inherit
    WINDOW
        redefine
            display
        select
            display
        end
    WINDOW
        rename
            display as window_display
        export
            {NONE} window_display
        end
feature
    ...
end
```

Если такая схема должна применяться к целому ряду компонентов, их можно перечислить вместе. При этом нередко возникает необходимость разрешить все конфликты именно в пользу переопределенных компонентов. В этом случае можно воспользоваться `select all`.

Предложение **export** (см. [лекцию 16](#)) определяет статус экспорта наследуемых компонентов класса. Так, WINDOW может экспортировать компонент display, а BUTTON сделать window_display скрытым (поскольку его клиенты в нем не нуждаются). Экспорт исходной версии наследуемого компонента может сделать класс формально некорректным, если она не соответствует новому инварианту класса.

Для скрытия всех компонентов, полученных "в наследство" по одной из ветвей иерархии, служит запись **export {NONE} all**.

Такой вариант экспорта переопределенных компонентов и скрытия исходных компонентов под новыми именами весьма распространен, но отнюдь не универсален. Нередко классу наследнику необходимо скрывать или экспортировать оба варианта (если исходная версия не нарушает инвариант класса).

Насколько полезна такая техника дублируемого наследования для сохранения исходной версии компонента при переопределении? Обычно в ней нет необходимости, так как достаточно обратиться к Precursor. Поэтому этот способ следует использовать, когда старая версия нужна не только в целях переопределения, но и как один из компонентов нового класса.

Пример повышенной сложности

Вот более сложный пример применения разных аспектов дублируемого наследования.

Проблема, близкая по духу нашему примеру, возникла из интересного обсуждения в основной книге по C++ [Stroustrup 1991].

Рассмотрим класс WINDOW с процедурой display и двумя наследниками: WINDOW_WITH_BORDER и WINDOW_WITH_MENU. Эти классы описывают абстрактные окна, первое из них имеет рамку, а второе поддерживает меню. Переопределяя display, каждый класс выводит на экран стандартное окно, а затем добавляет к нему рамку (в первом случае) и меню (во втором).

Опишем окно с рамкой и с поддержкой меню. В результате мы породим класс WINDOW_WITH_BORDER_AND_MENU.

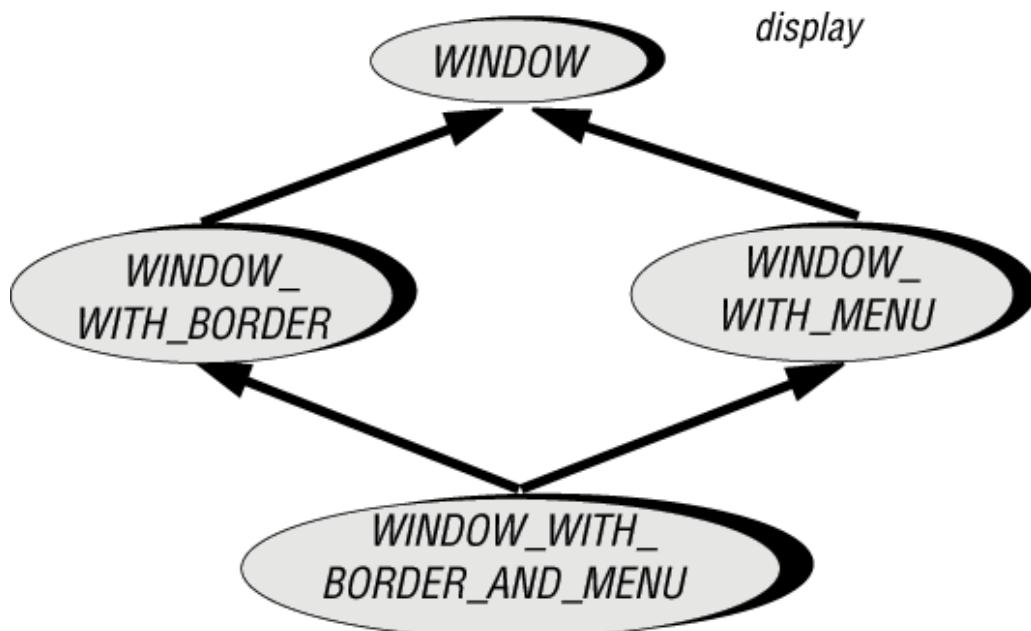


Рис. 15.24. Варианты окна

Переопределим метод display в новом классе; новая версия вначале вызывает исходную, затем строит рамку, а потом строит меню. Исходный класс WINDOW имеет вид:

```

class WINDOW feature
    display is
        do
            -- Отобразить окно (общий алгоритм)
        ...
    end
    ... Другие компоненты ...
end

```

Наследник WINDOW_WITH_BORDER осуществляет вызов родительской версии display и затем отображает рамку. В дублируемом наследовании нет необходимости, достаточно воспользоваться механизмом Precursor:

```
class WINDOW_WITH_BORDER inherit
```

```

WINDOW
    redefine display end
feature -- Output
    display is
        do
            Precursor
            draw_border
        end
feature {NONE} -- Implementation
    draw_border is do ... end
    ...
end

```

Обратите внимание на процедуру `draw_border`, рисующую рамку окна. Она скрыта от клиентов класса `WINDOW_WITH_BORDER` (экспорт классу `NONE`), поскольку для них вызов `draw_border` не имеет смысла. Класс `WINDOW_WITH_MENU` аналогичен:

```

class WINDOW_WITH_MENU inherit
    WINDOW
        redefine display end
feature -- Output
    display is
        do
            Precursor
            draw_menu
        end
feature {NONE} -- Implementation
    draw_menu is do ... end
    ...
end

```

Осталось описать общего наследника `WINDOW_WITH_BORDER_AND_MENU` этих двух классов, дублируемого потомка `WINDOW`. Предпримем первую попытку:

```

indexing
WARNING: "Первая попытка - версия не будет работать корректно!"
class WINDOW_WITH_BORDER_AND_MENU inherit
    WINDOW_WITH_BORDER
        redefine display end
    WINDOW_WITH_MENU
        redefine display end
feature
    display is
        do
            Precursor {WINDOW_WITH_BORDER}
            Precursor {WINDOW_WITH_MENU}
        end
    ...
end

```

Заметьте: при каждом обращении к `Precursor` мы вынуждены называть имя предка. Каждый предок имеет собственный компонент `display`, переопределенный под тем же именем.

Впрочем, как замечает Страуструп, это решение некорректно: версии родителей дважды вызывают исходную версию `display` класса `WINDOW`, что приведет к появлению "мусора" на экране. Для исправления ситуации добавим еще один класс, получив тройку наследников класса `WINDOW`:

```

indexing
    note: "Это корректная версия"
class WINDOW_WITH_BORDER_AND_MENU inherit
    WINDOW_WITH_BORDER
        redefine
            display
        export {NONE}
            draw_border
        end

```

```

WINDOW_WITH_MENU
    redefine
        display
    export {NONE}
        draw_menu
    end
WINDOW
    redefine display end
feature
    display is
        do
            -- Рисует окно, его рамку и меню.
            Precursor {WINDOW}
            draw_border
            draw_menu
        end
    ...
end

```

Заметьте, что компоненты `draw_border` и `draw_menu` в новом классе являются скрытыми, поскольку мы не видим причин, по которым клиенты `WINDOW_WITH_BORDER_AND_MENU` могли бы их вызывать непосредственно.

Несмотря на активное применение дублируемого наследования, класс переопределяет все унаследованные им варианты `display`, что делает выражения `select` ненужными. В этом состоит преимущество спецификатора `Precursor` в сравнении с репликацией компонентов.

Неплохим тестом на понимание дублируемого наследования станет решение этой задачи без применения `Precursor`, путем репликации компонентов промежуточных классов. При этом, разумеется, вам понадобится `select` (см. упражнение 15.10).

В полученном варианте класса присутствует лишь совместное использование, но не репликация компонентов. Расширим пример Страуструпа: пусть `WINDOW` имеет запрос `id` (возможно, целого типа), направленный на идентификацию окон. Если идентифицировать любое окно только одним "номером", то `id` будет использоваться совместно, и нам не придется ничего менять. Если же мы хотим проследить историю окна, то экземпляр `WINDOW_WITH_BORDER_AND_MENU` будет иметь три `id` - независимых "номера". Новый текст класса комбинирует совместное использование и репликацию `id` (изменения в тексте класса помечены стрелками):

```

indexing
    note: "Усложненная версия с независимыми id."
class WINDOW_WITH_BORDER_AND_MENU inherit
    WINDOW_WITH_BORDER
        rename
            id as border_id
        redefine
            display
        export {NONE}
            draw_border
        end
    WINDOW_WITH_MENU
        rename
            id as menu_id
        redefine
            display
        export {NONE}
            draw_menu
        end
    WINDOW
        rename
            id as window_id
        redefine
            display
        select
            window_id
        end
feature
    .... Остальное, как ранее...
end

```

Обратите внимание на необходимость выбора (`select`) одного из вариантов `id`.

Дублируемое наследование и универсальность

В завершение мы должны рассмотреть особый случай дублируемого наследования. Он касается компонентов, содержащих родовые параметры. Рассмотрим следующую схему (подобная ситуация может возникнуть не только при прямом, но и при косвенном дублируемом наследовании):

```
class A [G] feature
    f: G;...
end
class B inherit
    A [INTEGER]
    A [REAL]
end
```

В классе B по правилу дублируемого наследования компонент f должен использоваться совместно. Но из-за универсализации возникает неоднозначность, - какой результат должен возвращать компонент - real или integer? Та же проблема возникнет, если f имеет параметр типа G.

Подобная неоднозначность недопустима. Отсюда правило:

Универсальность в правиле дублируемого наследования

Тип компонента, совместно используемого в правиле дублируемого наследования, а также тип любого из его аргументов не может быть родовым параметром класса, от которого произошло дублируемое наследование компонента.

Для устранения неоднозначности можно выполнить переименование в точке наследования.

Правила об именах

(В этом разделе мы только формализуем сказанное выше, поэтому при первом чтении книги его можно пропустить.)

Мы уже видели, что в случае возможной неоднозначности конфликты имен пресекаются, хотя некоторые ситуации бывают вполне корректны. Чтобы в представлении множественного и дублируемого наследования не оставить никакой неоднозначности, полезно обобщить ограничения на конфликт имен в едином правиле: Заканчивая этот раздел, сведем изложенный ранее материал в единое правило:

Конфликты имен: определение и правило

В классе, образованном в результате множественного наследования, возникает **конфликт имен**, если два компонента, наследованные от разных родителей, имеют одно и то же финальное имя.

Конфликт имен делает класс некорректным за исключением следующих случаев:

1. Оба компонента унаследованы от общего предка, и ни один из них не получен повторным объявлением версии предка.
2. Оба компонента имеют совместимые сигнатуры, и, по крайней мере, один из них наследуется в отложенной форме.
3. Оба компонента имеют совместимые сигнатуры и переопределются в новом классе.

Ситуация (1) описывает совместное использование при дублируемом наследовании.

Для случая (2) "наследование в отложенной форме" возможно по двум причинам: либо отложенная форма задана родительским классом, либо компонент был эффективным, но порожденный класс отменил его реализацию (**undefine**).

Ситуации (2) и (3) рассматриваются отдельно, однако, их можно представить как один вариант - вариант **соединения (join)**. Переходя к n компонентам (n >= 2), можно сказать, что ситуации (2) и (3) возникают, когда от разных родителей класс принимает n одноименных компонентов с совместимыми сигнатурами. Конфликт имен не делает класс некорректным, если эти компоненты могут быть соединены, иными словами:

- все n компонентов отложены, так что некому вызвать конфликт определений;
- существует единственный эффективный компонент. Его реализация станет реализацией остальных компонентов;
- два или несколько компонентов эффективны. Класс должен их переопределить. Новая реализация будет использоваться как для переопределяемых компонентов, так и для любых отложенных компонентов, участвующих в конфликте.

И, наконец, точное правило употребления конструкции **Precursor**. Если в переопределении используется **Precursor**, то неоднозначность может возникнуть из-за того, что неясно, версию какого родителя следует вызывать. Чтобы решить эту проблему, следует использовать вызов вида **Precursor {PARENT} (...)**, где PARENT - имя желаемого родителя. В остальных случаях указывать имя родителя не обязательно.

Обсуждение

Давайте проанализируем следствия некоторых решений, принятых в этой лекции.

Переименование

Любой язык, поддерживающий множественное наследование, должен как-то решать проблему конфликта имен. Коль скоро мы не можем и не должны требовать от разработчиков внесения изменений в исходные классы, есть всего два решения, помимо тех, что были описаны выше:

- требовать от клиентов устранения всех неоднозначностей;
- выбирать некую интерпретацию по умолчанию.

В соответствии с первым подходом, класс C, наследующий компонент f от A и B, будет нормально откомпилирован, возможно, с выдачей предупреждения. Ничего страшного не произойдет, пока в тексте клиента C не обнаружится нечто подобное:

```
x: C  
... x.f ...
```

Клиенту придется квалифицировать ссылку на f, используя нотацию, например, такую: x.f | A, либо x.f | B, чтобы указать подразумеваемый класс.

Это решение противоречит, однако, одному из принципов, важность которого мы подчеркивали в этой лекции: структура наследования класса касается лишь самого класса и его предков, но не клиентов, за исключением случаев полиморфного применения компонентов. Пользуясь f из C, я не должен знать о том, введена эта функция классом C либо получена им от A или B.

Согласно второй стратегии, запись x.f корректна. Выбор одного из вариантов делается средствами языка. Критерием выбора является, например, порядок, в котором C перечисляет своих родителей. Для обращения к другим вариантам может существовать особая форма записи.

Данный подход реализован в нескольких производных от Lisp языках с поддержкой множественного наследования. Тем не менее, выбор семантики по умолчанию весьма опасен ввиду потенциальной несовместимости со статической типизацией.

Эти проблемы решает смена имен. Одним из ее преимуществ является возможность создания клиентского интерфейса с "понятными" именами компонентов.

ОО-разработка и перегрузка

Анализ роли имен, сделанный в этой лекции, позволяет вернуться к вопросу о внутриклассовой перегрузке (in-class name overloading).

Напомню, что в таких языках, как Ada 83 и Ada 95, перегрузка разрешена - можно давать одно имя разным компонентам одного синтаксического модуля. Например, в одном пакете возможны определения:

```
infix "+" (a, b: VECTOR) is...  
infix "+" (a, b: MATRIX) is...
```

Языки Java и C++ позволяют делать то же самое в пределах класса.

Ранее мы называли эту возможность **синтаксической** перегрузкой. Это - статический механизм. Для однозначного разрешения вызова, например, x + y, достаточно посмотреть на тип аргументов x и y, который очевиден из текста программы.

В объектной технологии применяется и более мощный механизм **семантической** (или **динамической**) перегрузки. Так, если классы VECTOR и MATRIX наследуют от общего предка NUMERIC компонент

```
infix "+" (a: T) is...
```

и каждый из них переопределяет его нужным образом, то понять, о какой операции + идет речь в выражении x + y, можно только динамически во время выполнения программы. Семантическая перегрузка - действительно интересный механизм, позволяющий использовать единое имя в тексте различных классов для представления разных вариантов по сути **одной и той же операции**, такой, как сложение в NUMERIC. Правила для утверждений, рассматриваемые в следующей лекции, уточнят эту ситуацию, требуя, чтобы переопределения компонента сохраняли его фундаментальную семантику.

Сохраняется ли роль синтаксической перегрузки в объектной технологии? Трудно найти разумные аргументы в ее

поддержку. Можно понять, почему язык Ada 83, не имеющий классов, ее использовал. Но в ОО-языке выбор одного имени для обозначения **разных операций** - это прямой путь к созданию беспорядка.

Проблема состоит еще и в том, что синтаксическая форма перегрузки вступает в конфликт с семантической, в активе которой - полиморфизм и динамическое связывание. Рассмотрим вызов `x . f (a)`. Если он следует за полиморфными операторами присваивания `x := y` и `a := b`, то при сохранении имен его результат будет в точности тем же, что и для `y . f (b)`, даже если типы `b` и `y` отличны от типов `a` и `x`. Но при перегрузке это свойство не сохраняется! Теперь `f` может быть перегруженным именем двух разных компонентов: одного - типа `a`, другого - типа `b`. Чему отдать предпочтение: синтаксической перегрузке или динамическому связыванию? Хуже того, базовый класс типа `y` может переопределять один или оба перегруженных компонента. И таким комбинациям, как и причинам ошибок, нет числа.

То, что мы наблюдаем, является нежелательным результатом взаимодействия двух отдельных языковых черт. Предусмотрительный разработчик, предлагая новый язык и "поиграв" с некой новой возможностью, быстро откажется от нее, встретив несовместимость с более важными компонентами языка.

Таковы риски синтаксической перегрузки, а каковы все же ее плюсы? Ответить на этот вопрос нелегко. Простой принцип доступности кода гласит, что в тексте одного модуля читатель должен быть совершенно уверен в соответствии имени и значения. При внутриклассовой перегрузке это свойство теряется.

Типичный пример, иногда приводимый в подтверждение полезности перегрузки, связан с компонентами класса `STRING`. Чтобы к одной строке, при отсутствии перегрузки, добавить другую строку или отдельный символ, используются разные имена компонентов: `s1.add_string (s2)` и `s1.add_character ('A')`, или в инфиксной записи `s := s1++ s2` и `s := s1 + 'A'`. При перегрузке обе операции можно назвать одинаково. Так ли это необходимо? Объекты типов `CHARACTER` и `STRING` наделены совершенно разными свойствами. Добавление символа всегда увеличивает длину строки на 1. Сцепление строк может оставить длину неизменной (если вторая строка пуста) или увеличить ее произвольным образом. Применение разных имен кажется не только разумным, но и желательным, особенно потому, что приведенные выше примеры ошибок действительно вполне возможны.

Предположим, даже, что решено использовать перегрузку, но и в этом случае придется подумать о более точном критерии, позволяющем выбирать нужный компонент. Общепринятый критерий синтаксической перегрузки различает компоненты по их сигнатуре, что не исключает неоднозначности. Типичный пример - процедуры создания точек в полярной или декартовой системе координат: `make_cartesian` и `make_polar`. Сигнатуры обеих процедур одинаковы, - они имеют два аргумента типа `REAL`, однако, работают совершенно по-разному. Перегрузку здесь использовать нельзя. Для отражения этого факта, что оба компонента и в самом деле различны, им следует дать разные имена.

Реализацию процедур создания ("конструкторов") в Java и C++ нельзя описывать без иронии. Так, вы **не вправе** давать конструкторам разные имена, а вынуждены полагаться на перегрузку. Пытаясь решить эту проблему, я не нашел ничего лучше, чем ввести искусственный третий параметр.

В итоге (внутриклассовая) синтаксическая перегрузка в ОО-среде создает немало проблем, не давая видимых преимуществ. (Тем же, кто использует Java, C++ или Ada 95, можно посоветовать полностью отказаться от перегрузки, прибегая к ней лишь при создании конструкторов, то есть тогда, когда язык не оставляет другого выбора.) Стараясь умело применять объектный подход, придерживайтесь простого правила: каждый компонент имеет имя, каждое имя означает только один компонент.

Ключевые концепции

- Подход к конструированию ПО, подобный конструированию из кубиков, требует возможности объединения нескольких абстракций в одну. Это достигается благодаря множественному наследованию.
- В самых простых и наиболее общих случаях множественного наследования два родителя представляют независимые абстракции.
- Множественное наследование часто необходимо как для моделирования систем, так и для повседневной разработки ПО, в частности, создания повторно используемых библиотек.
- Конфликты имен при множественном наследовании должны устраняться переименованием.
- Переименование позволяет ввести в классе контекстно-адаптированную терминологию.
- Компоненты следует отделять от их имен. Один и тот же компонент в разных классах может быть известен под разными именами. Класс определяет отображение имен в компоненты.
- Дублируемое наследование - мощная техника - возникает как результат множественного наследования, при котором один класс становится потомком другого несколькими способами.
- При дублируемом наследовании компонент общего предка становится одним компонентом, если он наследуется под одним именем, и несколькими независимыми компонентами в противном случае.
- Конкурирующие версии общего предка при динамическом связывании должна устраняться предложением `select`.
- Механизм репликации при дублируемом наследовании не должен дублировать компоненты, включающие родовые параметры.
- В ОО-среде семантическая перегрузка, поддерживаемая динамическим связыванием, более полезна, чем синтаксическая перегрузка.

Библиографические замечания

Механизм переименования, а также правила дублируемого наследования были разработаны при написании этой книги. Механизм отмены определений предложен Михаэлем Швайцером (Michael Schweitzer), механизм выбора Джоном Поттером (John Potter).

Пример с выпадающим меню взят из книги [M 1988c].

Упражнения

У15.1 Окна как деревья

Класс WINDOW порожден от TREE [WINDOW]. Поясните суть родового параметра. Покажите, какое новое утверждение появится в связи с этим в инварианте класса.

У15.2 Является ли окно строкой?

Окно содержит ассоциированный с ним текст, представленный атрибутом text типа STRING. Стоит ли отказаться от атрибута и объявить WINDOW наследником класса STRING?

У15.3 Завершение строительства

Завершите проектирование класса WINDOW, показав точно, что необходимо от лежащего в основе механизма управления выводом?

У15.4 Итераторы фигур

При обсуждении COMPOSITE FIGURE мы говорили о применении итераторов для выполнения операций над составными фигурами. Разработайте соответствующие классы итераторов. (**Подсказка:** в [M 1994a] приведены классы библиотеки итераторов, которые послужат основой вашей работы.)

У15.5 Связанные стеки

Основываясь на классах STACK и LINKED_LIST, постройте класс LINKED_STACK, описывающий реализацию стека как связного списка.

У15.6 Кольцевые списки и цепи

Объясните, почему LIST нельзя использовать для создания кольцевых списков. (**Подсказка:** в этом вам может помочь изучение формальных утверждений, обсуждение которых вы найдете в начале следующей лекции.) Опишите класс CHAIN, который может служить родителем как для LIST, так и для нового класса кольцевых списков CIRCULAR. Обновите класс LIST и, если нужно, его потомков. Дополните структуру класса, обеспечивающую разные варианты реализации кольцевых списков.

У15.7 Деревья

Согласно одной из интерпретаций, дерево - это рекурсивная структура, представляющая собой список деревьев. Замените приведенное в этой лекции описание класса TREE как наследника LINKED_LIST и LINKABLE новым вариантом

```
class TREE [G] inherit
    LIST [TREE [G]]
feature ...end
```

Расширьте это описание до полнофункционального класса. Сравните это расширение с тем, что было описано в тексте данной лекции.

У15.8 Каскадные или "шагающие" (walking) меню

Оконные системы вводят понятие меню, реализуемое классом MENU с запросом, возвращающим список элементов, и командами отображения, перехода к следующему элементу и т.д. Меню составлено из элементов, поэтому нам понадобится класс MENU_ENTRY с такими запросами, как parent_menu и operation (операция, выполняемая при выборе элемента) и такими командами, как execute (выполняет операцию operation).

Среди меню нередко встречаются каскадные, или шагающие меню (walking menu), где выбор элемента приводит к появлению подменю (submenu). На рисунке приведено шагающее меню среди Open Windows, созданной корпорацией Sun:

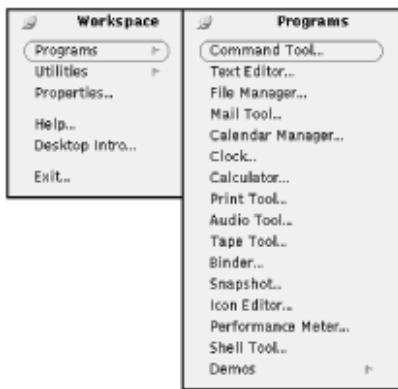


Рис. 15.25. Выпадающее меню

Предложите описание класса SUBMENU. (**Подсказка:** подменю одновременно является меню и элементом меню, чья операция должна отображать подменю.) Можно ли это понятие с легкостью описать в языке без множественного наследования?

У15.9 Плоский precursor (предшественник)

Что должна показывать плоская форма класса при встрече с инструкцией, использующей Precursor?

У15.10 Дублируемое наследование и репликация

Напишите класс WINDOW_WITH_BORDER_AND_MENU без обращения к Precursor. Для доступа к родительскому варианту переопределенного компонента используйте репликацию при дублируемом наследовании. Убедитесь в том, что вы используете правильные предложения **select** и назначаете каждому компоненту правильный статус экспорта.

Основы объектно-ориентированного программирования

16. Лекция: Техника наследования

Наследование - ключевая составляющая ОО-подхода к повторному использованию и расширяемости. В этой лекции нам предстоит исследовать новые возможности, разнородные, но демонстрирующие замечательные следствия красоты базисных идей. Связь наследования с утверждениями и Проектированием по Контракту. Глобальная структура наследования, где все классы согласованы. Замороженные компоненты, для которых не применим принцип Открыт-Закрыт. Ограниченнная универсальность: как задавать требования на родовые параметры. Попытка присваивания: как безопасно приводить к типу. Как и когда изменять свойства типа при повторных объявлениях. Закрепленные объявления, помогающие избежать лавины переобъявлений. Непростые отношения между наследованием и скрытием информации. Вопросам наследования будут посвящены еще две лекции: обзор проблем типизации представлен в [лекции 17](#), а подробное обсуждение методологии наследования - в [лекции 6](#) курса "Основы объектно-ориентированного проектирования". Большинство разделов этой лекции строится по единому принципу: экзаменуются следствия идей предыдущих двух лекций, обнаруживаются проблемы, они подробно анализируются, предлагается обоснованное решение. Ключевым является шаг анализа - как только проблема становится ясной, зачастую решение ее находится сразу же.

Наследование и утверждения

Обладая изрядной мощью, наследование может быть и опасным. Не будь механизма утверждений, создатели классов могли бы весьма "вероломно" пользоваться повторными объявлениями и динамическим связыванием для изменения семантики операций без возможности контроля со стороны клиента. Утверждения способны на большее: они дают нам более глубокое понимание природы наследования. Не будет преувеличением сказать, что лишь понимание принципов Проектирования по Контракту позволяет в полной мере постичь сущность концепции наследования.

Вкратце мы уже очертили основные правила, управляющие взаимосвязью наследования и утверждений: все утверждения (предусловие и постусловия подпрограмм, инварианты классов), заданные в классах-родителях, остаются в силе и для их потомков. В этом разделе мы уточним эти правила и используем полученные результаты, чтобы дать новый взгляд на наследование как на субподряды (subcontracts).

Инварианты

С правилом об инвариантах класса мы встречались и прежде:

Правило родительских инвариантов

Инварианты всех родителей применимы и к самому классу.

Инварианты родителей добавляются к классу. Инварианты соединяются логической операцией **and then**. (Если у класса нет явного инварианта, то инвариант True играет эту роль.) По индукции в классе действуют инварианты всех его предков, как прямых, так и косвенных.

Как следствие, выписывать инварианты родителей в инварианте потомка еще раз не нужно (хотя семантически такая избыточность не вредит: **a and then a** есть то же самое, что **a**).

Полностью восстановленный инвариант класса можно найти в плоской и краткой плоской форме последнего (см. [лекцию 15](#)).

Предусловия и постусловия при наличии динамического связывания

В случае с предусловиями и постусловиями ситуация чуть сложнее. Общая идея, как отмечалось, состоит в том, что любое повторное объявление должно удовлетворять утверждениям оригинальной подпрограммы. Это особенно важно, если подпрограмма отложена: без такого ограничения на будущую реализацию, задание предусловие и постусловий для отложенных подпрограмм было бы бесполезным или, хуже того, привело бы к нежелательному результату. Те же требования к предусловию и постусловию остаются и при переопределении эффективных подпрограмм.

Анализируя механизмы повторного объявления, полиморфизма и динамического связывания, можно дать точную формулировку искомого правила. Но для начала представим типичный случай.

Рассмотрим класс и его подпрограммы, имеющие как предусловие, так и постусловие:

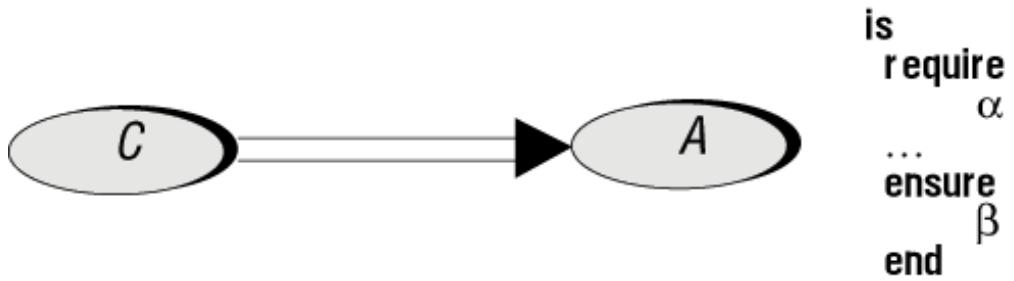


Рис. 16.1. Подпрограмма, клиент и контракт

На [рис. 16.1](#) показан клиент С класса А. Чтобы быть клиентом, класс С, как правило, включает в одну из своих подпрограмм объявление и вызов вида:

```

a1: A
...
a1.r

```

Для простоты мы проигнорируем все аргументы, которые может требовать *r*, и положим, что *r* является процедурой, хотя наши рассуждения в равной мере применимы и к функциям.

Вызов будет корректен лишь тогда, когда он удовлетворяет предусловию. Гарантировать, что С соблюдает свою часть контракта, можно, к примеру, предварив вызов проверкой предусловия, написав вместо *a1.r* конструкцию:

```

if a1.α then
    a1.r
        check a1.β end      -- постусловие должно выполняться
        ... Инструкции, которые могут предполагать истинность a1.. ...
end

```

(Как отмечалось при обсуждении утверждений, не всегда требуется проверка: достаточно, с помощью **if** или без него, гарантировать выполнение условия *a* перед вызовом *r*. Для простоты будем использовать **if**-форму, игнорируя предложение **else**.)

Обеспечив соблюдение предусловия, клиент С рассчитывает на выполнение постусловия *a1.β* при возврате из *r*.

Все это является основой Проектирования по Контракту: в момент вызова подпрограммы клиент **должен** обеспечить соблюдение предусловия, а в ответ при возврате из подпрограммы он полагается на выполнение постусловия.

Что происходит, когда вводится наследование?

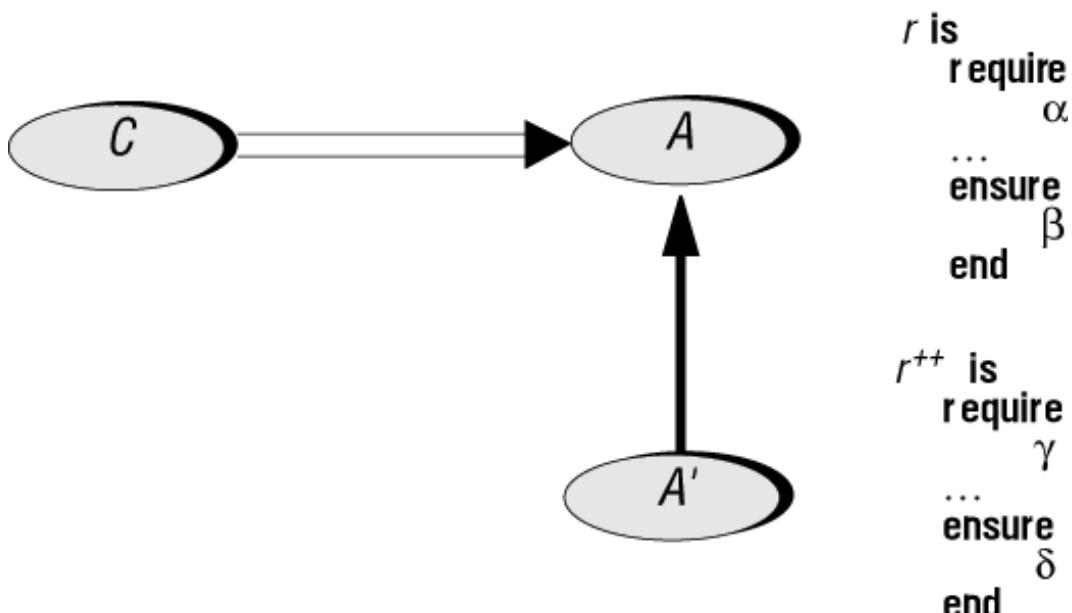


Рис. 16.2. Подпрограмма, клиент, контракт и потомок

Пусть новый класс A' порожден от A и содержит повторное объявление r. Как он может, если вообще может, заменить прежнее предусловие α новым γ , а прежнее постусловие β - новым δ ?

Чтобы найти ответ, рассмотрим обязательства клиента. В вызове a1.r цель a1 может - в силу полиморфизма - иметь тип A'. Однако С об этом не знает! Единственным объявлением a1 остается исходная строка

a1: A

где упоминается A, но не A'. На деле С может использовать A', даже если его автор не знает о наличии такого класса. Вызов подпрограммы r может произойти, например, в процедуре С вида:

```
some_routine_of_C (a1: A) is
  do
    ...; a1.r;...
  end
```

Тогда при вызове some_routine_of_C из другого класса в нем может использоваться фактический параметр типа A', даже если в тексте клиента С класс A' нигде не упоминается. Динамическое связывание как раз и означает тот факт, что обращение к r приведет в этом случае к использованию переопределенной версии A'.

Итак, может сложиться ситуация, в которой С, являясь только клиентом A, фактически во время выполнения использует версии компонентов класса A'. (Можно сказать, что С - "динамический клиент" A', хотя в тексте С об этом и не говорится.)

Что это значит для С? Только одно - проблемы, которые возникнут, если не предпринять никаких действий. Клиент С может добросовестно выполнять свою часть контракта, и все же в результате он будет обманут. Например,

```
if a1. $\alpha$  then a1.r end
```

если a1 полиморфно присоединена к объекту типа A', инструкция вызовет подпрограмму, ожидающую выполнения γ и гарантирующую выполнение δ , в то время как клиент получил указание соблюдать α и ожидать выполнения β . Налицо возможное расхождение во взглядах клиента и поставщика на контракт.

Как обмануть клиентов

Чтобы понять, как удовлетворить клиентов, мы должны сыграть роль адвокатов дьявола и на секунду представить себе, как их обмануть. Так поступает опытный криминалист, разгадывая преступление. Как мог бы поступить поставщик, желающий ввести в заблуждение своего честного клиента С, гарантирующего при вызове α и ожидающего выполнения β ? Есть два пути:

- **Потребовать больше**, чем предписано предусловием α . Формулируя более сильное предусловие, мы позволяем себе исключить случаи, которые, согласно исходной спецификации, были совершенно приемлемы.
- **Гарантировать меньше**, чем это следует из начального постусловия β . Более слабое постусловие позволяет нам дать в результате меньше, чем было обещано исходной спецификацией.

Вспомните, что мы неоднократно говорили при обсуждении Проектирования по Контракту: усиление предусловия облегчает задачу поставщика ("клиент чаще не прав"), иллюстрацией чего служит крайний случай - предусловие **false** (когда "клиент всегда не прав").

Как уже было сказано, утверждение A называется более сильным, чем B, если A логически влечет B, но отличается от него: например, $x \geq 5$ сильнее, чем $x \geq 0$. Если утверждение A сильнее утверждения B, говорят еще, что утверждение B слабее утверждения A.

Как быть честным

Теперь нам понятно, как обманывать. Но как же быть честным? Объявляя подпрограмму повторно, мы можем сохранить ее исходные утверждения, но также мы вправе:

- заменить предусловие более **слабым**;
- заменить постусловие более **сильным**.

Первый подход символизирует щедрость и великодушие: мы допускаем большее число случаев, чем изначально. Это не причинит вред клиенту, который на момент вызова удовлетворяет исходному предусловию. Второй подход означает, что мы выдаем больше, чем от нас требовалось. Это не причинит вред клиенту, полагающемуся на выполнение по завершении вызова исходных постусловий.

Итак, основное правило:

Правило (1) Утверждения Переобъявления (Assertion Redeclaration)

При повторном объявлении подпрограммы предусловие может заменяться лишь равным ему или более слабым, постусловие - лишь равным ему или более сильным.

Это правило отражает тот факт, что новый вариант подпрограммы не должен отвергать вызовы, допустимые в оригинале, и должен, как минимум, представлять гарантии, эквивалентные гарантиям исходного варианта. Он вправе, хоть и не обязан, допускать большее число вызовов или давать более сильные гарантии.

Как явствует из названия, это правило применимо к обеим формам повторного объявления: переопределению и реализации отложенного компонента. Второй случай важен особо, - утверждения будут связаны со всеми эффективными версиями потомков.

Утверждения подпрограммы, как отложенной, так и эффективной, задают ее семантику, применимую к ней самой и ко всем повторным объявлениям ее потомков. Точнее говоря, они специфицируют **область допустимого поведения** подпрограммы и ее возможных версий. Любое повторное объявление может лишь сужать эту область, не нарушая ее.

Как следствие, создатель класса должен быть осторожным при написании утверждений эффективной подпрограммы, не привнося **излишнюю спецификацию (overspecification)**. Утверждения должны описывать намерения подпрограммы, - ее абстрактную семантику, - но не свойства реализации. Иначе можно закрыть возможность создания иной реализации подпрограммы у будущих потомков.

Пример

Предположим, я написал класс MATRIX, реализующий операции линейной алгебры. Среди прочих возможностей я предлагаю своим клиентам подпрограмму расчета обратной матрицы. Фактически это сочетание команды и двух запросов: процедура `invert` инвертирует матрицу, присваивает атрибуту `inverse` значение обратной и устанавливает логический атрибут `inverse_valid`. Значение атрибута `inverse` имеет смысл тогда и только тогда, когда `inverse_valid` является истинным; в противном случае матрицу инвертировать не удалось, так как она вырождена. В ходе нашего обсуждения случай вырожденной матрицы мы можем проигнорировать.

Конечно же, я могу найти лишь приближенное значение обратной матрицы и готов гарантировать определенную точность расчетов, однако, не владея численными подпрограммами в совершенстве, буду принимать лишь запросы с точностью не выше 10^{-6} . В итоге, моя подпрограмма будет выглядеть приблизительно так:

```
invert (epsilon: REAL) is
    -- Обращение текущей матрицы с точностью epsilon
    require
        epsilon >= 10 ^ (-6)
    do
        "Вычисление обратной матрицы"
    ensure
        ((Current * inverse) | - | One) <= epsilon
    end
```

Постусловие предполагает, что класс содержит инфиксную функцию `infix "| - |"` такую, что $m1 | - | m2$ есть $|m1 - m2|$ (норма разности матриц $m1$ и $m2$), а также функцию `infix "*"`, результатом которой является произведение двух матриц. `One` - единичная матрица.

Как человек негордый, летом я приглашу программиста, и он перепишет мою подпрограмму `invert`, используя более удачный алгоритм, лучше аппроксимирующий результат и допускающий меньшее значение `epsilon` (как повторное объявление, эта запись синтаксически некорректна):

```
require
    epsilon >= 10 ^ (-20)
...
ensure
    ((Current * inverse) | - | One) <= (epsilon / 2)
```

Автор новой версии достаточно умен, чтобы не переписывать MATRIX в целом. Изменения коснутся лишь нескольких подпрограмм. Они будут включены в состав порожденного от MATRIX класса NEW_MATRIX.

Если повторное объявление содержит новые утверждения, они должны иметь иной синтаксис, нежели приведенный выше. Правило появится чуть позднее.

Изменения, внесенные в утверждения, удовлетворяют правилу повторного объявления: новое предусловие $\text{epsilon} \geq 10^{-20}$ слабее исходного $\text{epsilon} \geq 10^{-6}$, новое же постусловие сильнее сформулированного вначале.

Вот как все должно происходить. Клиент исходного класса MATRIX запрашивает расчет обратной матрицы именно у него, но на деле - ввиду динамического связывания - вызывает реализацию класса NEW_MATRIX. Тот же клиент может иметь в своем составе подпрограмму

```
some_client_routine (m1: MATRIX; precision: REAL) is
    do
        ... ; m1.invert (precision); ...
        -- Возможен вызов версии как MATRIX, так и NEW_MATRIX
    end
```

которой один из его собственных клиентов передает первый параметр типа NEW_MATRIX.

NEW_MATRIX должен воспринимать и корректно обрабатывать любой вызов, который принимается его предком. Используя более слабое предусловие и более сильное постусловие, мы корректно обработаем все обращения клиентов MATRIX и предложим своим клиентам решение, лучше прежнего.

При усилении предусловия invert, например, $\text{epsilon} \geq 10^{-5}$, вызов, корректный для класса MATRIX, мог стать теперь некорректным. При ослаблении постусловия возвращаемый результат стал бы хуже, чем гарантированный для MATRIX.

Устранение посредника

Последний комментарий указывает на весьма интересное следствие правила Утверждений Переобъявления. В общей схеме

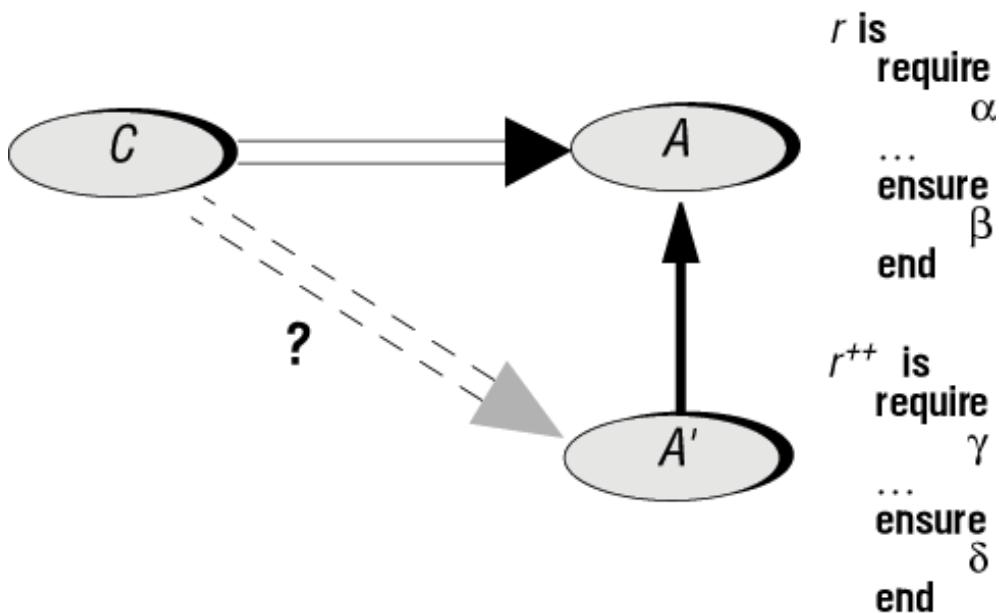


Рис. 16.3. Подпрограмма, клиент и подрядчик

утверждения γ и δ , введенные при повторном объявлении, предпочтительнее для клиентов, если они отличаются от α и β (предусловия - более слабые, постусловия - более сильные). Но клиент класса A, использующий A', благодаря полиморфизму и динамическому связыванию, не может в полной мере воспользоваться более выгодным контрактом, ибо единственный контракт клиента заключен с классом A.

Воспользоваться преимуществом нового контракта можно лишь став непосредственным клиентом A' (пунктирная связь с вопросительным знаком на [рисунке 16.3](#)), как в случае:

```

a1: A'
...
if a1.y then a1.r end
    check a1.δ end      -- постусловие выполняется

```

При этом вы, естественно, объявляете a1 как объект типа A', а не объект типа A, как прежде. В результате теряется универсальность полиморфизма, идущая от A.

Компромисс ясен. Клиент класса MATRIX должен обеспечивать выполнение исходного (более сильного) предусловия, а в ответ вправе ожидать выполнения исходного (более слабого) постусловия. Даже если его запрос динамически подготовлен к обслуживанию классом NEW_MATRIX, воспользоваться новыми возможностями - большей толерантностью входа и большей точностью выхода - ему никак не удастся. Для обращения к улучшенной спецификации клиент должен объявить матрицу типа NEW_MATRIX, тем самым, потеряв доступ к иным порожденным от MATRIX реализациям, не являющимся производными классами самого NEW_MATRIX.

Субподряды

Правило Утверждения Переобъявления великолепно сочетается с теорией Проектирования по Контракту.

Мы видели, что утверждения подпрограммы описывают связанный с ней контракт, в котором клиент гарантирует выполнение предусловия, получая право рассчитывать на истинность постусловия; для поставщика все наоборот.

Наследование совместно с повторным объявлением и динамическим связыванием приводит к созданию субподрядов. Приняв условия контракта, вы не обязаны выполнять его сами. Подчас вы знаете кого-то еще, способного сделать это лучше и с меньшими издержками. Так происходят, когда клиент запрашивает подпрограмму из MATRIX, но благодаря динамическому связыванию может на этапе выполнения фактически вызывать версию, переопределенную в потомке. "Меньшие издержки" означают здесь более эффективную реализацию, как в знакомом нам примере с периметром прямоугольника, а "лучше" - усовершенствование утверждений, в описанном здесь смысле.

Правило Утверждения Переобъявления просто устанавливает, что честный субподрядчик, приняв условия контракта, должен выполнить работу на тех же условиях, что и подрядчик или лучших, но никак не худших.

С позиции Проектирования по Контракту, инварианты классов - это ограничения общего характера, применимые и к подрядчикам, и к клиентам. Правило родительских инвариантов отражает тот факт, что все подобные ограничения передаются субподрядчикам.

Свое истинное значение для ОО-разработки наследование приобретает лишь совместно с утверждениями и двумя приведенными выше правилами. Метафора контрактов и субподрядов - прекрасная аналогия, помогающая разрабатывать корректное ОО-ПО. Несомненно, в этом - одна из центральных идей теории проектирования.

Абстрактные предусловия

Правило ослабления предусловий может оказаться чересчур жестким в случае, когда наследник понижает уровень абстракции, характерный для его предка. К счастью, есть легкий обходной путь, полностью согласующийся с теорией.

Типичным примером этого является порождение BOUNDED_STACK от универсального класса стека (STACK). Процедура занесения в стек элемента (put) в порожденном классе имеет предусловие count <= capacity, где count - текущее число элементов в стеке, capacity - физическая емкость накопителя.

В общем понятии стека нет понятия емкости. Поэтому создается впечатление, будто при переходе к BOUNDED_STACK предусловие приходится **усилить** (от бесконечной емкости перейти к конечной). Как выстроить структуру наследования, не нарушая правило Утверждения Переобъявления?

Ответ становится очевиден, если мы ближе познакомимся с требованиями к клиенту. То, что нужно сохранить или ослабить, не обязательно является конкретным предусловием, как оно видится в реализации поставщика (реализация это его забота), но касается предусловия, **как оно видится клиенту**. Пусть процедура put класса STACK имеет вид:

```

put (x: G) is
    -- Поместить x на вершину.
    require
        not full

```

```
deferred  
ensure  
...  
end
```

где функция `full` всегда возвращает ложное значение, а значит, стек по умолчанию никогда не бывает полным.

```
full: BOOLEAN is  
    -- Заполнено ли представление стека?  
    -- (По умолчанию, нет)  
    do Result := False end
```

Тогда в `BOUNDED_STACK` достаточно переопределить `full`:

```
full: BOOLEAN is  
    -- Заполнено ли представление стека?  
    -- (Да, если число элементов равно емкости стека)  
    do Result := (count = capacity) end
```

Предусловие, такое как `not full`, включающее свойство, которое переопределяется потомками, называется абстрактным (*abstract*) предусловием.

Такое использование абстрактных предусловий для соблюдения правила Утверждения Переобъявления может показаться обманом, однако это не так. Несмотря на то, что конкретное предусловие фактически становится более сильным, абстрактное предусловие не меняется. Важно не то, как реализуется утверждение, а то, как оно представлено клиентам в интерфейсе класса (краткой или плоско-краткой форме). Предваренный условием вызов

```
if not s.full then s.put (a) end
```

будет корректен независимо от вида `STACK`, присоединенного к `s`.

Впрочем, есть доля справедливой критики этого подхода, так как он вступает в противоречие с принципом Открыт-Закрыт. При проектировании класса `STACK` мы должны предвидеть ограниченную емкость отдельных стеков. Не проявив должной предусмотрительности, нам придется вернуться к проектированию `STACK` и изменить интерфейс класса. Это неизбежно. Из следующих двух свойств только одно должно выполняться:

- ограниченный стек является стеком;
- в стек всегда можно добавить еще один элемент.

Если предпочесть первое свойство и допускать порождение `BOUNDED_STACK` от `STACK`, мы должны согласиться с тем, что общее понятие стека включает предположение о невозможности в ряде случаев выполнить операцию `put`, абстрактно выраженное запросом `full`.

Было бы ошибкой включить в виде постусловия подпрограммы `full` в классе `STACK` выражение `Result = False` или (придерживаясь рекомендуемого стиля, эквивалентный ему) инвариант `not full`. Это - случай излишней спецификации, ограничивающей свободу реализации компонентов потомками класса.

Правило языка

Правило Утверждений Переобъявления, так как оно сформулировано, является концептуальным руководством. Как преобразовать его в безопасное и проверяемое правило языка?

В принципе, чтобы убедиться в том, что старые предусловия влекут новые, а новые постусловия - старые, следует провести логический анализ тех и других утверждений. К сожалению, это требует наличия сложного **механизма доказательства теорем** (несмотря на десятилетия исследований в области искусственного интеллекта). Его применение в компиляторе пока не реально.

К счастью, возможно простое техническое решение. Нужное нам правило можно сформулировать через простое лингвистическое соглашение, основанное на том наблюдении, что для любых утверждений `a` и `b`:

- α влечет α or γ независимо от значения γ ;
- β and δ влечет β независимо от значения δ .

Итак, гарантируется, что новое предусловие слабее исходного α либо равно ему, если оно имеет вид $\alpha \text{ or } y$.

Гарантируется, что новое постусловие сильнее исходного β либо равно ему, если оно имеет вид $\beta \text{ and } \delta$.

Отсюда следует искомое языковое правило:

Правило (2) Утверждения Переобъявления

При повторном объявлении подпрограммы нельзя использовать предложения **require** или **ensure**. Вместо них следует использовать предложение, начинающееся с:

- **require else**, объединенное с исходным предусловием логической связкой **or**
- **ensure then**, объединенное с исходным постусловием логической связкой **and**.

При отсутствии таких предложений действуют исходные утверждения.

Заметим, что используются нестрогие булевы операторы **and then** и **or else**, а не обычные **and** и **or**, хотя чаще всего это различие несущественно.

Иногда получаемые утверждения могут оказаться сложнее, чем необходимо на самом деле. В примере с подпрограммой обращения матриц, где исходным было утверждение

```
invert (epsilon: REAL) is
    -- Обращение текущей матрицы с точностью epsilon
    require
        epsilon >= 10 ^ (-6)
    ...
    ensure
        ((Current * inverse) |-| One) <= epsilon
мы не вправе в повторном объявлении использовать require и ensure, поэтому результат примет вид
...
    require else
        epsilon >= 10 ^ (-20)
    ...
    ensure then
        ((Current * inverse) |-| One) <= (epsilon / 2)
```

а стало быть, предусловие формально станет таким: $(\text{epsilon} \geq 10^{-6}) \text{ or else } (\text{epsilon} \geq 10^{-20})$.

Ситуация с постусловием аналогична. Такое расширение не имеет особого значения, поскольку преобладает более слабое предусловие или более сильное постусловие. Если y влечет α , то $\alpha \text{ or else } y$ имеет то же значение, что и α . Если β влечет δ , то $\beta \text{ and then } \delta$ имеет то же значение, что и β . Поэтому математически предусловие повторного объявления есть: $\text{epsilon} \geq 10^{-6}$, а его постусловие есть: $((\text{Current} * \text{inverse}) |-| \text{One}) \leq (\text{epsilon} / 2)$, хотя запись утверждений в программе (а также, вероятно, их расчет во время выполнения при отсутствии средств символьных преобразований) является более сложной.

Повторное объявление функции как атрибута

Правило Утверждения Переобъявления нуждается в небольшом дополнении ввиду возможности при повторном объявлении задать функцию как атрибут. Что произойдет с предусловием функции и ее постусловием, если такие имелись?

Атрибут доступен всегда, а потому мы вправе считать, что его предусловие равно `True`. В итоге можно полагать, что предусловие атрибута, согласно правилу Утверждения Переобъявления, было ослаблено.

Но атрибут не имеет постусловий. Мы же должны гарантировать, что он наделен всеми свойствами, заданными исходной функцией. Поэтому (в дополнение к правилу Утверждения Переобъявления) будем считать, что в этом случае автоматически постусловие добавляется к инварианту класса. Плоская форма класса будет содержать это условие в составе своего инварианта.

Для функции без параметров, формулируя некое свойство ее результата, вы всегда можете выбрать, включать ли его в постусловие или в инвариант. С точки зрения стиля предпочтительно пользоваться инвариантом. Соблюдение этого правила позволит отказаться от внесения изменений в утверждения в будущем, если при повторном объявлении функция становится атрибутом.

Замечание математического характера

Неформально, правило Утверждения Переобъявления гласит: "Повторное объявление утверждений может лишь сужать область допустимого поведения, не нарушая ее". Сейчас, завершая обсуждение этой темы, приведем строгую формулировку данного свойства.

Пусть подпрограмма реализует частичную функцию r , отображающую множество возможных входных состояний I в множество возможных выходных состояний O . Утверждения подпрограммы определяют правила действия r и ее возможных переопределений.

- Предусловие задает область определения DOM функции r (подмножество I , на котором r гарантированно вырабатывает результат).
- Постусловие задает для каждого x из DOM подмножество $RESULTS(x)$ множества O , такое, что $r(x) \subseteq RESULTS(x)$. Так как постусловие не всегда однозначно описывает результат, это подмножество может иметь больше одного элемента.

Правило Утверждения Переобъявления означает, что повторное объявление может расширять область определения и сужать множество результатов. Пометив новые множества знаком ' \sqsubseteq ', запишем требования, закрепленные этим правилом:

$DOM' \sqsubseteq DOM$

$RESULTS'(x) \subseteq RESULTS(x)$ для всех x из DOM

Предусловие устанавливает, что подпрограмма и ее повторные объявления, **как минимум, должны** принимать некоторые входы (DOM), хотя повторные объявления могут это множество и расширить. Постусловие говорит, что результаты, возвращаемые подпрограммой и ее повторными объявлениями, **могут, самое большое,** содержать значения из **RESULTS(x)**, однако, постусловия при повторных объявлениях могут это множество сузить.

В этом описании состояние системы в период выполнения определяется состоянием (значениями) всех достижимых объектов. Кроме того, входные состояния (элементы I) также включают в себя значения аргументов. Более подробное введение в математическое описание программ и языков программирования см. в [М 1990].

Глобальная структура наследования

Ранее мы уже ссылались на универсальные (universal) классы GENERAL и ANY, а также на безобъектный (objectless) класс NONE. Пришло время пояснить их роль и представить глобальную структуру наследования.

Универсальные классы

Удобно использовать следующее соглашение:

Правило Универсального Класса

Любой класс, не содержащий предложение наследования, неявно содержит предложение вида:

inherit ANY,

ссылающееся на класс ANY из библиотеки Kernel.

Тем самым становится возможным определить по умолчанию целый ряд компонентов, наследуемых всеми классами. Эти компоненты реализуют общие, универсальные операции: копирование, клонирование, сравнение, базовый ввод и вывод.

Для большей гибкости поместим эти компоненты в класс GENERAL, чьим потомком является ANY. Сам класс ANY по умолчанию не имеет никаких компонентов, будучи классом вида: **class ANY inherit GENERAL end.** При создании нового проекта его менеджер может решить, какие общие для проекта компоненты следует включить в класс ANY, в то время как GENERAL остается всегда неизменным.

Для построения нетривиального ANY можно прибегнуть к наследованию. В самом деле, класс ANY можно породить от некоторого HOUSE_STYLE или нескольких таких классов, не вводя циклы в иерархию наследования и не нарушая правило об универсальном классе: достаточно сделать класс HOUSE_STYLE и другие классы потомками GENERAL. Вынесенный на [рис. 16.4](#) текст "Классы разработчика" означает все классы, написанные разработчиком и не порожденные от GENERAL явным образом.

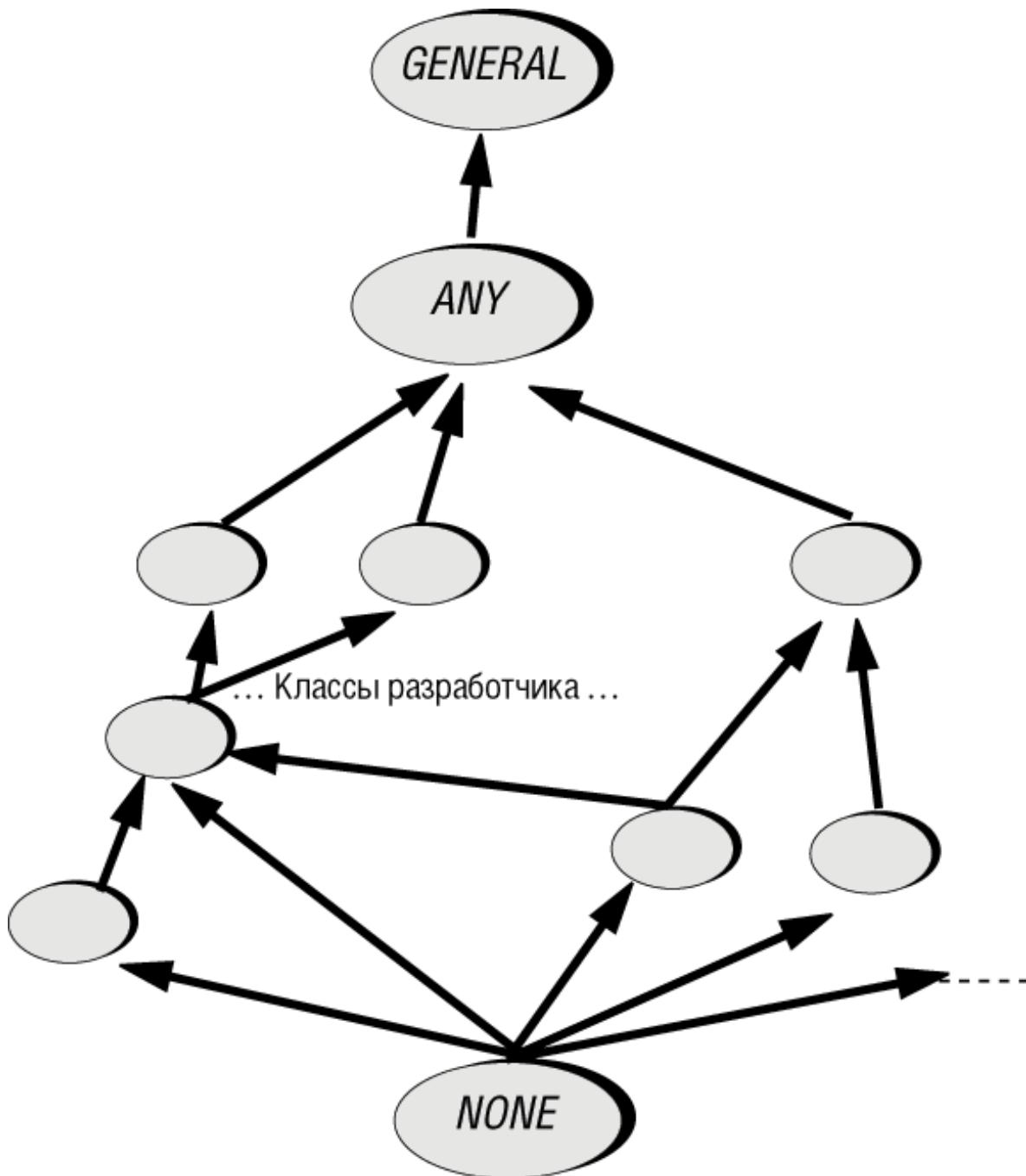


Рис. 16.4. Глобальная структура наследования

Нижняя часть иерархии

На [рис. 16.4](#) представлен также класс **NONE**, антипод класса **ANY**, потомок всех классов, не имеющих собственных наследников и превращающий глобальную иерархию наследования классов в решетку (математическую структуру). **NONE** не имеет потомков, его нельзя переопределить - это лишь удобная фикция, однако, теоретическое существование такого класса оправдано и служит двум практическим целям:

- **Void** - пустая ссылка, используемая наряду с другими ссылками, по соглашению имеет тип **NONE**. (Фактически, **Void** - это один из компонентов класса **GENERAL**.)
- Чтобы скрыть компонент от всех клиентов, достаточно экспорттировать его только классу **NONE**. Предложение **feature {NONE}** (практически эквивалентное **feature {}**, но записанное явно) или предложение наследования **export {NONE}** (на практике дающее тот же результат, что и **export {}**), делает компонент недоступным для любого класса, написанного разработчиком, ибо **NONE** не имеет потомков. Обратите внимание на то, что **NONE** скрывает и все свои компоненты.

Первое свойство объясняет, почему значение **Void** можно присвоить любому элементу ссылочного типа данных. До сих пор статус **Void** оставался некой загадкой, теперь, когда **Void** связано с классом **NONE**, этот статус становится очевидным, официальным и согласующимся с системой типов: по построению **NONE** является потомком всех классов, а потому мы можем использовать **Void** как допустимое значение любой ссылки, не нарушая правил описания типов.

По симметрии ко второму свойству заметим, что объявление, начинающееся с **feature** и экспортирующее все компоненты во все классы, написанные разработчиком, считается сокращением от **feature {ANY}**. Для повторного экспорта во все классы компонента родителя, доступ к которому был ограничен, можно использовать предложение **export {ANY}** или его не столь очевидное сокращение **export**.

Классы ANY и NONE обеспечивают замкнутость системы типов и полноту структуры наследования: решетка (это строго определенный математический термин) имеет свой верхний и нижний элемент.

Универсальные компоненты

Вот лишь некоторые компоненты, содержащиеся в классе GENERAL, а значит, доступные всем другим классам. Часть из них была введена и использована в предшествующих лекциях курса:

- `clone` для создания клона (дубля) объекта, а также его "глубинный" вариант `deep_clone` для рекурсивного дублирования полной структуры объекта;
- `copy` для копирования содержимого одного объекта в другой;
- `equal` для сравнения объектов (поле-с-полем), а также его "глубинный" вариант `deep_equal`;
- `print` и `print_line` - печать простого представления по умолчанию любого объекта (default representation);
- `tagged_out` - строка, содержащая представление по умолчанию любого объекта, в котором каждое поле сопровождается своей меткой (tag) (соответствующим именем атрибута);
- `same_type` и `conforms_to` - булевые функции, сопоставляющие тип текущего объекта с типом другого;
- `generator` - возвращает имя порождающего (generating) класса объекта, то есть класса, экземпляром которого является данный объект.

Замороженные компоненты

При обсуждении идеи наследования неоднократно подчеркивался принцип Открыт-Закрыт - право, взятое компонентом класса-родителя, переопределить его, возложив на него иные задачи. Могут ли появиться причины запрета такой возможности?

Запрет повторного объявления

Обсуждение утверждений в начале лекции дало нам теоретическое понимание сути переопределений. Часть "Открыт" принципа Открыт-Закрыт дает возможность изменять компоненты потомков, но под контролем утверждений. Разрешены лишь те повторные объявления, для которых реализация согласуется со спецификацией, заданной предусловием и постусловиям оригинала.

В ряде случаев клиентам класса и клиентам классов потомков нужна гарантия, что компонент не только соблюдает спецификацию, но и пользуется в точности исходной реализацией. Достичь этого можно лишь "заморозив" его реализацию - полностью запретив переопределение компонента. Подобную возможность дает простая языковая конструкция:

```
frozen feature_name ... is... Остальные объявления - как обычно...
```

При таком описании ни один из потомков класса не может включать данный компонент в предложения **redefine** и **undefine** ни под своим, ни под любым другим именем (смена имен, конечно же, по-прежнему разрешена). Отложенный компонент по своей сути должен быть переопределен и, следовательно, не может быть заморожен.

Фиксированная семантика компонентов `copy`, `clone` и `equality`

Чаще всего замороженные (frozen) компоненты применяются в операциях общего назначения, подобных тем, что входили в состав класса GENERAL. Так, есть две версии базовой процедуры копирования:

```
copy, frozen standard_copy (other: ...) is
    -- скопировать поля other в поля текущего объекта.
    require
        other_not_void: other /= Void
    do
        ...
    ensure
        equal (Current, other)
    end
```

Два компонента (copy и standard_copy) описаны как синонимы. Правила разрешают совместно описывать два компонента класса, если они имеют общее определение. Заметьте, в данном случае только один из компонентов допускает повторное объявление, второй - заморожен. В итоге потомки вправе переопределить copy, что необходимо, например классам ARRAY и STRING, которые сравнивают содержимое, а не значение указателей. Однако параллельно удобно иметь и замороженный вариант компонента для вызова при необходимости исходной операции - standard_copy.

Компонент clone, входящий в состав класса GENERAL, тоже имеет "двойника" standard_clone, однако обе версии заморожены. Зачем понадобилось замораживать clone? Причина кроется не в запрете задания иной семантики операции клонирования, а в необходимости сохранения совместимости семантик copy и clone, что, как побочный эффект, облегчает задачу разработчика. Общий вид объявления clone таков:

```
frozen clone (other:...): ... is
-- Void если other пуст; иначе вернуть новый объект, содержимое которого скопировано
из other.
do
    if other /= Void then
        Result := "Новый объект того же типа, что other"
        Result.copy (other)
    end
ensure
    equal (Result, other)
end
```

Фраза "**Новый объект того же типа, что other**" есть неформальное обозначение вызова функции, которая создает и возвращает объект того же типа, что и other. (Result равен Void, если other - "пустой" указатель.)

Несмотря на замораживание компонента clone, он будет изменяться, соответствуя любому переопределению copy, например в классах ARRAY и STRING. Это удобно (для смены семантики copy-clone достаточно переопределить copy) и безопасно (задать иную семантику clone было бы, скорее всего, ошибкой).

Переопределять clone не нужно (да и нельзя), однако при переопределении copy понадобится переопределить и семантику равенства. Как сказано в постусловиях компонентов copy и clone, результатом копирования должны быть тождественные объекты. Сама функция equal, по сути, зафиксирована, как и clone, но она зависит от компонентов, допускающих переопределение:

```
frozen equal (some, other: ...): BOOLEAN is
    -- Обе сущности some и other пусты или присоединены
    -- к объектам, которые можно считать равными?
do
Result := ((some = Void) and (other = Void)) or else some.is_equal (other)
ensure
Result = ((some = Void) and (other = Void)) or else some.is_equal (other)
end
```

Вызов equal (a, b) не соответствует строгому ОО-варианту a.is_equal (b), но на практике выгодно отличается от него, будучи применим, даже если a или b пусто. Базовый компонент is_equal не заморожен и требует согласованного переопределения в любом классе, переопределяющем copy. Это делается для того, чтобы семантика равенств оставалась совместимой с семантикой copy-clone, а постусловия copy и clone были по-прежнему верными.

Не злоупотребляйте замораживанием

Приведенные примеры замораживания - это типичные образцы применения механизма, гарантирующего точное соответствие копий и клонов семантике исходного класса.

Замораживание компонентов не следует делать по соображениям эффективности. (Эту ошибку иногда совершают программисты, работающие на C++ или Smalltalk, которым внущили мысль, будто динамическое связывание накладно и его нужно по возможности избегать.) Хотя вызов замороженных компонентов означает отсутствие динамического связывания, это лишь побочный эффект механизма **frozen**, а не его конечная цель. Выше мы подробно говорили о том, что безопасное статическое связывание - это проблема оптимизации, и решает ее компилятор, а не программист. В грамотно спроектированном языке компилятор обладает всем необходимым для такой и даже более сильной оптимизации, скажем, для подстановки тела функции в точку вызова (routine inlining). Поиск возможностей оптимизации - задача машин, а не человека. Пользуйтесь **frozen** в редких, но важных для себя случаях, когда это действительно необходимо (для обеспечения точного

соответствия семантике исходной реализации), и пусть ваш язык и ваш компилятор делают свою работу.

Ограниченнaя универсальность

Расширяя базовое понятие класса, мы представляли наследование и универсальность (genericity) как своего рода "партнеров". Объединить их нам позволило знакомство с **полиморфными структурами данных**: в контейнер - объект, описанный сущностью типа SOME_CONTAINER_TYPE [T] с родовым параметром T - можно помещать объекты не только самого типа T, но и любого потомка T. Однако есть и другая интересная комбинация партнерства, в которой наследование используется для задания ограничения на возможный тип фактического родового параметра класса.

Векторa, допускающие сложение

Приведем простой, но характерный пример, демонстрирующий необходимость введения ограниченной универсальности. Он поможет в обосновании метода решения поставленной задачи и в выборе соответствующей конструкции языка.

Предположим, что мы хотим объявить класс VECTOR, над элементами которого определена операция сложения. Потребность в подобном базовом классе неоспорима. Вот первый вариант:

```
indexing
    description: "Векторы со сложением"
class
    VECTOR [G]
feature -- Доступ
    count: INTEGER
        -- Количество элементов
    item, infix "@" (i: INTEGER): G is
        -- Элемент вектора с индексом i (нумерация с 1)
        require ... do
            ...
        end
feature -- Основные операции
    infix "+" (other: VECTOR [G]): VECTOR is
        -- Поэлементное сложение текущего вектора с other
        require ... do
            ...
        end
    ... Прочие компоненты ...
invariant
    non_negative_count: count >= 0
end
```

Применение инфиксной записи продиктовано соображениями удобства. Для удобства введены и синонимы в обозначении i-го компонента вектора: v.item (i) или просто v @ i.

Обратимся к функции "+". Сначала сложение двух векторов кажется очевидным и состоящим в суммировании элементов на соответствующих местах. Общая его схема такова:

```
infix "+" (other: VECTOR [G]): VECTOR is
    -- Поэлементное сложение текущего вектора с other
    require
        count = other.count
local
    i: INTEGER
do
    "Создать Result как массив из count элементов"
    from i := 1 until i > count loop
        Result.put(item (i) + other.item (i), i)
        i := i + 1
    end
end
```

Выражение в прямоугольнике - результат сложения i-го элемента текущего вектора с i-м элементом other. Процедура put сохраняет это значение в i-м элементе Result, и хотя она не показана в классе VECTOR, данная процедура в нем, безусловно, присутствует.

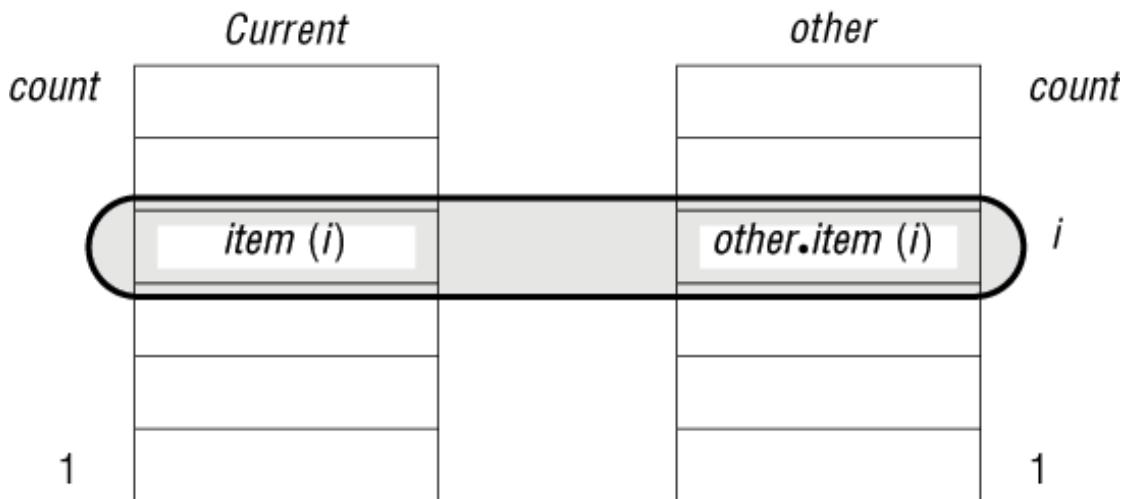


Рис. 16.5. Поэлементное сложение векторов

Но подобная схема не работает! Операция +, которую мы определили для сложения векторов (VECTOR), здесь применяется к объектам совсем другого типа (G), являющегося родовым параметром. По определению, родовой параметр представлен неизвестным типом - фактическим параметром, появляющимся только тогда, когда нам понадобится для каких либо целей родовой класс. Процесс порождения класса при задании фактического родового параметра называется родовым порождением (**generic derivation**). Если фактическим параметром служит INTEGER либо иной тип (класс), содержащий функцию infix "+" правильной сигнатуры, корректная работа обеспечена. Но что если параметром станет ELLIPSE, STACK, EMPLOYEE или другой тип без операции сложения?

С прежними родовыми классами: контейнерами STACK, LIST и ARRAY - этой проблемы не возникало, поскольку их действия над элементами (типа G как формального параметра) были универсальны - операции (присваивание, сравнение) могли выполняться над элементами любого класса. Но для абстракций, подобных векторам, допускающих сложение, нужно ограничить круг допустимых фактических родовых параметров, чтобы быть уверенными в допустимости проектируемых операций.

Этот случай отнюдь не является исключением. Вот еще два примера того же рода.

- Предположим, вы проектируете класс, описывающий структуру данных с операцией sort, упорядочивающей элементы структуры в соответствии с некоторым критерием сортировки. Тогда элементы этой структуры должны принадлежать типу, для которого определена операция сравнения infix "<=", задающая порядок для любой пары соответствующих объектов.
- При разработке таких базисных структур данных как словари зачастую используется для хранения данных **хеш-таблица**, в которой место элемента определяется ключом, вычисляемым по значению элемента. Элементы, размещаемые в словаре должны принадлежать классу, допускающему применение хеш-функции, вычисляющей ключ каждого элемента.

Не ОО-подход

Переходя к решению этой проблемы, посмотрим, как с такой задачейправлялись другие, не ОО-языки.

В языке Ada нет классов, но зато есть пакеты для группировки взаимосвязанных типов и операций. Пакет может быть родовым, с родовыми параметрами, представляющими типы. При этом возникает та же проблема: пакет VECTOR_PROCESSING может включать объявление типа VECTOR и эквивалент нашей функции infix "+".

Решение в языке Ada рассматривает необходимые операции, например инфиксное сложение, как родовые параметры. Параметрами пакета могут быть не только типы, как при объектном подходе, но и подпрограммы. Например:

```
generic
    type G is private;
    with function "+" (a, b: G) return G is <>;
    with function "*" (a, b: G) return G is <>;
    zero: G; unity: G;
package VECTOR_HANDLING is
    ... Интерфейс пакета ...
end VECTOR_HANDLING
```

Заметим, что наряду с типом G и подпрограммами родовым параметром служит значение zero - нулевой элемент сложения. Типичное использования пакета:

```
package BOOLEAN_VECTOR_HANDLING is
    new VECTOR_HANDLING (BOOLEAN, "or", "and", false, true);
```

В этом примере логическая операция **or** используется как сложение, **and** - умножение, а также задаются соответствующие значения для **zero** и **unity**. Подробнее мы обсудим этот пример в одной из следующих лекций курса.

Являясь решением для Ada, данный прием не применим в объектной среде. Основа ОО-подхода - приоритет типов данных над операциями при декомпозиции ПО, чьим следствием является отсутствие независимых операций. **Всякая операция принадлежит некоторому типу данных, основанному на классе.** Следовательно, возникшая "на пустом месте" функция, скажем, **infix "+"**, не может быть фактическим родовым параметром, стоящим в одном ряду с типами **INTEGER** и **BOOLEAN**. То же касается и значений, таких как **zero** и **unity**, обязанных знать свое место - быть компонентами класса - вполне респектабельными членами ОО-сообщества.

Ограничение родового параметра

Эти наблюдения дают решение. Мы должны оперировать исключительно терминами классов и типов.

Потребуем, чтобы любой фактический параметр, используемый классом **VECTOR** (в других примерах по аналогии), был типом, поставляемым с множеством операций: **infix "+"**, **zero** для инициализации суммы и т.д. Владея наследованием, мы знаем, как снабдить тип нужными операциями, - нужно просто сделать его потомком класса, отложенного или эффективного, обладающего этими операциями.

Синтаксически это выглядит так:

```
class C [G -> CONSTRAINING_TYPE] ... Все остальное как обычно ...
```

где **CONSTRAINING_TYPE** - произвольный тип, именуемый родовым ограничением (generic constraint). Символ **->** обозначает стрелку на диаграммах наследования. Результат этого объявления в том, что:

- в роли фактических родовых параметров могут выступать лишь типы, совместимые с **CONSTRAINING_TYPE**;
- в классе **C** над сущностью типа **G** допускаются только те операции, которые допускаются над сущностью **CONSTRAINING_TYPE**, другими словами, представляющими собой компоненты базового класса этого типа.

Какое родовое ограничение использовать для класса **VECTOR**? Обсуждая множественное наследование, мы ввели в рассмотрение **NUMERIC** - класс объектов, допускающих базисные арифметические операции: сложение и умножение с нулем и единицей (лежащая в его основе математическая структура называется кольцом). Эта модель кажется вполне уместной, хотя нам необходимо пока только сложение. Соответственно, класс будет описан так:

```
indexing
    description: "Векторы, допускающие сложение"
class
    VECTOR [G -> NUMERIC]
... Остальное - как и раньше (но теперь правильно!) ...
```

После чего ранее некорректная конструкция в теле цикла

```
Result.put(item (i) + other.item (i), i)
```

становится допустимой, поскольку **item (i)** и **other.item (i)** имеют тип **G**, а значит, к ним применимы все операции **NUMERIC**, включая, инфиксный **"+"**.

Следующие родовые порождения корректны, если полагать, что все классы, представленные как фактические родовые параметры, являются потомками **NUMERIC**:

```
VECTOR [NUMERIC]
VECTOR [REAL]
VECTOR [COMPLEX]
```

Класс EMPLOYEE не порожден от NUMERIC, так что попытка использовать VECTOR [EMPLOYEE] приведет к ошибке времени компиляции.

Абстрактный характер NUMERIC не вызывает никаких проблем. Фактический параметр при порождении может быть как эффективным (примеры выше), так и отложенным (VECTOR [NUMERIC_COMPARABLE]), если он порожден от NUMERIC.

Аналогично описываются класс словаря и класс, поддерживающий сортировку:

```
class DICTIONARY [G, H -> HASHABLE] ...
class SORTABLE [G -> COMPARABLE] ...
```

Игра в рекурсию

Вот некий трюк с нашим примером: спросим себя, возможен ли вектор векторов? Допустим ли тип VECTOR [VECTOR [INTEGER]]?

Ответ следует из предыдущих правил: только если фактический родовой параметр совместим с NUMERIC. Сделать это просто - породить класс VECTOR от класса NUMERIC (см. упражнение 16.2):

```
indexing
    description: "Векторы, допускающие сложение"
class
    VECTOR [G -> NUMERIC]
inherit
    NUMERIC
... Остальное - как и раньше...
```

Векторы, подобные этому, можно и впрямь считать "числовыми". Операции сложение и умножение дают структуру кольца, в котором роль нуля (zero) играет вектор из G-нулей, и роль единицы (unity) - вектор из G-единиц. Операция сложения в этом кольце - это, строго говоря, векторный вариант infix "+", речь о котором шла выше.

Можно пойти дальше и использовать VECTOR [VECTOR [VECTOR [INTEGER]]] и так далее - приятное рекурсивное приложение ограниченной универсальности.

И снова неограниченная универсальность

Конечно же, не все случаи универсальности ограничены. Форма - STACK [G] или ARRAY [G] - по-прежнему существует и называется неограниченной универсальностью. Пример DICTIONARY [G, H -> HASHABLE] показывает, что класс одновременно может иметь как ограниченные, так и неограниченные родовые параметры.

Изучение ограниченной универсальности дает шанс лучше понять неограниченный случай. Вы, конечно же, вывели правило, по которому **class C [G]** следует понимать как **class C [G -> ANY]**. Поэтому если G - неограниченный типовой параметр (например, класса STACK), а x - сущность, имеющая тип G, то мы точно знаем, что можем делать с сущностью x: читать и присваивать значения, сравнивать (=, /=), передавать как параметр и применять в универсальных операциях clone, equal и прочее.

Попытка присваивания

Наша следующая техника адресуется к тем областям Объектной страны, в которых из страха тиранического поведения мы не можем позволить править простым правилам типизации, не встречая никакого сопротивления.

Когда правила типов становятся несносными

Цель правил типов, введенных вместе с наследованием, в достижении статически проверяемого динамического поведения, так чтобы система, прошедшая проверку при компиляции, не выполняла неадекватных операций над объектами во время выполнения.

Вот два основных правила, представленных в первой лекции о наследовании ([лекция 14](#)).

- **Правило Вызыва Компонентов:** запись x . f осмыслена лишь тогда, когда базовый класс x содержит и экспортирует компонент f.
- **Правило Совместимости Типов:** при передаче a как аргумента или при присваивании его некой

сущности необходимо, чтобы тип *a* был совместим с ожидаемым, то есть основан на классе, порожденным от класса сущности.

Правило Вызова Компонентов не является причиной каких-либо проблем - это фундаментальное условие всякой работы с объектами. Естественно, что обращаясь к компоненту объекта, нужно проверить, действительно ли данный класс предлагает и экспортирует данный компонент.

Правило Совместимости Типов требует больше внимания. Оно предполагает наличие у нас всей информации о типах объектов, с которыми мы работаем. Как правило, это так, - создав объекты, мы знаем, чем они являются, но иногда информация может частично отсутствовать. Вот два таких случая.

- В полиморфной структуре данных мы располагаем лишь информацией, общей для всех объектов структуры; однако нам может понадобиться и специфическая информация, применимая только к отдельному объекту.
- Если объект приходит из внешнего мира - файл или по сети - мы обычно не можем доверять тому, что он принадлежит определенному типу.

Давайте займемся исследованием примеров этих двух случаев. Рассмотрим для начала полиморфную структуру данных, такую как список геометрических фигур:

```
figlist: LIST [FIGURE]
```

В предыдущих лекциях рассматривалась иерархия наследования фигур. Пусть нам необходимо найти самую длинную диагональ среди всех прямоугольников списка (и вернуть -1, если прямоугольников нет). Сделать это непросто. Выражение *item* (*i*).diagonal, где *item* (*i*) - *i*-й элемент списка, идет вразрез с правилом вызова компонентов: *item* (*i*) имеет тип FIGURE, а этот класс, в отличие от его потомка RECTANGLE, не содержит в своем составе компонента diagonal. Решение, используемое до сих пор, изменяло определение класса, - в нем появлялся атрибут, задающий тип фигуры. Однако это решение не столь элегантно, как нам хотелось бы.

Теперь пример второго рассматриваемого случая. Пусть имеется механизм хранения объектов в файле или передачи их по сети, аналогичный универсальному классу STORABLE, описанному нами ранее. Для получения объекта используем:

```
my_last_book: BOOK
...
my_last_book := retrieved (my_book_file)
```

Значение, возвращаемое retrieved, имеет тип STORABLE библиотеки Kernel, хотя с тем же успехом оно может иметь тип ANY. Но мы не ожидали STORABLE или ANY, - мы надеялись получить именно BOOK. Присваивание my_last_book нарушает правило Совместимости Типов.

Даже если написать собственную функцию retrieved, учитывающую специфику приложения и объявленную с подходящим типом, вам не удастся полностью на нее положиться. В отличие от объектов вашего ПО, в котором согласованность типов гарантируется действующими правилами, данный объект к вам поступает со стороны. При его получении вы могли ошибиться в выборе имени файла и прочитать объект EMPLOYEE вместо объекта BOOK, файл мог быть подделан, а при сетевом доступе данные могли быть искажены при передаче.

Проблема

Из этих примеров ясно: нам может понадобиться механизм удостоверения типа объекта.

Решение этой проблемы, возникающей в специфических, но критически важных случаях, должно быть найдено без потери преимуществ ОО-стиля разработки. В частности, мы не хотим возвращаться к той схеме, которую сами и осудили:

```
if "f типа RECTANGLE" then
...
elseif "f типа CIRCLE" then
...
и т.д.
```

Это решение идет вразрез с принципами Единственного Выбора и Открыт-Закрыт. Избежать риска потерь нам помогут два обстоятельства.

- Нет смысла создавать универсальный механизм выяснения типа объектов. В том и другом случае тип

объекта **предположительно известен**. Все, что нам нужно, - это способ проверки гипотезы. Определение принадлежности объекта данному типу носит более частный характер, чем запрос на определение типа. Кроме того, нам **не** требуется вводить в наш язык никаких операций над типами, к примеру, их сравнение - ужасающая мысль.

- Как уже говорилось, мы не должны влиять на правило Вызова Компонентов. Ни при каких обстоятельствах мы не должны проверять применимость вызова компонента, если класс прошел статистическую проверку. Все, что нам нужно, - это более свободная версия другого правила - правила совместимости типов, позволяющая "испытать тип" и проверить результат.

Механизм решения

И снова запись механизма решения напрямую вытекает из анализа поставленной проблемы. Введем новую форму присваивания, назвав ее **попыткой присваивания (assignment attempt)**:

```
target ?= source
```

Знак вопроса указывает на предварительный характер операции. Пусть сущность target имеет тип T, тогда попытка присваивания дает следующий результат:

- если source ссылается на объект совместимого с T типа, присоединить target к объекту так, как это делает обычное присваивание;
- иначе (если source равно void или ссылается на объект несовместимого типа) присвоить target значение void.

На эту инструкцию не действуют никакие ограничения типов, кроме одного: тип target (T) должен быть ссылочным.

Новое средство быстро и элегантно решает поставленные проблемы и, прежде всего, дает возможность обращаться к объектам полиморфной структуры с учетом их типа:

```
maxdiag (figlist: LIST [FIGURE]): REAL is
-- Максимальная длина диагонали прямоугольника в списке;
-- если прямоугольников нет, то -1.
    require
        list_exists: figlist /= Void
    local
        r: RECTANGLE
    do
        from
            figlist.start; Result := -1.0
        until
            figlist.after
        loop
            r ?= figlist.item
            if r /= Void then
                Result := Result.max (r.diagonal)
            end
            figlist.forth
        end
    end
end
```

Здесь применяются обычные итерационные механизмы работы с последовательными структурами данных ([лекция 5](#) курса "Основы объектно-ориентированного проектирования"). Компонент start служит для перехода к первому элементу (если он есть), after - для выяснения того, имеются ли еще не пройденные элементы, forth - для перехода на одну позицию, item (определенный, если **not after**) - для выборки текущего элемента.

В попытке присваивания используется локальная сущность r типа RECTANGLE. Успех присваивания проверяется сравнением значения r с Void. Если r не Void, то r - прямоугольник и можно обратиться к r.diagonal. Эта схема проверки вполне типична.

Заметим, что мы никогда не нарушаем правило Вызова Компонентов: обращения к r.diagonal защищены дважды: статически - компилятором, проверяющим, является ли diagonal компонентом класса RECTANGLE, и динамически - нашей гарантией того, что r не Void, а имеет присоединенный объект.

Обращение к элементу списка - потомку класса RECTANGLE, например SQUARE (квадрат), связывает r с

объектом, и его диагональ будет участвовать в вычислениях.

Пример с универсальной функцией чтения объектов `retrieval` выглядит так:

```
my_last_book: BOOK
... Сравните с := в первой попытке
my_last_book ?= retrieved (my_book_file)
if my_last_book /= Void then
    ... "Обычные операции над my_last_book" ...
else
    ... "Полученное не соответствует ожиданию" ...
end
```

Правильное использование попытки присваивания

Необходимость попытки присваивания обусловлена, как правило, тем, что на статически объявленный тип сущности положиться нельзя, а опознать тип фактически адресуемого объекта необходимо "на лету".

Например, при работе с полиморфными структурами данных и получении объектов из третьих рук.

Заметьте, как тщательно был спроектирован механизм, дающий разработчикам шанс забыть об устаревшем стиле разбора вариантов (`case-by-case`). Если вы действительно хотите перехитрить динамическое связывание и отдельно проверять каждый вариант типа, вы можете это сделать, хотя вам и придется немало потрудиться. Так, вместо обычного `f.display`, использующего ОО-механизмы полиморфизма и динамического связывания, можно, - но **не** рекомендуется, - писать:

```
display (f: FIGURE) is
-- Отобразить f, используя алгоритм,
-- адаптируемый к истинной природе объекта.
local
    r: RECTANGLE; t: TRIANGLE; p: POLYGON; s: SQUARE
    sg: SEGMENT; e: ELLIPSE; c: CIRCLE;?
    do
        r ?= f; if r /= Void then "Использовать алгоритм вывода прямоугольника" end
        t ?= f; if t /= Void then "Использовать алгоритм вывода треугольника" end
        c ?= f; if c /= Void then "Использовать алгоритм вывода окружности" end
        ... и т.д. ...
    end
```

На практике такая схема даже хуже, чем кажется, так как структура наследования имеет несколько уровней, а значит, усложнения управляющих конструкций не избежать.

Из-за трудностей написания таких закрученных конструкций попытки присваивания новичкам вряд ли придет в голову использовать их вместо привычной ОО-схемы. Однако опытные специалисты должны помнить о возможности неправильного использования конструкции.

Немного похожий на попытку присваивания механизм "сужения" (**narrowing**) есть в языке Java. В случае несоответствия типов он выдает исключение. Это похоже на самоубийство, неуспех присваивания вовсе не является чем-то ненормальным, это ожидаемый результат. Оператор `instanceof` в языке Java выполняет проверку типов на совместимость.

Из-за отсутствия в языке универсальности Java активно использует оба механизма. Отчасти это связано с тем, что в отсутствие множественного наследования Java не содержит класса `NONE`, а потому не может выделить эквиваленту `Void` надежное место в собственной системе типов.

Типизация и повторное объявление

Повторное объявление компонентов не требует сохранения сигнатуры. Пока оно виделось нам как замена одного алгоритма другим или - для отложенного компонента - запись алгоритма, соответствующего ранее заданной спецификации.

Но, воплощая идею о том, что класс способен предложить более специализированную версию элемента, описанного его предком, мы вынуждены иногда изменять типы данных. Приведем два характерных примера.

Устройства и принтеры

Вот простой пример переопределения типа. Рассмотрим понятие устройства, включив предположение о том,

что для любого устройства есть альтернатива, так что устройство можно заменить, если оно по каким-либо причинам недоступно:

```
class DEVICE feature
    alternate: DEVICE
    set_alternate (a: DEVICE) is
        -- Пусть a - альтернативное устройство.
        do
            alternate := a
        end
    ... Прочие компоненты ...
end
```

Принтер является устройством, так что использование наследования оправдано. Но альтернативой принтера может быть только принтер, но не дисковод для компакт-дисков или сетевая карта, - поэтому мы должны переопределить тип:

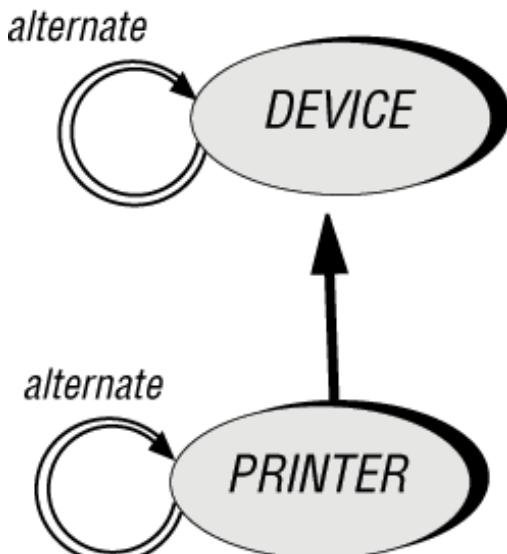


Рис. 16.6. Устройства и принтеры

```
class PRINTER inherit
    DEVICE
        redefine alternate, set_alternate
feature
    alternate: PRINTER
    set_alternate (a: PRINTER) is
        -- Пусть a - альтернативное устройство.
        ... Тело как у класса DEVICE ...
    ... Прочие компоненты ...
end
```

В этом и проявляется специализирующая природа наследования.

Одно- и двусвязные элементы

В следующем примере мы обратимся к базовым структурам данных. Рассмотрим библиотечный класс LINKABLE, описывающий односвязные элементы, используемые в LINKED_LIST - одной из реализаций списков. Вот частичное описание класса:

```
indexing
    description: "Односвязные элементы списка"
class LINKABLE [G] feature
    item: G
    right: LINKABLE [G]
    put_right (other: LINKABLE [G]) is
        -- Поместить other справа от текущего элемента.
        do right := other end
    ... Прочие компоненты ...
end
```

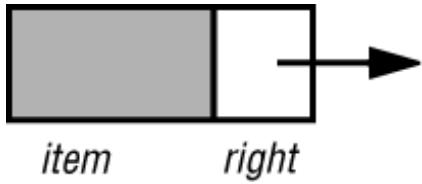


Рис. 16.7. Односвязный элемент списка

Ряд приложений требуют двунаправленных списков. Класс TWO_WAY_LIST - наследник LINKED_LIST должен быть также наследником класса BI_LINKABLE, являющегося наследником класса LINKABLE.

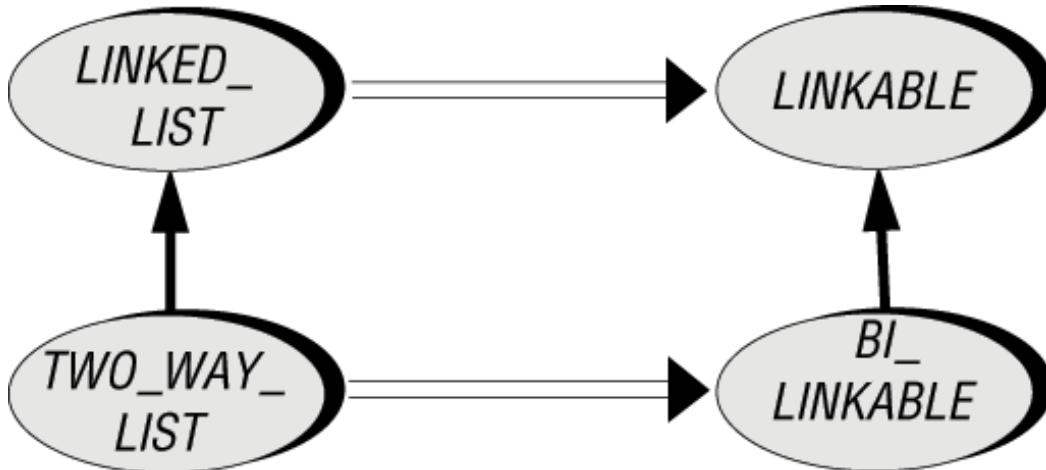


Рис. 16.8. Параллельные иерархии

Двусвязный элемент списка имеет еще одно поле:

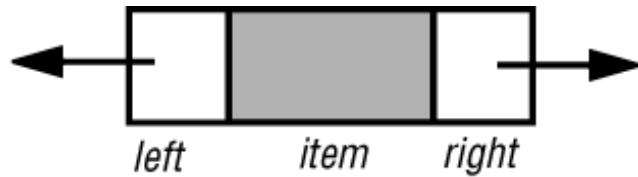


Рис. 16.9. Двусвязный элемент списка

В состав двунаправленных списков должны входить лишь двусвязные элементы (хотя последние, в силу полиморфизма, вполне можно внедрять и в односторонние структуры). Переопределив right и put_right, мы гарантируем однородность двусвязных списков.

```

indexing
    description: "Элементы двусвязного списка"
class BI_LINKABLE [G] inherit
    LINKABLE [G]
        redefine right, put_right end
feature
    left, right: BI_LINKABLE [G]
    put_right (other: BI_LINKABLE [G]) is
        -- Поместить other справа от текущего элемента.
do
    right := other
    if other /= Void then other.put_left (Current) end
end
put_left (other: BI_LINKABLE [G]) is
    -- Поместить other слева от текущего элемента.
    ... Упражнение для читателя ...
    ... Прочие компоненты ...
invariant
    right = Void or else right.left = Current
    left = Void or else left.right = Current
end

```

(Попробуйте написать put_left. Здесь скрыта ловушка! См. приложение А.)

Правило повторного объявления типов

Примеры, рассмотренные выше, несмотря на все их различия, объединяет необходимость повторного объявления типов. Спуск по иерархии наследования означает специализацию классов, и в соответствии со специализацией изменяются типы функций и типы аргументов подпрограмм, как, например, а в `set_alternate` и `other` в `put_right`; изменяются типы запросов - `alternate` и `right`.

Этот аспект повторного объявления выражает следующее правило:

Правило повторного объявления типов

При повторном объявлении компонента можно заменить тип компонента (для атрибутов и функций) или тип формального параметра (для подпрограмм) любым совместимым типом.

Правило использует понятие совместимости типов. Связка "или", стоящая в тексте правила, не исключает того, что при повторном объявлении функции мы можем одновременно изменить как тип результата функции, так и тип одного или нескольких ее аргументов.

Любое повторное объявление ведет к специализации, а, следовательно, к изменению типов. Так, с переходом к двунаправленным спискам параметры и результаты функций сменили свой тип на `BT_LINKABLE`. Отсюда становится понятен тот термин, которым часто описывают политику редекларации типов, - **ковариантная типизация (covariant typing)**, где приставка "ко" указывает на параллельное изменение типов при спуске по диаграмме наследования.

Ковариантная типизация таит в себе немало проблем, которые возникают у создателей компиляторов, нередко перекладывающих их решение на плечи разработчиков приложений.

Закрепленные объявления

Правило повторного объявления типов способно свести на нет целый ряд преимуществ наследования. Почему это происходит и каково решение данной проблемы?

Несогласованность типов

Рассмотрим пример с участием класса `LINKED_LIST`. Пусть мы имеем процедуру добавления в список нового элемента с заданным значением, который вставляется справа от текущего элемента. В деталях процедуры нет ничего необычного, но все же обратим внимание на потребность создания локальной сущности `new` типа `LINKABLE`, представляющей элемент списка, который будет создан и включен в список.

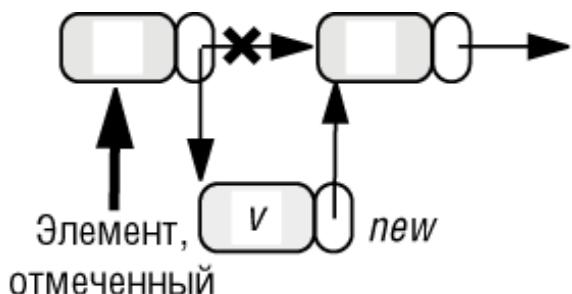


Рис. 16.10. Добавление элемента

```
put_right (v: G) is
-- Вставить элемент v справа от курсора.
    -- Не передвигать курсор.
    require
        not after
    local
        new: LINKABLE [T]
    do
        create new.make (v)
        put_linkable_right (new)
        ...
    ensure
        ... См. приложение А ...
    end
```

Для вставки нового элемента, имеющего значение v , необходимо предварительно создать элемент типа `LINKABLE [G]`. Вставка производится закрытой процедурой `put_linkable_right`, принимающей `LINKABLE` как параметр (и связывающей его с текущим элементом, используя процедуру `put_right` класса `LINKABLE`). Эта процедура осуществляет все нужные манипуляции со ссылками.

У потомков `LINKED_LIST`, таких как `TWO_WAY_LIST` или `LINKED_TREE`, процедура `put_right` тоже должна быть применимой. Но у них она работать не будет! Хотя алгоритм ее остается корректным, сущность `new` для них должна иметь другой тип - `BI_LINKABLE` или `LINKED_TREE`. Поэтому в каждом потомке нужно переопределять и переписывать целую процедуру, и это притом, что ее тело будет идентично оригиналу, за исключением переопределения `new!` Для подхода, претендующего на решение проблемы повторного использования, это серьезный порок.

Примеры из практики

Было бы ошибочно полагать, что проблема неоправданного переопределения возникает лишь там, где структура ориентирована на реализацию, как в `LINKED_LIST`. В любой схеме вида

```
some_attribute: SOME_TYPE
set_attribute (a: SOME_TYPE) is do ... end
```

переопределение `some_attribute` подразумевает соответствующее переопределение `set_attribute`. В случае с `put_right` из `BI_LINKABLE` (не путайте с подпрограммой из `LINKED_LIST`) повторное определение необходимо, поскольку фактически меняется алгоритм. Но во многих широко распространенных случаях (к примеру, в `set_alternate`) новый алгоритм идентичен исходному.

Вот еще один пример, показывающий глубину проблемы (не ограниченной лишь процедурами `set_xxx`, которые сами появились в силу принципа Скрытия информации). Добавим в класс `POINT` функцию, которая возвращает точку, сопряженную с данной, - ее зеркальное отражение относительно горизонтальной оси:

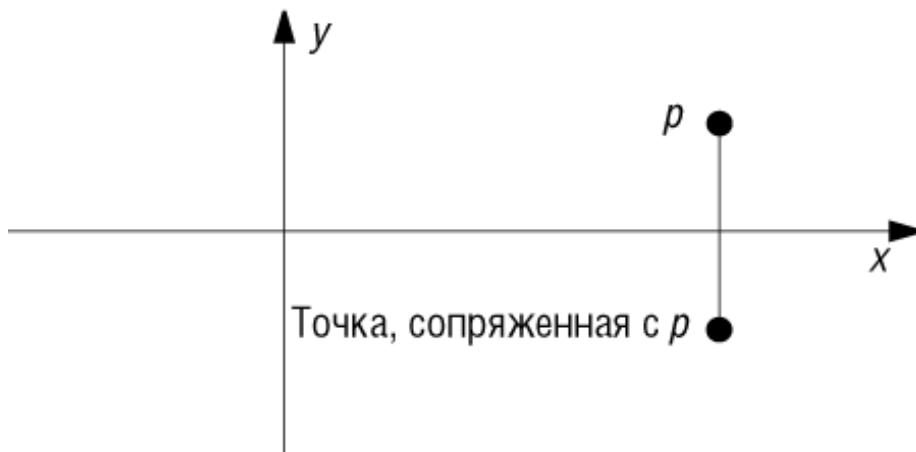


Рис. 16.11. Исходная и сопряженная точка

```
conjugate: POINT is
    -- Точка, сопряженная с текущей
do
    Result := clone (Current) -- Получить копию текущей точки
    Result.move (0, -2*y) -- Перенести результат по вертикали
end
```

Рассмотрим теперь некий класс, порожденный от `POINT`, например `PARTICLE`. К атрибутам частиц, помимо координат, относятся, вероятно, масса и скорость. По идеи, функция `conjugate` применима и к `PARTICLE` и выдает в результате ту же частицу с противоположным значением координаты y . Но если оставить все как есть, функция работать не будет из-за несоблюдения правила совместимости типов:

```
p1, p2: PARTICLE; create p1.make (...); ...
p2 := p1.conjugate
```

Правая часть подчеркнутого оператора имеет тип `POINT`, левая часть - тип `PARTICLE`. Правило совместимости типов этого не допускает. Поэтому мы должны переписать `conjugate` для `PARTICLE` с единственной целью - обеспечить соблюдение правила.

Предприняв попытку присваивания, мы не решим проблему, а лишь запишем в р2 пустой указатель.

Серьезное затруднение

Изучив класс LINKED_LIST в тексте приложения А, вы поймете, что проблема еще масштабнее. В теле класса содержится множество объявлений со ссылкой на тип LINKABLE [G], а с переходом к двунаправленным спискам почти все они потребуют повторного определения. Так, вариант представления списка включает четыре ссылки на отдельные элементы:

```
first_element, previous, active, next: LINKABLE [G]
```

В классе TWO_WAY_LIST каждая из этих сущностей должна быть объявлена заново. Аналогичная процедура ждет и другие порожденные классы. Многие функции, такие как put_right, имеют "односвязные" аргументы и нуждаются в повторном определении. В итоге реализация TWO_WAY_LIST будет во многом дублировать оригинал.

Понятие опорного элемента

В отличие от других проблем, решение которых предложено в этой лекции, такое тиражирование кода не связано с тем, что система типов препятствует нам в выполнении задуманного. Повторное объявление ковариантных типов разрешает их переопределение, но заставляет нас заниматься утомительным копированием текста.

Заметим: наши примеры действительно требуют переопределения типа, но ничего более. Все сводится только к этому. Из этого следует решение проблемы - необходимо создать механизм не абсолютного, а **относительного** объявления типа сущности.

Назовем такое объявление закрепленным (anchored). Пусть закрепленное объявление типа имеет вид

```
like anchor
```

где anchor, именуемый опорным (anchor) элементом объявления, - это либо запрос (атрибут или функция) текущего класса, либо предопределенное выражение Current. Описание **my_entity: like anchor** в классе А, где anchor - запрос, означает выбор для сущности типа, аналогичного anchor, с оговоркой, что любое переопределение anchor вызовет неявное переопределение my_entity.

Если anchor имеет тип Т, то в силу закрепленного объявления my_entity в классе А будет трактоваться так, будто тоже имеет тип Т. Рассматривая лишь класс А, вы не найдете различий между объявлениями:

```
my_entity: like anchor
my_entity: T
```

Различия проявятся только в потомках А. Будучи описана подобной (**like**) anchor, сущность my_entity автоматически будет следовать всем переопределениям типа anchor, освобождая от них автора класса.

Обнаружив, что класс содержит ряд сущностей, чьи потомки должны переопределяться одинаково, вы можете избавить себя от всех переопределений, кроме одного, объявив все элементы "подобными" (**like**) первому и определяя заново лишь его. Остальное будет сделано автоматически.

Вернемся к LINKED_LIST. Выберем first_element в качестве опорного для других сущностей типа LINKABLE [G]:

```
first_element: LINKABLE [G]
previous, active, next: like first_element
```

Локальная сущность new процедуры put_right класса LINKED_LIST тоже должна иметь тип **like first_element**, и это - единственное изменение в процедуре. Теперь достаточно переопределить first_element как BI_LINKABLE в классе TWO_WAY_LIST, как LINKED_TREE в LINKED_TREE и т.д. Сущности, описанные как **like**, не нужно указывать в предложении **redefine**. Не требуется и повторное определение put_right.

Итак, закрепленные определения есть весьма важное средство сохранения возможности повторного использования при статической типизации.

Опорный элемент Current

В качестве опорного элемента можно использовать **Current**, обозначающий текущий экземпляр класса (о текущем экземпляре см. [лекцию 7](#)). Сущность, описанная в классе A как **like Current**, будет считаться в нем имеющей тип A, а в любом B, порожденном от A, - имеющей тип B.

Эта форма закрепленного объявления помогает решить оставшиеся проблемы. Исправим объявление **conjugate**, получив правильный тип результата функции класса **POINT**:

```
conjugate: like Current is
    ... Все остальное - в точности, как раньше ...
```

Теперь в каждом порожденном классе тип результата **conjugate** автоматически определяется заново. Так, в классе **PARTICLE** он меняется на класс **PARTICLE**.

В классе **LINKABLE**, найдя объявления

```
right: LINKABLE [G]
put_right (other: LINKABLE [G]) is...
```

замените **LINKABLE [G]** на **like Current**. Компонент **left** класса **BI_LINKABLE** объявите аналогично.

Эта схема применима ко многим процедурам **set_attribute**. В классе **DEVICE** имеем:

```
class DEVICE feature
    alternate: like Current
    set_alternate (a: like Current) is
        -- Пусть a - альтернативное устройство.
        do
            alternate := a
        end
    ... Прочие компоненты ...
end
```

Еще раз о базовых классах

С введением закрепленных типов нуждается в расширении понятие базового класса типа.

Сначала классы и типы были для нас едины, и это их свойство - отправной пункт ОО-метода, - **по существу**, сохраняется, хотя нам пришлось немного расширить систему типов, добавляя в классы родовые параметры. Каждый тип основан на классе и для типа определено понятие базового класса. Для типов, порожденных универсальным классом с заданными фактическими родовыми параметрами, базовым классом является универсальный класс, в котором удалены фактические параметры. Так, например, для **LIST [INTEGER]** базовым классом является **LIST**. На классах основаны и развернутые типы; и для них аналогично: для **expanded SOME_CLASS [...]** базовый класс - **SOME_CLASS**.

Закрепление типов - это еще одно расширение системы типов, которое, подобно двум предыдущим, сохраняет свойство выводимости каждого типа непосредственно из класса. Базовым для **like anchor** является базовый класс типа сущности **anchor** в текущем классе. Если **anchor** есть **Current**, базовым будет класс, в котором это объявление содержится.

Правила о закрепленных типах

Теоретически ничто не мешает нам записать **like anchor** для самого элемента **anchor** как сущности закрепленного типа. Достаточно ввести правило, которое запрещало бы циклы в декларативных цепочках.

Вначале закрепленные опорные элементы (**anchored anchor**) были запрещены, но это новое, более либеральное правило придает системе типов большую гибкость.

Пусть **T** - тип **anchor** (текущий класс, если **anchor** есть **Current**). Тогда тип **like anchor** совместим как с самим собой, так и с **T**.

Обратное определение не симметрично: единственный тип, совместимый с **like anchor**, - это он сам. В частности, с ним не совместим тип **T**. Если бы следующий код был верен:

```
anchor, other: T; x: like anchor
...
create other
x := other -- предупреждение: ошибочное присваивание
```

то в порожденном классе, где `anchor` меняет свой тип на `U`, совместимый с `T`, но основанный на его потомке, сущности `x` был бы присвоен объект типа `T`, а не объект типа `U` или `U`-совместимого типа, что некорректно.

Будем говорить, что `x` опорно-эквивалентен `y`, если `x` есть `y` или имеет тип `like z`, где `z` по рекурсии опорно-эквивалентен `y`. Присваивания: `x := anchor`, `anchor := x`, как и присваивания опорно-эквивалентных (`anchor-equivalent`) элементов, конечно же, допустимы.

При закреплении формального параметра или результата, как в случае

```
r (other: like Current)
```

фактический параметр вызова, например, `b` в `a.r(b)`, должен быть опорно-эквивалентен `a`.

Когда не используются закрепленные объявления

Не всякое объявление вида `x: A` в классе `A` следует менять на `x: like Current` и не в каждой паре компонентов одного типа следует один из них делать опорным, а другой - закрепленным.

Закрепленное объявление - это своего рода обязательство изменения типа закрепленной сущности при смене типа опорного элемента. Как мы видели, оно не имеет обратной силы: объявив тип сущности как `like anchor`, вы теряете право на переопределение его в будущем (коль скоро новый тип должен быть совместим с исходным, а с закрепленным типом совместим только он сам). Пока не введено закрепление, остается свобода: если `x` типа `T`, то потомок может переопределить тип, введя более походящий тип `U`.

Достоинства и недостатки закрепления сущностей очевидны. Закрепление гарантирует, что вам не придется выполнять повторные объявления вслед за изменением типа опорного элемента, но оно раз и навсегда привязывает вас к типу опорного элемента. Это типичный случай "свободы выбора". (В каком-то смысле Фауст объявил себя `like Мефистофель`.)

Как пример нежелательного закрепления рассмотрим компонент `first_child` для деревьев, описывающий первого сына данного узла дерева. (При построении дерева он аналогичен компоненту `first_element` для списков, типом которого изначально является `CELL [G]` или `LINKABLE [G]`.) Для деревьев требуется повторное объявление. Может показаться, что уместным использовать закрепленное объявление:

```
first_child: like Current
```

Но на практике это накладывает слишком много ограничений. Класс дерева может иметь потомков, представляющих разные виды деревьев (их узлов): `UNARY_TREE` (узлы с одним сыном), `BINARY_TREE` (узлы с двумя сыновьями) и `BOUNDED_ARITY_TREE` (узлы с ограниченным числом сыновей). При закреплении `first_child` все сыновья каждого узла должны иметь один и тот же отцовский тип.

Это может быть нежелательным при построении более гибких структур, например бинарного узла с унарным потомком. Для этого компонент нужно описать без закрепления:

```
first_child: TREE [G]
```

Это решение не связано с какими-то ограничениями, и для создания деревьев с узлами одного типа вы, оставив класс `TREE` без изменений, можете породить от него `HOMOGENEOUS_TREE`, где переопределить `first_child` как

```
first_child: like Current
```

что гарантирует неизменность типов всех узлов дерева.

Статический механизм

Устранить последнее неясности в понимании закрепленного объявления поможет следующее замечание: это чисто статический механизм, не предполагающий никаких изменений объектов в период выполнения. Все ограничения могут быть проверены в период компиляции.

Закрепленное объявление можно считать синтаксическим приемом, позволяющим переложить переопределения на компилятор. Кроме того, оно является важнейшим инструментом достижения компромисса между повторным использованием и контролем типов.

Наследование и скрытие информации

Последний вопрос, оставшийся пока без ответа, как наследование взаимодействует с принципом Скрытия информации.

В отношениях между классом и его клиентами скрытие информации определяет разработчик класса. Именно он определяет политику в отношении каждого компонента класса: экспортируя его всем клиентам, разрешая выборочный экспорт, или делая компонент закрытым.

Кое-что о политике

Что происходит со статусом экспорта при передаче компонента потомку? Наследование и скрытие информации - ортогональные механизмы. Наследование определяет отношение между классом и его потомками, экспорт - между классом и его клиентами. Класс B может свободно экспортировать или скрывать любой из компонентов f, унаследованных им от класса A. При этом доступны все возможные комбинации:

- f экспортируется в классе A и в классе B (хотя и не обязательно одним и тем же клиентам);
- f скрыто в A и B;
- f скрыто в A, но полностью или частично экспортируется в B;
- f экспортируется в A, но скрыто в B.

Правило гласит: по умолчанию f сохраняет тот статус экспорта, которым компонент был наделен в A. Однако его можно изменить, добавив предложение **export** в предложение наследования класса. Например:

```
class B inherit
    A
export {NONE} f end
-- Скрыть f (возможно, экспортируемый в классе A)
...
```

или

```
class B inherit
    A
export {ANY} f end
-- Экспортировать f (возможно, скрытый в классе A)
...
```

или

```
class B inherit
    A
export {X, Y, Z} f end
-- Сделать f доступным определенным классам
...
```

Применение

Характерным примером является создание нескольких вариантов одной абстракции.

Представим себе GENERAL_ACCOUNT - класс, содержащий все необходимые операции для работы с банковскими счетами: процедуры open, withdraw, deposit, code (для снятия денег через банкомат), change_code и т.д., - но не предназначенный для использования клиентами напрямую, а потому не экспортирующий никаких подпрограмм. Его потомки выступают как разные облики родителя: они не содержат новых компонентов и отличаются лишь предложениями экспорта. Один экспортирует open и deposit, второй, наряду с ними, - withdraw и code, и т. д.

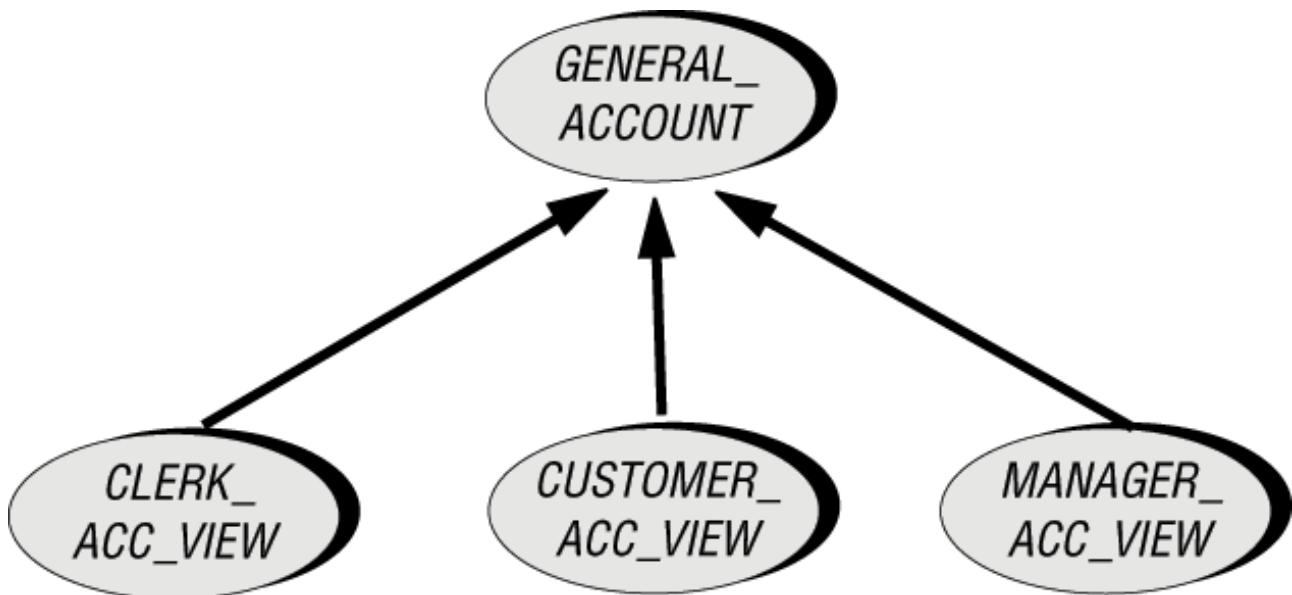


Рис. 16.12. Разные облики одной абстракции

Эта схема в обсуждении методологии наследования (см. [лекцию 6](#) курса "Основы объектно-ориентированного проектирования") носит название наследования функциональных возможностей (facility inheritance).

Понятие облика (view) является классическим в области баз данных, где необходимо дифференцировать пользователей, работающих с данными, предоставляя им разные права.

Другой пример касается классов, введенных, когда речь шла о множественном наследовании. Компонент `right` класса `CELL` скрыт в нем или, точнее говоря, экспортируется лишь классу `LIST`. Фактически так обстоят дела со всеми компонентами `CELL`, поскольку этот класс был изначально нацелен на работу со списками. Однако в дереве (классе `TREE`), помимо как `CELL`, так и `LIST`, `right` теперь означает доступ к правому брату и является респектабельным членом общества экспортруемых компонентов.

Зачем нужна такая гибкость?

Стратегия экспортации, согласно которой каждый потомок класса имеет свою политику, хотя и усложняет проверку типов, но придает необходимую гибкость действиям разработчика.

Предпринимались и иные попытки. Так, отдельные объектные языки определяют не только то, будет ли компонент экспортирован клиентам класса, но и то, будет ли он доступен его потомкам. Преимущества этого подхода неочевидны. В частности:

- мне не известно о публикации рекомендаций по применению этой возможности, неясно, когда компонент должен передаваться потомкам, а когда быть скрытым. Конструкции языка, за которыми нет ни малейшей теории, имеют весьма сомнительную ценность. (Для сравнения: правило, посвященное методологии скрытия информации, совершенно прозрачно: то, что принадлежит АТД, и надлежит экспортовать; прочее следует скрыть.)
- механизмы ограничения порожденных классов, введенные в языке Simula и др., редко используются разработчиками.

При близком рассмотрении отсутствие ясных методологических установок не удивляет. Наследование является воплощением принципа Открыт-Закрыт: позволяя выбрать готовый класс, написанный вами или другим программистом вчера или 20 лет назад, и обнаружить, что с ним можно делать нечто полезное, что даже не предполагалось при его проектировании.

Позволить создателю класса определять, что могут и что не могут использовать потомки класса, - значит лишиться основного свойства наследования.

Пример классов `CELL` и `TREE` характерен: при разработке `CELL` его целью была лишь поддержка работоспособности `LIST`, а потому `right` и `put_right` служили в нем исключительно внутренним целям. И лишь позднее этим компонентам нашли новое применение в классе-потомке `TREE`. Не будь этой открытости, наследование почти полностью утратило бы свой шарм.

Если нет основы для принятия решения об экспорте компонентов потомкам, то еще более абсурдно пытаться догадаться, что потомки могут экспортовать своим клиентам. Единственная задача разработчика порожденного класса - предоставление своим клиентам как можно более удобного для них класса. Наследование - это лишь средство, которое позволяет быстрее добиться желаемого результата. Все правила ОО-игры определяются утверждениями и ограничениями типизации, - не более того. Найти полезный для

клиентов потомка компонент предка - это большая удача, ну а то, как поступал предок с этим компонентом, - экспортировал ли он его, это дело предка и волнует потомка меньше всего.

В итоге, единственной стратегией, сочетающейся с принципиальной открытостью наследования, нам кажется та, что была описана выше: предоставить каждому разработчику возможность самостоятельно решать, что делать с компонентами предка, выбирая собственную политику экспорта в интересах своих клиентов.

Интерфейс и повторное использование реализаций

Знакомясь с объектным подходом по другим источникам, вы могли видеть в них предостережения использования "наследования реализаций". Однако в нем нет ничего плохого.

Повторное использование имеет две формы: использование интерфейсов и использование реализаций. Любой класс - это реализация (возможно, частичная) АТД. Он содержит как интерфейс, выражающий спецификацию АТД и образующий лишь "вершину айсберга", так и набор решений, определяющих реализацию. Повторное использование интерфейса означает согласие со спецификацией, повторное использование реализации - ваше согласие положиться на свойства класса, а не только на АТД.

Совместно для одних и тех же целей эти две возможности не применяются. Если вы хотите получить некоторое множество возможностей только через их абстрактные свойства и хотите быть защищенными от будущих изменений реализации, выбирайте повторное использование интерфейсов. Но в некоторых случаях вам может понравиться определенная реализация, поскольку она обеспечивает нужную основу вашего решения.

Эти формы повторного использования взаимно дополняют друг друга и обе совершенно законны.

По сути, их воплощением являются два вида межмодульных отношений, имеющих место при ОО-проектировании программ: клиент обеспечивает повторное использование интерфейсов, наследование поддерживает повторное использование реализаций.

Повторно используя реализацию, вы безусловно принимаете более ответственное решение, так как не можете рассчитывать на неизменность реализации в перспективе. По этой причине, став наследником класса, вы связываете себя более сильными обязательствами.

Слово в защиту реализаций

В чем же причина недоверия к наследованию реализаций? Я пришел к выводу, что ответ лежит в области психологии. Тридцатилетний программистский опыт оставил нам лишь сомнения насчет самой идеи реализаций. И даже слово "реализация" приобрело в отдельных кругах почти неприличный характер. По этой причине мы ведем речь о проектировании и анализе, а если и упоминаем реализацию, то начинаем разговор с "но", "лишь" или "только".

Объектная технология в корне меняет все: ОО-реализации настолько элегантны, полезны, с ясно выраженной корректностью, что уже можно забыть об неприятных оттенках этого слова в языке. Для многих из нас программа часто оказывается вещью наиболее абстрактной, дает описание на самом высоком уровне и наиболее понимаема, чем большая часть того, что в анализе и проектировании провозглашается "величайшим достижением мысли".

Два стиля

Ряд основных различий между понятиями, о которых шла речь, мы представили в виде таблицы.

Итак, есть два отношения - "быть потомком" и "быть клиентом"; две формы повторного использования - интерфейсов и реализаций; скрытие информации и его отсутствие; защита от изменений в поставляемых модулях и отсутствие таковой.

Наличие альтернатив в любом случае не вносит противоречий, и в зависимости от контекста каждый из вариантов вполне оправдан. Отважимся на смелый шаг и сведем эти противоположности в одно целое:

Таблица 16.1. Слияние четырех противоположностей

Клиент	Потомок
Повторное использование интерфейсов	Повторное использование реализаций
Информация скрывается	Информация не скрывается
Исходная реализация защищена	Исходная реализация не защищена

Возможно, есть и другие подходы к решению этой проблемы, но я не знаю ни одного столь же простого, доступного и практичного.

Выборочный экспорт

Говоря о наследовании и скрытии информации, нельзя обойти вопрос о выборочном экспорте компонентов. Класс A, выборочно экспортирующий f классу B:

```
class A feature {B, ...}
  f...
  ...
```

делает f доступным в реализации собственных компонентов B. Потомки B, в свою очередь, имеют доступ к реализации предка, а потому они должны быть вправе обращаться ко всем доступным B возможностям, в том числе, к f.

Практические наблюдения подтверждают этот теоретический обоснование. Все, что необходимо классу, обычно требуется и его потомкам. Однако нам не хотелось бы с появлением очередного порожденного класса B возвращаться в A и расширять его предложение экспорта.

Согласно принципу Скрытия информации, а также принципу Открыт-Закрыт, разработчику A дано право решать, делать ли f доступным для B, однако, ему запрещено ограничивать свободу разработчика B. Тем самым, имеет место правило:

Правило наследования при выборочном экспорте

Выборочно экспортенный компонент доступен как самому классу, так и всем его потомкам.

Ключевые концепции

- К инвариантам класса автоматически добавляются инварианты его родителей.
- В подходе Проектирования по Контракту наследование, переопределение и динамическое связывание приводят к идеи субподрядов.
- Повторное объявление подпрограммы (переопределение или создание реализации) может сохранить или ослабить предусловие, сохранить или усилить постусловие.
- Повторное объявление утверждений может использовать только **require else** (при объединении с предусловием связкой "или") и **ensure then** (при объединении с постусловием связкой "и"). Применение **require/ensure** запрещено. В отсутствие названных предложений подпрограмма сохраняет исходные утверждения.
- Универсальный класс GENERAL и допускающий настройку его наследник обеспечивают переопределяемые компоненты, представляющие общий интерес для всех создаваемых разработчиком классов. Класс NONE замыкает решетку наследования снизу.
- Заморозив компонент, можно гарантировать его вечную семантическую уникальность.
- Ограниченная универсальность дает возможность использовать только родовые параметры со специфическими свойствами.
- Попытка присваивания позволяет динамически проверить, принадлежит ли объект ожидаемому типу. Эта операция не должна использоваться как замена динамического связывания.
- Потомок вправе переопределять тип любой сущности (атрибута, результата функции, формального параметра подпрограммы). Повторное определение должно быть ковариантным - заменять исходные типы соответствующими, согласуясь с требованиями потомка.
- Закрепленные объявления (**like anchor**) - это важная часть системы типов, облегчающая применение ковариантной типизации и позволяющая отказаться от избыточных повторных объявлений.
- Наследование и скрытие информации - это независимые механизмы. Потомки могут скрывать экспортированные компоненты и экспорттировать скрытые компоненты.
- Компонент, доступный самому классу, доступен и его потомкам.

Библиографические замечания

Иную точку зрения на взаимосвязь наследования и скрытия информации см. в [Snyder 1986].

Упражнения

У16.1 Наследование: простота и эффективность

Перепишите и упростите ранее созданную реализацию защищенного стека, сделав класс STACK3 потомком, а не клиентом STACK, чтобы избежать излишних обходных путей. (**Подсказка:** см. правила взаимодействия наследования и скрытия информации.)

У16.2 Векторы

Напишите класс VECTOR, представляющий числовые вектора (кольцо) с обычными математическими операциями. Сам класс рекурсивно должен относиться к численному типу, допуская вектора векторов. Возможно, для этого вам придется самостоятельно дописать класс NUMERIC (или воспользоваться готовым из [М 1994а]).

У16.3 Экстракт?

В случае, когда x_1 имеет тип X, y_1 имеет тип Y, и Y является потомком X, оператор $y_1 := x_1$ будет недопустимым. Однако полезным мог бы показаться универсальный компонент extract, такой, что $y_1.extract(x_1)$ копирует значения полей объекта x_1 в соответствующие поля объекта y_1 при условии, что ни в одной из этих ссылок не содержится Void.

Объясните, почему компонент extract стоит отвергнуть. (**Подсказка:** обратитесь к вопросам корректности, в частности, к понятию инварианта.) Выясните, можно ли спроектировать удовлетворительный механизм, решающий эту задачу каким-то иным способом.

Основы объектно-ориентированного программирования

17. Лекция: Типизация

Эффективное применение объектной технологии требует четкого описания в тексте системы типов всех объектов, с которыми она работает на этапе выполнения. Это правило, известное как статическая типизация (static typing), делает наше ПО: более надежным, позволяя компилятору и другим инструментальным средствам устраниить несоответствия прежде, чем они смогут нанести вред; более понятным, обеспечивая точной информацией читателей: авторов клиентских систем и тех, кто будет сопровождать систему; более эффективным, поскольку информация о типах данных позволит компилятору генерировать оптимальный код. Хотя в вопросах типизации данных активно занимались и в нее объектной среды, да и сама статическая типизация применяется в языках, не поддерживающих ООП, особенно ярко эти идеи проявили себя именно при объектном подходе, во многом основанном на понятии типа, которое, слившись с понятием модуля, образует базовую ОО-конструкцию - класс.

Проблема типизации

О типизации при ОО-разработке можно сказать одно: эта задача **проста** в своей постановке, но решить ее подчас нелегко.

Базисная конструкция

Простота типизации в ОО-подходе есть следствие простоты объектной вычислительной модели. Опуская детали, можно сказать, что при выполнении ОО-системы происходят события только одного рода - вызов компонента (feature call):

`x.f (arg)`

означающий выполнение операции `f` над объектом, присоединенным к `x`, с передачей аргумента `arg` (возможно несколько аргументов или ни одного вообще). Программисты Smalltalk говорят в этом случае о "передаче объекту `x` сообщения `f` с аргументом `arg`", но это - лишь отличие в терминологии, а потому оно несущественно.

То, что все основано на этой Базисной Конструкции (Basic Construct), объясняет частично ощущение красоты ОО-идей.

Из Базисной Конструкции следуют и те ненормальные ситуации, которые могут возникнуть в процессе выполнения:

Определение: нарушение типа

Нарушение типа в период выполнения или, для краткости, просто нарушение типа (type violation) возникает в момент вызова `x.f (arg)`, где `x` присоединен к объекту ОВJ, если либо:

- не существует компонента, соответствующего `f` и применимого к ОВJ,
- такой компонент имеется, однако, аргумент `arg` для него недопустим.

Проблема типизации - избегать таких ситуаций:

Проблема типизации ОО-систем

Когда мы обнаруживаем, что при выполнении ОО-системы может произойти нарушение типа?

Ключевым является слово **когда**. Рано или поздно вы поймете, что имеет место нарушение типа. Например, попытка выполнить компонент "Пуск торпеды" для объекта "Служащий" не будет работать и при выполнении произойдет отказ. Однако возможно вы предпочитаете находить ошибки как можно раньше, а не позже.

Статическая и динамическая типизация

Хотя возможны и промежуточные варианты, здесь представлены два главных подхода:

- **Динамическая типизация**: ждать момента выполнения каждого вызова и тогда принимать решение.
- **Статическая типизация**: с учетом набора правил определить по исходному тексту, возможны ли нарушения типов при выполнении. Система выполняется, если правила гарантируют отсутствие ошибок.

Эти термины легко объяснимы: при динамической типизации проверка типов происходит во время работы системы (динамически), а при статической типизации проверка выполняется над текстом статически (до выполнения).

Термины типизированный и нетипизированный (typed/untyped) нередко используют вместо статически типизированный и динамически типизированный (statically/dynamically typed). Во избежание любых недоразумений мы будем придерживаться полных именований.

Статическая типизация предполагает автоматическую проверку, возлагаемую, как правило, на компилятор. В итоге имеем простое определение:

Определение: статически типизированный язык

ОО-язык статически типизирован, если он поставляется с набором согласованных правил, проверяемых компилятором, соблюдение которых гарантирует, что выполнение системы не приведет к нарушению типов.

В литературе встречается термин "сильная типизация" (**strong**). Он соответствует ультимативной природе определения, требующей полного отсутствия нарушения типов. Возможны и **слабые** (**weak**) формы статической типизации, при которых правила устраниют определенные нарушения, не ликвидируя их целиком. В этом смысле некоторые ОО-языки являются статически слабо типизированными. Мы будем бороться за наиболее сильную типизацию.

В динамически типизированных языках, известных как нетипизированные, отсутствуют объявления типов, а к сущностям в период выполнения могут присоединяться любые значения. Статическая проверка типов в них невозможна.

Правила типизации

Наша ОО-нотация является статически типизированной. Ее правила типов были введены в предыдущих лекциях и сводятся к трем простым требованиям.

- При объявлении каждой сущности или функции должен задаваться ее тип, например, `acc : ACCOUNT`. Каждая подпрограмма имеет 0 или более формальных аргументов, тип которых должен быть задан, например: `put (x : G; i : INTEGER)`.
- В любом присваивании `x := y` и при любом вызове подпрограммы, в котором `y` - это фактический аргумент для формального аргумента `x`, тип источника `y` должен быть совместим с типом цели `x`. Определение совместимости основано на наследовании: В совместим с `A`, если является его потомком, - дополненное правилами для родовых параметров ([лекцию 14](#)).
- Вызов `x.f (arg)` требует, чтобы `f` был компонентом базового класса для типа цели `x`, и `f` должен быть экспортирован классу, в котором появляется вызов (см. 14.3).

Реализм

Хотя определение статически типизированного языка дано совершенно точно, его недостаточно, - необходимы неформальные критерии при создании правил типизации. Рассмотрим два крайних случая.

- **Совершенно корректный язык**, в котором каждая синтаксически правильная система корректна и в отношении типов. Правила описания типов не нужны. Такие языки существуют (представьте себе польскую запись выражения со сложением и вычитанием целых чисел). К сожалению, ни один реальный универсальный язык не отвечает этому критерию.
- **Совершенно некорректный язык**, который легко создать, взяв любой существующий язык и добавив правило типизации, делающее **любую** систему некорректной. По определению, этот язык типизирован: так как нет систем, соответствующих правилам, то ни одна система не вызовет нарушения типов.

Можно сказать, что языки первого типа **пригодны**, но **бесполезны**, вторые, возможно, полезны, но не пригодны.

На практике необходима система типов, пригодная и полезная одновременно: достаточно мощная для реализации потребностей вычислений и достаточно удобная, не заставляющая нас идти на усложнения для удовлетворения правил типизации.

Будем говорить, что язык **реалистичен**, если он пригоден к применению и полезен на практике. В отличие от определения статической типизации, дающего безапелляционный ответ на вопрос: "**Типизирован ли язык X статически?**", определение реализма отчасти субъективно.

В этой лекции мы убедимся, что предлагаемая нами нотация реалистична.

Пессимизм

Статическая типизация приводит по своей природе к "пессимистической" политике. Попытка дать гарантию, что **все вычисления не приводят к отказам**, отвергает **вычисления, которые могли бы закончиться без ошибок**.

Рассмотрим обычный, необъектный, Pascal-подобный язык с различными типами `REAL` и `INTEGER`. При описании `n : INTEGER; r : REAL` оператор `n := r` будет отклонен, как нарушающий правила. Так, компилятор отвергнет все нижеследующие операторы:

<code>n := 0.0</code>	[A]
<code>n := 1.0</code>	[B]
<code>n := -3.67</code>	[C]
<code>n := 3.67 - 3.67</code>	[D]

Если мы разрешим их выполнение, то увидим, что [A] будет работать всегда, так как любая система счисления имеет точное представление вещественного числа 0,0, недвусмысленно переводимое в 0 целых. [B] почти наверняка также будет работать. Результат действия [C] не очевиден (хотим ли мы получить итог округлением или отбрасыванием дробной части?). [D] справится со своей задачей, как и оператор:

```
if n ^ 2 < 0 then n := 3.67 end [E]
```

куда входит недостижимое присваивание (`n ^ 2` - это квадрат числа `n`). После замены `n ^ 2` на `n` правильный результат даст только ряд запусков. Присваивание `n` большого вещественного значения, не представимого целым, приведет к отказу.

В типизированных языках все эти примеры (работающие, неработающие, иногда работающие) безжалостно трактуются как нарушения правил описания типов и отклоняются любым компилятором.

Вопрос не в том, **будем** ли мы пессимистами, а в том, **насколько** пессимистичными мы можем позволить себе быть. Вернемся к требованию реализма: если правила типов настолько пессимистичны, что препятствуют простоте записи вычислений, мы их отвергнем. Но если достижение безопасности типов достигается небольшой потерей выразительной силы, мы примем их. Например, в среде разработки, предоставляющей функции округления и выделения целой части - `round` и `truncate`, оператор `n := r` считается некорректным справедливо, поскольку заставляет вас явно записать преобразование вещественного числа в целое, вместо использования двусмысленных преобразований по умолчанию.

Статическая типизация: как и почему

Хотя преимущества статической типизации очевидны, неплохо поговорить о них еще раз.

Преимущества

Причины применения статической типизации в объектной технологии мы перечислили в начале лекции. Это надежность, простота понимания и эффективность.

Надежность обусловлена обнаружением ошибок, которые иначе могли проявить себя лишь во время работы, и только в некоторых случаях. Первое из правил, заставляющее объявлять сущности, как, впрочем, и функции, вносит в программный текст избыточность, что позволяет компилятору, используя два других правила, обнаруживать несоответствия между задуманным и реальным применением сущностей, компонентов и выражений.

Раннее выявление ошибок важно еще и потому, что чем дольше мы будем откладывать их поиск, тем сильнее вырастут издержки на исправление. Это свойство, интуитивно понятное всем программистам-профессионалам, количественно подтверждают широко известные работы Бема (Boehm). Зависимость издержек на исправление от времени отыскания ошибок приведена на графике, построенном по данным ряда больших промышленных проектов и проведенных экспериментов с небольшим управляемым проектом:



Рис. 17.1. Сравнительные издержки на исправление ошибок ([Boehm 1981], публикуется с разрешения)

Читабельность или **Простота понимания** (readability) имеет свои преимущества. Во всех примерах этой книги появление типа у сущности дает читателю информацию о ее назначении. Читабельность крайне важна на этапе сопровождения.

Исключив читабельность из круга приоритетов, можно было бы получить другие преимущества, не вводя явных объявлений. В самом деле, возможна неявная форма типизации, когда компилятор, не требуя явного указания типа, пытается автоматически определить его из контекста применения сущности. Эта стратегия известна как **выведение типов** (type inference). Но в программной инженерии явные объявления типов это помощь, а не наказание, - тип должен быть ясен не только машине, но и читающему текст человеку.

Наконец, **эффективность** может определять успех или отказ от объектной технологии на практике. В отсутствие статической типизации на выполнение `x.f(arg)` может уйти сколько угодно времени. Причина этого в том, что на этапе выполнения, не найдя `f` в базовом классе цели `x`, поиск будет продолжен у ее потомков, а это верная дорога к неэффективности. Снять остроту проблемы можно, улучшив поиск компонента по иерархии. Авторы языка Self провели большую работу, стремясь генерировать лучший код для языка с динамической типизацией. Но именно статическая типизация позволила такому ОО-продукту приблизиться или сравняться по эффективности с традиционным ПО.

Ключом к статической типизации является уже высказанная идея о том, что компилятор, генерирующий код для конструкции `x.f(arg)`, знает тип `x`. Из-за полиморфизма нет возможности однозначно определить подходящую версию компонента `f`. Но объявление сужает множество возможных типов, позволяя компилятору построить таблицу, обеспечивающую доступ к правильному `f` с минимальными издержками, - с ограниченной константой сложностью доступа. Дополнительно выполняемые оптимизации **статического связывания (static binding)** и **подстановки (inlining)** - также облегчаются благодаря статической типизации, полностью устранив издержки в тех случаях, когда они применимы.

Аргументы в пользу динамической типизации

Несмотря на все это, динамическая типизация не теряет своих приверженцев, в частности, среди Smalltalk-программистов. Их аргументы основаны прежде всего на реализме, речь о котором шла выше. Они уверены, что статическая типизация чрезсчур ограничивает их, не давая им свободно выражать свои творческие идеи, называя иногда ее "поясом целомудрия".

С такой аргументацией можно согласиться, но лишь для статически типизированных языков, не поддерживающих ряд возможностей. Стоит отметить, что все концепции, связанные с понятием типа и введенные в предыдущих лекциях, необходимы - отказ от любой из них чреват серьезными ограничениями, а их введение, напротив, придает нашим действиям гибкость, а нам самим дает возможность в полной мере насладиться практичностью статической типизации.

Типизация: слагаемые успеха

Каковы механизмы реалистичной статической типизации? Все они введены в предыдущих лекциях, а потому нам остается лишь кратко о них напомнить. Их совместное перечисление показывает согласованность и мощь их объединения.

Наша система типов полностью основана на понятии **класса**. Классами являются даже такие базовые типы, как `INTEGER`, а стало быть, нам не нужны особые правила описания предопределенных типов. (В этом наша нотация отличается от "гибридных" языков наподобие Object Pascal, Java и C++, где система типов старых языков сочетается с объектной технологией, основанной на классах.)

Развернутые типы дают нам больше гибкости, допуская типы, чьи значения обозначают объекты, наряду с типами, чьи значения обозначают ссылки.

Решающее слово в создании гибкой системы типов принадлежит **наследованию** и связанному с ним понятию **совместимости**. Тем самым преодолевается главное ограничение классических типизированных языков, к примеру, Pascal и Ada, в которых оператор `x := y` требует, чтобы тип `x` и `y` был одинаковым. Это правило слишком строго: оно запрещает использовать сущности, которые могут обозначать объекты взаимосвязанных типов (`SAVINGS_ACCOUNT` и `CHECKING_ACCOUNT`). При наследовании мы требуем лишь совместимости типа `y` с типом `x`, например, `x` имеет тип `ACCOUNT`, `y` - `SAVINGS_ACCOUNT`, и второй класс - наследник первого.

На практике статически типизированный язык нуждается в поддержке **множественного наследования**. Известны принципиальные обвинения статической типизации в том, что она не дает возможность по-разному интерпретировать объекты. Так, объект `DOCUMENT` (документ) может передаваться по сети, а потому нуждается в наличии компонентов, связанных с типом `MESSAGE` (сообщение). Но эта критика верна только для языков, ограниченных единственным наследованием.



Рис. 17.2. Множественное наследование

Универсальность необходима, например, для описания гибких, но безопасных контейнерных структур данных (например `class LIST [G] ...`). Не будь этого механизма, статическая типизация потребовала бы объявления разных классов для списков, отличающихся типом элементов.

В ряде случаев универсальность требуется **ограничить**, что позволяет использовать операции, применимые лишь к сущностям родового типа. Если родовой класс `SORTABLE_LIST` поддерживает сортировку, он требует от сущностей типа `G`, где `G` - родовой параметр, наличия операции сравнения. Это достигается связыванием с `G` класса, задающего родовое ограничение, - `COMPARABLE`:

```
class SORTABLE_LIST [G -> COMPARABLE] ...
```

Любой фактический родовой параметр `SORTABLE_LIST` должен быть потомком класса `COMPARABLE`, имеющего необходимый компонент.

Еще один обязательный механизм - **попытка присваивания** - организует доступ к тем объектам, типом которых ПО не управляет. Если у - это объект базы данных или объект, полученный через сеть, то оператор `x ?= y` присвоит `x` значение `y`, если `y` имеет совместимый тип, или, если это не так, даст `x` значение `Void`.

Утверждения, связанные, как часть идеи Проектирования по Контракту, с классами и их компонентами в форме предусловий, постусловий и инвариантов класса, дают возможность описывать семантические ограничения, которые не охватываются спецификацией типа. В таких языках, как Pascal и Ada, есть типы-диапазоны, способные ограничить значения сущности, к примеру, интервалом от 10 до 20, однако, применяя их, вам не удастся добиться того, чтобы значение `j` являлось отрицательным, всегда вдвое превышая `j`. На помощь приходят инварианты классов, призванные точно отражать вводимые ограничения, какими бы сложными они не были.

Закрепленные объявления нужны для того, чтобы на практике избегать лавинного дублирования кода. Объявляя `u`: `like x`, вы получаете гарантию того, что `u` будет меняться вслед за любыми повторными объявлениями типа `x` у потомка. В отсутствие этого механизма разработчики беспрестанно занимались бы повторными объявлениями, стремясь сохранить соответствие различных типов.

Закрепленные объявления - это особый случай последнего требуемого нам языкового механизма - **ковариантности**, подробное обсуждение которого нам предстоит позже.

При разработке программных систем на деле необходимо еще одно свойство, присущее самой среде разработки - **быстрая, возрастающая (fast incremental) перекомпиляция**. Когда вы пишите или модифицируете систему, хотелось бы как можно скорее увидеть эффект изменений. При статической типизации вы должны дать компилятору время на перепроверку типов. Традиционные подпрограммы компиляции требуют повторной трансляции всей системы (и ее **сборки**), и этот процесс может быть мучительно долгим, особенно с переходом к системам большого масштаба. Это явление стало аргументом в пользу **интерпретирующих** систем, таких как ранние среды Lisp или Smalltalk, запускавшие систему практически без обработки, не выполняя проверку типов. Сейчас этот аргумент позабыт. Хороший современный компилятор определяет, как изменился код с момента последней компиляции, и обрабатывает лишь найденные изменения.

"Типизирована ли кроха"?

Наша цель - **строгая** статическая типизация. Именно поэтому мы и должны избегать любых лазеек в нашей "игре по правилам", по крайней мере, точно их идентифицировать, если они существуют.

Самой распространенной лазейкой в статически типизированных языках является наличие преобразований, меняющих тип сущности. В C и производных от него языках их называют "приведением типа" или кастингом (cast). Запись (`OTHER_TYPE`) `x` указывает на то, что значение `x` воспринимается компилятором, как имеющее тип `OTHER_TYPE`, при соблюдении некоторых ограничениях на возможные типы.

Подобные механизмы обходят ограничения проверки типов. Приведение широко распространено при программировании на языке C, включая диалект ANSI C. Даже в языке C++ приведение типов, хотя и не столь частое, остается привычным и, возможно, необходимым делом.

Придерживаться правил статической типизации не так просто, если в любой момент их можно обойти путем приведения.

Далее будем полагать, что система типов является строгой и не допускает приведения типа.

Возможно, вы заметили, что попытка присваивания - неотъемлемый компонент реалистичной системы типов - напоминает приведение. Однако есть существенное отличие: попытка присваивания выполняет проверку, действительно ли текущий тип соответствует заданному типу, - это безопасно, а иногда и необходимо.

Типизация и связывание

Хотя как читатель этой книги вы наверняка отличите статическую типизацию от статического **связывания**, есть люди, которым подобное не под силу. Отчасти это может быть связано с влиянием языка Smalltalk, отстаивающего динамический подход к обеим задачам и способного сформировать неверное представление, будто они имеют одинаковое решение. (Мы же в своей книге утверждаем, что для создания надежных и гибких программ желательно объединить статическую типизацию и динамическое связывание.)

Как типизация, так и связывание имеют дело с семантикой Базисной Конструкции `x.f(arg)`, но отвечают на два разных вопроса:

Типизация и связывание

- **Вопрос о типизации:** когда мы должны точно знать, что во время выполнения появится операция, соответствующая `f`, применимая к объекту, присоединенному к сущности `x` (с параметром `arg`)?
- **Вопрос о связывании:** когда мы должны знать, какую операцию инициирует данный вызов?

Типизация отвечает на вопрос о наличии **как минимум одной** операции, связывание отвечает за выбор **нужной**.

В рамках объектного подхода:

- проблема, возникающая при типизации, связана с **полиморфизмом**: поскольку `x` **во время выполнения** может обозначать объекты нескольких различных типов, мы должны быть уверены, что операция, представляющая `f`,

- доступна в каждом из этих случаев;
- проблема связывания вызвана **повторными объявлениями**: так как класс может менять наследуемые компоненты, то могут найтись две или более операции, претендующие на то, чтобы представлять *f* в данном вызове.

Обе задачи могут быть решены как динамически, так и статически. В существующих языках представлены все четыре варианта решения.

- Ряд необъектных языков, скажем, Pascal и Ada, реализуют как статическую типизацию, так и статическое связывание. Каждая сущность представляет объекты только одного типа, заданного статически. Тем самым обеспечивается надежность решения, платой за которую является его гибкость.
- Smalltalk и другие ОО-языки содержат средства динамического связывания и динамической типизации. При этом предпочтение отдается гибкости в ущерб надежности языка.
- Отдельные необъектные языки поддерживают динамическую типизацию и статическое связывание. Среди них - языки ассемблера и ряд языков сценариев (scripting languages).
- Идеи статической типизации и динамического связывания воплощены в нотации, предложенной в этой книге.

Отметим своеобразие языка C++, поддерживающего статическую типизацию, хотя и не строгую ввиду наличия приведения типов, статическое связывание (по умолчанию), динамическое связывание при явном указании виртуальных (**virtual**) объявлений.

Причина выбора статической типизации и динамического связывания очевидна. Первый вопрос: "Когда мы будем знать о существовании компонентов?" - предполагает статический ответ: "**Чем раньше, тем лучше**", что означает: во время компиляции. Второй вопрос: "Какой из компонентов использовать?" предполагает динамический ответ: " **тот, который нужен**", - соответствующий динамическому типу объекта, определяемому во время выполнения. Это единственно приемлемое решение, если статическое и динамическое связывание дает различные результаты.

Следующий пример иерархии наследования поможет прояснить эти понятия:

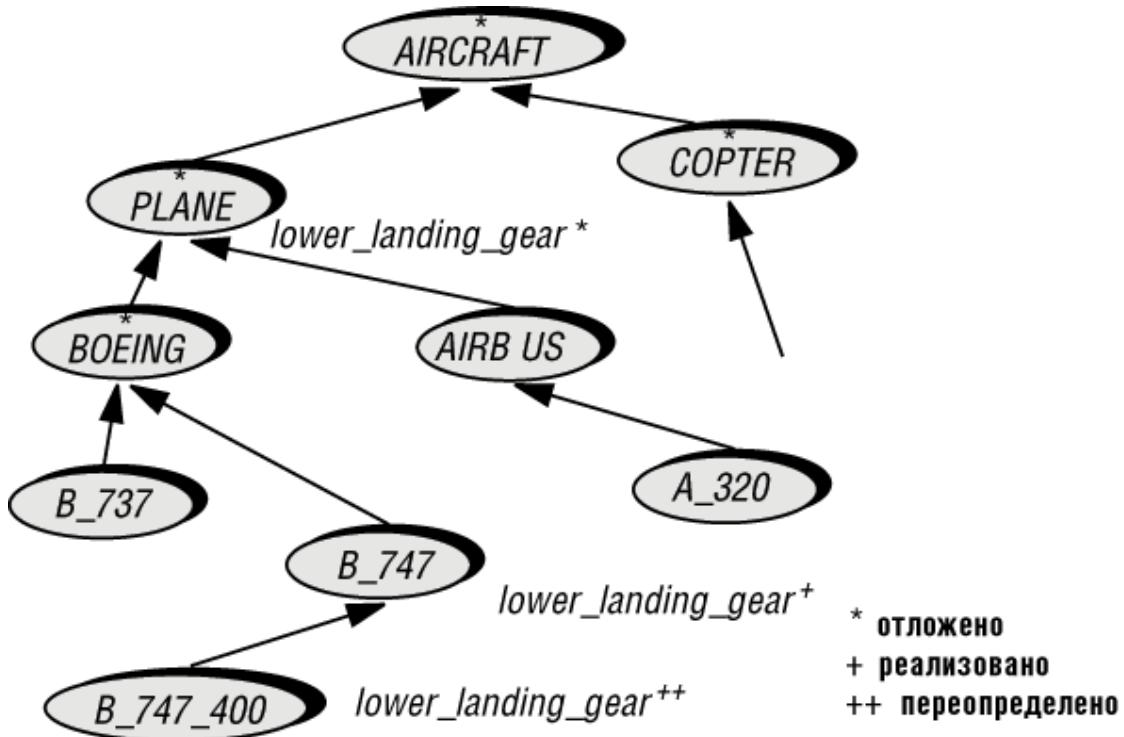


Рис. 17.3. Виды летательных аппаратов

Рассмотрим вызов:

`my_aircraft.lower_landing_gear`

Вопрос о типизации: когда убедиться, что здесь будет компонент `lower_landing_gear` ("выпустить шасси"), применимый к объекту (для COPTER его не будет вовсе) Вопрос о связывании: какую из нескольких возможных версий выбрать.

Статическое связывание означало бы, что мы игнорируем тип присоединяемого объекта и полагаемся на объявление сущности. В итоге, имея дело с Boeing 747-400, мы вызвали бы версию, разработанную для обычных лайнеров серии 747, а не для их модификации 747-400. Динамическое связывание применяет операцию, требуемую объектом, и это правильный подход.

При статической типизации компилятор не отклонит вызов, если можно гарантировать, что при выполнении программы к сущности `my_aircraft` будет присоединен объект, поставляемый с компонентом, соответствующим `lower_landing_gear`. Базисная техника получения гарантий проста: при обязательном объявлении `my_aircraft` требуется, чтобы базовый класс его типа включал такой компонент. Поэтому `my_aircraft` не может быть объявлен как

AIRCRAFT, так как последний не имеет lower_landing_gear на этом уровне; вертолеты, по крайней мере в нашем примере, выпускать шасси не умеют. Если же мы объявим сущность как PLANE, - класс, содержащий требуемый компонент, - все будет в порядке.

Динамическая типизация в стиле SmallTalk требует дождаться вызова, и в момент его выполнения проверить наличие нужного компонента. Такое поведение возможно для прототипов и экспериментальных разработок, но недопустимо для промышленных систем - в момент полета поздно спрашивать, есть ли у вас шасси.

Ковариантность и скрытие потомком

Если бы мир был прост, то разговор о типизации можно было бы и закончить. Мы определили цели и преимущества статической типизации, изучили ограничения, которым должны соответствовать реалистичные системы типов, и убедились в том, что предложенные методы типизации отвечают нашим критериям.

Но мир не прост. Объединение статической типизации с некоторыми требованиями программной инженерии создает проблемы более сложные, чем это кажется с первого взгляда. Проблемы вызывают два механизма: **ковариантность (covariance)** - смена типов параметров при переопределении, **скрытие потомком (descendant hiding)** - способность класса потомка ограничивать статус экспорта наследуемых компонентов.

Ковариантность

Что происходит с аргументами компонента при переопределении его типа? Это важнейшая проблема, и мы уже видели ряд примеров ее проявления: устройства и принтеры, одно- и двухсвязные списки и т. д. (см. разделы 16.6, 16.7).

Вот еще один пример, помогающий уяснить природу проблемы. И пусть он далек от реальности и метафоричен, но его близость к программным схемам очевидна. К тому же, разбирая его, мы будем часто возвращаться к задачам из практики.

Представим себе готовящуюся к чемпионату лыжную команду университета. Класс GIRL включает лыжниц, выступающих в составе женской сборной, BOY - лыжников. Ряд участников обеих команд ранжированы, показав хорошие результаты на предыдущих соревнованиях. Это важно для них, поскольку теперь они побегут первыми, получив преимущество перед остальными. (Это правило, дающее привилегии уже привилегированным, возможно и делает слалом и лыжные гонки столь привлекательными в глазах многих людей, являясь хорошей метафорой самой жизни.) Итак, мы имеем два новых класса: RANKED_GIRL и RANKED_BOY.

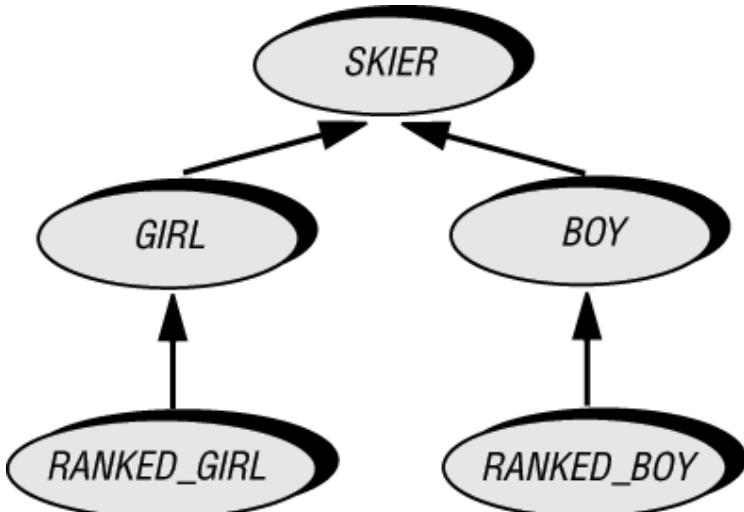


Рис. 17.4. Классификация лыжников

Для проживания спортсменов забронирован ряд номеров: только для мужчин, только для девушек, только для девушек-призеров. Для отображения этого используем параллельную иерархию классов: ROOM, GIRL_ROOM и RANKED_GIRL_ROOM.

Вот набросок класса SKIER:

```

class SKIER feature
  roommate: SKIER
    -- Сосед по номеру.
    share (other: SKIER) is
      -- Выбрать в качестве соседа other.
      require
        other /= Void
      do
        roommate := other
      end
    ...
  ...
end
  
```

... другие возможные компоненты, опущенные в этом и последующих классах ...

Нас интересуют два компонента: атрибут `roommate` и процедура `share`, "размещающая" данного лыжника в одном номере с текущим лыжником:

```
s1, s2: SKIER
...
s1.share (s2)
```

При объявлении сущности `other` можно отказаться от типа `SKIER` в пользу закрепленного типа `like roommate` (или `like Current` для `roommate` и `other` одновременно). Но давайте забудем на время о закреплении типов (мы к ним еще вернемся) и посмотрим на проблему ковариантности в ее изначальном виде.

Как ввести переопределение типов? Правила требуют раздельного проживания юношей и девушек, призеров и остальных участников. Для решения этой задачи при переопределении изменим тип компонента `roommate`, как показано ниже (здесь и далее переопределенные элементы подчеркнуты).

```
class GIRL inherit
    SKIER
        redefine roommate end
feature
    roommate: GIRL
        -- Сосед по номеру.
end
```

Переопределим, соответственно, и аргумент процедуры `share`. Более полный вариант класса теперь выглядит так:

```
class GIRL inherit
    SKIER
        redefine roommate, share end
feature
    roommate: GIRL
        -- Сосед по номеру.
    share (other: GIRL) is
        -- Выбрать в качестве соседа other.
        require
            other /= Void
        do
            roommate := other
        end
end
```

Аналогично следует изменить все порожденные от `SKIER` классы (закрепление типов мы сейчас не используем). В итоге имеем иерархию:

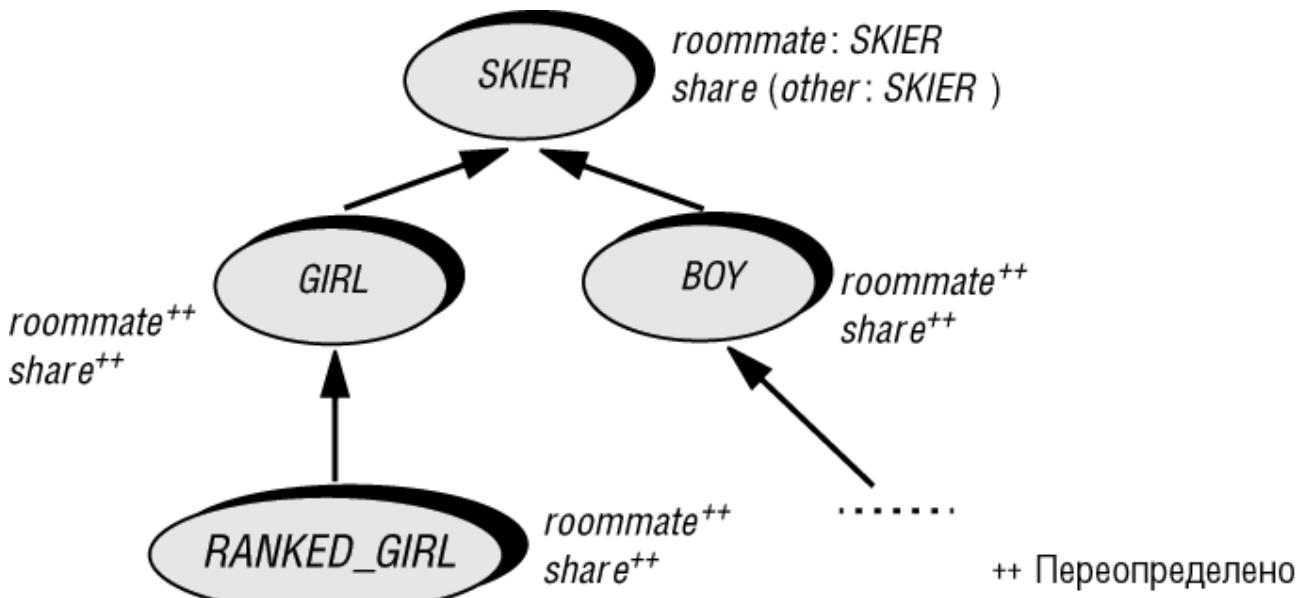


Рис. 17.5. Иерархия участников и повторные определения

Так как наследование является специализацией, то правила типов требуют, чтобы при переопределении результата компонента, в данном случае `roommate`, новый тип был потомком исходного. То же касается и переопределения типа аргумента `other` подпрограммы `share`. Эта стратегия, как мы знаем, именуется ковариантностью, где приставка "ко" указывает на совместное изменение типов параметра и результата. Противоположная стратегия называется

контравариантностью.

Все наши примеры убедительно свидетельствуют о практической необходимости ковариантности.

- Элемент односвязного списка LINKABLE должен быть связан с другим подобным себе элементом, а экземпляр BI_LINKABLE - с подобным себе. Ковариантно потребуется переопределяется и аргумент в put_right.
- Всякая подпрограмма в составе LINKED_LIST с аргументом типа LINKABLE при переходе к TWO_WAY_LIST потребует аргумента BI_LINKABLE.
- Процедура set_alternate принимает DEVICE-аргумент в классе DEVICE и PRINTER-аргумент - в классе PRINTER.

Ковариантное переопределение получило особое распространение потому, что скрытие информации ведет к созданию процедур вида

```
set_attrib (v: SOME_TYPE) is
    -- Установить attrib в v.
    ...
```

для работы с attrib типа SOME_TYPE. Подобные процедуры, естественно, ковариантны, поскольку любой класс, который меняет тип атрибута, должен соответственно переопределять и аргумент set_attrib. Хотя представленные примеры укладываются в одну схему, но ковариантность распространена значительно шире. Подумайте, например, о процедуре или функции, выполняющей конкатенацию односвязных списков (LINKED_LIST). Ее аргумент должен быть переопределен как двусвязный список (TWO_WAY_LIST). Универсальная операция сложения infix "+" принимает NUMERIC-аргумент в классе NUMERIC, REAL - в классе REAL и INTEGER - в классе INTEGER. В параллельных иерархиях телефонной службы процедуре start в классе PHONE_SERVICE может требоваться аргумент ADDRESS, представляющий адрес абонента, (для выписки счета), в то время как этой же процедуре в классе CORPORATE_SERVICE потребуется аргумент типа CORPORATE_ADDRESS.

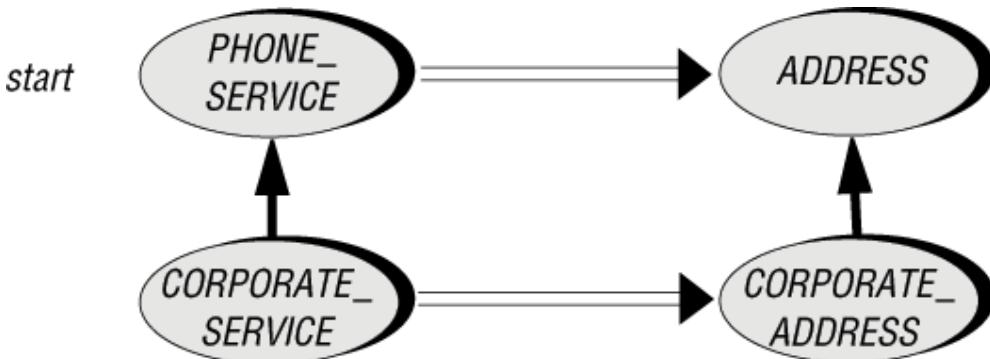


Рис. 17.6. Службы связи

Что можно сказать о контравариантном решении? В примере с лыжниками оно означало бы, что если, переходя к классу RANKED_GIRL, тип результата roommate переопределили как RANKED_GIRL, то в силу контравариантности тип аргумента share можно переопределить на тип GIRL или SKIER. Единственный тип, который не допустим при контравариантном решении, - это RANKED_GIRL! Достаточно, чтобы возбудить наихудшие подозрения у родителей девушки.

Параллельные иерархии

Чтобы не оставить камня на камне, рассмотрим вариант примера SKIER с двумя параллельными иерархиями. Это позволит нам смоделировать ситуацию, уже встречавшуюся на практике: TWO_WAY_LIST > LINKED_LIST и BI_LINKABLE > LINKABLE; или иерархию с телефонной службой PHONE_SERVICE.

Пусть есть иерархия с классом ROOM, потомком которого является GIRL_ROOM (класс BOY опущен):

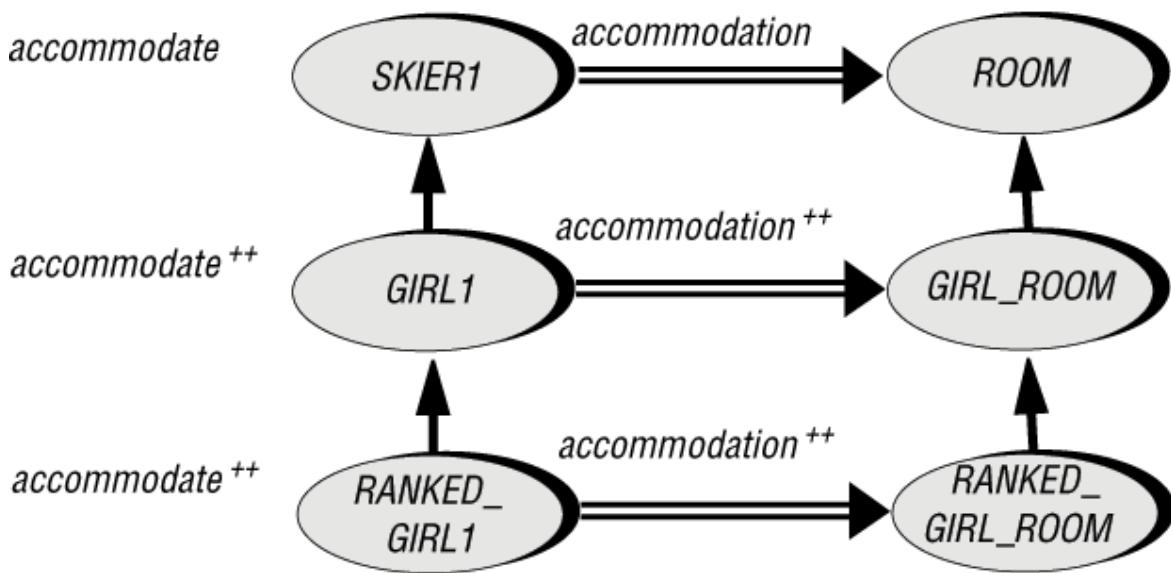


Рис. 17.7. Лыжники и комнаты

Наши классы лыжников в этой параллельной иерархии вместо roommate и share будут иметь аналогичные компоненты accommodation (**размещение**) и accommodate (**разместить**):

```

indexing
    description: "Новый вариант с параллельными иерархиями"
class SKIER1 feature
    accommodation: ROOM
    accommodate (r: ROOM) is ... require ... do
        accommodation:= r
    end
end

```

Здесь также необходимы ковариантные переопределения: в классе GIRL1 как accommodation, так и аргумент подпрограммы accommodate должны быть заменены типом GIRL_ROOM, в классе BOY1 - типом BOY_ROOM и т.д. (Не забудьте: мы по-прежнему работаем без закрепления типов.) Как и в предыдущем варианте примера, контравариантность здесь бесполезна.

Своенравие полиморфизма

Не довольно ли примеров, подтверждающих практичность ковариации? Почему же кто-то рассматривает контравариантность, которая вступает в противоречие с тем, что необходимо на практике (если не принимать во внимание поведения некоторых молодых людей)? Чтобы понять это, рассмотрим проблемы, возникающие при сочетании полиморфизма и стратегии ковариантности. Придумать вредительскую схему несложно, и, возможно, вы уже создали ее сами:

```

s: SKIER; b: BOY; g: GIRL
...
create b; create g;-- Создание объектов BOY и GIRL.
s := b; -- Полиморфное присваивание.
s.share (g)

```

Результат последнего вызова, вполне возможно приятный для юношей, - это именно то, что мы пытались не допустить с помощью переопределения типов. Вызов share ведет к тому, что объект BOY, известный как b и благодаря полиморфизму получивший псевдоним s типа SKIER, становится соседом объекта GIRL, известного под именем g. Однако вызов, хотя и противоречит правилам общежития, является вполне корректным в программном тексте, поскольку share -экспортируемый компонент в составе SKIER, а GIRL, тип аргумента g, совместим со SKIER, типом формального параметра share.

Схема с параллельной иерархией столь же проста: заменим SKIER на SKIER1, вызов share - на вызов s.accommodate (gr), где gr - сущность типа GIRL_ROOM. Результат - тот же.

При контравариантном решении этих проблем не возникло бы: специализация цели вызова (в нашем примере s) требовала бы обобщения аргумента. Контравариантность в результате ведет к более простой математической модели механизма: наследование - переопределение - полиморфизм. Данный факт описан в ряде теоретических статей,лагающих эту стратегию. Аргументация не слишком убедительна, поскольку, как показывают наши примеры и другие публикации, контравариантность не имеет практического использования.

В литературе для программистов нередко встречается призыв к методам, основанных на простых математических моделях. Однако математическая красота - всего лишь один из критериев ценности результата, - есть и другие - полезность и реалистичность.

Поэтому, не пытаясь натянуть контравариантную одежду на ковариантное тело, следует принять ковариантную действительность и искать пути устранения нежелательного эффекта.

Скрытие потомком

Прежде чем искать решение проблемы ковариантности, рассмотрим еще один механизм, способный в условиях полиморфизма привести к нарушениям типа. Скрытие потомком (descendant hiding) - это способность класса не экспортировать компонент, полученный от родителей.

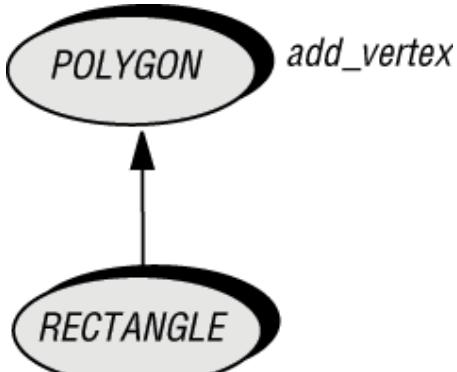


Рис. 17.8. Скрытие потомком

Типичным примером является компонент add_vertex (добавить вершину), экспортруемый классом POLYGON, но скрываемый его потомком RECTANGLE (ввиду возможного нарушения инварианта - класс хочет оставаться прямоугольником):

```
class RECTANGLE inherit
    POLYGON
        export {NONE} add_vertex end
feature
    ...
invariant
    vertex_count = 4
end
```

Не программистский пример: класс "Страус" скрывает метод "Летать", полученный от родителя "Птица".

Давайте на минуту примем эту схему такой, как она есть, и поставим вопрос, будет ли легитимным сочетание наследования и скрытия. Моделирующая роль скрытия, подобно ковариантности, нарушается из-за трюков, возможных из-за полиморфизма. И здесь не трудно построить вредоносный пример, позволяющий, несмотря на скрытие компонента, вызвать его и добавить прямоугольнику вершину:

```
p: POLYGON; r: RECTANGLE
...
create r;           -- Создание объекта RECTANGLE.
p := r;            -- Полиморфное присваивание.
p.add_vertex (...)
```

Так как объект r скрывается под сущностью p класса POLYGON, а add_vertex экспортруемый компонент POLYGON, то его вызов сущностью p корректен. В результате выполнения в прямоугольнике появится еще одна вершина, а значит, будет создан недопустимый объект.

Корректность систем и классов

Для обсуждения проблем ковариантности и скрытия потомком нам понадобится несколько новых терминов. Будем называть **классово-корректной (class-valid)** систему, удовлетворяющую трем правилам описания типов, приведенным в начале лекции. Напомним их: каждая сущность имеет свой тип; тип фактического аргумента должен быть совместимым с типом формального, аналогичная ситуация с присваиванием; вызываемый компонент должен быть объявлен в своем классе и экспортован классу, содержащему вызов.

Система называется **системно-корректной (system-valid)**, если при ее выполнении не происходит нарушения типов.

В идеале оба понятия должны совпадать. Однако мы уже видели, что классово-корректная система в условиях наследования, ковариантности и скрытия потомком может не быть системно-корректной. Назовем такую ошибку **нарушением системной корректности (system validity error)**.

Практический аспект

Простота проблемы создает своеобразный парадокс: пытливый новичок построит контрпример за считанные минуты, в реальной практике изо дня в день возникают ошибки классовой корректности систем, но нарушения системной корректности даже в больших, многолетних проектах возникают исключительно редко.

Однако это не позволяет игнорировать их, а потому мы приступаем к изучению трех возможных путей решения данной проблемы.

Далее мы будем затрагивать весьма тонкие и не столь часто дающие о себе знать аспекты объектного подхода. Читая книгу впервые, вы можете пропустить оставшиеся разделы этой лекции. Если вы лишь недавно занялись вопросами ОО-технологии, то лучше усвоите этот материал после изучения лекций 1-11 курса "Основы объектно-ориентированного проектирования", посвященной методологии наследования, и в особенности [лекции 6](#) курса "Основы объектно-ориентированного проектирования", посвященной методологии наследования.

Корректность систем: первое приближение

Давайте сконцентрируемся вначале на проблеме ковариантности, более важной из двух рассматриваемых. Этой теме посвящена обширная литература, предлагающая ряд разнообразных решений.

Контравариантность и безвариантность

Контравариантность устраниет теоретические проблемы, связанные с нарушением системной корректности. Однако при этом теряется реалистичность системы типов, по этой причине рассматривать этот подход в дальнейшем нет никакой необходимости.

Оригинальность языка C++ в том, что он использует стратегию **безвариантности (n ovariance)**, не позволяя менять тип аргументов в переопределяемых подпрограммах! Если бы язык C++ был строго типизированным языком, его системной типов было бы трудно пользоваться. Простейшее решение проблемы в этом языке, как и обход иных ограничений C++ (скажем, отсутствия ограниченной универсальности), состоит в использовании кастинга - приведения типа, что позволяет полностью игнорировать имеющийся механизм типизации. Это решение не кажется привлекательным. Заметим, однако, что ряд предложений, обсуждаемых ниже, будет опираться на безвариантность, смысл которой придаст введение новых механизмов работы с типами взамен ковариантного переопределения.

Использование родовых параметров

Универсальность лежит в основе интересной идеи, впервые высказанной Францем Вебером (Franz Weber). Объявим класс SKIER1, ограничив универсализацию родового параметра классом ROOM:

```
class SKIER1 [G -> ROOM] feature
    accommodation: G
        accommodate (r: G) is ... require ... do accommodation := r end
end
```

Тогда класс GIRL1 будет наследником SKIER1 [GIRL_ROOM] и т. д. Тем же приемом, каким бы странным он не казался на первый взгляд, можно воспользоваться и при отсутствии параллельной иерархии: **class SKIER [G -> SKIER]**.

Этот подход позволяет решить проблему ковариантности. При любом использовании класса необходимо задать фактический родовой параметр ROOM или GIRL_ROOM, так что неверная комбинация просто становится невозможной. Язык становится безвариантным, а система полностью отвечает потребностям ковариантности благодаря родовым параметрам.

К сожалению, эта техника неприемлема как общее решение, поскольку ведет к разрастанию списка родовых параметров, по одному на каждый тип возможного ковариантного аргумента. Хуже того, добавление ковариантной подпрограммы с аргументом, тип которого отсутствует в списке, потребует добавления родового параметра класса, а, следовательно, изменит интерфейс класса, повлечет изменения у всех клиентов класса, что недопустимо.

Типовые переменные

Ряд авторов, среди которых Ким Брюс (Kim Bruce), Дэвид Шэнг (David Shang) и Тони Саймонс (Tony Simons), предложили решение на основе типовых переменных (type variables), значениями которых являются типы. Их идея проста:

- взамен ковариантных переопределений разрешить объявление типов, использующее типовые переменные;
- расширить правила совместимости типов для управления такими переменными;
- считать язык (в остальном) безвариантным;
- обеспечить возможность присваивания типовым переменным в качестве значений типы языка.

Подробное изложение этих идей читатели могут найти в ряде статей по данной тематике, а также в публикациях Карделли (Cardelli), Кастаньи (Castagna), Вебера (Weber) и др. Начать изучение вопроса можно с источников, указанных в библиографических заметках к этой лекции. Мы же не будем заниматься этой проблемой, и вот почему.

- Надлежащее реализованный механизм типовых переменных относится к категории, позволяющей использовать тип без полной его спецификации. Эта же категория включает универсальность и закрепление объявлений. Этот механизм мог бы заменить другие механизмы этой категории. Вначале это можно истолковать в пользу типовых

переменных, но результат может оказаться плачевным, так как не ясно, сможет ли этот всеобъемлющий механизм справиться со всеми задачами с той легкостью и простотой, которая присуща универсальности и закреплению типов.

- Предположим, что разработан механизм типовых переменных, способный преодолеть проблемы объединения ковариантности и полиморфизма (все еще игнорируя проблему скрытия потомком). Тогда от разработчика классов потребуется **незаурядная интуиция** для того, чтобы заранее решить, какие из компонентов будут доступны для переопределения типов в порожденных классах, а какие - нет. Ниже мы обсудим эту проблему, имеющую место в практике создания программ и, увы, ставящую под сомнение применимость многих теоретических схем.

Это заставляет нас вернуться к уже рассмотренным механизмам: ограниченной и неограниченной универсальности, закреплению типов и, конечно, наследованию.

Полагаясь на закрепление типов

Почти готовое решение проблемы ковариантности мы найдем, присмотревшись к известному нам механизму закрепленных объявлений.

При описании классов SKIER и SKIER1 вас не могло не посетить желание, воспользовавшись закрепленными объявлениями, избавиться от многих переопределений. Закрепление - это типичный ковариантный механизм. Вот как будет выглядеть наш пример (все изменения подчеркнуты):

```
class SKIER feature
    roommate: like Current
    share (other: like Current) is ... require ... do
        roommate := other
    end
    ...
end
class SKIER1 feature
    accommodation: ROOM
    accommodate (r: like accommodation) is ... require ... do
        accommodation := r
    end
end
```

Теперь потомки могут оставить класс SKIER без изменений, а в SKIER1 им понадобится переопределить только атрибут accommodation. Закрепленные сущности: атрибут roommate и аргументы подпрограмм share и accommodate - будут изменяться автоматически. Это значительно упрощает работу и подтверждает тот факт, что при отсутствии закрепления (или другого подобного механизма, например, типовых переменных) написать ОО-программный продукт с реалистичной типизацией невозможно.

Но удалось ли устраниТЬ нарушения корректности системы? Нет! Мы, как и раньше, можем перехитрить проверку типов, выполнив полиморфные присваивания, вызывающие нарушения системной корректности.

Правда, исходные варианты примеров будут отклонены. Пусть:

```
s: SKIER; b: BOY; g: GIRL
...
create b; create g;-- Создание объектов BOY и GIRL.
s := b;-- Полиморфное присваивание.
s1 share (g)
```

Аргумент g, передаваемый share, теперь неверен, так как здесь требуется объект типа **like s**, а класс GIRL не совместим с этим типом, поскольку по правилу закрепленных типов ни один тип не совместим с **like s**, кроме него самого.

Впрочем, радоваться нам не долго. В другую сторону это правило говорит о том, что **like s** совместим с типом s. А значит, используя полиморфизм не только объекта s, но и параметра g, мы можем снова обойти систему проверки типов:

```
s: SKIER; b: BOY; g: like s; actual_g: GIRL;
...
create b; create actual_g-- Создание объектов BOY и GIRL.
s := actual_g; g := s-- Через s присоединить g к GIRL.
s := b-- Полиморфное присваивание.
s.share (g)
```

В результате незаконный вызов проходит.

Выход из положения есть. Если мы всерьез готовы использовать закрепление объявлений как единственный механизм ковариантности, то избавиться от нарушений системной корректности можно, полностью запретив полиморфизм закрепленных сущностей. Это потребует изменения в языке: введем новое ключевое слово **anchor** (эта гипотетическая конструкция нужна нам исключительно для того, чтобы использовать ее в данном обсуждении):

anchor s: SKIER

Разрешим объявления вида `like s` лишь тогда, когда `s` описано как `anchor`. Изменим правила совместимости так, чтобы гарантировать: `s` и элементы типа `like s` могут присоединяться (в присваиваниях или передаче аргумента) только друг к другу.

В исходном варианте правила существовало понятие **опорно-эквивалентных** элементов. При новом подходе опорно-эквивалентными должны быть как правая, так и левая часть любого присваивания, в котором участвует опорная или закрепленная сущность.

При таком подходе мы устранием из языка возможность переопределения типа любых аргументов подпрограммы. Помимо этого, мы могли запретить переопределять тип результата, но в этом нет необходимости. Возможность переопределения типа атрибутов, конечно же, сохраняется. **Все** переопределения типов аргументов теперь будут выполняться неявно через механизм закрепления, инициируемый ковариантностью. Там, где при прежнем подходе класс D переопределял наследуемый компонент как:

```
r (u: Y) ...
```

тогда как у класса C - родителя D это выглядело

```
r (u: X) ...
```

где Y соответствовало X, то теперь переопределение компонента r будет выглядеть так:

```
r (u: like your_anchor) ...
```

Остается только в классе D переопределить тип `your_anchor`.

Это решение проблемы ковариантности - полиморфизма будем называть подходом **Закрепления (Anchoring)**. Более аккуратно следовало бы говорить: "Ковариация только через Закрепление". Свойства подхода привлекательны:

- Закрепление основано на идеи строгого разделения **ковариантных** и потенциально полиморфных (или, для краткости, полиморфных) элементов. Все сущности, объявленные как `anchor` или `like some_anchor` ковариантны; прочие-полиморфны. В каждой из двух категорий допустимы любые присоединения, но нет сущности или выражения, нарушающих границу. Нельзя, например, присвоить полиморфный источник ковариантной цели.
- Это простое и элегантное решение нетрудно объяснить даже начинающим.
- Оно полностью устраняет возможность нарушения системной корректности в ковариантно построенных системах.
- Оно сохраняет заложенную выше концептуальную основу, в том числе понятия ограниченной и неограниченной универсальности. (В итоге это решение, по-моему, предпочтительнее типовых переменных, подменяющих собой механизмы ковариантности и универсальности, предназначенных для решения разных практических задач.)
- Оно требует незначительного изменения языка, - добавляя одно ключевое слово, отраженное в правиле соответствия, - и не связано с ощутимыми трудностями в реализации.
- Оно реалистично (по крайней мере, теоретически): любую ранее возможную систему можно переписать, заменив ковариантные переопределения закрепленными повторными объявлениями. Правда, некоторые присоединения в результате станут неверными, но они соответствуют случаям, которые могут привести к нарушениям типов, а потому их следует заменить попытками присваивания и разобраться в ситуации во время выполнения.

Казалось бы, дискуссию можно на этом закончить. Так почему же подход Закрепления не полностью нас устраивает? Прежде всего, мы еще не касались проблемы скрытия потомком. Кроме этого, основной причиной продолжения дискуссии является проблема, уже высказанная при кратком упоминании типовых переменных. Раздел сфер влияния на полиморфную и ковариантную часть, чем-то похож на результат Ялтинской конференции. Он предполагает, что разработчик класса обладает незаурядной интуицией, что он в состоянии для каждой введенной им сущности, в частности для каждого аргумента раз и навсегда выбрать одну из двух возможностей:

- Сущность является потенциально полиморфной: сейчас или позднее она (посредством передачи параметров или путем присваивания) может быть присоединена к объекту, чей тип отличается от объявленного. Исходный тип сущности не сможет изменить ни один потомок класса.
- Сущность является субъектом переопределения типов, то есть она либо закреплена, либо сама является опорным элементом.

Но как разработчик может все это предвидеть? Вся привлекательность ОО-метода во многом выраженная в принципе Открыт-Закрыт как раз и связана с возможностью изменений, которые мы вправе внести в ранее сделанную работу, а также с тем, что разработчик универсальных решений **не** должен обладать бесконечной мудростью, понимая, как его продукт смогут адаптировать к своим нуждам потомки.

При таком подходе переопределение типов и скрытие потомком - своего рода "предохранительный клапан", дающий возможность повторно использовать существующий класс, почти пригодный для достижения наших целей:

- Прибегнув к переопределению типов, мы можем менять объявления в порожденном классе, не затрагивая

оригинал. При этом чисто ковариантное решение потребует правки оригинала путем описанных преобразований.

- Скрытие потомком защита от многих неудач при создании класса. Можно критиковать проект, в котором RECTANGLE, используя тот факт, что он является потомком POLYGON, пытается добавить вершину. Взамен можно было бы предложить структуру наследования, в которой фигуры с фиксированным числом вершин отделены от всех прочих, и проблемы не возникало бы. Однако при разработке структур наследования предпочтительнее всегда те, в которых нет **таксономических исключений**. Но можно ли их полностью устраниć? Обсуждая ограничение экспорта в одной из следующих лекций, мы увидим, что подобное невозможно по двум причинам. Во-первых, это наличие конкурирующих критериев классификации. Во-вторых, вероятность того, что разработчик не найдет идеального решения, даже если оно существует.

Желая сохранить гибкость адаптации порожденных классов для наших нужд, мы должны разрешить и ковариантное переопределение типов, и скрытие потомком. Далее мы узнаем, как этого добиться.

Глобальный анализ

Этот раздел посвящен описанию промежуточного подхода. Основные практические решения изложены в [лекции 17](#).

Изучая вариант с закреплением, мы заметили, что его основной идеей было разделение ковариантного и полиморфного наборов сущностей. Так, если взять две инструкции вида

```
s := b ...
s.share (g)
```

каждая из них служит примером правильного применения важных ОО-механизмов: первая - полиморфизма, вторая - переопределения типов. Проблемы начинаются при объединении их для одной и той же сущности s. Аналогично:

```
p := r ...
p.add_vertex (...)
```

проблемы начинаются с объединения двух независимых и совершенно невинных операторов.

Ошибочные вызовы ведут к нарушению типов. В первом примере полиморфное присваивание присоединяет объект BOY к сущности s, что делает g недопустимым аргументом share, так как она связана с объектом GIRL. Во втором примере к сущности r присоединяется объект RECTANGLE, что исключает add_vertex из числа экспортруемых компонентов.

Вот и идея нового решения: заранее - статически, при проверке типов компилятором или иными инструментальными средствами - определим **набор типов (typeset)** каждой сущности, включающий типы объектов, с которыми сущность может быть связана в период выполнения. Затем, опять же статически, мы убедимся в том, что каждый вызов является правильным для каждого элемента из наборов типов цели и аргументов.

В наших примерах оператор s := b указывает на то, что класс BOY принадлежит набору типов для s (поскольку в результате выполнения инструкции создания **create b** он принадлежит набору типов для b). GIRL, ввиду наличия инструкции **create g**, принадлежит набору типов для g. Но тогда вызов share будет недопустим для цели s типа BOY и аргумента g типа GIRL. Аналогично RECTANGLE находится в наборе типов для p, что обусловлено полиморфным присваиванием, однако, вызов add_vertex для p типа RECTANGLE окажется недопустимым.

Эти наблюдения наводят нас на мысль о создании **глобального** подхода на основе нового правила типизации:

Правило системной корректности

Вызов x.f (arg) является системно-корректным, если и только если он классово-корректен для x, и arg, имеющих любые типы из своих соответствующих наборов типов.

В этом определении вызов считается классово-корректным, если он не нарушает правила Вызыва Компонентов, которое гласит: если C есть базовый класс типа x, компонент f должен экспортirоваться C, а тип arg должен быть совместим с типом формального параметра f. (Вспомните: для простоты мы полагаем, что каждый подпрограмма имеет только один параметр, однако, не составляет труда расширить действие правила на произвольное число аргументов.)

Системная корректность вызова сводится к классовой корректности за тем исключением, что она проверяется не для отдельных элементов, а для любых пар из наборов множеств. Вот основные правила создания набора типов для каждой сущности:

1. Для каждой сущности начальный набор типов пуст.
2. Встретив очередную инструкцию вида **create {SOME_TYPE} a**, добавим SOME_TYPE в набор типов для a. (Для простоты будем полагать, что любая инструкция **create a** будет заменена инструкцией **create {ATYPE} a**, где ATYPE - тип сущности a.)
3. Встретив очередное присваивание вида a := b, добавим в набор типов для a все элементы набора типов для b.
4. Если a есть формальный параметр подпрограммы, то, встретив очередной вызов с фактическим параметром b, добавим в набор типов для a все элементы набора типов для b.
5. Будем повторять шаги (3) и (4) до тех пор, пока наборы типов не перестанут изменяться.

Данная формулировка не учитывает механизма универсальности, однако расширить правило нужным образом можно без особых проблем. Шаг (5) необходим ввиду возможности цепочек присваивания и передач (от b к a, от c к b и т. д.).

Нетрудно понять, что через конечное число шагов этот процесс прекратится.

Число шагов ограничено длиной максимальной цепочки присоединений; другими словами максимум равен n , если система содержит присоединения от x_{i+1} к x_i для $i=1, 2, \dots, n-1$. Повторение шагов (3) и (4) известно как метод "неподвижной точки".

Как вы, возможно, заметили, правило не учитывает последовательности инструкций. В случае

```
create {TYPE1} t; s := t; create {TYPE2} t
```

в набор типов для s войдет как TYPE1, так и TYPE2, хотя s , учитывая последовательность инструкций, способен принимать значения только первого типа. Учет расположения инструкций потребует от компилятора глубокого анализа потока команд, что приведет к чрезмерному повышению уровня сложности алгоритма. Вместо этого применяются более пессимистичные правила: последовательность операций:

```
create b  
s := b  
s.share (g)
```

будет объявлена системно-некорректной, несмотря на то, что последовательность их выполнения не приводит к нарушению типа.

Глобальный анализ системы был (более детально) представлен в 22-й главе монографии [М 1992]. При этом была решена как проблема ковариантности, так и проблема ограничений экспорта при наследовании. Однако в этом подходе есть досадный практический недочет, а именно: предполагается проверка **системы в целом**, а не каждого класса в отдельности. Убийственным оказывается правило (4), которое при вызове библиотечной подпрограммы будет учитывать все ее возможные вызовы в других классах.

Хотя затем были предложены алгоритмы работы с отдельными классами в [М 1989b], их практическую ценность установить не удалось. Это означало, что в среде программирования, поддерживающей возрастающую компиляцию, необходимо будет организовать проверку всей системы. Желательно проверку вводить как элемент (быстрой) локальной обработки изменений, внесенных пользователем в некоторые классы. Хотя примеры применения глобального подхода известны, - так, программисты на языке C используют инструмент **lint** для поиска несоответствий в системе, не обнаруживаемых компилятором, - все это выглядит не слишком привлекательно.

В итоге, как мне известно, проверка системной корректности осталась никем не реализованной. (Другой причиной такого исхода, возможно, послужила сложность самих правил проверки.)

Классовая корректность предполагает проверку, ограниченную классом, и, следовательно, возможна при возрастающей компиляции. Системная корректность предполагает глобальную проверку всей системы, что входит в противоречие с возрастающей компиляцией.

Однако, несмотря на свое имя, фактически можно проверить системную корректность, используя только возрастающую проверку классов (в процессе работы обычного компилятора). Это и будет финальным вкладом в решение проблемы.

Остерегайтесь полиморфных кэтколлов!

Правило Системной Корректности пессимистично: в целях упрощения оно отвергает и вполне безопасные комбинации инструкций. Как ни парадоксально, но последний вариант решения мы построим на основе **еще более пессимистического правила**. Естественно, это поднимет вопрос о том, насколько реалистичным будет наш результат.

Назад, в Ялту

Суть решения **Кэтколл (Catcall)**, - смысл этого понятия мы поясним позднее, - в возвращении к духу Ялтинских соглашений, разделяющих мир на полиморфный и ковариантный (и спутник ковариантности - скрытие потомков), но без необходимости обладания бесконечной мудростью.

Как и прежде, сузим вопрос о ковариантности до двух операций. В нашем главном примере это полиморфное присваивание: $s := b$, и вызов ковариантной подпрограммы: $s.share (g)$. Анализируя, кто же является истинным виновником нарушений, исключим аргумент g из числа подозреваемых. Любой аргумент, имеющий тип SKIER или порожденный от него, нам не подходит ввиду полиморфизма s и ковариантности $share$. А потому если статически описать сущность $other$ как SKIER и динамически присоединить к объекту SKIER, то вызов $s.share (other)$ статически создаст впечатление идеального варианта, но приведет к нарушению типов, если полиморфно присвоить s значение b .

Фундаментальная проблема в том, что мы пытаемся использовать s двумя несовместимыми способами: как полиморфную сущность и как цель вызова ковариантной подпрограммы. (В другом нашем примере проблема состоит в использовании r как полиморфной сущности и как цели вызова подпрограммы потомка, скрывающего компонент add_vertex .)

Решение Кэтколл, как и Закрепление, носит радикальный характер: оно запрещает использовать сущность как полиморфную и ковариантную одновременно. Подобно глобальному анализу, оно статически определяет, какие

сущности могут быть полиморфными, однако, не пытается быть слишком умным, отыскивая для сущностей наборы возможных типов. Вместо этого всякая полиморфная сущность воспринимается как достаточно подозрительная, и ей запрещается вступать в союз с кругом почтенных лиц, включающих ковариантность и скрытие потомком.

Одно правило и несколько определений

Правило типов для решения Кэтколл имеет простую формулировку:

Правило типов для Кэтколл

Полиморфные кэтколлы некорректны.

В его основе - столь же простые определения. Прежде всего, полиморфная сущность:

Определение: полиморфная сущность

Сущность x ссылочного (не развернутого) типа полиморфна, если она обладает одним из следующих свойств:

1. Встречается в присваивании $x := y$, где сущность y имеет иной тип или по рекурсии полиморфна.
2. Встречается в инструкциях создания `create {OTHER_TYPE} x`, где `OTHER_TYPE` не является типом, указанным в объявлении x .
3. Является формальным аргументом подпрограммы.
4. Является внешней функцией.

Цель этого определения - придать статус полиморфной ("потенциально полиморфной") любой сущности, которую при выполнении программы можно присоединить к объектам разных типов. Это определение применимо лишь к ссылочным типам, так как развернутые сущности по природе не могут быть полиморфными.

В наших примерах лыжник s и многоугольник r - полиморфны по правилу (1). Первому из них присваивается объект `BOY b`, второму - объект `RECTANGLE r`.

Если вы познакомились с формулировкой понятия набора типов, то заметили, насколько пессимистичнее выглядит определение полиморфной сущности, и насколько проще его проверить. Не пытаясь отыскать все всевозможные динамические типы сущности, мы довольствуемся общим вопросом: может данная сущность быть полиморфной или нет? Наиболее удивительным выглядит правило (3), по которому **полиморфным** считается **каждый формальный параметр** (если его тип не расширен, как в случае с целыми и т. д.). Мы даже не утруждаем себя анализом вызовов. Если у подпрограммы есть аргумент, то он находится в полном распоряжении клиента, а значит, и полагаться на указанный в объявлении тип нельзя. Это правило тесно связано с повторным использованием - целью объектной технологии, - где любой класс потенциально может быть включен в состав библиотеки, и будет многократно вызываться различными клиентами.

Характерным свойством этого правила является то, что оно не требует никаких глобальных проверок. Для выявления полиморфности сущности достаточно просмотреть текст самого класса. Если для всех запросов (атрибутов или функций) сохранять информацию об их статусе полиморфности, то не приходится изучать даже тексты предков. В отличие от отыскания наборов типов, можно обнаружить полиморфные сущности, проверяя класс за классом в процессе возрастающей компиляции.

Как было сказано при обсуждении наследования, подобный анализ может также представлять ценность при оптимизации кода.

Вызовы, как и сущности, могут быть полиморфными:

Определение: полиморфный вызов

Вызов является полиморфным, если его цель полиморфна.

Оба вызова в наших примерах полиморфны: `s.share(g)` ввиду полиморфизма s , `r.add_vertex(...)` ввиду полиморфизма r . Согласно определению, только квалифицированные вызовы могут быть полиморфны. (Придав неквалифицированному вызову `f(...)` вид квалифицированного `Current.f(...)`, мы не меняем суть дела, поскольку `Current`, присвоить которому ничего нельзя, не является полиморфным объектом.)

Далее нам потребуется понятие Кэтколла, основанное на понятии САТ. (САТ - это аббревиатура Changing Availability or Type - изменение доступности или типа). Подпрограмма является САТ подпрограммой, если некоторое ее переопределение потомком приводит к изменениям одного из двух видов, которые, как мы видели, являются потенциально опасными: изменяет тип аргумента (ковариантно) или скрывает ранее экспортировавшийся компонент.

Определение: САТ-подпрограммы

Подпрограмма называется САТ-подпрограммой, если некоторое ее переопределение изменяет статус экспорта или тип любого из ее аргументов.

Это свойство опять-таки допускает возрастающую проверку: любое переопределение типа аргумента или статуса экспорта делают процедуру или функцию САТ-подпрограммой. Отсюда следует понятие Кэтколла: вызова САТ-подпрограммы, который может оказаться ошибочным.

Определение: Кэтколл

Вызов называется Кэтколлом, если некоторое переопределение подпрограммы сделало бы его ошибочным из-за изменения статуса экспорта или типа аргумента.

Созданная нами классификация позволяет выделять специальные группы вызовов: полиморфные и кэтколлы. Полиморфные вызовы придают выразительную мощь объектному подходу, кэтколлы позволяют переопределять типы и ограничивать экспорт. Используя терминологию, введенную ранее в этой лекции, можно сказать, что полиморфные вызовы расширяют **полезность (usefulness)**, кэтколлы - **используемость (usability)**.

Вызовы `share` и `add_vertex`, рассмотренные в наших примерах, являются кэт-коллами. Первый осуществляет ковариантное переопределение своего аргумента. Второй экспортируется классом `RECTANGLE`, но скрыт классом `POLYGON`. Оба вызова также и полиморфны, а потому они служат прекрасным примером полиморфных кэтколлов. Они являются ошибочными согласно правилу типов Кэтколл.

Оценка

Прежде чем мы сведем воедино все, что узнали о ковариантности и скрытии потомком, вспомним еще раз о том, что нарушения корректности систем возникают действительно редко. Наиболее важные свойства статической ОО-типовизации были обобщены в начале лекции. Этот впечатляющий ряд механизмов работы с типами совместно с проверкой классовой корректности, открывает дорогу к безопасному и гибкому методу конструирования ПО.

Мы видели три решения проблемы ковариантности, два из которых затронули и вопросы ограничения экспорта. Какое же из них правильное?

На этот вопрос нет окончательного ответа. Следствия коварного взаимодействия ОО-типовизации и полиморфизма изучены не так хорошо, как вопросы, изложенные в предыдущих лекциях. В последние годы появились многочисленные публикации по этой теме, ссылки на которые приведены в библиографии в конце лекции. Кроме того, я надеюсь, что в настоящей лекции мне удалось представить элементы окончательного решения или хотя бы к нему приблизиться.

Глобальный анализ кажется непрактичным из-за полной проверки всей системы. Тем не менее, он помог нам лучше понять проблему.

Решение на основе Закрепления чрезвычайно привлекательно. Оно простое, интуитивно понятное, удобное в реализации. Тем сильнее мы должны сожалеть о невозможности поддержки в нем ряда ключевых требований ОО-метода, отраженных в принципе Открыт-Закрыт. Если бы мы и впрямь обладали прекрасной интуицией, то закрепление стало бы великолепным решением, но какой разработчик решится утверждать это, или, тем более, признать, что такой интуицией обладали авторы библиотечных классов, наследуемых в его проекте?

Это предположение сужает сферу применения многих опубликованных методов, в том числе, основанных на типовых переменных. Если бы мы были уверены в том, что разработчик всегда заранее знает о будущих изменениях типов, задача бы упростилась в теоретическом плане, но из-за ошибочности гипотезы она не имеет практической ценности.

Если от закрепления мы вынуждены отказаться, то наиболее подходящим кажется Кэтколл-решение, достаточно легко объяснимое и применимое на практике. Его пессимизм не должен исключать полезные комбинации операторов. В случае, когда полиморфный кэтколл порожден "легитимным" оператором, всегда можно безопасно допустить его, введением попытки присваивания. Тем самым ряд проверок можно перенести на время выполнения программы. Однако количество таких случаев должно быть предельно мало.

В качестве пояснения я должен заметить, что на момент написания книги решение Кэтколл не было реализовано. До тех пор, пока компилятор не будет адаптирован к проверке правила типов Кэтколл и не будет успешно применен к репрезентативным системам - большим и малым, - рано говорить, что в проблеме примирения статической типизации с полиморфизмом, сочетаемым с ковариантностью и скрытием потомком, сказано последнее слово.

Полное соответствие

Завершая обсуждение ковариантности, полезно понять, как общий метод можно применить к решению достаточно общей проблемы. Метод появился как результат Кэтколл-теории, но может использоваться в рамках базисного варианта языка без введения новых правил.

Пусть существуют два согласованных списка, где первый задает лыжников, а второй - соседа по комнате для лыжника из первого списка. Мы хотим выполнять соответствующую процедуру размещения `share`, только если она разрешена правилами описания типов, которые разрешают поселять девушек с девушками, девушек-призеров с девушками-призерами и так далее. Проблемы такого вида встречаются часто.

Возможно простое решение, основанное на предыдущем обсуждении и попытке присваивания. Рассмотрим универсальную функцию `fitted` (согласовать):

```
fitted (other: GENERAL): like other is
-- Текущий объект (Current), если его тип соответствует типу объекта,
-- присоединенного к other, иначе void.
do
    if other /= Void and then conforms_to (other) then
        Result ?= Current
```

```

        end
    end

```

Функция `fitted` возвращает текущий объект, но известный как сущность типа, присоединенного к аргументу. Если тип текущего объекта не соответствует типу объекта, присоединенного к аргументу, то возвращается `Void`. Обратите внимание на роль попытки присваивания. Функция использует компонент `conforms_to` из класса `GENERAL`, выясняющий совместимость типов пары объектов.

Замена `conforms_to` на другой компонент `GENERAL` с именем `same_type` дает нам функцию `perfect_fitted` (**полное соответствие**), которая возвращает `Void`, если типы обоих объектов не идентичны.

Функция `fitted` - дает нам простое решение проблемы соответствия лыжников без нарушения правил описания типов. Так, в код класса `SKIER` мы можем ввести новую процедуру и использовать ее вместо `share`, (последнюю можно сделать скрытой процедурой).

```

safe_share (other: SKIER) is
    -- Выбрать, если допустимо, other как соседа по номеру.
    -- gender_ascertained - установленный пол
local
    gender_ascertained_other: like Current
do
    gender_ascertained_other := other .fitted (Current)
    if gender_ascertained_other /= Void then
        share (gender_ascertained_other)
    else
        "Вывод: совместное размещение с other невозможно"
    end
end

```

Для `other` произвольного типа `SKIER` (а не только `like Current`) определим версию `gender_ascertained_other`, имеющую тип, закрепленный за `Current`. Гарантировать идентичность типов нам поможет функция `perfect_fitted`.

При наличии двух параллельных списков лыжников, представляющих планируемое размещение:

```
occupant1, occupant2: LIST [SKIER]
```

можно организовать цикл, выполняя на каждом шаге вызов:

```
occupant1.item.safe_share (occupant2.item)
```

сопоставляющий элементы списков, если и только если их типы полностью совместимы.

Ключевые концепции

- Статическая типизация - залог надежности, читабельности и эффективности.
- Чтобы быть реалистичной, статической типизации требуется совместное применение механизмов: утверждений, множественного наследования, попытки присваивания, ограниченной и неограниченной универсальности, закрепленных объявлений. Система типов не должна допускать ловушек (приведений типа).
- Практические правила повторного объявления должны допускать ковариантное переопределение. Типы результатов и аргументов при переопределении должны быть совместимыми с исходными.
- Ковариантность, также как и возможность скрытия потомком компонента, экспортированного предком, в сочетании с полиморфизмом порождают редко встречающуюся, но весьма серьезную проблему нарушения типов.
- Этих нарушений можно избежать, используя: глобальный анализ, (что непрактично) ограничивая ковариантность закрепленными типами (что противоречит принципу "Открыт-Закрыт"), решение Кэтколл, препятствующее вызову полиморфной целью подпрограммы с ковариантностью или скрытием потомком.

Библиографические замечания

Ряд материалов этой лекции представлен в докладах на форумах OOPSLA 95 и TOOLS PACIFIC 95, а также опубликован в [M 1996a]. Ряд обзорных материалов заимствован из статьи [M 1989e].

Понятие автоматического выведения типов введено в [Milner 1989], где описан алгоритм выведения типов функционального языка ML. Связь между полиморфизмом и проверкой типов была исследована в работе [Cardelli 1984a].

Приемы повышения эффективности кода динамически типизированных языков в контексте языка Self можно найти в [Ungar 1992].

Теоретическую статью, посвященную типам в языках программирования и оказавшую большое влияние на специалистов, написали Лука Карделли (Luca Cardelli) и Петер Вегнер (Peter Wegner) [Cardelli 1985]. Эта работа, построенная на базе лямбда-исчисления (см. [M 1990]), послужила основой многих дальнейших изысканий. Ей предшествовала другая фундаментальная статья Карделли [Cardelli 1984].

Руководство по ISE включает введение в проблемы совместного применения полиморфизма, ковариантности и скрытия потомком [M 1988a]. Отсутствие надлежащего анализа в первом издании этой книги послужило причиной ряда критических дискуссий (первыми из которых стали комментарии Филиппа Элинка (Philippe Elinck) в бакалаврской работе "De la Conception-Programmation par Objets", Memoire de licence, Universite Libre de Bruxelles (Belgium), 1988), высказанных в работах [Cook 1989] и [America 1989a]. В статье Кука приведены несколько примеров, связанных с проблемой ковариантности, и предпринята попытка ее решения. Решение на основе типовых параметров для ковариантных сущностей на TOOLS EUROPE 1992 предложил Франц Вебер [Weber 1992]. Точные определения понятий системной корректности, а также классовой корректности, даны в [M 1992], там же предложено решение с применением полного анализа системы. Решение Кэтколл впервые предложено в [M 1996a]; см. также [M-Web].

Решение Закрепления было представлено в моем докладе на семинаре TOOLS EUROPE 1994. Тогда я, однако, не усмотрел необходимости в **anchor**-объявлениях и связанных с этим ограничениях совместимости. Пол Дюбуа (Paul Dubois) и Амирам Йехудай (Amiram Yehudai) не преминули заметить, что в этих условиях проблема ковариантности остается. Они, а также Рейнхардт Будде (Reinhardt Budde), Карл-Хайнц Зилла (Karl-Heinz Sylla), Ким Вальден (Kim Walden) и Джеймс Мак-Ким (James McKim) высказали множество замечаний, имевших принципиальное значение в той работе, которая привела к написанию этой лекции.

Вопросам ковариантности посвящено большое количество литературы. В [Castagna 1995] и [Castagna 1996] вы найдете как обширную библиографию, так и обзор математических аспектов проблемы. Перечень ссылок на онлайновые материалы по теории типов в ООП и Web-страницы их авторов см. на странице Лорана Дами (Laurent Dami) [Dami-Web]. Понятия ковариантности и контравариантности заимствованы из теории категорий. Их появлением в контексте программной типизации мы обязаны Луке Карделли, который начал использовать их в своих выступлениях с начала 80-х гг., но до конца 80-х не прибегал к ним в печати.

Приемы на основе типовых переменных описаны в [Simons 1995], [Shang 1996], [Bruce 1997].

Контравариантность была реализована в языке Sather. Пояснения даны в [Szypersky 1993].

Основы объектно-ориентированного программирования

18. Лекция: Глобальные объекты и константы

Локальных знаний не достаточно - компонентам ПО необходима глобальная информация: разделяемые данные, общее окно для вывода ошибок, шлюз для подключения к базе данных или сети. В классическом подходе достаточно объявить такой объект глобальной переменной главной программы. В ОО-системах нет ни главной программы, ни глобальных переменных. Но разделяемые (shared) объекты по-прежнему нужны. Глобальные объекты - некий вызов ОО-методу, провозглашающему идеи децентрализации, модульности и автономности. Борьба шла за независимость модулей, за избавление от произвола центральной власти. Теперь этой власти нет. Как же построить систему, в которой компоненты совместно используют данные, не теряя своей автономности, гибкости, допускают повторное использование? Передавать модулю разделяемые объекты как параметры не разумно, поскольку число их может быть достаточно велико. Да и сама передача параметров предполагает существование владельца, хотя при подлинном разделении владеть значениями не может ни один модуль. Поиск более удачного решения мы начнем с хорошо известного понятия, необходимого как в объектной, так и в традиционной методологии проектирования. Речь пойдет о константах. Что такое константа Pi, как не простой, совместно используемый объект? Обобщив это понятие на более сложные объекты, мы сделаем первый шаг на пути к разделению объектов.

Константы базовых типов

Начнем с формы записи констант.

Правило стиля - **принцип символических констант** - гласит, что обращение к конкретному значению (числу, символу или строке) **почти** всегда должно быть косвенным. Должно существовать **определение константы**, задающее имя, играющее роль символьической константы (**symbolic constant**), и связанное с ним значение - константа, называемая манифестной (**manifest constant**). Далее в алгоритме следует использовать символьическую константу. Тому есть два объяснения.

- Читабельность: читающему текст легче понять смысл US_states_count, чем числа 50;
- Расширяемость: символьическую константу легко обновить, исправив лишь ее определение.

Принцип допускает применение манифестных или, как часто говорят, неименованных констант в качестве "начальных" элементов разнообразных операций, как в случае с циклом **from i = 1 until i > n** (Но n, конечно, должно быть символьической константой).

Итак, нам нужен простой и ясный способ определения символьических констант.

Атрибуты-константы

Как и все сущности, символьические константы должны быть определены внутри класса. Будем рассматривать константы как атрибуты с фиксированным значением, одинаковым для всех экземпляров класса.

Синтаксически вновь используем служебное слово **is**, применяемое при описании методов, только здесь за ним будет следовать не алгоритм, а значение нужного типа. Вот примеры определения констант базовых типов INTEGER, BOOLEAN, REAL и CHARACTER:

```
Zero: INTEGER is 0
Ok: BOOLEAN is True
Pi: REAL is 3.1415926524
Backslash: CHARACTER is '\'
```

Как видно из этих примеров, имена атрибутов-констант рекомендуется начинать с заглавной буквы, за которой следуют только строчные символы.

Потомки не могут переопределять значения атрибутов-констант.

Как и другие атрибуты, класс может экспортить константы или скрывать. Так, если C - класс, экспортирующий выше объявленные константы, а у клиента класса к сущности x присоединен объект типа C, то выражение x.Backslash обозначает символ '\'.

В отличие от атрибутов-переменных, константы не занимают в памяти места. Их введение не связано с издержками в период выполнения, а потому не страшно, если их в классе достаточно много.

Использование констант

Вот пример, показывающий, как клиент может применять константы, определенные в классе:

```
class FILE feature
    error_code: INTEGER;           -- Атрибут-переменная
```

```

Ok: INTEGER is 0
Open_error: INTEGER is 1
...
open (file_name: STRING) is
    -- Открыть файл с именем file_name
    -- и связать его с текущим файловым объектом
do
    error_code := Ok
...
    if "Что-то не так" then
        error_code := Open_error
    end
end
... Прочие компоненты ...
end

```

Клиент можем вызвать метод open и проверить успешность операции:

```

f: FILE; ...
f.open
if f.error_code = f.Open_error then
    "Принять меры"
else
    ...
end

```

Нередко нужны и наборы констант, не связанных с конкретным объектом. Их, как и раньше, можно объединить в класс, выступающий в роли родителя всех классов, которым необходимы константы. В этом случае можно не создавать экземпляр класса:

```

class EDITOR_CONSTANTS
feature
    Insert: CHARACTER is 'i'
    Delete: CHARACTER is 'd'; -- и т.д.
    ...
end
class SOME_CLASS_FOR_THE_EDITOR
inherit
    EDITOR_CONSTANTS
    ...Другие возможные родители ...
feature ...
... подпрограммы класса имеют доступ к константам, описанным в EDITOR_CONSTANTS ...
end

```

Класс, подобный EDITOR_CONSTANTS, служит лишь для размещения в нем группы констант, и его роль как "реализации АТД" (а это - наше рабочее определение класса) не столь очевидна, как в предыдущих примерах. Теоретическое обоснование введения таких классов мы обсудим позднее. Представленная схема работоспособна только при множественном наследовании, поскольку классу SOME_CLASS_FOR_THE_EDITOR могут потребоваться и другие родители.

Константы пользовательских классов

Символические константы полезны не только при работе с предопределеными типами, такими как INTEGER. Они нужны и тогда, когда их значениями являются объекты классов, созданных разработчиком. В этом случае решение не столь очевидно.

Константы с манифестом для этого непригодны

Первым примером служит класс, описывающий комплексное число:

```

class COMPLEX creation
    make_cartesian, make_polar
feature
    x, y: REAL
        -- Действительная и мнимая часть

```

```

make_cartesian (a, b: REAL) is
    -- Установить действительную часть a, мнимую - b.
do
    x := a; y := b
end
... Прочие методы (помимо x и y, других атрибутов нет) ...
end

```

Пусть мы хотим определить константу - комплексное число *i*, действительная часть которого равна 0, а мнимая 1. Первое, что приходит в голову, - это буквальная константа вида

```
i: COMPLEX is "Выражение, определяющее комплексное число (0, 1)"
```

Как записать выражение после **is**? Для пользовательских типов данных никакой формы записи неименованных констант не существует.

Можно представить себе вариант нотации на основе атрибутов класса:

```
i: COMPLEX is COMPLEX (0, 1)
```

Но этот подход, хотя и реализован в некоторых ОО-языках, противоречит принципу модульности - основе объектной методологии. Приняв этот подход, мы согласились бы с тем, что клиенты COMPLEX должны описывать константы в терминах реализации класса, а это нарушает принцип Скрытия информации.

Кроме того, как гарантировать соответствие неименованной константы инварианту класса, если таковой имеется?

Последнее замечание позволяет найти правильное решение. Мы уже говорили о том, что в момент рождения объекта ответственность за соблюдение инварианта возлагается на **процедуру создания**. Создание объекта иным путем (помимо безопасного клонирования *clone*) ведет к ситуациям ошибки. Поэтому мы должны найти путь, основанный на обычном методе создания объектов класса.

Однократные функции

Пусть константный объект - это функция. Например, *i* можно (в иллюстративных целях) описать внутри самого класса COMPLEX как

```

i: COMPLEX is
    -- Комплексное число, re= 0, a im= 1
do
    create Result.make_cartesian (0, 1)
end

```

Это почти решает нашу задачу, поскольку функция всегда возвратит ссылку на объект нужного вида. Коль скоро мы полагаемся на обычную процедуру создания объекта, условие инварианта будет соблюдено, - как следствие, получим корректный объект.

Однако результат не соответствует потребностям: каждое обращение клиента к *i* порождает новый объект, идентичный всем остальным, а это - траты времени и пространства. Поэтому необходим особый вид функции, выполняемой только при первом вызове. Назовем такую функцию **однократной (once function)**. В целом она синтаксически аналогична обычной функции и отличается лишь служебным словом **once**, начинающим вместо **do** ее тело:

```

i: COMPLEX is
    -- Комплексное число, re= 0, im= 1
once
    create Result.make_cartesian (0, 1)
end

```

При первом вызове однократной функции она создает объект, который представляет желаемое комплексное число, и возвращает на него ссылку. Каждый последующий вызов приведет к немедленному завершению функции и возврату результата, вычисленного в первый раз. Что касается эффективности, то обращение к *i* во второй, третий и т.д. раз должно отнимать времени ненамного больше, чем операция доступа к атрибуту.

Результат, найденный при первом вызове однократной функции, может использоваться во всех экземплярах класса, включая экземпляры потомков, где эта функция не переопределена. Переопределение однократных функций как обычных (и обычных как однократных) допускается без всяких ограничений. Так, если COMPLEX1, порожденный от класса COMPLEX, заново определяет *i*, то обращение к *i* в экземпляре COMPLEX1 означает вызов переопределенного варианта, а обращение к *i* в экземпляре самого COMPLEX или его потомка, отличного от COMPLEX1, означает вызов однократной функции, то есть значения, найденного ею при первом вызове.

Применение однократных подпрограмм

Понятие однократных подпрограмм расширяет круг задач, позволяя включить разделяемые объекты, глобальные системные параметры, инициализацию общих свойств.

Разделяемые объекты

Для ссылочных типов, таких как COMPLEX, наш механизм фактически предлагает константные ссылки, а не обязательно константные **объекты**. Он гарантирует, что тело функции выполняется при первом обращении, возвращая результат, который будет также возвращаться при последующих вызовах, уже не требуя никаких действий.

Если функция возвращает значение ссылочного типа, то в ее теле, как правило, есть инструкция создания объекта, и любой вызов приведет к получению ссылки на этот объект. Хотя создание объекта не повторяется, ничто не мешает изменить сам объект, воспользовавшись полученной ссылкой. В итоге мы имеем **разделяемый** объект, не являющийся константным.

Пример такого объекта - окно вывода информации об ошибках. Пусть все компоненты интерактивной системы могут направлять в это окно свои сообщения:

```
Message_window.put_text ("Соответствующее сообщение об ошибке")
```

где Message_window имеет тип WINDOW, чей класс описан следующим образом:

```
class WINDOW
creation
    make
feature
    make (...) is
        -- Создать окно; аргументы задают размер и положение.
    do ... end
    text: STRING
        -- Отображаемый в окне текст
    put_text (s: STRING) is
        -- Сделать s отображаемым в окне текстом.
    do
        text := s
    end
    ... Прочие компоненты ...
end -- класс WINDOW
```

Ясно, что объект Message_window должен быть одним для всех компонентов системы. Это достигается описанием соответствующего компонента как однократной функции:

```
Message_window: WINDOW is
    -- Окно для вывода сообщений об ошибках
once
    create Result.make (... Аргументы размера и положения ...)
end
```

В данном случае окно сообщений должно находиться в совместном пользовании всех сторон, но не являться константным объектом. Каждый вызов put_text будет изменять объект, помещая в него новую строку текста. Лучшим местом описания Message_window станет класс, от которого порождены все компоненты системы, нуждающиеся в окне выдачи сообщений.

Создав разделяемый объект, играющий роль константы, (например, *i*), вы можете запретить вызовы *i.some_procedure*, способные его изменять. Для этого, например, в классе COMPLEX достаточно ввести в

инвариант класса предложения `i.x = 0` и `i.y = 1`.

Однократные функции с результатами базовых типов

Еще одним применением однократных функций является моделирование глобальных значений - "системных параметров", которые обычно нужны сразу нескольким классам, но не меняются в ходе программной сессии. Их начальная установка требует информации от пользователя или операционной среды. Например:

- компонентам низкоуровневой системы может понадобиться объем доступной им памяти, выделенный средой при инициализации;
- система эмуляции терминала может начать работу с отправки среде запроса о числе терминальных портов. Затем эти данные будут использоваться в ряде модулей приложения.

Такие глобальные данные аналогичны совместно используемым объектам, хотя обычно они являются значениями базовых типов. Схема их реализации однократными функциями такова:

```
Const_value: T is
    -- Однократно вычисляемый системный параметр
local
    envir_param: T' -- Любой тип (T и не только)
once
    "Получить envir_param из операционной среды"
    Result := "Значение, рассчитанное на основе envir_param"
end
```

Такие однократные функции описывают динамически вычисляемые константы.

Предположим, данное объявление находится в классе ENVIR. Класс, которому надо воспользоваться константой `Const_value`, получит ее значение, указав ENVIR в списке своих родителей. В отличие от классического подхода к расчету константы, здесь не нужна процедура инициализации системы, вычисляющая все глобальные параметры системы, как это делается в классическом подходе. Как отмечалось в начальных лекциях, такая процедура должна была бы иметь доступ к внутренним деталям многих модулей, что нарушило бы ряд критериев и принципов модульности: декомпозиции, скрытия информации и других. Наоборот, классы, подобные ENVIR, могут разрабатываться как согласованные модули, каждый задающий множество логически связанных глобальных значений. Процесс вычисления такого параметра, к примеру, `Const_value`, инициирует первый из компонентов, который запросит этот параметр при выполнении системы. Хотя `Const_value` является функцией, использующие его компоненты могут полагать, что имеют дело с константным атрибутом.

Как уже говорилось, ни один модуль не имеет больше прав на разделяемые данные, чем остальные. Это особенно справедливо для только что рассмотренных случаев. Если расчет значения способен инициировать любой модуль, нет смысла и говорить о том, будто один из них выступает в роли владельца. Такое положение дел и отражает модульная структура системы.

Однократные процедуры

Функция `close` должна вызываться только один раз. Контроль над количеством ее вызовов рекомендуется возложить на глобальную переменную приложения.

Из руководства к коммерческой библиотеке функций языка C

Механизм однократных функций интересен и при работе с процедурами. Однократные процедуры могут применяться для инициализации общесистемного свойства, когда заранее неизвестно, какому компоненту это свойство понадобится первому.

Примером может стать графическая библиотека, в которой любая функция, вызываемая первой, должна предварительно провести настройку, учитывая параметры дисплея. Автор библиотеки мог, конечно, потребовать, чтобы каждый клиент начинал работу с библиотекой с вызова функции настройки. Этот нюанс, в сущности, не решает проблему - чтобы справиться с ошибками, любая функция должна обнаруживать, не запущена ли она без настройки. Но если функции такие "умные", то зачем что-то требовать от клиента, когда можно нужную функцию настройки вызывать самостоятельно.

Однократные процедуры решают эту проблему лучше:

```
check_setup is
    -- Настроить терминал, если это еще не сделано.
once
```

```
    terminal_setup -- Фактические действия по настройке.  
end
```

Теперь каждая экранная функция должна начинаться с обращения к `check_setup`, первый вызов которой приведет к настройке параметров, а остальные не сделают ничего. Заметьте, что `check_setup` не должна экспортироваться клиентам.

Однократная процедура - это важный прием, упрощающий применение библиотек и других программных пакетов.

Параметры

Однократные процедуры и функции могут иметь параметры, необходимые, по определению, лишь при первом вызове.

Однократные функции, закрепление и универсальность

В этом разделе мы обсудим конкретную техническую проблему, поэтому при первом чтении книги его можно пропустить.

Однократные функции, тип которых не является встроенным, вносят потенциальную несовместимость с механизмом закрепления типов и универсальностью.

Начнем с универсальности. Пусть в родовом классе `EXAMPLE [G]` есть однократная функция, чей тип родовой параметр:

```
f: G is once ... end
```

Рассмотрим пример ее использования:

```
character_example: EXAMPLE [CHARACTER]  
...  
print (character_example.f)
```

Пока все в порядке. Но если попытаться получить константу с другим родовым параметром:

```
integer_example: EXAMPLE [INTEGER]  
...  
print (integer_example.f + 1)
```

В последней инструкции мы складываем два числа. Первое значение, результат вызова `f`, к сожалению, уже найдено, поскольку `f` - однократная функция, причем символьного, а не числового типа. Сложение окажется недопустимым.

Проблема заключается в попытке разделения значения разными формами родового порождения, ожидающими значения, тип которого определяется родовым параметром. Аналогичная ситуация возникает и с закреплением типов. Представим себе класс `B`, добавляющий еще один атрибут к компонентам своего родителя `A`:

```
class B inherit A feature  
    attribute_of_B: INTEGER  
end
```

Пусть `A` имеет однократную функцию `f`, возвращающую результат закрепленного типа:

```
f: like Current is once create Result1 make end
```

и пусть первый вызов функции `f` имеет вид:

```
a2 := a1.f
```

где `a1` и `a2` имеют тип. Вычисление `f` создаст экземпляр `A` и присоединит его к сущности `a2`. Все прекрасно.

Но предположим, далее следует:

```
b2 := b1.f
```

где b1 и b2 имеют тип B. Не будь f однократной функцией, никакой проблемы бы не возникло. Вызов f породил бы экземпляр класса B и вернул его в качестве результата. Но функция является однократной, а ее результат был уже найден при первом вызове. И это - экземпляр A, но не B. Поэтому инструкция вида:

```
print (b2.attribute_of_B)
```

попытается обратиться к несуществующему полю объекта A.

Проблема в том, что закрепление вызывает неявное переопределение типов. Если бы f была переопределена явно, с применением в классе B объявления

```
f: B is once create Result1 make end
```

при условии, что исходный вариант f в классе A возвращает результат типа A (а не **like Current**), все было бы замечательно: экземпляры A обращались бы к версии f для A, экземпляры B - к версии f для B. Однако закрепление типов было введено как раз для того, чтобы избавить нас от таких явных переопределений.

Эти примеры - свидетельства несовместимости семантики однократных функций (с процедурами все прекрасно) с результатами применения закрепленных типов и формальных родовых параметров. Одно из решений проблемы в том, чтобы трактовать такие случаи как явные переопределения, приняв за правило то, что результат однократной функции совместно используется лишь в пределах одной формы родовой порождения, а при закреплении результата - лишь среди экземпляров своего класса. Недостатком такого подхода, впрочем, является, что он не отвечает интуитивной семантике однократных функций, которые, с позиции клиента, должны быть эквивалентны разделяемым атрибутам. Во избежание недоразумений и возможных ошибок можно пойти на более суровые меры, наложив полный запрет на сценарии подобного рода:

Правило для однократной функции

Тип результата однократной функции не может быть закреплен и не может включать любой родовой параметр.

Константы строковых типов

В начале этой лекции были введены символьные константы, значением которых является символ. Например:

```
Backslash: CHARACTER is '\'
```

Однако нередко классам требуются строковые константы, использующие, как обычно, для записи константы двойные кавычки:

[S1]

```
Message: STRING is "Syntax error" -- "Синтаксическая ошибка"
```

Вспомните, что STRING - не простой тип. Это - библиотечный класс, поэтому значение, связанное с сущностью Message во время работы программы, является объектом, то есть экземпляром STRING. Как вы могли догадаться, такое описание является сокращенной формой объявления однократной функции вида:

[S2]

```
Message: STRING is
-- Стока из 12 символов
once
create Result.make (12)
Result.put ('S', 1)
Result.put ('y', 2)
...
Result.put ('r', 12)
end
```

Строковые значения являются не константами, а ссылками на разделяемые объекты. Любой класс, имеющий доступ к Message, может изменить значение одного или нескольких символов строки. Строковые константы можно использовать и как выражения при передаче параметров или присваивании:

```
Message_window.display ("НАЖМИТЕ ЛЕВУЮ КНОПКУ ДЛЯ ВЫХОДА")
greeting := "Привет!"
```

Unique-значения

Иногда при разработке программ возникает потребность в сущности, принимающей лишь несколько значений, характеризующих возможные ситуации. Так, операция чтения может вернуть код результата, значениями которого будут признаки успешной операции, ошибки при открытии и ошибки при считывании. Простым решением проблемы было бы применение целочисленного атрибута:

```
code: INTEGER
```

и набора символьных констант

```
[U1]
Successful: INTEGER is 1
Open_error: INTEGER is 2
Read_error: INTEGER is 3
```

которые позволяют записывать условные инструкции вида

```
[U2]
if code = Successful then ...
```

или инструкции выбора

```
[U3]
inspect
    code
    when Successful then
    ...
    when ...
end
```

Но такой перебор значений констант утомляет. Следующий вариант записи действует так же, как [U1]:

```
[U4]
Successful, Open_error, Read_error: INTEGER is unique
```

Спецификатор **unique**, записанный вместо буквального значения в объявлении атрибута-константы целого типа, указывает на то, что это значение выбирает компилятор, а не сам разработчик. При этом условная инструкция [U2] и оператор выбора [U3] по-прежнему остаются в силе.

Каждое **unique**-значение в теле класса положительно и отличается от других. Если, как в случае [U4], константы будут описаны вместе, то их значения образуют последовательность. Чтобы ограничить значение code этими тремя константами, в инвариант класса можно включить условие

```
code >= Successful; code <= Read_error
```

Располагая подобным инвариантом, производные классы, обладающие правом специализации инварианта, но не его расширением, могут сузить, но не расширить перечень возможных значений code, сведя его, скажем, всего к двум константам.

Значения, заданные как **unique**, следует использовать только для представления фиксированного набора возможных значений. Если допустить его пополнение, то это приведет к необходимости внесения изменений в тексты инструкций, подобных [U3]. В общем случае для классификации не рекомендуется использовать **unique**-значения, так как ОО-методология располагает лучшими приемами решения этой задачи. Данный выше пример является образцом правильного обращения с описанным механизмом. Правильными можно

считать и объявления цветов семафора: **green, yellow, red: INTEGER is unique;** но: **do, re, mi, ...: INTEGER is unique.** Объявление **savings, checking, money_market: INTEGER is unique** возможно будет неверным, поскольку различные финансовые инструменты, список которых здесь приведен, имеют различные свойства или допускают различную реализацию. Более удачным решением в этом случае, пожалуй, станут механизмы наследования и переопределения.

Объединим сказанное в форме правила:

Принцип дискриминации

Используйте **unique** для описания фиксированного набора возможных альтернатив. Используйте наследование для классификации абстракций с изменяющимися свойствами.

Хотя объявление **unique**-значений напоминает определение перечислимых типов (enumerated type) языков Pascal и Ada, оно не вводит новые типы, а только целочисленные значения. Дальнейшее обсуждение позволит объяснить разницу подходов.

Обсуждение

В этом разделе термин "глобальный объект" относится как к глобальным константам встроенных типов, так и к разделяемым сложным объектам, требующим в последнем случае создания объекта при инициализации.

Инициализация: подходы языков программирования

Проблема, решаемая в этой лекции, - это общая проблема языков программирования: как работать с глобальными константами и разделяемыми объектами, в частности, как выполнять их инициализацию в библиотеках компонентов?

Для библиотек более общей задачей является включение в каждый компонент возможности определения того, что его вызов является первым запросом к службам библиотеки, что и позволяет определить, была ли сделана инициализация.

Последнюю задачу можно свести к более простой: как разделять переменные булевого типа и согласованно их инициализировать? Связем с глобальным объектом *r* или группой глобальных объектов, нуждающихся в одновременной инициализации, булеву переменную, скажем, *ready*, истинную, если и только если инициализация проведена. Тогда любому обращению к *r* нетрудно предпослать инструкцию

```
if not ready then
    "Создать или вычислить r"
    ready := True
end
```

Теперь проблема инициализации касается только *ready* - еще одного глобального объекта, который необходимо инициализировать значением *False*.

Как же решается эта задача в языках программирования? С момента их появления в этом плане почти ничего не менялось. В блочно-структурированных языках, среди которых Algol и Pascal, типичным было описание *ready* как глобальной переменной на верхнем синтаксическом уровне; ее инициализация производилась в главной программе. Но такая техника непригодна для библиотек автономных модулей.

В языке Fortran, позволяющем независимую компиляцию подпрограмм (что придает им известную автономность), можно поместить все глобальные объекты в общий блок (*common block*), идентифицируемый по имени. Всякая подпрограмма, обращающаяся к общему блоку, должна содержать такую директиву:

```
COMMON /common_block_name/ data_item_names
```

При этом возникают две проблемы:

- Две совокупности подпрограмм могут использовать одноименные общие блоки, что приведет к конфликту, если одной из программ понадобится как первый, так и второй блок. Смена имени блока вызовет трудности у других программ.
- Как инициализировать сущности общего блока, такие как *ready*? Из-за отсутствия инициализации по умолчанию, ее нужно выполнять в особом модуле, называемом блоком данных (*block data unit*). В Fortran 77 допускаются именованные модули, что позволяет разработчикам объединять глобальные данные разных общих блоков. При этом есть немалый риск несогласованности инициализации и объявления глобальных объектов.

Принцип решения этой задачи в языке С по сути не отличается от решения Fortran 77. Признак `ready` нужно описать как "внешнюю" переменную, общую для нескольких "файлов" (единиц компиляции языка). Объявление переменной с указанием ее значения может содержать только один файл, остальные, используя директиву `extern`, подобную COMMON в Fortran 77, лишь заявляют о необходимости доступа к переменной. Обычно такие определения объединяют в "заголовочные" (header) .h-файлы, которые соответствуют блоку данных в Fortran. При этом наблюдаются те же проблемы, отчасти решаемые утилитами `make`, призванными отслеживать возникающие зависимости.

Решение может быть близко к тому, что предлагают модульные языки наподобие Ada или Modula 2, подпрограммы которых можно объединять в модули более высокого уровня. В Ada эти модули называют "пакетами" (package). Если все подпрограммы, использующие группу взаимосвязанных глобальных объектов, собраны в одном пакете, то соответствующие признаки `ready` можно описать в этом же пакете и здесь же выполнить их инициализацию. Однако этот подход (применимый также в С и Fortran 77) не решает проблему инициализации автономных библиотек. Еще более деликатный вопрос связан с тем, как поступать с глобальными объектами, разделяемых подпрограммами **разных** независимых модулей. Языки Ada и Modula не дают простого ответа на этот вопрос.

Механизм "однократных" методов, сохраняя независимость классов, допускает контекстно-зависимую инициализацию.

Строковые константы

Строковые константы (а точнее, разделяемые строковые объекты) объявляются в языках программирования в манифестной форме с использованием двойных кавычек. Это находит отражение в правилах языка, и как следствие любой компилятор предполагает присутствие в библиотеке класса STRING. Это - своего рода компромисс между "полярными" решениями.

- STRING рассматривается как встроенный тип, каким он является во многих языках программирования. Это означает введение в язык операций над строками: конкатенации, сравнения, выделения подстроки и других, что усложняет язык. Преимуществом введения такого класса является возможность снабдить его операции точными спецификациями, благодаря утверждениям, и способность порождать от него другие классы.
- STRING рассматривается как обычный класс, создаваемый разработчиком. Тогда задавать его константы в манифестной форме [S1] уже нельзя, от разработчиков потребуется соблюдение формата [S2]. Кроме того, данный подход препятствует оптимизации компилятором таких операций, как прямой доступ к символам строки.

Поэтому строки STRING, как и массивы ARRAY, ведут "двойную жизнь", принимая вид предопределенного типа при задании констант и оптимизации кода, и становясь классом, когда речь заходит о гибкости и универсальности.

Unique-значения и перечислимые типы

Pascal и производные от него языки допускают описание переменной вида

code : ERROR

где ERROR - это "перечислимый тип":

```
type ERROR = (Normal, Open_error, Read_error)
```

Переменная code может принимать только значения типа ERROR. Мы уже видели, как добиться того же самого в ОО-нотации: при выполнении кода результат будет почти идентичен, поскольку Pascal-компиляторы традиционно реализуют значения перечислимого типа как целые числа. Введение объявления `unique` не порождает нового типа. Понятие перечислимых типов, кажется, трудно совместить с объектным подходом. Все наши типы основаны на классах, характеризующих реально осуществимые операции и их свойства. Перечислимые типы не обладают такими характеристиками, а представляют обычные множества чисел. Проблемы с этими типами данных возникают и в необъектных языках.

- Статус символьических имен не вполне ясен. Могут ли два перечислимых типа иметь общие символьические имена (скажем, Orange в составе типов FRUIT и COLOR)? Можно ли их экспортовать как переменные и распространять на них те же правила видимости?
- Значения перечислимых типов трудно получать и передавать программам, написанным на других языках, к примеру, С и Fortran, не поддерживающих такое понятие. В тоже время значения, описанные как `unique`, - это обычные числа, работа с которыми не вызывает никаких проблем.
- Перечислимые типы данных могут требовать специальных операторов. Так, можно представить себе

оператор **next**, возвращающий следующее значение и неопределенный для последнего элемента перечисления. Помимо него потребуется оператор, сопоставляющий элементу целое значение (индекс). В итоге синтаксическое и семантическое усложнение языка кажется непропорциональным вкладу этого механизма.

Объявления перечислимых типов в Pascal и Ada обычно принимают вид:

```
type FIGURE_SORT = (Circle, Rectangle, Square, ...)
```

и используются совместно с вариантными полями записей:

```
FIGURE =
record
    perimeter: INTEGER;
    ... Другие атрибуты, общие для фигур всех типов ...
    case fs: FIGURE_SORT of
        Circle: (radius: REAL; center: POINT);
        Rectangle:... Специальные атрибуты прямоугольника ...;
    ...
end
```

Этот механизм позволяет организовать разбор случаев в операторе выбора **case**:

```
procedure rotate (f: FIGURE)
begin case f of
    Circle:... Специальные операции поворота окружности ...;
    Rectangle:... ;
    ...
```

Мы уже познакомились с лучшим способом решения этой проблемы, сохраняющим расширяемость при появлении новых вариантов, - достаточно определить различные версии процедур, подобных `rotate` для каждого нового варианта, представленного классом.

Когда это наиболее важное применение перечислимых типов исчезло, все, что осталось необходимым в некоторых случаях, - это выбор целочисленных кодов для фиксированного множества возможных значений. Определив их как обычные целые, мы избежим многих семантических неопределенностей, связанных с перечислимыми типами, например, нет ничего необычного в выражении `Circle +1`, если известно, что `Circle` типа `integer`. Введение **unique**-значения позволяет обойти единственное неудобство, связанное с необходимостью инициализации значений, позволяя выполнять ее автоматически.

Ключевые концепции

- При любом подходе к конструированию ПО возникает проблема работы с глобальными объектами, совместно используемыми компонентами разных модулей, и инициализируемыми в период выполнения, когда какой-либо из компонентов первым к ним обратился.
- Константы могут быть манифестными и **символическими**. Первые задаются значениями, синтаксис которых определен так, что значение одновременно описывает тип константы, а потому является манифестом. Символические константы представлены именами, а их значение указывается в определении константы.
- Манифестные константы базовых типов можно объявлять как константные атрибуты, не требующие памяти в объектах.
- За исключением строк, типы, определенные пользователем, не имеют манифестных констант, нарушающих принципы Скрытия информации и расширяемости.
- Однократная подпрограмма синтаксически отличается от обычной лишь ключевым словом **once**, заменяющим **do**. Она полностью выполняется лишь один раз (при первом вызове). При последующих вызовах однократной функции возвращается результат, вычисленный при первом вызове, последующие вызовы процедуры не имеют эффекта и могут быть проигнорированы.
- Разделяемые объекты могут быть реализованы как однократные функции. Можно использовать инвариант для указания их константности.
- Однократные процедуры используются там, где операции должны быть выполнены только однажды во время выполнения системы, чаще всего, это связано с инициализацией глобальных параметров системы.
- Тип однократной функции не может быть закрепленным или родовым типом.
- Константы строковых типов внутренне интерпретируются как однократные функции, однако, внешне они

- выглядят как манифестные константы, значения которых заключается в двойные кавычки.
- Перечислимые типы в стиле языка Pascal не соответствуют объектной методологии. Для представления объектов с несколькими возможными вариантами значений используются символические **unique** константы. Инициализация значений таких констант выполняется компилятором.

Библиографические замечания

Проблемы перечислимых типов были изучены в работах [Welsh 1977] и [Moffat 1981]. Некоторые приемы, рассмотренные в этой лекции, впервые представлены в [M 1988b].

Упражнения

У18.1 Эмуляция перечислимых типов однократными функциями

Покажите, что при отсутствии **unique**-типов перечислимый тип языка Pascal

```
type ERROR = (Normal, Open_error, Read_error)
```

может быть представлен классом с однократной функцией для каждого значения типа.

У18.2 Однократные функции для эмуляции unique-значений

Покажите, что в языке без поддержки **unique**-объявлений результат, аналогичный

```
value: INTEGER is unique
```

можно получить, воспользовавшись объявлением вида

```
value: INTEGER is once...end
```

где вам необходимо написать тело однократной функции и все, что может еще понадобиться.

У18.3 Однократные функции в родовых классах

Приведите пример однократной функции, чей результат включает родовой параметр, и, если он не корректен, порождает ошибку времени выполнения.

У18.4 Однократные атрибуты?

Исследуйте полезность понятия "однократного атрибута", полученного по образцу однократной функции? Будет ли такой атрибут общим для всех экземпляров класса? Как инициализировать однократные атрибуты? Являются ли они избыточными при наличии однократных функций без аргументов? Если нет, объясните, когда использовать тот или иной механизм. Предложите хороший синтаксис объявления однократных атрибутов.

Основы объектно-ориентированного проектирования

1. Лекция: Объектно-ориентированная методология: Правильно применяйте метод

В этих лекциях рассматривается методология объектной ориентации: как применять мощное множество концепций и технических приемов для получения преимуществ наших проектов и успешной их организации.

О методологии

Следующие несколько лекций с 1-й по 11-ю, составляющие часть этого курса, полностью посвящены методологии. В них исследуются проблемы, возникающие при создании ОО-проектов: как находить классы, как не делать ошибок при использовании наследования, роль и место ОО-анализа, фундаментальные идеи проектирования ("образцы"), как учить Методу, новый цикл жизни ПО. В результате, я надеюсь, придет понимание того, как наилучшим образом использовать преимущества ОО-техники, изученной в предыдущих лекциях курса "Основы объектно-ориентированного программирования".

Прежде чем изучать методологические правила, рассмотрим общую роль методологии в построении ПО. Это позволит определить метаправила, помогающие обоснованию методологических советов и выделению лучшего из того, что есть в литературе. Попутно мы изобреем таксономию правил и покажем, что некоторый вид правил предпочтительнее других. Наконец, мы покажем привлекательную и опасную роль **метафор** и отметим полезность скромности.

Методология: что и почему

Люди верят заповедям. Сражения за незыблемые "Принципы Истинной Веры" не являются чем-то новым и характерны не только для разработчиков ПО.

Программистская литература, включая ОО-ветвь, учитывает эти естественные желания и предлагает массу рецептов. В результате существует много полезных советов, наряду с еще большим количеством весьма спорных идей.

Следует помнить, что нет простых путей, ведущих к созданию качественного ПО. В предыдущих лекциях несколько раз звучала мысль, что конструирование ПО - это не тривиальная задача, каждый раз бросающая вызов разработчику. За последние годы наше понимание проблем существенно усовершенствовалось, о чем свидетельствует техника, представленная в этой книге. Одновременно выросли наши амбиции и желание создавать проекты больших размеров, работающие быстрее. В конечном счете проблемы остались такими же трудными, как и ранее.

По этим причинам важно понимать достоинства и ограничения, присущие методологии конструирования ПО. От последующих лекций этой книги, как и от всей обширной ОО-литературы, вы имеете право ожидать полезных советов и тех преимуществ, которые может дать опыт людей, создававших ПО. Но ни здесь, и нигде вы не найдете надежного и легкого пути создания качественного ПО.

Во многих отношениях построение ПО сродни построению математической теории. Математике, как и разработке ПО, можно учить на примерах и общих принципах, помогающих талантливым студентам достигать выдающихся результатов, но никакое обучение не может гарантировать успех в этой деятельности.

Конечно, не все так безнадежно. Если оставаться в пределах одной проблемной области, где уже существует множество образцов, то возможно определить пошаговый процесс, приводящий к успеху. Эта ситуация встречается в некоторых областях обработки данных, где методология выработала сравнительно небольшое число широко применимых схем решения. Как правило, в результате таких схем создаются тиражируемые программные комплексы или повторно используемые библиотеки программ. Но как только вы переходите к новой проблемной области, простые подходы перестают работать, и разработчик должен проявить все свое искусство. Методология может служить общим руководством благодаря примерам предыдущих удачных решений, а также примерам того, что не работает, - но не более того.

Пусть эти рассуждения остаются в подсознании как при чтении лекций 19-29 этого курса, так и при чтении любой другой методологической литературы, где предлагаются разнообразные советы. Нет причин их полного отрицания - они могут быть полезными, - но отделяйте зерна от плевел.

Замечание к терминологии: в литературе некоторого сорта становится обычным говорить о специфических "методологиях", в действительности означающих методы, а чаще всего вариации одного общего ОО-метода. Эта практика может рассматриваться как пример вербального мыльного пузыря - это все равно, что называть ремонтника **инженером по сопровождению**, - опасность в том, что читатель может подозревать отсутствие содержания за красивой вывеской. В этой книге слово "методология" используется в единственном числе и несет общепринятый смысл - изучение методов, применение принципов умозаключений в научных и философских исследованиях, система методов.

Как создавать хорошие правила: советы советчикам

Прежде чем перейти к специфическим правилам использования ОО-методов, необходимо спросить себя, что мы ожидаем от них. Методологи берут на себя большую ответственность: учить разработчиков, как следует писать их программы и как не следует этого делать. Трудно избежать сравнения с проповедниками, - позиция чреватая, как известно, злоупотреблениями, тем не менее, определим несколько правил над правилами: дадим советы советчикам.

Необходимость методологических руководств

Методология разработки ПО не является новой областью. Ее истоки восходят к известной работе Дейкстры "Go To Statement Considered Harmful" (О вреде оператора Go To) и последующим работам этого же автора и его коллег по структурному программированию. Но не все последующие методологические работы поддерживают достигнутый уровень стандартов.

На самом деле довольно просто запретить ту или иную программистскую конструкцию, но есть величайшая опасность в создании бесполезных правил, плохо обдуманных и даже вредных. Следующие заповеди, основанные на анализе роли методологии в создании ПО, помогут нам избежать подобных ловушек.

Теория

Первая обязанность советчика - давать совет, согласующийся с предметной областью:

Теоретический базис: принцип методологии

Правила методологии ПО должны базироваться на теории предметной области.

Пример Дейкстры может служить хорошей иллюстрацией. Он не пытался атаковать оператор Goto по причинам вкуса или чьих либо мнений - он приводил тщательно выверенную систему выводов. Кто-то мог не соглашаться с некоторыми из аргументов, но не мог отрицать, что заключение построено на хорошо продуманном взгляде на процесс разработки ПО. Не считаться с точкой зрения Дейкстры можно лишь при обнаружении изъяна в его теории и построении своей собственной теории для этого случая.

Практика

Теория - это дедуктивная часть методологии ПО. Но правила, основанные только на теории, могут быть опасными. Эмпирическая компонента столь же важна:

Практический базис: принцип методологии

Правила методологии ПО должны основываться на широком практическом опыте.

Возможно, кто-нибудь и когда-нибудь изобретет блестящий и применимый метод конструирования ПО, исходя из теоретических рассуждений. В физике такие примеры хорошо известны, и теоретики получали вполне практические результаты, не выполняя никаких практических экспериментов. Но в инженерии программ такие случаи не наблюдались, - все великие методологи одновременно были программистами и лидерами при разработке больших программных проектов. И в объектной технологии каждый может освоить основные концепции, читая литературу, выполняя небольшие проекты и размышляя о больших разработках, но этой подготовки недостаточно для того, чтобы давать методологические советы. Опыт играет ключевую роль в построения больших систем, состоящих из тысяч классов, десятков тысяч строк кода, - здесь опыт незаменим.

Такой опыт должен включать все этапы жизненного цикла ПО: анализ, проектирование, реализацию и, конечно же, сопровождение (заключительный аккорд, который только и показывает, выдержали ли ваши решения, принятые на предыдущих этапах, проверку временем и изменениями).

Опыта анализа или даже анализа и проектирования явно недостаточно. Не один раз приходилось видеть, как консультанты по анализу выполняли свою работу, получали плату, и оставляли компанию с не более чем схемами с квадратиками и стрелками - документом анализа. А затем компания должна была извлекать нечто полезное из этих кусочков и делать свою трудную работу; иногда работа аналитиков оказывалась полностью бесполезной, поскольку не учитывала важных практических ограничений. Подход "только анализ" противоречит фундаментальным идеям **бесшовности (seamlessness)** и **обратимости (reversibility)**, интегрированному жизненному циклу, характерному для объектной технологии, где анализ и проектирование свиваются с реализацией и сопровождением. Кто не прошел все этапы этого пути, вряд ли может давать методологические советы.

Повторное использование

Принимать ключевое участие в ряде больших проектов необходимо, но недостаточно. В области ОО-разработок необходим опыт создания повторно используемых программных продуктов.

К отличительным свойствам ОО-метода относится возможность создавать повторно используемые компоненты. Никто не может считать себя экспертом, если он не создавал повторно используемую ОО-библиотеку - не просто компоненты, заявленные как повторно используемые, но библиотеку, которая бы фактически и многократно использовались бы пользователями, не входящими в группу разработки. Вот следующий принцип:

Повторное использование: принцип методологии

Для получения статуса эксперта в ОО-области необходимо играть ключевую роль в разработке библиотеки классов, которая бы широко использовалась в различных проектах и в различных контекстах.

Типология правил

Обратимся теперь к форме методологических правил. Какой вид советов является эффективным?

Правило может быть рекомендацией (**advisory**) - приглашением следовать определенному стилю, или абсолютным (**absolute**) - предписывающим выполнять работу определенным образом. Правило может быть выражено в положительной (**positive**) форме - говорящей, что следует делать, или в отрицательной (**negative**) форме - чего не следует делать. Это дает нам четыре вида:

Классификация методологических правил

- Абсолютно положительное: "Всегда делай а".
- Абсолютно отрицательное: "Никогда не используй б".
- Рекомендательно положительное: "Используй с, если это возможно".
- Рекомендательно отрицательное: "Избегай д, если это возможно".

В каждом случае требования слегка отличаются.

Абсолютная положительность

Правила абсолютно положительного вида наиболее полезны для разработчиков ПО, так как дают точное и недвусмысленное руководство к действию.

К сожалению, они являются наиболее редким видом правил, появляющимся в методологической литературе. Частично этому есть разумные причины - для точных советов можно написать соответствующий инструментарий, автоматически выполняющий требуемую задачу, избавляя тем самым от необходимости давать методологические указания. Но часто за этим стоит осторожность методологов, которые, подобно адвокатам, никогда не говорящим "да" или "нет", опасаются последствий, когда их клиент будет действовать, полагаясь на их рекомендации.

Абсолютная положительность: принцип методологии

При выработке методологических правил отдавайте предпочтение абсолютной положительности. Для каждого такого правила рассмотрите возможность его автоматического выполнения благодаря специальному инструментарию или конструкциям языка.

Абсолютная отрицательность

Абсолютная отрицательность - весьма чувствительная область. Можно только пожелать, что всякий, кто рискнет пойти по стопам Дейкстры, проявит ту же тщательность в проверке отрицательного эффекта, как это сделал Дейкстра по отношению к оператору Goto. Следующее предписание справедливо для таких правил:

Абсолютная отрицательность: принцип методологии

Любое абсолютное отрицание должно сопровождаться точным пояснением того, почему отвергаемый механизм является плохой практикой. Это пояснение должно дополняться точным описанием механизма, заменяющего отвергаемый.

Рекомендации

Рекомендательные правила, положительные или отрицательные, несут в себе риск бесполезности.

Чтобы отличить **принцип от обычной банальности** (*platitude*), следует рассмотреть отрицание: для принципов отрицание имеет смысл независимо от того, согласны вы с ним или нет. Например, часто цитируемый методологический совет: "Используйте имена переменных, имеющие смысловое содержание", - не является принципом с этой точки зрения, так как никто в здравом уме не будет выбирать бессмысленные имена переменных. Для превращения этого правила в принцип, следует задать точный стандарт именования переменных. Конечно, поступив так, вы обнаружите, что некоторые читатели не будут согласны с предложенным стандартом, - вот почему банальности более комфорtabельны, но методологи должны брать на себя подобные риски.

Рекомендательные правила, избегая абсолютных запретов, частично склонны превращаться в банальности, особенно когда они имеют форму "**всегда, когда это возможно**" или для случая отрицательной рекомендации: "**если это не является абсолютно необходимым**", - наиболее бесчестная формулировка в методологии ПО.

Следующее предписание позволит избежать этого риска, сохраняя нашу честность:

Рекомендательные правила: принцип методологии

Создавая рекомендательные правила (положительные или отрицательные), используйте принципы, а не банальности. Для того чтобы отличить одно от другого, используйте отрицание.

Вот пример отрицательной рекомендации, извлеченный из обсуждения преобразования типов (*casts*) в одном из справочников по C++:

Явного преобразования типов лучше избегать. Использование кастинга подавляет проверку типов, обеспечиваемую компилятором, и тем самым может приводить к сюрпризам, если только программист действительно не был прав.

Все это не сопровождается пояснением, как же выяснить, что "**программист действительно был прав**". Так что читателю предлагается не использовать некоторый механизм языка (*type casts*), предупреждая, что это может быть опасно и "**приводить к сюрпризам**", неявно заявляя, что иногда этот механизм следует применять, не давая никакого указания, как различать случай легитимного использования.

Такие советы, обычно, бесполезны. Более точно, они несут в себе отрицательный эффект - неявно заставляя читателя полагать, что описываемые средства, в данном случае язык программирования, имеют небезопасные области неопределенности, так что им не следует полностью доверять.

Исключения

Многие правила имеют исключения. Но если вы представляете методологическое правило и желаете указать, что оно не всегда применимо, следует точно определить исключения. В противном случае правило будет неэффективным: каждый раз, когда разработчик действительно нуждается в совете, он будет судорожно размышлять применимо правило или нет.

В одной из статей по методологии ПО после представления довольно строгого множества правил приводится следующий абзац:

Строгая версия формы класса, вытекающая из закона Деметры (см. курс [Основы объектно-ориентированного программирования](#)), предназначена быть нормой, хотя это и не является абсолютным ограничением. Минимизированная версия законной формы класса дает нам выбор, насколько мы хотим следовать строгой версии закона: чем больше классов с непредпочтительным знакомством вы используете, тем менее следует придерживаться строгой формы. В некоторых ситуациях цена соблюдения строгой версии может быть выше тех преимуществ, которые она предоставляет.

После чтения этого пассажа довольно трудно решить, насколько серьезно авторы предлагают свои собственные правила, - когда их следует применять и когда лучше ими не пользоваться?

Что ошибочного, если исключения не присутствуют в общих руководствах? Поскольку проектирование ПО является сложной задачей, то иногда необходимо (хотя всегда нежелательно) к абсолютно положительному правилу "**Всегда делай X в ситуации A**" или к абсолютно отрицательному "**Никогда не делай Y в ситуации A**" добавлять квалификацию "**за исключением случаев B, C и D**". Такое квалифицированное правило остается абсолютно положительным или отрицательным: просто его область применения сужается, теперь это не все A, но A лишенное B, C и D. Расплывчатая формулировка исключений неприемлема ("**в некоторых ситуациях** **потери могут быть больше преимуществ**" - что это за ситуации?). Позже в цитируемой статье показан пример, нарушающий правило, но исключение проверяется в терминах, специально подобранных для данного случая, а оно должно быть частью правила:

Включение исключений: принцип методологии

Если методологическое правило представляет общеприменимое руководство, предполагающее исключения, то исключения должны быть частью правила.

Если исключения включены в правило, то они перестают быть исключениями из правила! Вот почему в приведенном принципе говорится о "руководстве", связанном с правилом. У руководства могут быть исключения, но они не являются исключениями для правила. В правиле "**Переходите улицу только при зеленом свете светофора, если только светофор в порядке**" руководство "переходите улицу только на зеленый" имеет исключения, но у правила как у целого исключений нет.

Этот принцип возвращает каждому правилу форму: "Делай это..." абсолютно положительного правила или "Не делай этого..." абсолютно отрицательного.

Сомнения - это прекрасное качество во многих жизненных обстоятельствах, но не тогда, когда оно встречается в методологических правилах. Почти каждый согласится с тем, что сомневающийся, не говорящий ничего определенного методолог, хуже выдающегося, кто может случайно ошибиться. Советы первого, в большей части, бесполезны, поскольку никогда нельзя определить, применимы ли они к данной ситуации. Если при изучении правила, вы не согласны с ним, то аргументам автора следует противопоставить собственные. Независимо от исхода вы чему-нибудь научитесь: либо вы ошиблись и тогда вы придетете к более глубокому пониманию правила и его применимости к вашей проблеме, либо обнаружите ограничения правила.

Абстракция и точность

Общая суть последних нескольких принципов состоит в том, что правила должны быть точными и директивными.

Конечно, это в большей мере относится к правилам, чем к общим руководствам по проектированию. Когда ищешь советы по выявлению того, какими должны быть классы проекта, или по построению нужной иерархии наследования, то нельзя ожидать рецептов вида: шаг-1, шаг-2, шаг-3.

Но даже тогда общность и абстракция не означают неопределенность. Большинство из принципов ОО-проектирования не могут выполнить за вас вашу работу. Но все же они достаточно точны, чтобы быть непосредственно применимыми, позволяя однозначно решить, применимы ли они в данном конкретном случае.

Если это необычно, зафиксируй это

Совет по кастингу типов C++, процитированный выше, иллюстрирует проблему, общую для советов отрицательного типа: подобные рекомендации своим существованием обязаны ограничениям соответствующего инструментария или языка. Для совершенного инструментария никогда не приходится давать отрицательные советы; каждое свойство такого инструментария сопровождается точным определением, когда оно применимо, а когда нет - критерий абсолютного вида. Поскольку совершенства в мире нет, то в хорошем инструментарии число отрицательных советов должно оставаться относительно небольшим. Если при изучении нового средства приходится часто сталкиваться с комментариями в форме "Пытайтесь избегать этого механизма, если только в нем нет обязательной необходимости", то это в большей степени говорит о проблеме с изучаемым инструментарием, а не о том, чтобы чему-нибудь вас обучить.

В таких ситуациях вместо того, чтобы давать подобные советы, лучше улучшить само средство или построить лучшее.

Типичные фразы, свидетельствующие о подобной ситуации:

...разве только вы знаете, что делаете.

...разве только вы абсолютно должны.

Избегайте... если можете.

Не пытайтесь...

Обычно это предпочтительно, но...

Лучше этого избегать...

Литература по C/C++/Java имеет особую любовь к таким формулировкам. Типичным является совет "Не пишите в поля данных ваших структур, если только вы не должны" от того же эксперта по C++, предупреждавшего в одной из предыдущих лекций о нежелательности использования ОО-механизмов.

Этот совет загадочен. По каким причинам разработчики не должны записывать данные?

Нарастающая Проблема Программистов, Пищущих в Поля Структуры Данных, Не Заботящихся Об Индустрии ПО США. Почему они делают это? Говорит Джил Добраядуша, Старший Программист из Санта Барбары, Калифорния: "Мое сердце склонно к добрым делам. В пространстве свопинга чувствуется такое одиночество! Я считаю своим долгом записывать данные в поле каждого из моих объектов, по меньшей мере один раз в день, даже если приходится писать его собственное значение. Иногда я возвращаюсь во время уикенда, просто чтобы сделать это". Действия программистов, подобных Джилу, приводят к растущим озабоченностям поставщиков ПО, заявляющих о необходимости специальных мер, чтобы справиться с этими проблемами.

Другая известная ситуация возникает при попытках с помощью методологических советов устранить пороки языка разработки, - перекладывая ответственность за чьи-то ошибки на пользователей языка. В одной из предыдущих лекций (Исключения) цитировался совет программистам Java ("**однако программист может повредить объекты**"), направленный против прямого присваивания полю `a.x := y` как нарушающий принципы скрытия информации. Это удивительный подход; если вы думаете, что конструкция плоха, то зачем включать ее в язык программирования, а затем писать книгу, предписывающую будущим пользователям языка избегать ее.

Закон Деметры, цитированный ранее, дает еще один пример. Он ограничивает тип `x` в вызове `x.f (. . .)`, появляющемся в программе `r` класса `C`: типами аргументов `r`; типами атрибутов `C`; типами создания (типами `i` в `create i`) для инструкций создания, появляющихся в `r`. Такие правила, если они обоснованы, должны быть частью языка. Но сами авторы полагают, что это было бы чересчур жесткое требование. Это правило делает невозможным, например, написать вызов `my_stack.item.some_routine`, применяющий `some_routine` к элементу в вершине стека.

Исследуя возможные причины появления этого закона и его исключений, можно предположить, что авторы не рассматривали понятия селективного выбора, позволяющего предоставлять разные права доступа разным клиентам. С этим механизмом необходимость в законах, подобных закону Деметры, просто бы отпала.

Эти наблюдения приводят к последнему принципу:

Фиксируйте все, что приводит к поломкам: принцип методологии

Если вы встретились с необходимостью многих отрицательных советов:

- L Исследуйте поддерживающий инструментарий или язык, чтобы определить, не связано ли это с дефектами используемых средств.
- L Если так, рассмотрите возможность изменений.
- L Рассмотрите также возможность исключить проблему, переключившись на лучшее средство.

Об использовании метафор

АНДРОМАХА: Я не понимаю абстракций.

КАССАНДРА: Как пожелаешь. Давай пользоваться метафорами.

Жан Жирадо, Троянская война не произошла, Акт I

В этом метаметодологическом обсуждении полезно вкратце отразить область и границы применения мощного средства, полезного при объяснениях, - метафор.

Все используют метафоры - аналогии - для обсуждения и обучения техническим проблемам. Эта книга не является исключением. Центральными метафорами для нее являются понятия наследования и Проектирования по контракту. Метафорой является и слово "объект", термин, нагруженный повседневным смыслом, но используемый для специфических целей.

В научных областях метафоры являются мощным, но опасным средством. Это в полной мере относится к методологии разработки ПО.

Мой коллега однажды поклялся, что он покинет конференцию, если еще раз услышит сравнение с автомобилями ("если бы программы были подобны автомобилям"). Если бы он выполнял свой зарок, сколько дискуссий ему пришлось бы пропустить!

Хорошо или плохо применять метафоры? Это может быть очень хорошо или очень плохо - все зависит от целей, для которых они используются.

Ученые используют метафоры в своих исследованиях; употребляя для объяснения наиболее абстрактных понятий конкретные видимые образы. Великий математик Адамар, например, описывал яркие образы: сталкивающиеся красные шары, облака, "**ленты, становящиеся толще или темнее в местах, соответствующих возможно важным термам**". Эти сравнения использовались в математических

выпусках, в которых он и его последователи решали трудные задачи в наиболее абстрактных областях алгебры и анализа.

Метафоры могут служить великолепным обучающим средством. Великие ученые - Эйнштейн, Фейнман, Саган - бесподобны в изложении трудных вещей с использованием аналогий и концепций повседневного опыта. Все это превосходно.

Но существует и опасность. Если мы начинаем воспринимать метафоры в их повседневном смысле и начинаем на этом основании делать выводы, то мы можем столкнуться с серьезными проблемами. Псевдосиллогизм ("Доказательство по аналогии") в форме:

А походит на В
В имеет свойство р

Ergo: А имеет свойство р

обычно порочен. Ясно, что некоторые свойства В должны отличаться от свойств А, в противном случае А и В были бы одной и той же вещью. Вспомните академиков Лапуты из Путешествий Гулливера, которые полагали: "так как слова это - только имена вещей, то было бы намного удобнее носить при себе вещи, необходимые для выражения наших мыслей и желаний". Метафора включает и то, что есть общего, и то, что различается. По этой причине для истинности заключения следует проверить, что р включено в общую часть. Когда Адамар, используя интуицию, получал результат, он знал, что шаг за шагом он должен проверить его, основываясь на строгих законах математики. Блестящие образы - только начало процесса.

Метафоры могут вдохновлять нас, но следует вовремя остановиться, чтобы избежать путаницы между реальной вещью А и метафорой В.

Прекрасная книга [Bachelard 1960] "Формирование научного сознания", посвященная тому, как в 18-м веке происходил переход к научному сознанию, приводит историю, которую следует знать каждому, кто пытается использовать метафоры в научных рассуждениях. В попытке понять природу воздуха великий физик и философ Реомюр использовал общую метафору губки, которая, как показано у Бэчеларда, восходит еще к Декарту.

Весьма общей идеей является рассматривать воздух подобным хлопку, шерсти, губке. Эта идея в частности позволяет адекватно объяснить, почему воздух может становиться разреженным и занимать значительно больший объем, чем это было моментом ранее.

Воздух подобен губке, а потому воздух расширяется подобно губке! А потом приходит никто иной, как Бенджамин Франклайн и находит губки весьма удобными для объяснения электричества. Если материал подобен губке, электрический поток должен, конечно, быть подобным жидкости, текущей сквозь губку.

Обычный материал представляет некоторый вид губки для электрической жидкости. Губка не сможет получать воду, если частицы, из которых сделана жидкость, больше чем поры губки...

Комментарий Бэчеларда: "Франклин только думает в терминах губки. Губка для него эмпирическая категория".

В процитированных метафорах Реомюра и Франклина ощущается излучение интеллектуальных гигантов своего времени, за каждый из которых стоят серьезные научные достижения. Они послужат примером и нам при обсуждении программистских концепций и помогут оценить вещи в перспективе, когда автор отходит от своих собственных аналогий.

Как важно быть скромным

Последнее слово из общих советов, перед тем как перейти к изучению специфических правил проектирования. Чтобы создать великолепный продукт, проектировщики, даже самые лучшие, никогда не должны переоценивать значение своего опыта. Каждый серьезный программный проект - это всегда новый вызов, здесь нет гарантированных рецептов.

Проектирование программного продукта - это интеллектуальное приключение. Излишняя самоуверенность может только навредить. Книги, которые вы прочли или написали, созданные или изученные классы, языки программирования, изученные или созданные вами, и многое-многое другое поможет вам лучше справиться с новой разработкой. Но не думайте, что весь ваш опыт дает гарантию от ошибок. В серьезных проектах ничто не заменит свежего мышления и творческого вдохновения. Каждая новая проблема открыта для новых идей. Каждый, начиная от умудренного лидера до новичка, может иметь верное понимание конкретной проблемы и каждый может ошибаться. Великого проектировщика отличает не то, что у него мало плохих идей, а его умение отказываться от них, умение подавлять свою гордость и отбирать хорошие идеи независимо от того, сгенерированы они им самим или кем-то другим. Некомпетентность и неопытность являются очевидными

препятствиями в борьбе за верное решение. Зазнайство может быть столь же плохим.

Никто не удивится этому комментарию, если слышал Лучано Паваротти, заявившего, что он испытывает страх перед каждым своим выступлением. Одна из причин, по которой выдающиеся люди являются выдающимися, - это их требовательность к самим себе. Это правило в полной мере относится к проектированию ПО, где всегда есть риск попасть в интеллектуальную ловушку и принять простое, но ошибочное решение, о котором позже можно лишь с огорчением сожалеть.

Библиографические замечания

"Советы советчикам" часть этой лекции основана на работе [M 1995b].

Я впервые услышал о разнице между принципами и банальностями в докладе Джозефа Гурве [Joseph Gurvet, TOOLS EUROPE 1992]. Я благодарен Эрику Бизолту за комментарий о связи селективного выбора с законом Деметры.

Упражнения

У1.1 Самоприменение правил

Выполните критику методологических правил этой книги в свете рекомендаций этой лекции.

У1.2 Библиотека правил

[M 1994a] содержит широкое множество правил: как принципов проектирования, так и стандартов стиля для построения библиотеки классов. Выполните критику этих правил в свете рекомендаций этой лекции.

У1.3 Применение правил

Выберите любую программистскую книгу с методологическими советами и подвергните проверке данные в ней правила.

У1.4 Метафоры в сети

Понаруждайте несколько недель за программистскими конференциями в Интернете. Посмотрите, какие метафоры используются авторами, насколько они значимы, используют ли авторы "доказательство по аналогии".

Основы объектно-ориентированного проектирования

2. Лекция: Образец проектирования: многопанельные интерактивные системы

Создадим образец проектирования, иллюстрирующий ОО-метод. Он будет сравниваться с другими подходами, в частности с функциональной декомпозицией сверху вниз, что позволит продемонстрировать выдающиеся возможности Метода. Поскольку этот пример, имея небольшой размер, прекрасно охватывает принципиальные свойства конструирования ОО-ПО, я часто использую его, когда необходимо в течение нескольких часов познакомить аудиторию с возможностью Метода. Показывая на практике (даже для людей с весьма слабой математической подготовкой), как каждый может перейти от классической декомпозиции к ОО-взгляду на вещи, и преимущества, получаемые в результате такого перехода, наш пример является замечательным педагогическим средством. Читателям еще в начале книги предлагалось прочесть эту лекцию, представляющую "рекламный ролик" ОО-метода проектирования. Для облегчения задачи она написана как можно более независимо. Если эта лекция читается после изучения предыдущих лекций, то вы встретите некоторые повторения, в частности несколько коротких определений уже знакомых концепций.

Многопанельные системы

Наша задача - спроектировать систему, представляющую некоторый общий тип интерактивных систем. В этих системах работа пользователя состоит в выполнении ряда этапов, и каждый этап поддерживается полноэкранный диалоговой панелью.

В целом процесс является простым и хорошо определенным. Сессия работы пользователя проходит через некоторое число состояний. В каждом состоянии отображается некоторая панель, содержащая вопросы к пользователю. Пользователь дает требуемые ответы, проверяемые на согласованность (вопросы повторяются, пока не будет дан корректный ответ); затем ответ обрабатывается некоторым образом, например обновляется база данных. Частью пользовательского ответа является выбор следующего шага, интерпретируемый как переход к следующему состоянию, в котором процесс повторяется.

Примером может служить система резервирования авиабилетов, где состояния представляют такие шаги обработки, как User Identification (Идентификация Пользователя), Enquiry on Flights (Запрос Рейса в нужное место и требуемую дату), Enquiry on Seats (Запрос Места на выбранный рейс), Reservation (Резервирование).

Типичная панель для состояния Enquiry on Flights (Запрос Рейса) может выглядеть как на [рис. 2.1](#) (рисунок иллюстрирует только идею и не претендует на реалистичный или хороший эргономичный дизайн). Экран показан на шаге, завершающем состояние, ответы пользователя в соответствующих окнах показаны курсивом, реакция системы на эти ответы (показ доступных рейсов) дана жирным шрифтом.

- Запрос на Рейсы -

Вылет из: В:

Дата вылета, начиная с: Прилета, не позже:

Предпочитаемые авиакомпании:

Специальные требования:

СПЕЦИАЛЬНЫЕ ТРЕБОВАНИЯ: 1

Рейс № AA 42 Вылет 8:25 Прибытие 7:45 Через: Chicago

Выберите следующее действие:

0 — **Выход**

1 — **Справка**

2 — **Следующий запрос**

3 — **Резервирование места**

Рис. 2.1. Панель "Запрос Рейса"

Сессия начинается в начальном состоянии `Initial` и заканчивается в заключительном состоянии `Final`. Мы можем

представить всю структуру графом переходов, показывающим возможные состояния и переходы между ними. Ребра графа помечены целыми, соответствующими возможному выбору пользователя следующего шага при завершении состояния. На [рис. 2.2](#) показан граф переходов системы резервирования авиабилетов.

Проблема, возникающая при проектировании и реализации таких приложений, состоит в достижении максимально возможной общности и гибкости. В частности:

- **G1** Граф может быть большим. Довольно часто можно видеть приложения, включающие сотни состояний с большим числом переходов.
- **G2** Структура системы, как правило, изменяется. Проектировщики не могут предвидеть все возможные состояния и переходы. Когда пользователи начинают работать с системой, они приходят с запросами на изменение системы и расширение ее возможностей.
- **G3** В данной схеме нет ничего специфического для конкретного приложения. Система резервирования авиабилетов является лишь примером. Если вашей компании необходимо несколько таких систем для собственных целей или в интересах различных клиентов, то большим преимуществом было бы определить общий проект или, еще лучше, множество модулей, допускающих повторное использование в разных приложениях.

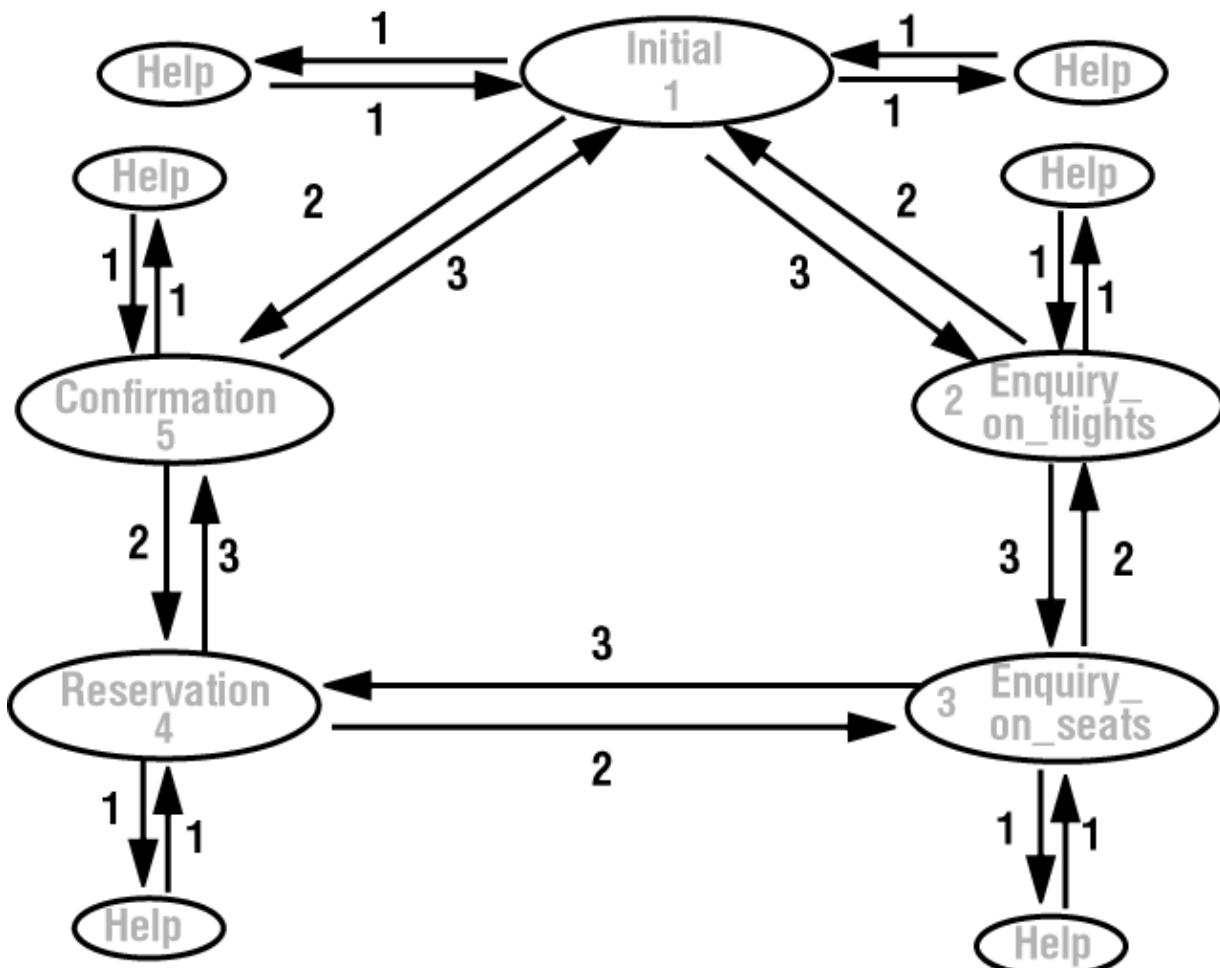


Рис. 2.2. Граф переходов в системе резервирования авиабилетов

Первая напрашивающаяся попытка

Давайте начнем с прямолинейной, без всяких ухищрений программной схемы. В этой версии наша система будет состоять из нескольких блоков по одному на каждое состояние системы: B_{Enquiry} , $B_{\text{Reservation}}$, $B_{\text{Cancellation}}$ и т. д. Типичный блок (выраженный не в ОО-нотации этой книги, а в специально подобранный для этого случая, хотя и удовлетворяющей некоторым синтаксическим соглашениям), выглядит следующим образом:

```

 $B_{\text{Enquiry}}:$ 
    "Отобразить панель Enquiry on flights"
    repeat
        "Чтение ответов пользователя и выбор С следующего шага"
        if "Ошибка в ответе" then
            "Вывести соответствующее сообщение" end
    until not "ошибки в ответе" end
    "Обработка ответа"
    case C in
         $C_0$ : goto Exit,
         $C_1$ : goto  $B_{\text{Help}}$ ,

```

```

C2: goto BReservation,
...
end

```

Аналогичный вид имеют блоки для каждого состояния.

Что можно сказать об этой структуре? Ее нетрудно спроектировать, и она делает свое дело. Но с позиций программной инженерии она оставляет желать много лучшего.

Наиболее очевидная критика связана с присутствием инструкций `goto` (реализующих условные переходы подобно переключателю `switch` языка C или "Вычисляемый Goto" Fortran), из-за чего управляющая структура выглядит подобно "блюду спагетти" и чревата ошибками.

Но `goto` - это симптом заболевания, а не настоящая причина. Мы взяли поверхностную структуру нашей проблемы - текущую форму диаграммы переходов - и перенесли ее в алгоритм. Ветвящаяся структура программы является точным отражением графа переходов. Из-за этого наш проект становится уязвимым к любым простым и общим изменениям, о чем уже говорилось выше. Когда кто-то попросит нас добавить состояние и изменить график переходов, нам придется менять центральную управляющую структуру системы. Нам придется забыть, конечно же, о надеждах повторного использования приложений - цели **G3** из нашего списка, требующей, чтобы структура покрывала все возможные приложения подобного вида.

Это пример является отрезвляющим напоминанием, когда приходится слышать о преимуществах "моделирования реального мира" или "вывода системы из анализа реальности". В зависимости от того, как вы его описываете, реальный мир может быть простым или выглядеть непонятной кашей. Плохая модель приводит к плохому ПО. Следует рассматривать не то, насколько близко ПО к реальному миру, а насколько хорошо его описание. В конце этой лекции этому вопросу еще будет уделено внимание.

Чтобы получить не просто систему, а хорошую систему, придется еще поработать.

Функциональное решение: проектирование сверху вниз

Повторяя на этом частном примере эволюцию, пройденную программированием в целом, перейдем от низкоуровневого подхода `goto` к иерархической структуре, создаваемой при проектировании сверху вниз. Это решение принадлежит общему стилю, известному как "структурное", хотя этот термин следует использовать с осторожностью.

ОО-решение является, конечно же, структурным, соответствующим духу "структурного программирования", как оно изначально было введено Э. Дейкстрой.

Функция переходов

Первым шагом на пути улучшения нашего решения будет приданье центральной роли алгоритму обхода в структуре ПО. Диаграмма переходов - это всего лишь одно из свойств нашей системы, и нет никаких оснований, чтобы она правила над всеми частями системы. Ее отделение от остального алгоритма позволит, по крайней мере, избавиться от `goto`. Можно ожидать и большей общности, поскольку диаграмма переходов специфична для приложения, такого как резервирование авиабилетов, в то время как алгоритм обхода графа более универсален.

Что представляет собой диаграмма переходов? Абстрактно это функция `transition`, имеющая два аргумента - состояние и выбор пользователя. Функция `transition` (`s, c`) возвращает новое состояние, определяемое пользовательским выбором `c` в состоянии `s`. Здесь слово "функция" используется в его математическом смысле. На программном уровне можно выбрать реализацию `transition` либо функцией в программистском смысле, либо структурой данных, например массивом. На данный момент решение можно отложить и рассматривать `transition` просто как абстрактное понятие.

В добавление к функции `transition` необходимо спроектировать начальное состояние `initial`, - точку, в которой начинаются все сессии, и одно или несколько заключительных состояний как булеву функцию `is_final`. И снова речь идет о функции в математическом смысле, независимо от ее возможной реализации.

Зададим функцию `transition` в табличной форме со строками, представляющими состояния, и столбцами, отображающими пользовательский выбор:

Таблица 2.1.

Таблица
переходов

Состояние	Выбор			
	0	1	2	3
1 (Initial)	-1	0	5	2
2 (Flights)	-1	0	1	3
3 (Seats)	0	2	4	
4 (Reserv.)	0	3	5	
5 (Confirm)	0	4	1	

Соглашения, используемые в таблице: здесь в состоянии Help с идентификатором 0 задан специальный переход Return, возвращающий в состояние, запросившее справку, задано также ровно одно финальное состояние -1. Эти соглашения не являются необходимыми для дальнейшего обсуждения, но позволяют проще сделать таблицу.

Архитектура программы

Следуя традиционным рекомендациям декомпозиции сверху вниз, выберем "вершину" - главную функцию нашей системы. Это должна быть, очевидно, программа `execute_session`, описывающая выполнение полной интерактивной сессии.

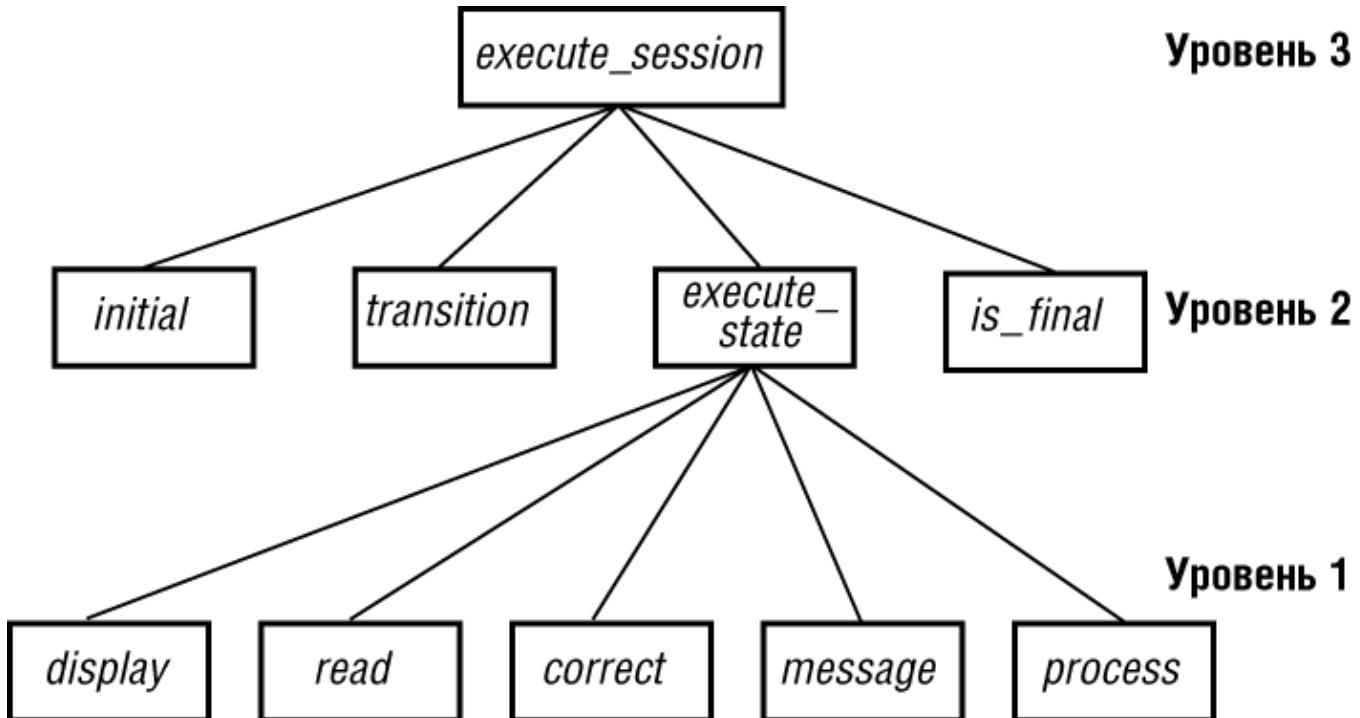


Рис. 2.3. Функциональная декомпозиция сверху-вниз

Непосредственно ниже (уровень 2) найдем операции, связанные с состояниями: определение начального и конечного состояний, структуру переходов и функцию `execute_state`, описывающую действия, выполняемые в каждом состоянии. На нижнем уровне 1 найдем операции, определяющие `execute_state`: отображение панели на экране и другие. Заметьте, что и это решение, также как и ОО-решение, описываемое чуть позже, отражает "реальный мир", в данном случае включающий состояния и элементарные операции данного мира. В этом примере и во многих других не в реальности мира состоит важная разница между ОО-подходом и другими решениями, а в том, как мы моделируем этот мир.

При написании программы `execute_session` попытаемся сделать наше приложение максимально независимым. (Наша нотация выбрана в соответствии с примером. Цикл `repeat until` заимствован из Pascal.)

```

execute_session is
  -- Выполняет полную сессию интерактивной системы
local
  state, next: INTEGER
do
  state := initial
  repeat
    execute_state (state, >next)
    -- Процедура execute_state обновляет значение next
    state := transition (state, next)
  until is_final (state) end
end
  
```

Это типичный алгоритм обхода диаграммы переходов. (Те, кто писал лексический анализатор, узнают образец.) На каждом этапе мы находимся в состоянии `state`, вначале устанавливаемом в `initial`; процесс завершается, когда состояние удовлетворяет `is_final`. Для состояний, не являющихся заключительными, вызывается `execute_state`, принимающее текущее состояние и возвращающее в аргументе `next` выбор перехода, сделанный пользователем. Функция `transition` определяет следующее состояние.

Техника, используемая в процедуре `execute_state`, изменяющая значение одного из своих аргументов, никогда не

подходит для хорошего ОО-проекта, но здесь она вполне приемлема. Для того чтобы сделать ее явной, используется "флажок" для "out" аргумента - next со стрелкой.

Для завершения проекта следует определить процедуру execute_state, описывающую действия, выполняемые в каждом состоянии. Ее тело реализует содержимое блока начальной goto-версии.

```
execute_state (in s: INTEGER; out c: INTEGER) is
    -- Выполнить действия, связанные с состоянием s,
    -- возвращая в c выбор состояния, сделанный пользователем
    local
        a: ANSWER; ok: BOOLEAN
    do
        repeat
            display (s)
            read (s, a)
            ok := correct (s, a)
            if not ok then message (s, a) end
        until ok end
        process (s, a)
        c := next_choice (a)
    end
```

Здесь вызываются программы уровня 1 со следующими ролями:

- display (s) выводит на экран панель, связанную с состоянием s;
- read (s, a) читает в a ответы пользователя, введенные в окнах панели состояния s;
- correct (s, a) возвращает true, если и только если a является приемлемым ответом; если да, то process (s, a) обрабатывает ответ a, например, обновляя базу данных или отображая некоторую информацию, если нет, то message (s, a) выводит соответствующее сообщение об ошибке.

Тип ANSWER объекта, представляющего ответ пользователя, не будет уточняться. Значение a этого типа глобально представляет ввод пользователя, включающий и выбор следующего шага (ANSWER фактически во многом подобен классу, даже если остальная структура не является объектной.)

Для получения работающего приложения необходимо задать реализации программ уровня 1: display, read, correct, message и process.

Критика решения

Получили ли мы удовлетворительное решение? Не совсем. Оно лучше, чем первая версия, но все еще далеко от заявленных целей повторного использования и расширяемости.

Статичность

Хотя с первого взгляда кажется, что нам удалось отделить общность, присущую приложениям такого типа, от специфических черт конкретного приложения, в реальности различные модули все еще тесно связаны друг с другом и с выбранным приложением. Главной проблемой остается структура данных, передаваемых в системе. Рассмотрим сигнатуры (типы аргументов и результатов) наших программ:

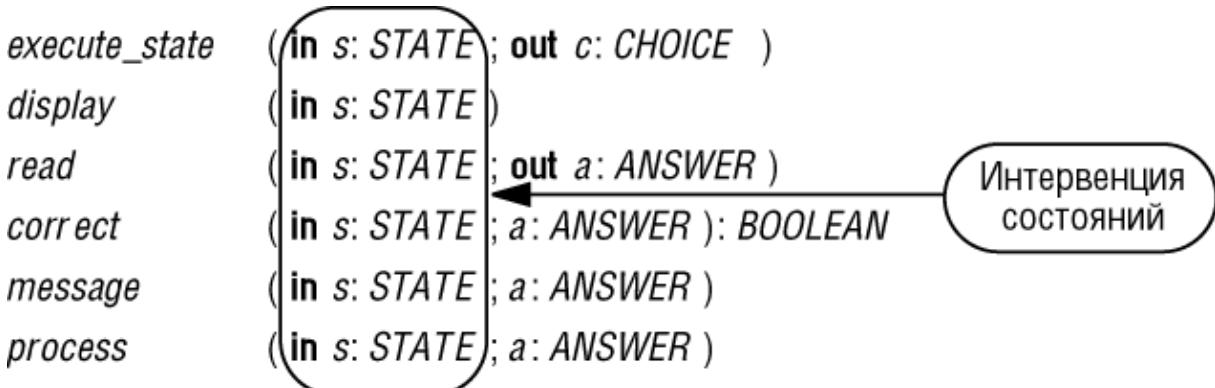


Рис. 2.4. Сигнатура программ

Замечание, звучащее подобно жалобе, состоит в том, что роль состояний всеобъемлюща. Текущее состояние s появляется как аргумент во всех программах, спускаясь с вершины execute_session, где оно известно как state. Так что кажущаяся простота и управляемость иерархической структуры, показанной на [рис. 2.3](#), является ложью или, более точно, фасадом. За спиной формальной элегантности функциональной декомпозиции стоит неразбериха передачи данных. Истинная картина должна включать данные.

В основе технологии лежит битва между функциями и данными (объектами) за управление архитектурой системы. В необъектных подходах функции берут вверх над данными, но затем данные начинают мстить.

Месть проявляется в форме саботажа. Атакуя основания архитектуры, данные не пропускают изменения, - пока, подобно правительству не способному руководить своей перестройкой (в оригинале - perestroika), система не рухнет под собственной тяжестью.

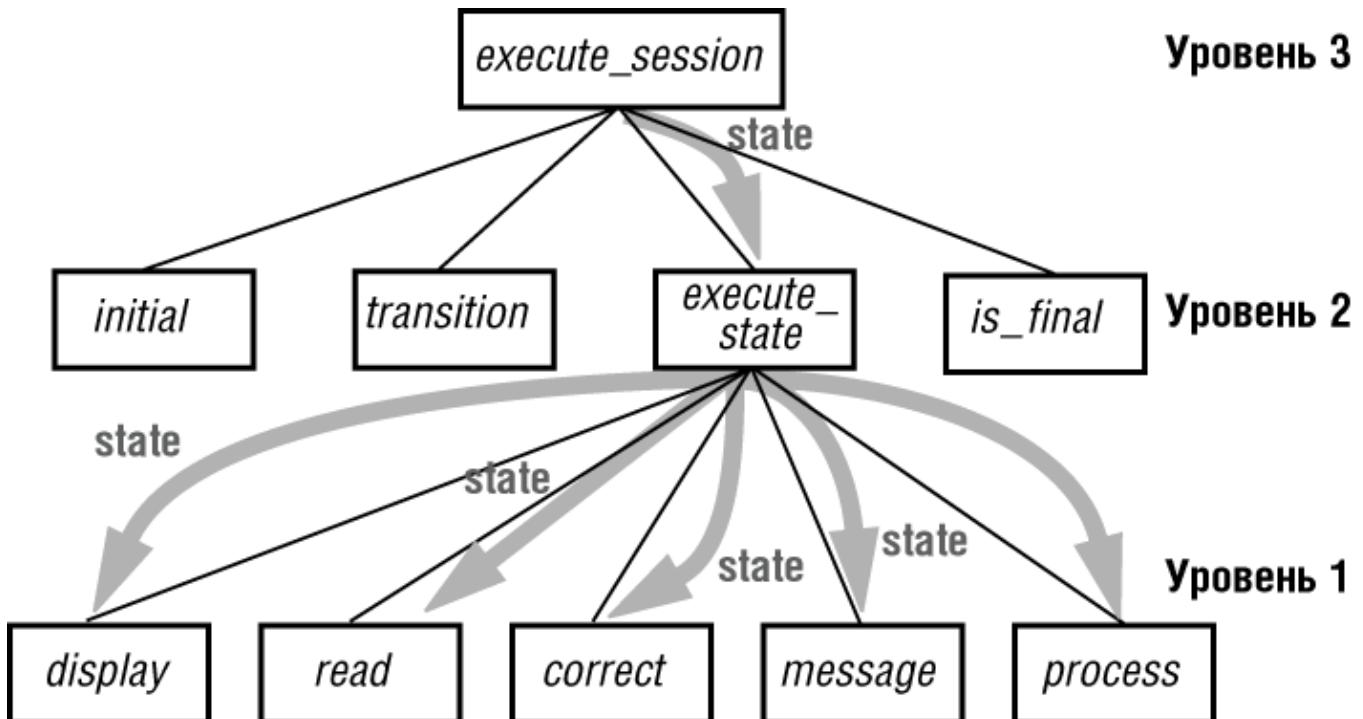


Рис. 2.5. Поток данных

В этом примере структура рушится из-за необходимости различать состояния. Все программы уровня 1 должны выполнять различные действия, зависящие от состояния s : отображать панель для некоторого состояния, читать и интерпретировать ответы пользователя, определять корректность ответов, - для всех этих задач необходимо знать состояние. Программы должны будут выполнять разбор случаев в форме:

```

inspect
  ...
when Initial then
  ...
when Enquiry_on_flights then
  ...
...
end
  
```

Это приводит к длинной и сложной структуре и, что хуже всего, к неустойчивой системе, - любое добавление состояния потребует изменения всей структуры. Имеет место типичный случай необузданного распределения знаний. Слишком много модулей системы используют одну и ту же информацию - список всех возможных состояний, являющийся предметом изменений.

Если надеяться на получение общего повторно используемого решения, то ситуация еще хуже, чем может показаться. Дело в том, что во всех программах неявно присутствует еще один аргумент - приложение - система резервирования авиабилетов или другая проектируемая система. Так что программы, такие как `display`, если они действительно носят общий характер, должны знать все состояния всех возможных приложений! Аналогично функция `transition` должна содержать графы переходов для всех приложений - совершенно нереалистическое предположение.

Объектно-ориентированная архитектура

Проблемы функциональной декомпозиции сверху вниз указывают, что нужно делать, чтобы получить хорошую ОО-версию.

Закон инверсии

Что пошло не так, в чем проблема? Слишком много передач данных свидетельствует обычно об изъянах в архитектуре ПО. Устранение этих изъянов приводит непосредственно к ОО-проекту, что находит отражение в следующем правиле проектирования:

Закон инверсии

Если программы пересыпают друг другу много данных, поместите программы в данные.

Ранее модули строились вокруг операций (таких как `execute_session` и `execute_state`) и данные распределялись

между программами со всеми неприятными последствиями, с которыми нам пришлось столкнуться. ОО-проектирование ставит все с головы на ноги, - оно использует наиболее важные типы данных как основу модульности, присоединяя каждую программу к тому типу данных, с которым она наиболее тесно связана. Когда объекты поддерживают победу, их бывшие хозяева - функции - становятся вассалами данных.

Закон инверсии является ключом получения ОО-проекта из классической функциональной (процедурной) декомпозиции, как это делается в данной лекции. Подобная необходимость возникает в случаях реверс-инженерии, когда существующая необъектная система преобразуется в объектную для обеспечения ее дальнейшей эволюции и сопровождения. К этому приему прибегают в командах, возможно, не столь хорошо знакомых с ОО-проектированием и предпочитающих начинать с привычной функциональной декомпозиции.

Конечно, желательно начинать ОО-проектирование с самого начала, - тогда не придется прибегать к инверсии. Но закон инверсии полезен не только в случае реверс-инженерии или для новичков-разработчиков. И в объектной разработке проекта на фоне объектного ландшафта могут встречаться области функциональной декомпозиции. Анализ передачи данных является хорошим способом обнаружения и корректировки подобных дефектов. Если вы обнаружили в структуре, разрабатываемой как объектной, образцы передачи данных, подобные состояниям в данном примере, то это должно привлечь ваше внимание и привести в большинстве случаев к осознанию абстракции данных, не нашедшей еще должного места в проекте.

Состояние как класс

Пример "состояния" является типичным. Такой тип данных, играющий всеобъемлющую роль в передаче данных между программами, является первым кандидатом на роль модуля в ОО-архитектуре, основанной на классах (абстрактно описанных типах данных).

Понятие состояния было важным в оригинальной постановке проблемы, но затем в функциональной архитектуре эта важность была потеряна, - состояние было представлено обычной переменной, передаваемой из программы в программу, как если бы это было существо низкого ранга. Мы уже видели, как оно отомстило за себя. Теперь мы готовы предоставить ему заслуженный статус. STATE должно быть классом, одним из владельцев структуры в нашей новой ОО-системе.

В этом классе мы найдем все операции, характеризующие состояние: отображение соответствующего экрана (display), анализ ответа пользователя (read), проверку ответа (correct), выработку сообщения об ошибке для некорректных ответов (message), обработку корректных ответов (process). Мы должны также включить сюда execute_state, выражающее последовательность действий, выполняемых всякий раз, когда сессия достигает заданного состояния (поскольку данное имя было бы сверхквалифицированным в классе, называемом STATE, заменим его именем execute).

Возвращаясь к рисунку, отражающему функциональную декомпозицию, выделим в нем множество программ, принадлежащих классу STATE.

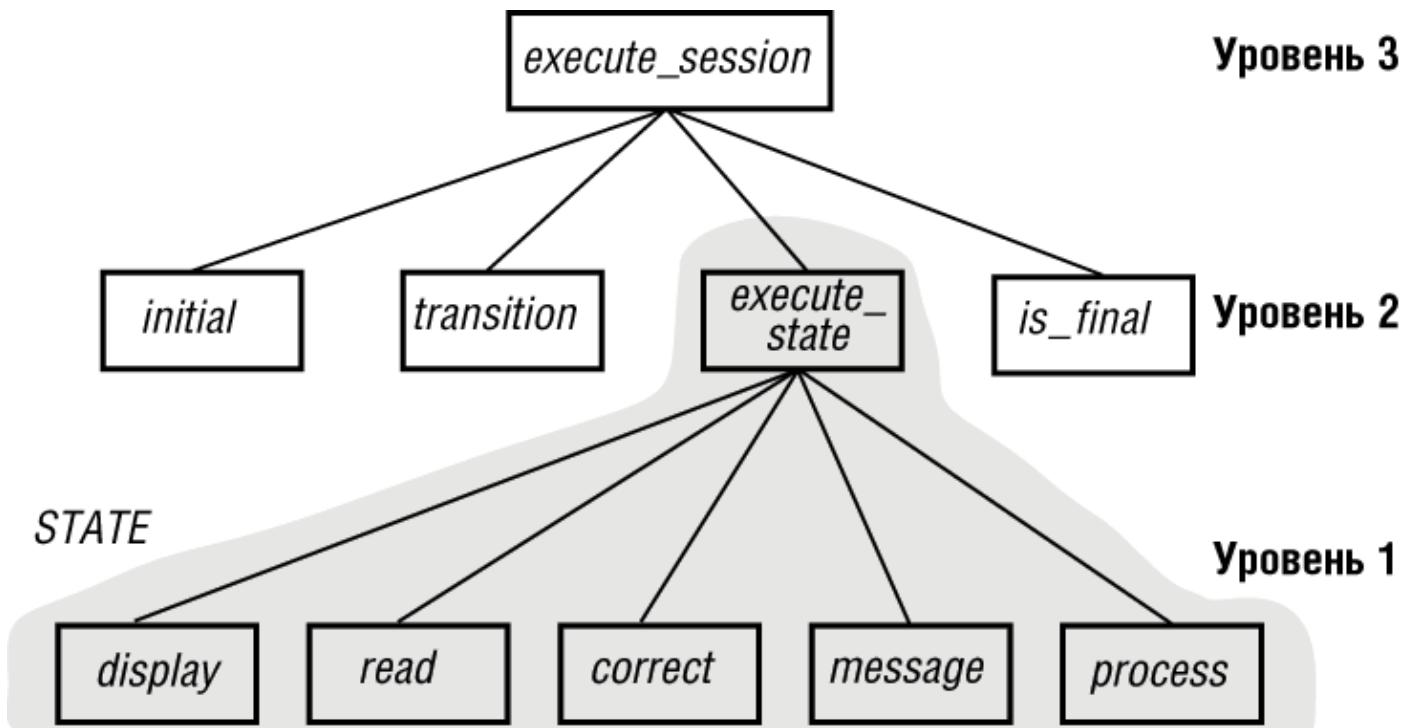


Рис. 2.6. Компоненты класса STATE

Класс имеет следующую форму:

```

...class STATE feature
  input: ANSWER
  choice: INTEGER
  
```

```

execute is do ... end
display is ...
read is ...
correct: BOOLEAN is ...
message is ...
process is ...
end

```

Компоненты `input` и `choice` являются атрибутами, остальные - подпрограммами (процедурами и функциями). В сравнении со своими двойниками при функциональной декомпозиции подпрограммы потеряли явный аргумент, задающий состояние, хотя он появится другим путем в клиентских вызовах, таких как `s.execute`.

В предыдущих подходах функция `execute` (ранее `execute_state`) возвращала пользовательский выбор следующего шага. Но такой стиль нарушает правила хорошего проектирования. Предпочтительнее сделать `execute` командой. Запрос "какой выбор сделал пользователь в последнем состоянии?" доступен благодаря атрибуту `choice`. Аналогично, аргумент `ANSWER` подпрограмм уровня 1 заменен теперь закрытым атрибутом `input`. Вот причина скрытия информации: клиентскому коду нет необходимости обращаться к ответам помимо интерфейса, обеспечиваемого компонентами класса.

Наследование и отложенные классы

Класс `STATE` описывает не частное состояние, а общее понятие состояния. Процедура `execute` - одна и та же для всех состояний, но другие подпрограммы зависят от состояния.

Наследование и отложенные классы идеально позволяют справиться с этими ситуациями. На уровне описания класса `STATE` мы знаем атрибуты и процедуру `execute` во всех деталях. Мы знаем также о существовании программ уровня 1 (`display` и др.) и их спецификации, но не их реализации. Эти программы должны быть отложенными, класс `STATE`, описывающий множество вариантов, а не полностью уточненную абстракцию, сам является отложенным классом. В результате имеем:

```

indexing
    description: "Состояния приложений, управляемых панелями"
deferred class
    STATE
feature -- Access
    choice: INTEGER
        -- Пользовательский выбор следующего шага
    input: ANSWER
        -- Пользовательские ответы на вопросы в данном состоянии
feature -- Status report
    correct: BOOLEAN is
        -- Является ли input корректным ответом?
    deferred
    end
feature -- Basic operations
    display is
        -- Отображает панель, связанную с текущим состоянием
    deferred
    end
    execute is
        -- Выполняет действия, связанные с текущим состоянием,
        -- и устанавливает choice - пользовательский выбор
    local
        ok: BOOLEAN
    do
        from ok := False until ok loop
            display; read; ok := correct
            if not ok then message end
        end
        process
    ensure
        ok
    end
    message is
        -- Вывод сообщения об ошибке, соответствующей input
    require
        not correct
    deferred
    end
    read is
        -- Получить ответы пользователя input и choice
    deferred

```

```

    end
process is
    -- Обработка input
    require
        correct
    deferred
end
end

```

Для описания специфических состояний следует ввести потомков класса STATE, задающих отложенную реализацию компонент.

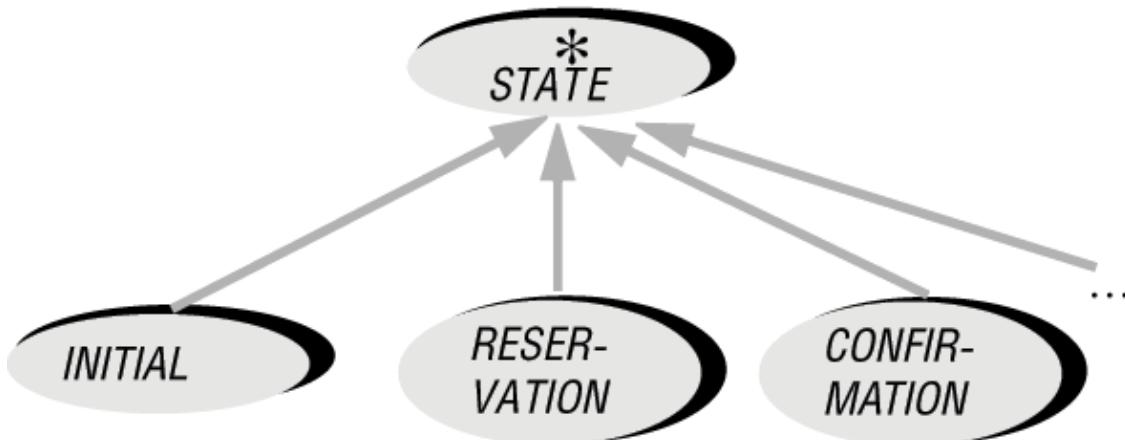


Рис. 2.7. Иерархия классов State

Пример мог бы выглядеть следующим образом:

```

class ENQUIRY_ON_FLIGHTS inherit
    STATE
feature
    display is
        do
            ...Специфическая процедура вывода на экран...
        end
    ...И аналогично для read, correct, message и process ...
end

```

Эта архитектура отделяет зерно от шелухи: элементы, общие для всех состояний, отделяются от элементов, специфичных для конкретного состояния. Общие элементы, такие как процедура execute, сосредоточены в классе STATE и нет необходимости в их повторном объявлении в потомках, таких ENQUIRY_ON_FLIGHTS. Принцип Открыт-Закрыт выполняется: класс STATE закрыт, поскольку он является хорошо определенным, компилируемым модулем, но он также открыт, так как можно добавлять в любое время любых его потомков.

Класс STATE является типичным представителем **поведенческих классов** (behavior classes, см. [лекцию 14](#) курса "Основы объектно-ориентированного программирования") - отложенных классов, задающих общее поведение большого числа возможных объектов, реализующих то, что полностью известно на общем уровне (execute) в терминах, зависящих от каждого варианта. Наследование и отложенный механизм задают основу представления такого поведения повторно используемых компонентов.

Описание полной системы

Для завершения проекта следует заняться управлением сессией. При функциональной декомпозиции эту задачу выполняла процедура execute_session - главная программа. Но мы знаем, как сделать это лучшим образом. Как ранее говорилось (см. [лекцию 5](#) курса "Основы объектно-ориентированного программирования") главная функция системы, позиционируемая как верхняя функция в проектировании сверху вниз, - это нечто мифическое. Большие программные системы выполняют множество одинаково важных функций. И здесь основанный на АТД подход является предпочтительным. Вся система в целом рассматривается как множество абстрактных объектов, способных выполнять ряд служб (services).

Ранее мы рассмотрели одну ключевую абстракцию - STATE. Какая же абстракция в нашем рассмотрении осталась пропущенной? Ответ очевиден: центральным в нашей системе является понятие APPLICATION, описывающее специфическую интерактивную систему, подобную системе резервирования билетов. Это приводит нас к следующему классу.

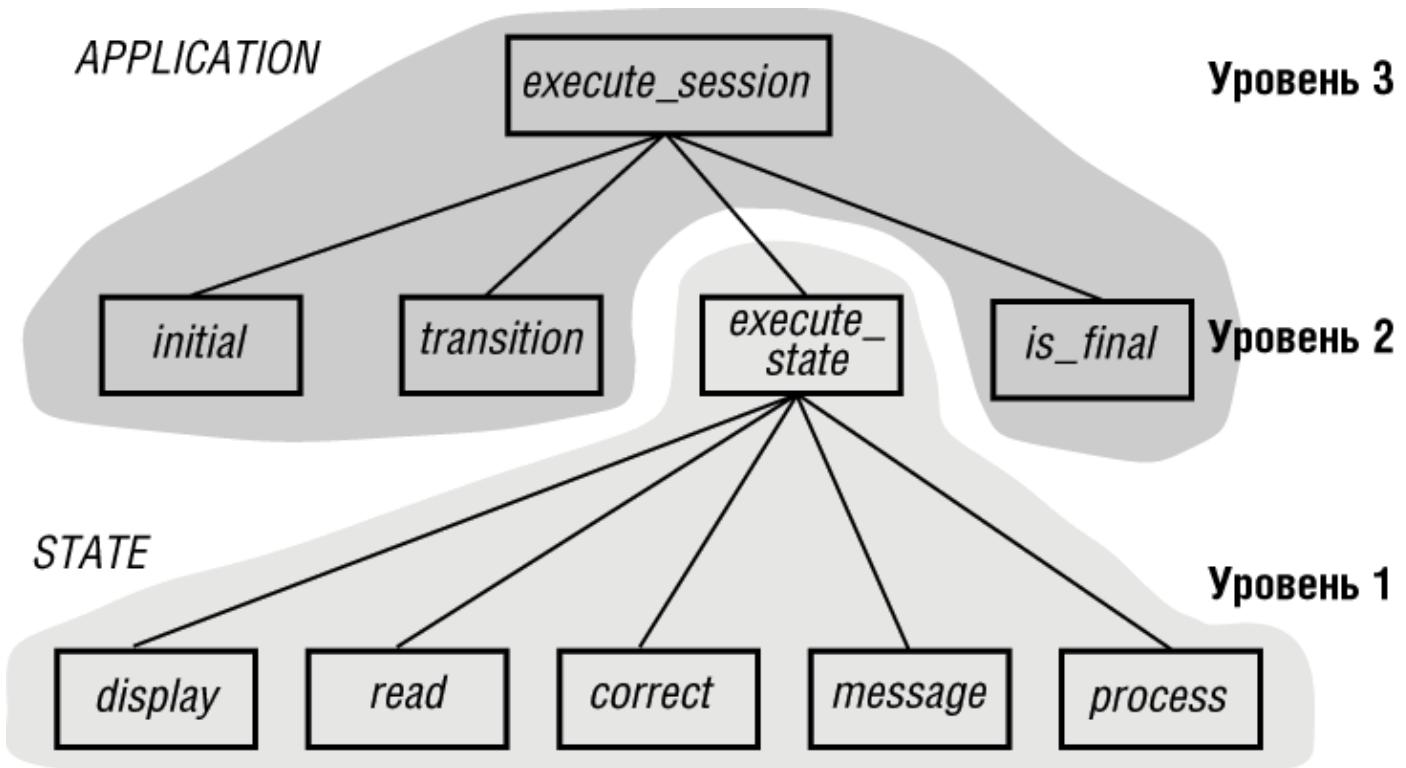


Рис. 2.8. Компоненты классов State и Application

Заметьте, все не вошедшие в класс STATE программы функциональной декомпозиции стали теперь компонентами класса APPLICATION:

- `Execute_session` - описывает, как выполнять сессию, ее имя теперь разумно упростить и называть просто `execute`, так как дальнейшую квалификацию обеспечивает имя класса.
- `Initial` и `is_final` - указывают, какие состояния имеют специальный статус в приложении. Конечно же, их разумно включить именно в класс `APPLICATION`, а не в класс `STATE`, поскольку они описывают свойства приложения, а не состояния, которое является заключительным или начальным только по отношению к приложению, а не само по себе. При повторном использовании состояние, бывшее заключительным в одном приложении вполне может не быть таковым для другого приложения.
- `Transition` - описывает переходы между состояниями приложения.

Все компоненты функциональной декомпозиции нашли свое место в ОО-декомпозиции: одни в классе `STATE`, другие в `APPLICATION`. Это не должно нас удивлять. Объектная технология, о чем многократно говорится в этой книге, является прежде всего архитектурным механизмом, в первую очередь предназначенным для организации программных элементов в согласованные структуры. Сами элементы, возможно, нижнего уровня, те же самые или похожие на элементы необъектных решений. Объектные механизмы: абстракция данных, скрытие информации, утверждения, наследование, полиморфизм, динамическое связывание позволяют сделать задачу проектирования более простой, общей и мощной.

Системе управления панелями, изучаемой в данной лекции, всегда необходимы: процедура обхода графа приложения (`execute_session`, теперь просто `execute`), чтение ввода пользователя (`read`), обнаружение заключительного состояния (`is_final`). Погружаясь в структуру, можно найти одни и те же элементы, независимо от выбранного подхода к проектированию. Что же меняется? - способ группирования элементов, создающий модульную архитектуру.

Конечно, нет необходимости ограничивать себя элементами, пришедшими из предыдущих решений. То, что для функционального решения было завершением процесса - построение функции `execute` и всего необходимого для ее работы - теперь становится только началом. Существует много других вещей, которые хотелось бы выполнять для подобных приложений:

- добавить новое состояние;
- добавить новый переход;
- построить приложение (Многократно повторяя комбинацию предшествующих двух операций);
- удалить состояние, переход;
- сохранить законченное приложение, его состояние и переходы в базе данных;
- промоделировать работу приложения (например, с заглушками для программ, проверяя работу только переходов);
- мониторинг использования приложения.

Все эти операции и другие будут в равной степени являться компонентами класса `APPLICATION`. Здесь нет более или менее важных программ, чем наша бывшая "главная программа", - процедура `execute`, ставшая теперь обычным компонентом класса, равная среди других, но не первая. Устранив понятие вершины, мы подготовливаем систему к эволюции и повторному использованию.

Класс приложения

Для завершения рассмотрения класса APPLICATION рассмотрим несколько возможных реализаций решений:

- Будем нумеровать состояния приложения числами от 1 до n. Заметьте, эти числа не являются абсолютными свойствами состояний, они связаны с определенным приложением, поэтому в классе STATE нет атрибута "номер состояния". Вместо этого, одномерный массив associated_state, атрибут класса APPLICATION, задает состояние, связанное с заданным номером.
- Представим функцию переходов transition еще одним атрибутом - двумерным массивом размерности n * m, где m - число возможных пользовательских выборов при выходе из состояния.
- Номер начального состояния хранится в атрибуте initial и устанавливается в подпрограмме choose_initial. Для конечных состояний мы используем соглашение, что переход в псевдосостояние 0 означает завершение сессии.
- Процедура создания в классе APPLICATION использует процедуры создания библиотечных классов ARRAY и ARRAY2. Последний описывает двумерные массивы и построен по образцу ARRAY; его процедура создания make принимает четыре аргумента, например create a.make (1, 25, 1, 10), а его подпрограммы item и put используют два индекса - a.put (x, 1, 2). Границы двумерного массива a можно узнать, вызвав a.lower1 и так далее.

Вот определение класса, использующего эти решения:

```
indexing
  description: "Интерактивные приложения, управляемые панелями"
class APPLICATION creation
  make
feature -- Initialization
  make (n, m: INTEGER) is
    -- Создает приложение с n состояниями и m возможными выборами
    do
      create transition.make (1, n, 1, m)
      create associated_state.make (1, n)
    end
feature -- Access
  initial: INTEGER
    -- Номер начального состояния
feature -- Basic operations
  execute is
    -- Выполняет сессию пользователя
    local
      st: STATE; st_number: INTEGER
    do
      from
        st_number := initial
      invariant
        0 <= st_number; st_number <= n
      until st_number = 0 loop
        st := associated_state.item (st_number)
        st.execute
      -- Вызов процедуры execute класса STATE.
      -- (Комментарии к этой ключевой инструкции даны в тексте.)
        st_number := transition.item (st_number, st.choice)
      end
    end
  feature -- Element change
    put_state (st: STATE; sn: INTEGER) is
      -- Ввод состояния st с индексом sn
      require
        1 <= sn; sn <= associated_state.upper
      do
        associated_state.put (st, sn)
      end
    choose_initial (sn: INTEGER) is
      -- Определить состояние с номером sn в качестве начального
      require
        1 <= sn; sn <= associated_state.upper
      do
        initial := sn
      end
    put_transition (source, target, label: INTEGER) is
      -- Ввести переход, помеченный label,
      -- из состояния с номером source в состояние target
      require
```

```

1 <= source; source <= associated_state.upper
0 <= target; target <= associated_state.upper
1 <= label; label <= transition.upper2
do
    transition.put (source, label, target)
end
feature {NONE} -- Implementation
    transition: ARRAY2 [STATE]
    associated_state: ARRAY [STATE]
    ... Другие компоненты ...
invariant
    transition.upper1 = associated_state.upper
end -- class APPLICATION

```

Обратите внимание на простоту и элегантность вызова `st.execute`. Компонент `execute` класса STATE является эффективным (полностью реализованным) поскольку описывает известное общее поведение состояний, но его реализация основана на вызове компонентов: `read`, `message`, `correct`, `display`, `process`, отложенных на уровне STATE, эффективизация которых выполняется потомками класса, такими как `RESERVATION`. Когда мы помещаем вызов `st.execute` в процедуру `execute` класса APPLICATION, у нас нет информации о том, какой вид состояния обозначает `st`, но благодаря статической типизации мы точно знаем, что это состояние. Далее включается механизм динамического связывания и в период исполнения `st` становится связанной с объектом конкретного вида, например `RESERVATION`, - тогда вызовы `read`, `message` и других царственных особ автоматически будут переключаться на нужную версию.

Значение `st`, полученное из `associated_state`, представляет полиморфную структуру данных (**polymorphic data structure**), содержащую объекты разных типов, все из которых согласованы (являются потомками) со STATE. Текущий индекс `st_number` определяет операции состояния.

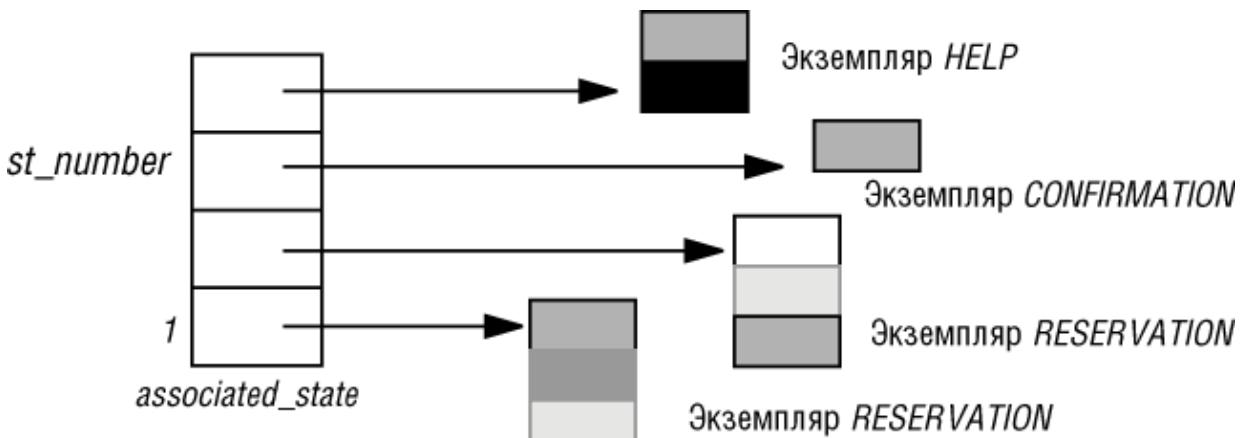


Рис. 2.9. Полиморфный массив состояний

Вот как строится интерактивное приложение. Приложение должно быть представлено сущностью, скажем `air_reservation`, класса APPLICATION. Необходимо создать соответствующий объект:

```
create air_reservation.make (number_of_states, number_of_possible_choices)
```

Далее независимо следует определить и создать состояния приложения, как сущности классов-потомков STATE, либо новые, либо уже готовые и взятые из библиотеки повторного использования. Каждое состояние `s` связывается с номером `i` в приложении:

```
air_reservation.put_state (s, i).
```

Затем одно из состояний выбирается в качестве начального:

```
air_reservation.choose_initial (i0)
```

Для установления перехода от состояния `sn` к состоянию с номером `tn`, с меткой `l` используйте вызов:

```
air_reservation.enter_transition (sn, tn, l)
```

Это включает и заключительные состояния, для которых по умолчанию `tn` равно 0. Затем можно запустить приложение:

```
air_reservation.execute_session.
```

При эволюциях системы можно в любой момент использовать те же подпрограммы для добавления состояний и переходов.

Конечно же, можно расширить класс APPLICATION, изменяя сам класс или добавляя новых потомков, включив новые

функциональные возможности - удаление, моделирование или любые другие.

Обсуждение

Этот пример, надеюсь, показал впечатляющую картину той разницы, которая существует между ОО-конструированием ПО и ранними подходами. В частности, он показал преимущества, получаемые при устраниении понятия главной программы. Сосредоточившись на понятии абстракции данных, забывая столь долго, пока это еще возможно, о том, что является главной функцией системы, мы получаем структуру, более подготовленную к будущим изменениям и повторному использованию в разнообразных вариантах.

Этот стабилизирующий эффект является одним из характеристических свойств Метода. Он предполагает некоторую дисциплину применения, поскольку такой способ проектирования постоянно наталкивается на сопротивление и естественное желание спросить, "А что же делает система?". Это один из тех навыков, по которому можно отличить ОО-профессионала от людей, которые не проникли в суть Метода, хотя и могут использовать ОО-язык программирования и объектную технику, но за объектным фасадом их систем по-прежнему стоит функциональная архитектура.

Как показано в этой лекции, идентифицировать ключевые абстракции часто удается, анализируя передачу данных и обращая внимание на те понятия, что чаще других используются в коммуникациях между компонентами системы. Часто это является прямым указанием на обращение ситуации - присоединение программ к абстрактным данным.

Заключительный урок этой лекции состоит в том, что следует быть осторожным и не придавать слишком большого значения тому факту, что ОО-системы выведены непосредственно путем моделирования "реального мира". Моделирующая мощь метода и в самом деле впечатляющая, и вполне приятно создавать программную архитектуру, чьи принципиальные компоненты непосредственно отражают абстракции внешней моделируемой системы. Но построить модель реального мира можно разными способами, не все они приводят к хорошей программной системе. Наша первая, goto версия была также близка к реальному миру, как и две другие, - фактически даже ближе, поскольку ее структура была построена по образцу диаграммы переходов системы, в то время как другие версии вводили промежуточные понятия. Но результат с точки зрения программной инженерии является катастрофическим.

Созданная в конечном итоге ОО-декомпозиция хороша из-за использования абстракций: STATE, APPLICATION, ANSWER - все они являются ясными, общими, управляемыми, готовыми к изменениям и повторному использованию в широкой области применения. Вы понимаете, что эти абстракции столь же реальны, как и все остальное, но новичку они могут казаться менее естественными, чем концепции, используемые в ранее изучаемых решениях.

При создании хорошего ПО следует учитывать не его близость к реальному миру, а то, насколько выбранные абстракции хороши как для моделирования внешней системы, так и для построения структуры ПО. Фактически в этом суть ОО-анализа, проектирования и реализации - работа, которую для успеха проекта необходимо выполнять хорошо как сегодня, так и завтра. Профессионала от любителя отличает умение находить правильные абстракции.

Библиографические замечания

Варианты примера, обсуждаемого в этой лекции, использовались для иллюстрации ОО-концепций в [М 1983] и [М 1987].

Основы объектно-ориентированного проектирования

3. Лекция: Наследование: "откат" в интерактивных системах

В нашем втором примере займемся задачей, возникающей перед проектировщиками почти любой интерактивной системы, - как обеспечить возможность "отката" команд. Покажем, что наследование и динамическое связывание позволяют получить простое и общее решение довольно сложной и многогранной проблемы.

Проделки дьявола

Человеку свойственно ошибаться - чтобы окончательно все запутать, дайте ему компьютер. Чем быстрее становятся наши интерактивные системы, тем проще выполнить совсем не желанные действия. Вот почему хотелось бы иметь способ стереть прошлое, но не "большой красной кнопкой", стирающей все, - одной из компьютерных шуток, а иметь Большую Зеленую Кнопку, нажатие которой избавляет нас от сделанных ошибок.

Откаты для пользы и для забавы

В интерактивных системах эквивалентом Большой Зеленой Кнопки является операция отката Undo, дающая пользователю системы возможность отменить действие последней выполненной команды.

Исходная цель механизма отката - справиться с потенциально опасными ошибками ввода (напечатан не тот символ, нажата не та кнопка). Откат позволяет достичь большего. Помимо освобождения от нервного напряжения и боязни сделать что-то не то, он поощряет использовать стратегию "**Что-Если**", - стиль взаимодействия, при котором пользователи сознательно испытывают различные варианты ввода, анализируя полученные результаты, зная при этом, что всегда есть возможность вернуться к предыдущему состоянию.

Каждая хорошая интерактивная система должна обеспечивать подобный механизм. (По этой причине на клавиатуре моего компьютера есть кнопка Undo, хотя она вовсе не зеленая и не особенно большая. Жаль только, что не все разработчики ПО предусматривают ее использование.)

Многоуровневый откат и повтор: undo и redo

В некоторых системах откат ограничен одним уровнем. Если не делать двух ошибок подряд, то этого достаточно. Но если вы пошли не по той дороге и хотите вернуться назад, то нужен многоуровневый откат.

Нет никаких причин ограничиваться одним уровнем. Как только механизм отката разработан, распространение его на несколько уровней не представляет особого труда, что и будет показано в этой лекции. И, пожалуйста, говорю уже как потенциальный пользователь, не ограничивайте число уровней отката, а если уж вынуждены это сделать, то пусть ограничение задает сам пользователь, во всяком случае по умолчанию оно должно быть не меньше 20. Затраты на откат невелики, если применять описываемую здесь технику.

Многоуровневый откат может быть избыточным. Потому необходима независимая операция повтора Redo, в которой нет нужды, если откат ограничен одним шагом.

Практические проблемы

Хотя при разумных усилиях механизм undo-redo может быть встроен в любую хорошо написанную ОО-систему, лучше всего с самого начала планировать его использование. На архитектуре ПО это скажется введением класса command, что может и не прийти в голову, если не думать об откате при проектировании системы.

Практический механизм undo-redo требует учета нескольких требований. Это свойство следует включить в интерфейс пользователя. Для начала можно полагать, что множество доступных операций обогащено двумя новыми командами: Undo и Redo. (Для них может быть введена соответствующая комбинация горячих клавиш, например **control-U** и **control-R**.) Команда Undo отменяет эффект последней еще не отмененной команды; Redo повторно выполняет команду, отмененную при откате. Следует определить соглашения для попыток отката на большее число шагов, чем их было сделано первоначально, при попытках повтора, когда не было отката, - такие запросы можно игнорировать или выдавать предупреждающее сообщение. (Таков возможный взгляд на интерфейс пользователя, поддерживающий undo-redo. В конце лекции мы увидим, что возможен лучший вариант интерфейса.)

Второе, что следует учитывать, действия не всех команд могут быть отменены. В некоторых ситуациях этого нельзя сделать фактически: после выполнения команды "запуск ракет" (которую может отдать, как известно, лишь президент) или менее драматичной "отпечатать страницу" действия этих команд необратимы. В других ситуациях эффект от действия команды может быть устранен, но ценой слишком больших усилий. Например, текстовые редакторы, как правило, не позволяют отменить действия команды Save, записывающей текущее состояние документа в файл. Реализация отката должна учитывать наличие таких необратимых команд, четко указывая их статус в интерфейсе пользователя. Ограничите необратимые команды случаями, для которых оправдание свойства может быть сформулировано в терминах пользователя.

Контрпримером является часто используемое мной приложение, обрабатывающее документы, которое изредка сообщает, что запрашиваемую команду нельзя отменить, хотя причины этого ясны только самой программе.

Интересно, что в каком-то смысле это утверждение ложно, - фактически вы **можете** отменить эффект команды, но не через Undo, а через команду "Вернуться к последней сохраненной версии документа". Это наблюдение приводит к следующему правилу: всякий раз, когда команду можно следует признать необратимой, не поступайте, как в приведенном выше примере, выводя сообщение "Эта команда будет необратимой". Вместо выбора двух возможностей - **Continue anyway** и **Cancel** - предоставьте пользователю **три**: сохранить документ и затем выполнить команду, выполнить без сохранения, отмена команды.

Наконец, можно попытаться предложить общую схему "Undo, Skip, Redo", позволяющую после нескольких операций Undo пропустить некоторые команды перед включением Redo. Интерфейс пользователя, показанный в конце этой лекции, поддерживает такое расширение, но возникают концептуальные проблемы: после пропуска некоторых команд может оказаться невозможным выполнить следующую команду. Рассмотрим тривиальный пример текстового редактора и сессию некоторого пользователя с набранной одной строкой текста. Предположим, пользователь выполнил две команды:

- (1) Добавить строку в конец текста.
- (2) Удалить вторую строку.

После отмены обеих команд пользователь захотел пропустить выполнение первой и повторно выполнить только вторую (skip (1) и redo (2)). К несчастью, в этом состоянии выполнение команды (2) бессмысленно, поскольку нет второй строки. Эта проблема не столько интерфейса, сколько реализации: команда "Удалить вторую строку" была применима к структуре объекта, полученного в результате выполнения команды (1), ее применение к структуре, предшествующей выполнению (1), может быть невозможным или приводить к непредсказуемым результатам.

Требования к решению

Механизм undo-redo, который мы намереваемся обеспечить, должен удовлетворять следующим свойствам:

- **U1** Механизм должен быть применим к широкому классу интерактивных приложений независимо от их проблемной области.
- **U2** Механизм не должен требовать перепроектирования при добавлении новых команд.
- **U3** Он должен разумно использовать ресурсы памяти.
- **U4** Он должен быть применим к откатам как на один, так и несколько уровней.

Первое требование следует из того, что ничего проблемно специфического в откатах и повторах нет. Только для облегчения обсуждения мы будем использовать в качестве примера знакомый каждому инструмент - текстовый редактор, (подобный Notepad или Vi), позволяющий пользователям вводить тексты и выполнять такие команды, как: **INSERT_LINE**, **DELETE_LINE**, **GLOBAL_REPLACEMENT** (одного слова в тексте другим) и другие. Но это только пример, и ни одна из концепций, обсуждаемых ниже, не является характерной только для текстовых редакторов.

Второе требование означает, что Undo и Redo имеют особый статус и не могут рассматриваться подобно любым другим командам интерактивной системы. Будь Undo обычной командой, ее структура **требовала** бы разбора случаев в форме:

```
If "Последняя команда была INSERT_LINE" then
    "Undo эффект INSERT_LINE"
elseif "Последняя команда была DELETE_LINE" then
    "Undo эффект DELETE_LINE"
и т.д.
```

Мы знаем (см. [лекцию 3](#) курса "Основы объектно-ориентированного программирования"), как плохи такие структуры, противоречащие принципу Единственного Выбора и затрудняющие расширяемость системы. Пришлось бы изменять программный текст при всяком добавлении новой команды. Хуже того, код каждой ветви отражал бы код соответствующей команды, например, первая ветвь должна бы знать достаточно много о том, что делает команда **INSERT_LINE**. Это было бы свидетельством изъянов проекта.

Третье требование заставляет нас бережно относиться к памяти. Понятно, что механизм undo-redo требует хранения **некоторой** информации для каждой команды Undo: например, при выполнении **DELETE_LINE**, нет возможности выполнить откат, если перед выполнением команды не запомнить где-нибудь удаляемую строку и ее позицию в тексте. Но следует хранить только то, что логически необходимо.

Вследствие третьего требования исключается такое очевидное решение, как сохранение полного состояния системы перед выполнением каждой команды. Такое решение можно было бы trivialно написать, используя свойства **STORABLE** (см. [лекцию 8](#) курса "Основы объектно-ориентированного программирования"), но оно было бы нереалистичным, так как просто пожирало бы память. Нужно придумать что-то более разумное.

Последнее требование поддержки произвольного числа уровней отката уже обсуждалось. В данном случае

оказывается проще рассмотреть откат на один уровень и затем обобщить решение на произвольное число уровней.

Этими требованиями заканчивается презентация проблемы. Хорошой идеей, как обычно, является попытка самостоятельного поиска решения, прежде чем продолжить чтение этой лекции.

Поиск абстракций

Ключом ОО-решения является поиск правильных абстракций. Здесь фундаментальное понятие буквально напрашивается.

Класс Command

Для нашей проблемы характерна фундаментальная абстракция данных COMMAND, представляющая любую операцию, отличающуюся от Undo и Redo. Выполнение операции это лишь один из многих компонентов, применимых к команде, - команду можно сохранить, тестировать или отменить. Так что нам понадобится класс и вот его первоначальная форма:

```
deferred class COMMAND feature
    execute is deferred end
    undo is deferred end
end
```

Класс COMMAND описывает абстрактное понятие команды и потому должен оставаться отложенным. Фактические типы команды будут представлены эффективными потомками этого класса, такими как:

```
class LINE_DELETION inherit
    COMMAND
feature
    deleted_line_index: INTEGER
    deleted_line: STRING
    set_deleted_line_index (n: INTEGER) is
        -- Устанавливает n номер следующей удаляемой строки
        do
            deleted_line_index := n
        end
    execute is
        -- Удаляет строку
        do
            "Удалить строку с номером deleted_line_index"
            "Записать текст удаляемой строки в deleted_line"
        end
    undo is
        -- Восстанавливает последнюю удаляемую строку
        do
            "Поместить deleted_line в позицию deleted_line_index"
        end
end
```

Аналогичный класс строится для каждой команды класса.

Что же представляют собой такие классы? Экземпляр LINE_DELETION, как будет показано ниже, является небольшим объектом, несущим всю необходимую информацию, связанную с выполнением команды: строку, подлежащую удалению, (deleted_line) и ее индекс в тексте (deleted_line_index). Эта информация необходима для выполнения команды undo, если она потребуется, или для повтора redo.



Рис. 3.1. Объект command

Атрибуты, такие как `deleted_line` и `deleted_line_index`, у каждой команды будут свои, но всегда они должны быть достаточными для поддержки локальных операций `execute` и `undo`. Объекты, концептуально описывающие разницу между двумя состояниями приложения: предшествующим и последующим за выполнением команды, дают возможность удовлетворить требование **U3** из нашего списка - хранить только то, что строго необходимо.

Структура наследования классов выглядит следующим образом:

*execute**

*undo**

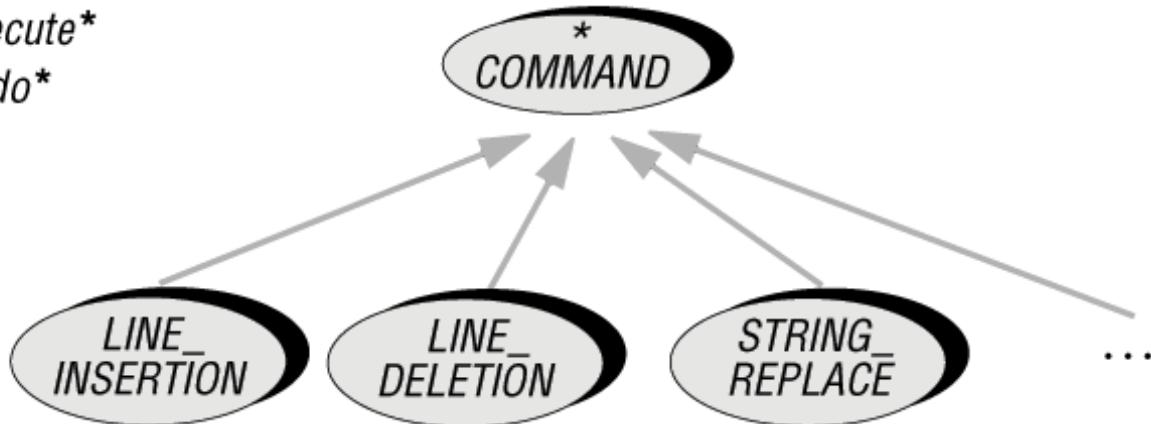


Рис. 3.2. Иерархия классов COMMAND

Граф показан плоским (все потомки COMMAND находятся на одном уровне), но ничто не мешает добавить некую структуру, группируя команды по типам, где каждая категория может иметь общие специфические черты.

При определении понятия важно указать, какие характеристики оно не покрывает. Здесь концепция команды не включает Undo и Redo; например, не имеет смысла выполнять откат самого Undo (если только не иметь в виду выполнение Redo). По этой причине в обсуждении используется термин **операция (operation)** для Undo и Redo и слово **команда (command)** для операций, допускающих откат и повтор, подобных вставке строки. Нет необходимости в классе, покрывающем понятие операции, так как такие операции, как Undo, имеют только одно связанное с ними свойство - быть выполненными.

Это хороший пример ограничений упрощенного подхода к "поиску объектов", подобному известному методу "Подчеркивание существительных", идея, изучаемая в последней лекции. В спецификациях проблемы существительные command и operation одинаково важны; но одно приводит к фундаментальному классу, второе - вообще не дает класса. Только изучение абстракций в терминах применимых операций и свойств может помочь в поиске классов проектируемой ОО системы.

Основной интерактивный шаг

Вначале посмотрим, как выглядит поддержка отката одного уровня. Обобщение на произвольное число уровней будет сделано позже.

В любой интерактивной системе в модуле, ответственном за коммуникацию с пользователем, должен быть некоторый фрагмент следующего вида:

```

basic_interactive_step is
  -- Декодирование и выполнение одного запроса пользователя
  do
    "Определить, что пользователь хочет выполнить"
    "Выполнить это (если возможно)"
  end
  
```

В традиционных структурированных системах, подобных редактору, эти операции будут частью цикла - базисного цикла программы:

```

from start until quit_has_been_requested_and_confirmed loop
  basic_interactive_step
end
  
```

где более сложные системы могут использовать событийно-управляемую схему, в которой цикл является внешним по отношению к системе и управляет системной графической оболочкой. Но во всех случаях существует нечто подобное процедуре basic_interactive_step.

С учетом наших абстракций тело процедуры можно уточнить следующим образом:

```

"Получить последний запрос пользователя"
"Декодировать запрос"
if "Запрос является нормальной командой (не Undo)" then
  "Определить соответствующую команду в системе"
  "Выполнить команду"
elseif "Запрос это Undo" then
  if "Есть обратимая команда" then
    "Undo последней команды"
  elseif "Есть команда для повтора" then
    ...
  end
end
  
```

```

    "Redo последней команды"
end
else
    "Отчет об ошибочном запросе"
end

```

Здесь реализуется соглашение, что Undo примененное сразу после Undo, означает Redo. Запрос Undo или Redo игнорируется, если нет возможности отката или повтора. В простом текстовом редакторе с клавиатурным интерфейсом, процедура "Декодировать запрос" будет анализировать ввод пользователя, отыскивая такие коды, как **control-I** (для вставки строки, **control-D** для удаления) и другие. В графическом интерфейсе будет проверяться выбор команды меню, нажатие кнопки или соответствующих клавиш.

Сохранение последней команды

Располагая понятием объекта **command**, можно добавить специфику в выполняемые операции, введя атрибуты:

```

requested: COMMAND
--Команда, запрашиваемая пользователем

```

Атрибут задает последнюю команду, подлежащую выполнению, отмене или повтору. Это позволяет уточнить нашу схему следующим образом:

```

"Получить и декодировать последний запрос пользователя"
if "Запрос является нормальной командой (не Undo)" then
    "Создать подходящий объект command и присоединить его к requested"
        -- requested создан как экземпляр некоторого потомка
        -- класса COMMAND, такого как LINE_DELETION.
        -- (Эта инструкция детализируется ниже.)
else
    requested.execute; undoing_mode := False
elseif "Запрос является Undo" and requested /= Void then
    if undoing_mode then
        "Это Redo; детали оставляем читателям"
    else
        requested.undo; undoing_mode := True
    end
else
    "Ошибкаочный запрос: вывод предупреждения или игнорирование"
end

```

Булева сущность `undoing_mode` определяет, была ли Undo последней операцией. В этом случае непосредственно следующий запрос Undo будет означать Redo, хотя непосредственные детали остаются за читателем, (упражнение УЗ.2); мы увидим полную реализацию Redo в более интересном случае многоуровневого механизма.

Информация, сохраняемая перед каждым выполнением команды, задается в экземпляре некоторого потомка `COMMAND`, такого как `LINE_DELETION`. Это означает, что, как и анонсировалось, решение удовлетворяет свойству **УЗ** в списке требований: хранится не все состояние, а только разница между новым состоянием и предыдущим.

Ключом решения - и его уточнением в оставшейся части лекции - является полиморфизм и динамическое связывание. Атрибут `requested` полиморфен: объявленный как `COMMAND` он присоединяется к объектам одного из эффективных потомков, таким как `LINE_INSERTION`. Вызовы `requested.execute` и `requested.undo` осмыслены из-за динамического связывания: подключаемый компонент должен быть версией, определенной в соответствующем классе, выполняя, например, откат `LINE_INSERTION`, `LINE_DELETION` или команду любого другого типа, определенного тем объектом, к которому присоединен `requested` во время вызова.

Действия системы

Ни одна из рассмотренных частей структуры не зависела до сих пор от специфики приложения. Фактические операции приложения, основанные на структурах специфических объектов, например, структурах, представляющих текст в текстовом редакторе, - находятся где-то в другом месте. Как же осуществляется соединение?

Ответ основан на процедурах `execute` и `undo` классов `command`, которые должны вызывать компоненты специфические для приложения. Например, процедура `execute` класса `LINE_DELETION` должна иметь доступ к классам, специфическим для текстового редактора, чтобы вызывать компоненты, вырабатывающие текст конкретной строки, задающие ее позицию в тексте.

Результатом является четкое разделение части системы, обеспечивающей взаимодействие с пользователем, и той ее части, которая зависит от специфики приложения. Вторая часть близка к концептуальной модели приложения - обработке текстов, CAD-CAM или чему-нибудь еще. Первая часть, особенно с учетом механизма истории действий пользователя, будет, как поясняется, широко использоваться в самых разных областях приложения.

Как создается объект command

После декодирования запроса система должна создать соответствующий объект command. Инструкцию, абстрактно появившуюся как "Создать подходящий объект command и присоединить его к requested", можно теперь выразить более точно, используя инструкцию создания:

```
if "Запрос является LINE INSERTION" then
    create {LINE_INSERTION} requested.make (input_text, cursor_index)
elseif "Запрос является LINE DELETION" then
    create {LINE_DELETION} requested.make (current_line, line_index)
elseif
...
...
```

Используемая здесь форма инструкции создания `create {SOME_TYPE}` x создает объект типа `SOME_TYPE` и присоединяет его к x. Тип `SOME_TYPE` должен соответствовать типу объявления x. Это имеет место в данном случае, так как `requested` имеет тип `COMMAND` и все классы команд являются потомками `COMMAND`.

Если каждый тип команды использует `unique`, то слегка упрощенная форма предыдущей записи может использовать `inspect`:

```
inspect
    request_code
when Line_insertion then
    create {LINE_INSERTION} requested.make (input_text, cursor_position)
и т.д.
```

Обе формы являются ветвящимся множественным выбором, но они не нарушают принцип Единственного Выбора. Как отмечалось при его обсуждении, если система предусматривает выбор, то некоторая часть системы **должна** знать полный список альтернатив. Оба рассмотренных варианта задают точку единственного выбора. Принцип запрещает лишь распространение этого знания на большое число модулей. В данном случае нет никакой другой части системы, которой нужен был бы доступ к списку команд; каждый командный класс имеет дело лишь с одной командой.

Фактически можно получить более элегантное решение и полностью избавиться от разбора случаев. Мы увидим его в конце презентации.

Многоуровневый откат и повтор: UNDO-REDO

Поддержка отката произвольной глубины и сопровождающего его повтора представляет прямое расширение рассмотренной нами схемы.

Список истории

Что не позволяло нам производить откат на большую глубину? Ответ очевиден - у нас был только один объект - последний созданный экземпляр `COMMAND`, доступный через `requested`.

Фактически мы создавали столь много объектов, сколько команд выполнял пользователь. Но поскольку в нашем проекте присутствует только одна ссылка на командный объект - `requested`, всегда присоединенная к последней команде, то каждый командный объект становится недоступным, как только пользователь создает новую команду. Нам нет необходимости заботиться о судьбе этих старых объектов. Важной частью, обеспечивающей элегантность и простоту хорошего ОО окружения, является сборщик мусора (см. [лекцию 9](#) курса "Основы объектно-ориентированного программирования"), в задачу которого входит освобождение памяти. Было бы ошибкой пытаться самим использовать память, так как все объекты имеют разную структуру и размеры.

Для обеспечения глубины отката достаточно заменить единственный объект `requested` списком, содержащим выполненные команды, - списком истории:

```
history: SOME_LIST [COMMAND]
```

Имя `SOME_LIST` не является именем настоящего класса, - в подлинном ОО стиле АТД мы исследуем, какие операции и свойства необходимы классу `SOME_LIST`, и позже вынесем заключение, какой же списочный класс

из базовой библиотеки (Base library) следует использовать. Принципиальные операции, нужные нам непосредственно, хорошо известны из предыдущего обсуждения:

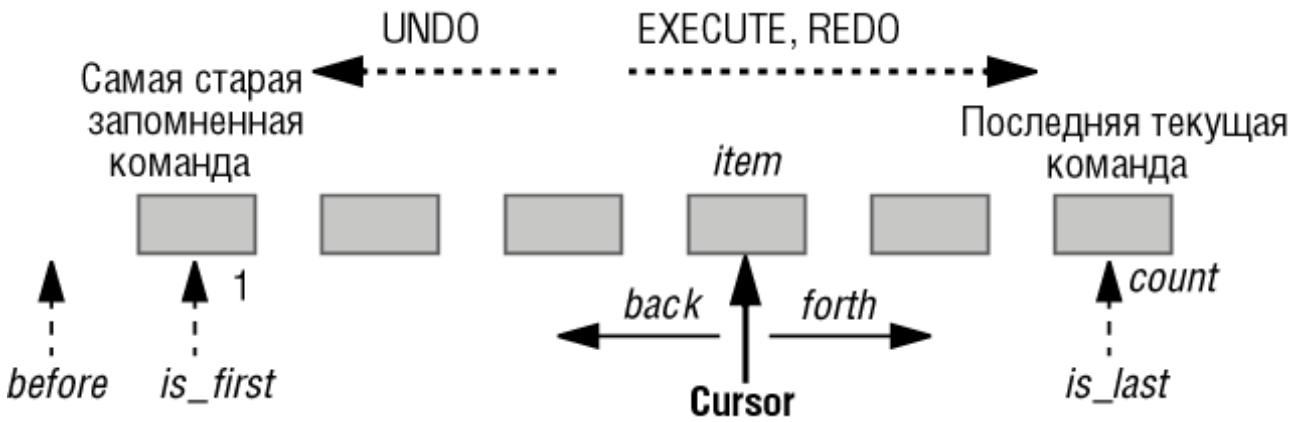


Рис. 3.3. Список истории

- Put - команда вставки элемента в конец списка (единственное необходимое нам место вставки). По соглашению, put позиционирует курсор списка на только что вставленном элементе.
- Empty - запрос определения пустоты списка.
- Before, is_first и is_last - запросы о позиции курсора.
- Back, forth - команды, передвигающие курсор назад, вперед на одну позицию.
- Item - запрос элемента в позиции, заданной курсором. Этот компонент имеет предусловие: (not empty) and (not before), которое можно выразить как запрос on_item.

В отсутствие откатов курсор всегда (за исключением пустого списка) будет указывать на последний элемент и is_last будет истинным. Если же пользователь начнет выполнять откат, курсор начнет передвигаться назад по списку вплоть до before, если отменяются все выполненные команды. Когда же начинается повтор, то курсор перемещается вперед.

На [рис. 3.3](#) курсор указывает на элемент, отличный от последнего. Это означает, что пользователь выполнял откат, возможно, перемежаемый повторами. Заметьте, число команд Undo всегда не меньше числа Redo (в состоянии на рисунке оно на два больше). Если в этом состоянии пользователь выберет обычную команду (ни Undo, ни Redo) соответствующий элемент будет вставлен непосредственно справа от курсора. Это означает, что остававшиеся справа в списке элементы будут потеряны, так для них не имеет смысла выполнение Redo. Здесь возникает та же ситуация, которая привела нас в начале лекции к введению понятия операции Skip (см. УЗ.4). Как следствие, в классе SOME_LIST понадобится еще один компонент - процедура remove_all_right, удаляющий все элементы справа от курсора.

Выполнение Undo возможно, если и только если курсор стоит на элементе с истинным значением on_item. Выполнение Redo возможно, если и только если был сделан откат, для которого еще не выполнена операция Redo, - это означает истинность выражения: (not empty) and (not is_last), которое будем называть запросом not_last.

Реализация Undo

Имея список истории, достаточно просто реализовать Undo:

```
if on_item then
    history.item.undo
    history.back
else
    message ("Нет команды для отката - undo")
end
```

И снова динамическое связывание играет основную роль. Список истории history является полиморфной структурой данных:

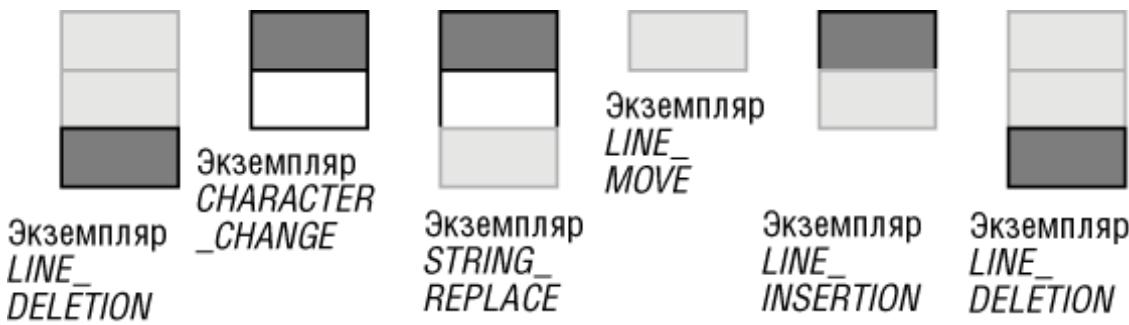


Рис. 3.4. Список истории с различными объектами command

При передвижении курсора влево каждое успешное значение `history.item` может быть присоединено к объекту любого доступного типа `command`. Динамическое связывание гарантирует, что в каждом случае `history.item.undo` автоматически выберет нужную версию `undo`.

Реализация Redo

Реализация Redo аналогична:

```

if not_last then
    history.forth
    history.item.redo
else
    message ("Нет команды для отката - undo")
end
    
```

Предполагается, что в классе `COMMAND` введена новая процедура `redo`. До сих пор считалось верным, что `redo` – это то же самое, что и `execute`. Это справедливо в большинстве случаев, но для некоторых команд повторное выполнение может отличаться от выполнения с нуля. Лучший способ справиться с такой ситуацией, не жертвуя общностью, – задать для `redo` поведение по умолчанию в классе `COMMAND`:

```

redo is
    -- Повтор команды, которую можно отменить,
    -- по умолчанию эквивалентно ее выполнению.
do
    execute
end
    
```

Наличие реализации превращает класс `COMMAND` в класс, определяющий поведение (см. [лекцию 4](#) курса "Основы объектно-ориентированного программирования"). Он имеет отложенные процедуры `execute` и `undo` и эффективную процедуру `redo`. Большинство из потомков сохранят поведение по умолчанию `redo`, но некоторые зададут поведение, соответствующее специфике команды.

Выполнение обычных команд

Обычная команда по-прежнему идентифицируется ссылкой `requested`. Такую команду следует не только выполнить, но и добавить ее в список истории, предварительно удалив все элементы справа от курсора. В результате получим:

```

if not is_last then remove_all_right end
history.put (requested)
    -- Напомним, put вставляет элемент в конец списка,
    -- курсор указывает на новый элемент
requested.execute
    
```

Мы рассмотрели все основные элементы решения. В оставшейся части лекции обсудим некоторые аспекты реализации и извлечем из нашего примера методологические уроки.

Аспекты реализации

Давайте займемся деталями, что позволит получить лучшую из возможных реализаций.

Аргументы команды

Некоторым командам нужны аргументы. Например, команде `LINE_INSERTION` нужен текст вставляемой строки.

Простым решением является добавление атрибута и процедуры в класс `COMMAND`:

```
argument: ANY
set_argument (a: like argument) is
    do argument := a end
```

Тогда любой потомок - командный класс - сможет переопределить `argument`, задав для него подходящий тип. Чтобы справиться с множественными аргументами, достаточно выбрать массив или списочный тип. Такова была техника, принятая выше, при передаче аргументов процедуре создания объектов командных классов.

Эта техника подходит для всех простых приложений. Заметьте, библиотечный класс `COMMAND` в среде ISE использует другую технику, немного более сложную, но более гибкую: здесь нет атрибута `argument`, но процедура `execute` имеет аргумент в обычном для процедур смысле:

```
execute (command_argument: ANY) is ...
```

Причина в том, что в графических системах удобнее позволять различным экземплярам одного и того же командного типа разделять один и тот же аргумент. Удаляя атрибут, мы получаем возможность повторно использовать тот же командный объект во многих различных контекстах, избегая создания нового командного объекта всякий раз, когда пользователь запрашивает команду.

Небольшое усложнение связано с тем, что теперь элементы списка истории уже не являются экземплярами `COMMAND` - они теперь должны быть экземплярами класса `COMMAND_INSTANCE` с атрибутами:

```
command_type: COMMAND
argument: ANY
```

Для серьезных систем стоит пойти на усложнение ради выигрыша в памяти и времени. В этом варианте создается один объект на каждый тип команды, а не на каждую выполняемую команду. Эта техника рекомендуется для производственных систем. Необходимо лишь изменить некоторые детали в рассмотренном ранее классе (см. УЗ.4).

Предвычисленные командные объекты

Еще до выполнения команды следует получить, а иногда и создать соответствующий командный объект. Для абстрактно написанной инструкции "Создать подходящий командный объект и присоединить его к `requested`" была предложена схема реализации:

```
inspect
    request_code
when Line_insertion then
    create {LINE_INSERTION} requested.make (...)
```

и т.д. (одна ветвь для каждого типа команды)

Как отмечалось, здесь **нет** нарушения принципа Единственного Выбора: фактически это и есть точка выбора - единственное место в системе, знающее, какое множество команд поддерживается. Но к этому времени у нас выработалось здоровое отвращение к инструкциям `if` или `inspect`, содержащим много ветвей. Давайте попытаемся избавиться от них, хотя их присутствие кажется на первый взгляд неизбежным.

Мы создадим широко применимый образец проектирования, который может быть назван **множество предвычисленных полиморфных экземпляров (precomputing a polymorphic instance set)**.

Идея достаточно проста - создать раз и навсегда полиморфную структуру данных, содержащую по одному экземпляру каждого варианта, затем, когда нужен новый объект, просто получаем его из соответствующего входа в структуру.

Хотя для этого возможны различные структуры, например списки, мы будем использовать массив `ARRAY [COMMAND]`, позволяющий идентифицировать каждый тип команды целым в интервале 1 и до `command_count` - числом типов команд. Объявим:

```
commands: ARRAY [COMMAND]
```

и инициализируем его элементы так, чтобы *i*-й элемент ($1 \leq i \leq n$) ссылался на экземпляр класса потомка `COMMAND`, соответствующего коду *i*; например, создадим экземпляр `LINE_DELETION`, свяжем его с первым элементом массива, так что удаление строки будет иметь код 1.

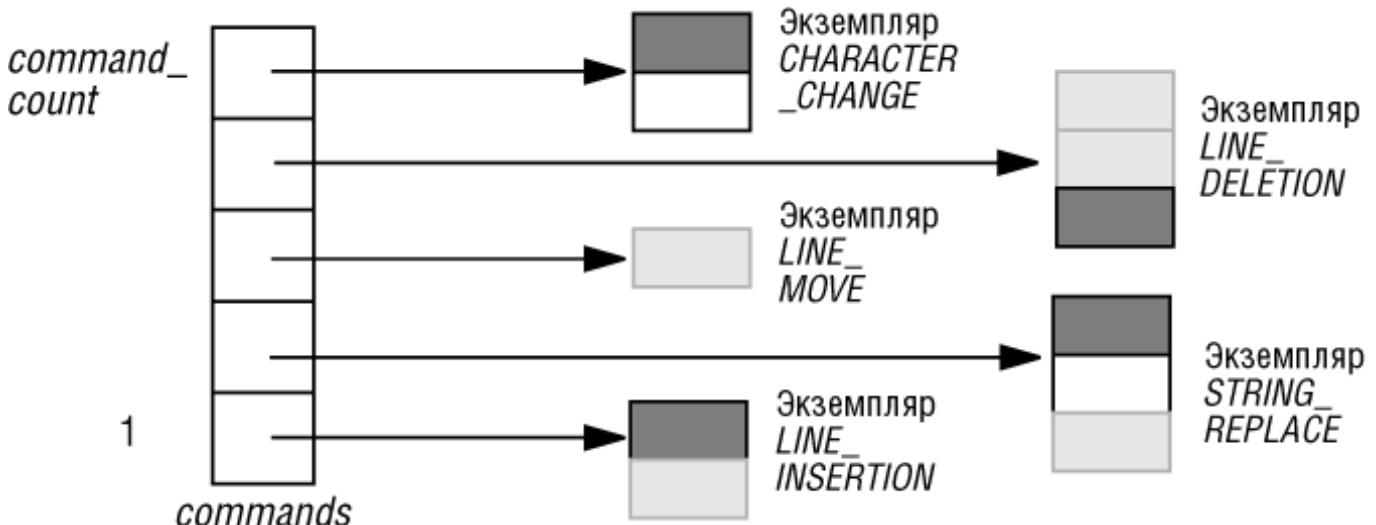


Рис. 3.5. Массив шаблонов команд

Подобная техника может быть применена к полиморфному массиву `associated_state`, используемому в ОО-решении предыдущей лекции для приложения, управляемого панелями.

Массив `commands` дает еще один пример мощи полиморфных структур данных. Его инициализация тривиальна:

```
create commands.make (1, command_count)
create {LINE_INSERTION} requested.make; commands.put (requested, 1)
create {STRING_REPLACE} requested.make; commands.put (requested, 2)
... И так для каждого типа команд ...
```

Заметьте, при этом подходе процедуры создания не должны иметь аргументов; если командный класс имеет атрибуты, то следует устанавливать их значения позднее в специально написанных процедурах, например `li.make (input_text, cursor_position)`, где `li` типа `LINE_INSERTION`.

Теперь исчезла необходимость применения разбора случаев и ветвящихся инструкций `if` или `inspect`. Приведенная выше инициализация служит теперь точкой Единственного Выбора. Теперь реализацию абстрактной операции "Создать подходящий командный объект и присоединить его к `requested`" можно записать так:

```
requested := clone (commands @ code)
```

где `code` является кодом последней команды. Так как каждый тип команды имеет теперь код, соответствующий его индексу в массиве, то базисная операция интерфейса, ранее написанная в виде "Декодировать запрос", анализирует запрос пользователя и определяет соответствующий код.

В присваивании `requested` используется клон (`clone`) шаблона команды из массива, так что можно получать более одного экземпляра одной и той же команды в списке истории (как это показано в предыдущем примере, где в списке истории присутствовали два экземпляра `LINE_DELETION`).

Если, однако, использовать предложенную технику, полностью отделяющую аргументы команды от командных объектов (так что список истории содержит экземпляры `COMMAND_INSTANCE`, а не `COMMAND`), то тогда в получении клонов нет необходимости, и можно перейти к использованию ссылок на оригинальные объекты из массива:

```
requested:= commands @ code
```

В длительных сессиях такая техника может давать существенный выигрыш.

Представление списка истории

Для списка истории был задан абстрактный тип `SOME_LIST`, обладающий компонентами: `put`, `empty`, `before`, `is_first`, `is_last`, `back`, `forth`, `item` и `remove_all_right`. (Есть также `on_item`, выраженный в терминах `empty` и `before`, и `not_last`, выраженный в терминах `empty` и `is_last`.)

Большинство из списочных классов базовой библиотеки можно использовать для реализации `SOME_LIST`; например, класс `TWO_WAY_LIST` или одного из потомков класса `CIRCULAR_LIST`. Для получения независимой версии рассмотрим специально подобранный класс `BOUNDED_LIST`. В отличие от ссылочной реализации списков, подобных `TWO_WAY_LIST`, этот класс основан на массиве, так что он хранит лишь ограниченное число команд в истории. Пусть `remembered` будет максимальным числом хранимых команд. Если используется в системе подобное свойство, то запомните (если не хотите получить гневное письмо от меня как от пользователя

вашей системы): этот максимум должен задаваться пользователем либо во время сессии, либо в профиле пользователя. По умолчанию он должен выбираться никак не менее 20.

Список BOUNDED_LIST может использовать массив с циклическим управлением, позволяющий использовать ранее занятые элементы, когда число команд переваливает за максимум remembered. Эта техника является общей для представления ограниченных очередей. Массив в этом случае представляется в виде баранки:

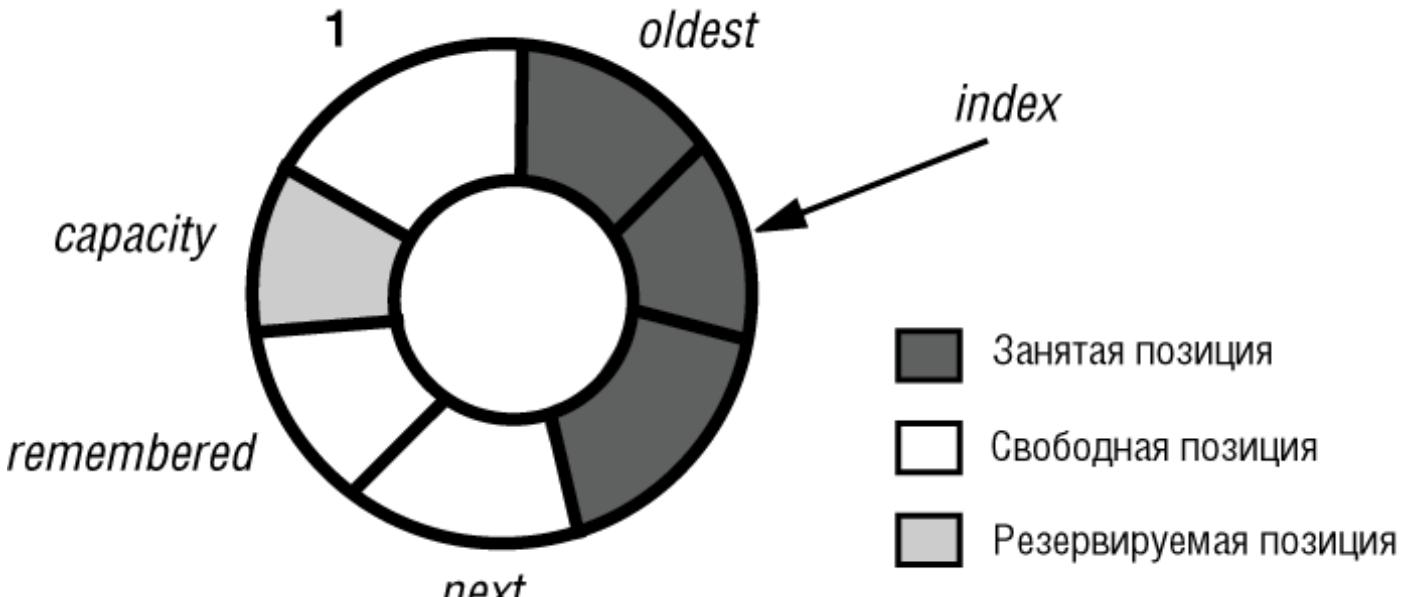


Рис. 3.6. Ограниченный циклический список, реализуемый массивом

Размером capacity массива является remembered + 1; это соглашение означает фиксирование одной из позиций (последней с индексом capacity), оно необходимо для различия пустого и полностью заполненного списка. Занятые позиции помечены двумя целочисленными атрибутами: oldest - является позицией самой старой запомненной команды, и next - первая свободная позиция (для следующей команды). Атрибут index указывает текущую позицию курсора.

Вот как выглядит реализация компонентов. Для put(c), вставляющей команду c в конец списка, имеем:

```
representation.put (x, next);
--где representation это имя массива
next:= (next\` remembered) + 1
index:= next
```

где операция \` представляет остаток от деления нацело. Значение empty истинно, если и только если next = oldest; значение is_first истинно, если и только если index = oldest; и before истинно, если и только если (index\` remembered) + 1 = oldest. Телом forth является:

```
index:= (index\` remembered) + 1
а телом back:
index:= ((index + remembered - 2) \` remembered) + 1
```

Терм +remembered математически избыточен, но он включен из-за отсутствия стандартного соглашения для операции взятия остатка в случае отрицательных operandов.

Запрос item возвращает элемент в позиции курсора - representation @ index, - элемент массива с индексом index. Наконец, процедура remove_all_right, удаляющая все элементы справа от курсора, реализована так:

```
next:= (index remembered) + 1
```

Интерфейс пользователя для откатов и повторов

Покажем, как выглядит возможный интерфейс пользователя, поддерживающий механизм undo-redo. Пример взят из ISE, но и некоторые другие наши продукты используют ту же схему.

Хотя горячие клавиши доступны для Undo и Redo, полный механизм включает показ окна истории (history window). В нем отображается список history. Однажды открытое, оно регулярно обновляется при выполнении команд. В отсутствие откатов оно выглядит так:



Рис. 3.7. Окно истории до выполнения откатов

Оно отображает список выполненных команд. При выполнении новой команды, она появится в конце списка. Текущая активная команда (отмеченная курсором) подсвеченна, как показано на рисунке для "change relation label".

Для отката достаточно щелкнуть по кнопке со стрелкой ↑ или использовать горячие клавиши (Alt-U). Если передвинуть курсор вверх (для списка это переход назад - back) то после нескольких операций Undo, окно примет вид, показанный на [рис. 3.8](#).

В этом состоянии есть выбор:

- Можно выполнить еще раз операцию Undo - подсветка передвинется к предыдущей строке.
- Можно выполнить один или несколько раз операцию повтора Redo, используя эквивалентную комбинацию горячих клавиш или щелкнув по кнопке со стрелкой вниз ↓ . Подсветка в окне передвинется к следующей строке, а список выполнит вызов forth.



Рис. 3.8. Окно истории в процессе откатов и повторов

Можно выполнить нормальную команду. Как мы знаем, из истории удаляются все команды, для которых был откат, но не было повтора; для списка это означает удаление элементов справа от курсора и вызов remove_all_right ; все команды ниже подсвеченной исчезнут.

Обсуждение

Образец проектирования, представленный в этой лекции, играет важную практическую роль, позволяя при незначительных усилиях создавать значительно лучшие интерактивные системы. Он также вносит интересный теоретический вклад, освещая некоторые аспекты ОО методологии, заслуживающие дальнейшего исследования.

Роль реализации

Замечательное свойство пользовательского интерфейса, представленного в последнем разделе, состоит в том, что оно **непосредственно выведено** из реализации, - взяв внутреннее, относящееся к разработке понятие **списка истории**, мы транслировали его во внешнее, относящееся к пользователю понятие **окна истории** с привлекательным пользовательским механизмом взаимодействия.

Можно представить, что кто-то мог бы вначале придумать внешнее представление независимо от реализации. Но так не получилось ни в этом изложении, ни при разработке наших программных продуктов.

Существование такого отношения между функциональностью системы и ее реализацией противоречит всему тому, чему учит традиционная методология разработки ПО. Нам говорят: выводите реализацию из спецификации, но не наоборот! Методы "итеративной разработки" и "жизненного цикла" немногое изменяют в том привычном подходе, когда реализация является рабом первичных концепций, а разработчики ПО должны делать то, что говорят их "пользователи". Здесь мы нарушаем это табу и утверждаем, что **реализация** может сказать нам, что **следует делать системе**. В прежние времена посягательство на освященные временем принципы -вокруг чего вращается мир - могло привести на костер.

Наивно верить, что пользователи могут предложить правильные свойства интерфейса. Иногда они могут это сделать, но, чаще всего, они будут исходить из свойств, знакомых им по прежним системам. Это понятно, у них своя работа, своя область, в которой они являются экспертами, и нельзя на них возлагать ответственность за то, что должно быть правильным в программной системе. Некоторые из худших интерактивных систем были спроектированы, находясь под **слишком большим** влиянием пользователей. Где действительно необходим вклад пользователей - так это их критические комментарии: они могут видеть изъяны в идее, которая могла казаться привлекательной разработчикам. Такой критицизм всегда необходим. Пользователи могут высказывать и блестящие положительные предложения тоже, но не следует быть зависимыми от них. Несмотря на критику иногда разработчикам удается склонить пользователей на свою сторону, возможно, после нескольких итераций и учета замечаний. И это происходит даже тогда, когда предложения вытекают из, казалось бы, чисто реализацийных аспектов, как это было со списком истории.

Равенство традиционных отношений представляет важный вклад в объектную технологию. Рассматривая процесс разработки бесшовным и обратимым (см. [лекцию 10](#)), мы допускаем влияние идей реализации на спецификации. Вместо одностороннего движения от анализа к проектированию и кодированию, приходим к непрерывному циклическому процессу с обратной связью. Реализация не должна рассматриваться как похлебка, низкоуровневый компонент конструирования системы. Разработанная с использованием методов, описанных в данной книге, она может и должна быть четкой, элегантной и абстрактной, ничуть не уступающей всему тому, что можно получить в презирающих реализацию традиционных формах анализа и проектирования.

Небольшие классы

Проект, описанный в этой лекции, может для типичных интерактивных систем включать достаточно много относительно небольших классов, по одному на каждую команду. Нет причин, однако, полагать, что это отражается на размере системы или ее сложности. Структура наследования классов остается простой, хотя она вовсе не должна быть плоской, - команды можно группировать по категориям.

При систематическом ОО-подходе такие вопросы возникают всякий раз, когда приходится вводить классы, представляющие действия. Хотя некоторые ОО-языки дают возможность передавать программы как аргументы в другие программы, такое свойство противоречит базисной идеи Метода - функции (программы) не существуют сами по себе, - они лишь **часть некоторой абстракции данных**. Поэтому вместо передачи операций следует передавать объект, поставляемый вместе с операцией, например, экземпляр COMMAND, поставляемый с операцией execute.

Иногда для операций приходится писать обертывающий класс, что кажется искусственным, особенно для людей, привыкших передавать процедуры в качестве аргументов. Но мне неоднократно приходилось видеть, что класс, введенный с единственной целью инкапсуляции операции, превращался позже в полноценную абстракцию данных с добавлением операций, о которых не шла речь в первоначальном замысле. Класс COMMAND не попадает в эту категорию, он с самого начала рассматривался как абстракция данных и имел два компонента (execute and undo). Но, что типично, серьезная работа с командами приводит к осознанию необходимости других компонентов, таких как:

- argument: ANY - для представления аргументов команды (как это было сделано в одной из версий проекта);
- help: STRING - для предоставления справки по каждой команде;
- **компоненты**, поддерживающие протоколирование и статистику вызова команд.

Еще один пример взят из области численных вычислений. Рассмотрим классическую задачу вычисления интеграла. Как правило, подынтегральная функция f передается как аргумент в программу, вычисляющую интеграл. Традиционная техника представляет f как функцию, при ОО-проектировании мы обнаруживаем, что

"Интегрируемая функция" является важной абстракцией со многими возможными свойствами. Для пришедших из мира C, Fortran и нынешнего проектирования необходимость написания класса в такой ситуации кажется простым программистским трюком. Возможно, первое время он неохотно будет принимать эту технику, смиряясь с ней. Продолжая проект, он скоро осознает, что интегрируемая функция - INTEGRABLE_FUNCTION - на самом деле является одной из главных абстракций проблемной области. В этом классе появятся новые полезные компоненты помимо компонента item (a: REAL): REAL, возвращающего значение функции в точке a.

То, что казалось лишь трюком, превращается в главную составляющую проекта.

Библиографические замечания

Механизм undo-redo, описанный в этой лекции, был реализован в конструкторе структурированных документов Сераге, разработанном Жан-Марк Нерсоном и автором в 1982 [M 1982] и позже был интегрирован во многие интерактивные средства ISE (включая ArchiText [ISE 1996], преемника Сераге).

На первой конференции OOPSLA в 1986 Larry Tesler говорил о механизме, основанном на той же идее, встроенным в интерактивный каркас Apple's MacApp.

В работе [Dubois 1997] объясняется в деталях, как применять ОО концепции в численных вычислениях, с приведением такой абстракции как "Integrable function".

Упражнения

У3.1 Небольшая интерактивная система (программистский проект)

Этот небольшой программистский проект является лучшим способом проверки понимания тем этой лекции и ОО-техники в целом.

Напишите текстовый редактор, ориентированный на работу со строками, поддерживающий следующие операции:

- p: печать введенного текста;
- ↓: передвигает курсор к следующей строке, если она есть (используйте код l, если это более удобно);
- ↑: передвигает курсор к предыдущей строке, если она есть (используйте код h, если это более удобно);
- i: вставляет новую строку после позиции курсора.
- d: удаляет строку в позиции курсора;
- u: откат последней операции, если она не была Undo; если же это Undo, то выполняется повтор redo.

Можно добавить новые команды или спроектировать более привлекательный интерфейс, но во всех случаях следует создать законченную, работающую систему. (Возможно, вы сразу начнете с улучшений, описанных в следующем упражнении.)

У3.2 Многоуровневый Redo

Дополните одноуровневую схему предыдущего упражнения переопределением смысла операции отката u:

- u: откат последней операции, отличной от Undo и Redo.

Добавьте операцию повтора Redo:

- r: повтор последней операции, если она применима.

У3.3 Undo-redo в Pascal

Объясните, как применить рассмотренную технику в не ОО-языках, подобных Pascal, Ada (используя записи с вариантами) или С (используя структуры и union типы). Сравните с ОО-решениями.

У3.4 Undo, Skip и Redo

С учетом проблем, поднятых в обсуждении, рассмотрите, как можно расширить механизм, разработанный в этой лекции так, чтобы он допускал поддержку Undo, Skip и Redo, а также делал возможным повтор и откат, перемежаемый обычными командами. Обсудите эффект обоих новинок как на уровне интерфейса, так и реализации.

У3.5 Сохранение командных объектов

Рассмотрите аргументы команды независимо от команд и создайте только один командный объект на каждый тип

команды.

Если вы выполнили предыдущее упражнение, примените эту технику к решению.

У3.6 Составные команды

В некоторых системах может быть полезным ввести понятие составной команды, выполнение которых включает выполнение нескольких других команд. Напишите соответствующий класс COMPOSITE_COMMAND, потомка COMMAND, убедитесь, что составные команды допускают откат и что компонента составной команды может быть составной командой.

Указание: используйте множественное наследование, представленное для составных фигур (см. [лекцию 15](#) курса "Основы объектно-ориентированного программирования").

У3.7 Необратимые команды

Система может включать необратимые команды либо по самой их природе ("Запуск ракет"), либо по прагматичным причинам больших расходов, связанных с отменой действия команды. Усовершенствуйте решение так, чтобы оно учитывало возможность присутствия необратимых команд. Внимательно изучите алгоритмы и интерфейс пользователя, в частности используйте окно истории.

Указание: введите наследников UNDOABLE и NON_UNDOABLE класса COMMAND.

У3.8 Библиотека команд (проектирование и реализация)

Напишите общечелевую библиотеку команд, предполагающую использование в произвольной интерактивной системе и поддерживающую неограниченный механизм undo-redo. Библиотека должна интегрировать свойства, обсуждаемые в последних трех упражнениях: отделение команд от их аргументов, составные команды, необратимые команды. Возможно также встраивание свойства "Undo, Skip и Redo". Проиллюстрируйте применимость библиотеки, построив на ее основе три демонстрационные системы различной природы, такие как текстовый редактор, графическая система, инструмент тестирования.

У3.9 Механизм истории

Полезным компонентом, встраиваемым в командно-ориентированный инструментарий, является механизм истории, запоминающий выполненную команду и позволяющий пользователю повторно ее выполнить, возможно, модифицировав. Под Unix, например, доступен командный язык C-shell, запоминающий несколько последних выполненных команд. Вы можете напечатать !-2, означающее, что нужно выполнить команду, предшествующую последней. Запись "yes^no" означает "выполнение последней команды с заменой yes на no". Другие окружения предлагают схожие свойства.

Механизмы истории, когда они существуют, построены в соответствии с модой. Под Unix многие интерактивные средства, выполняемые под C-shell, такие как текстовый редактор Vi или различные отладчики, будут получать преимущества от такого механизма, но он не будет предлагаться другим системам. Это тем более вызывает сожаление, что те же концепции истории команд и те же ассоциированные свойства полезны любой интерактивной системе независимо от выполняемых ею функций.

Спроектируйте класс, реализующий механизм истории общечелевого назначения, так чтобы любая интерактивная система, нуждающаяся в этом механизме, могла получить его путем простого наследования класса. (Заметьте, множественное наследование здесь необходимо.)

Обсудите расширение этого механизма на общий класс USER_INTERFACE.

У3.10 Тестирование окружения

Тестирование компонентов ПО, например, класса требует определенных свойств при подготовке теста: ввода тестовых данных, выполнения теста, записи результатов, сравнения с ожидаемыми результатами и так далее. Определите общий, допускающий наследование класс TEST, задающий тестирующее окружение. (Обратите внимание, что и здесь важно множественное наследование.)

У3.11 Интегрируемые функции

(Для читателей, знакомых с численными методами.) Напишите множество классов для интегрирования вещественных функций вещественной переменной на произвольном интервале. Сюда должен входить класс INTEGRABLE_FUNCTION, а также отложенный класс INTEGRATOR, описывающий метод интегрирования, и потомки класса, такие как RATIONAL_FIXED_INTEGRATOR.

Основы объектно-ориентированного проектирования

4. Лекция: Как найти классы

Важнейшая цель ОО-методологии - дать рекомендации по нахождению классов, на основе которых строится программная система. Этому посвящена данная лекция. Предупреждаю, не следует ожидать слишком много. Поиск классов - это одна из центральных проблем при ОО-конструировании ПО. Как в любой творческой деятельности ее решение определяется талантом и опытом, не исключая и доли везения. Нереально ожидать безошибочных рецептов поиска классов, как нельзя ожидать рецептов, позволяющих математику создавать новые теории или находить доказательства теорем. Хотя оба вида деятельности - конструирование ПО и конструирование теорий - могут извлекать пользу от советов общего вида и опыта удачливых предшественников, но в обоих случаях требуется созидательная работа, не регламентируемая механическими правилами. Если, как и многие в программной индустрии, вы полагаете, что эти два вида деятельности нельзя сравнивать, то рассматривайте нашу работу просто как одну из форм инженерного проектирования, хотя здесь и можно дать базисное руководство, но никакие придуманные пошаговые правила не могут гарантировать хорошего проектирования новых самолетов или мостов. При проектировании ПО никакие книги не могут заменить вашей интуиции и владения тем, что называется "ноухай" (know-how). Роль методологического обсуждения состоит в том, чтобы указать на хорошие идеи, привлечь внимание к удачным прецедентам и предупредить о типичных ловушках. Это относится к любому методу проектирования. В объектном случае есть некоторые хорошие новости, приходящие в форме повторного использования. Поскольку многое из необходимого уже изобретено, можно начинать строить, используя достижения других разработчиков. Есть и другие хорошие новости. Начиная со скромных притязаний, но внимательно изучив, что работает, а что нет, двигаясь мало-помалу в исключительно неблагоприятных условиях, мы получим возможность изобрести нечто, заслуживающее название метода поиска классов. Как всегда при поиске кандидатов, метод отбора включает два компонента - набор абитуриентов и их последующий отсев.

Изучение документа "технические требования"

Для понимания проблемы поиска классов, возможно, лучше всего начать с известного и широко опубликованного подхода.

Существительные и глаголы

В нескольких публикациях предлагается простое правило для получения классов, - начинайте с документа "технические требования" (считается, что кто-то его создал, но это уже другая история). В функционально-ориентированном проекте следует концентрироваться на глаголах, соответствующих действиям. При ОО-проектировании отмечайте существительные, описывающие объекты. Так из предложения:

Лифт закрывает дверь, прежде чем двигаться к следующему этажу (The elevator will close its door before it moves to another floor)

функционально-ориентированный разработчик извлечет необходимость создания функции "move", а ОО-разработчик увидит необходимость создания объектов трех типов: ELEVATOR, DOOR and FLOOR, приводящих к классам. Вот?!

Как было бы прекрасно, если бы жизнь была такой простой! Вы бы захватили документ с требованиями домой на вечерок, сыграли бы за обеденным столом в игру "Погоня За Объектами". Это был бы хороший способ отвлечь детей от телевизора, повторить заодно грамматику и помочь маме с папой в их важной работе по конструированию ПО.

К сожалению, такой примитивный метод не слишком помогает. Естественный язык, используемый экспертами при составлении технических требований, обладает столь многими нюансами, субъективными вариациями, двусмысленностями, что крайне опасно принимать важные решения на основе грамматического анализа такого документа. Вполне возможно, что учитывались бы не столько свойства проектируемой системы, сколько особенности авторского стиля.

Метод "**подчеркивание существительных**" дает лишь очевидные понятия. Любой разумный ОО-метод проектирования системы управления лифтом будет включать класс ELEVATOR. Получение подобных классов не самая трудная часть задачи. Повторяя сказанное в предыдущих обсуждениях, генерируемые понятия нуждаются в прополке - необходимо отделить зерна от плевел.

Хотя идея подчеркивания существительных не заслуживает особого рассмотрения, мы будем использовать ее для контраста, для лучшего понимания тех ограничений, подстерегающих нас на пути поиска классов.

Как избежать бесполезных классов

Существительные в документе с требованиями покрывают некоторое множество классов будущего проекта, но они также включают слишком много "ложных тревог" - концепций, не заслуживающих того, чтобы быть классами.

В примере с лифтовой системой door является существительным, но необходим ли класс DOOR? Может быть, да, может быть - нет. Возможно, что единственным свойством дверей лифта является их способность открываться и закрываться. Тогда проще включить это свойство в виде соответствующего запроса и команды

в класс ELEVATOR:

```
door_open: BOOLEAN;
close_door is
  ...
  ensure
    not door_open
  end;
open_door is
  ...
  ensure
    door_open
  end
```

В другом варианте понятие двери может заслуживать отдельного класса. Единственной реальной основой является здесь теория АТД. Вот вопрос, на который действительно следует ответить:

Является ли "door" независимым типом данных с собственными четко определенными операциями или все они уже включены в операции других типов данных, таких как, например, ELEVATOR?

Только наша интуиция и опыт проектировщика даст нам правильный ответ. Грамматические правила анализа документа требований малосодержательны. Вместо них следует обращаться к теории АТД, помогающей задавать правильные вопросы заказчикам и будущим пользователям системы.

Мы уже встречались (см. [лекцию 3](#)) с подобной ситуацией при проектировании механизма откатов и возвратов. Речь шла о понятии commands и более общем понятии operation, включающем запросы, подобные Undo. Оба слова фигурировали в документе требований, однако, только COMMAND приводил к абстракции данных - важнейшему классу проекта.

Нужен ли новый класс?

Еще одним примером существительного в примере с лифтом является слово floor. В отличие от дверей с их единственной операцией, понятие этажа является разумным АТД, однако этого мало, чтобы появился класс FLOOR.

Причина проста: в данном случае для целей лифтовой системы вполне достаточно представлять этажи целыми числами - их номерами, так расстояние между этажами может быть выражено простой разностью целых чисел.

Если, однако, этажи имеют свойства, не отображаемые номерами, тогда может потребоваться класс FLOOR. Например, некоторые этажи могут иметь специальные права доступа, разрешающие их посещение только избранным osobam, и тогда класс FLOOR может включать свойство:

```
rights: SET [AUTHORIZATION]
```

и связанные с ним процедуры. Но и в этом случае нет полной определенности. Возможно, стоит вместо создания класса включить в некоторый другой класс массив:

```
floor_rights: ARRAY [SET [AUTHORIZATION]]
```

связывающий множество значений AUTHORIZATION с каждым этажом, идентифицируемым его номером (см. У4.1).

Еще одним аргументом в пользу создания независимого класса FLOOR могла бы послужить возможность ограничения доступных операций над классом. Так операции вычитания и сравнения этажей должны быть доступными, а сложение и умножение лишены смысла и должны быть недоступны. Такой класс мог быть представлен как наследник класса INTEGER.

Это обсуждение снова приводит нас к теории АТД. Класс не должен представлять физические "объекты" в наивном смысле. Он должен описывать абстрактный тип данных - множество программных объектов, характеризуемых хорошо определенными операциями и их формальными свойствами. Типы реальных объектов могут и не иметь двойников в программном мире - классов. Когда решается вопрос - стать ли некоторому понятию классом, только АТД является правильным критерием, позволяя сказать, соответствует ли данное понятие классу программной системы или оно покрывается уже существующими классами.

"Релевантность системе" является определяющим критерием. Цель анализа системы не в том, чтобы "моделировать мир", - об этом пусть заботятся философы. Создатели ПО не могут позволить себе этого, по крайней мере, в своей профессиональной деятельности, их задачей является моделирование мира лишь в

той мере, которая касается создаваемого ПО. Подход АТД и соответственно ОО-метода основан на том, что **объекты определяются только тем, что мы можем с ними делать**, - это называлось (см. [лекцию 6](#) курса "Основы объектно-ориентированного программирования") Принципом Разумного Эгоизма. Если операция или свойство объекта не отвечают целям системы, то они и не включаются в состав класса, хотя и могут представлять интерес для других целей. Понятие PERSON может включать такие компоненты, как mother и father, но для системы уплаты налогов они не нужны, здесь личность выступает сама по себе, подобно сироте.

Если все операции и свойства некоторого типа не связаны с целями системы или покрываются другими классами, то сам тип не должен рассматриваться как самостоятельный класс.

Пропуск важных классов

Подчеркивание существительных может не только приводить к понятиям, не создающим классов, но и к пропуску понятий, которые должны быть классами. Есть, по меньшей мере, три причины возникновения таких ситуаций.

Напомню, мы анализируем ограничения подхода "подчеркивания существительных" лишь для лучшего понимания процесса поиска классов.

Первой причиной пропуска классов являются гибкость и неоднозначность естественного языка - те самые качества, благодаря которым он применим в самых широких областях - от ораторских речей и романов до любовных писем. Но эти же качества становятся недостатком при написании сухой и строгой технической документации. Предположим, что наш документ с требованиями к лифтовой системе содержит предложение:

Запись базы данных должна создаваться всякий раз, когда лифт перемещается от одного этажа к другому (A **database record** must be created every time the elevator moves from one floor to another).

Существительное "record" предполагает класс DATABASE_RECORD; но при этом можно пропустить более важную абстракцию данных: понятие move, определяющее перемещение между этажами. Из смысла данного предложения скорее следует необходимость класса MOVE, например, в форме:

```
class MOVE feature
    initial, final: FLOOR;           -- Или INTEGER, если нет класса FLOOR
    record (d: DATABASE) is ...
    ... другие компоненты...
end
```

Этот важный класс вполне мог быть пропущен при простом грамматическом разборе предложения. Правда, наше предложение могло появиться и в другой форме:

Каждое перемещение лифта приводит к созданию записи в базе данных (A database record must be created for every **move** of the elevator from one floor to another).

Здесь "move" из глагола переходит в разряд существительных, претендую на класс в соответствии с грамматическим критерием. Угрозы и абсурдность подхода, основанного на анализе документа, написанного на естественном языке, очевидны. Такое серьезное дело, как проектирование системы, в частности ее модульная структура, не может зависеть от причуд стиля и настроения автора документа.

Другая важная причина в пропуске критически важных абстракций состоит в том, что они могут не выводиться непосредственно из документа с требованиями. Примерами подобных ситуаций изобилует данная книга. Вполне возможно, что в документе, определяющем требования к системе, управляемой панелями (см. [лекцию 2](#)) ни слова нет о понятиях состояние или приложение (State, Application), задающих ключевые абстракции нашего заключительного проекта. Ранее уже отмечалось, что некоторые понятия внешнего мира могут не иметь двойников среди классов системы ПО. Имеет место и обратная ситуация: классы ПО могут не соответствовать никаким объектам внешнего мира. Аналогично, если автор требований к текстовому редактору, включающему откаты, написал: "система должна поддерживать вставку и удаление строк" (**the system must support line insertion and deletion**), то нам повезло, и мы обратим внимание на существительные insertion и deletion. Но необходимость этих свойств точно также должна следовать из предложения в форме:

Редактор должен позволять пользователям вставлять и удалять строки в текущей позиции курсора (The editor must allow its users to insert or delete a line at the current cursor position).

Наивный разработчик в этом тексте может обратить внимание на тривиальные понятия курсора и позиции, пропустив абстракции команд: вставка и удаление строк.

Третья главная причина пропуска классов характерна для любого метода, использующего документ с требованиями как основу анализа, поскольку такая стратегия не учитывает повторного использования. Удивительно, но литература по ОО-анализу (см. [лекцию 9](#)) исходит из традиционного взгляда на разработку - все начинается с документа с техническими требованиями на систему и движется к поиску решения проблемы, описанной в документе. Один из главных уроков объектной технологии как раз состоит в том, что не существует четко выраженного различия между постановкой проблемы и ее решением. Существующее ПО может и должно влиять на новые разработки.

Когда ОО-разработчик сталкивается с новым программным проектом, он не должен смотреть на документ с требованиями как на альфу и омегу мудрости по данной проблеме - он должен сочетать его со знанием предыдущей разработки и доступных программных библиотек. При необходимости он может предложить обновления документа, облегчающие конструирование системы, например, удалив свойство, представляющее ограниченный интерес для пользователей, но кардинально упрощающее систему, возможно, в интересах повторного использования. Подходящие абстракции, скорее всего, находятся в существующем ПО, а не в документе с требованиями для нового проекта.

Классы COMMAND и HISTORY_LOG из примера, посвященного откатам, являются в этом отношении типичными. Способ нахождения подходящих абстракций для этой проблемы состоит не в том, чтобы сушить мозги над документом с требованиями к текстовому редактору. Это может быть процесс интеллектуального озарения ("Эврика", для которого не существует рецептов), или кто-то до нас уже нашел решение, и нам остается повторно использовать его абстракции. Конечно, можно повторно использовать и существующую реализацию, если она доступна как часть библиотеки, в этом случае вся работа по анализу, проектированию и реализации уже была бы сделана для нас.

Обнаружение и селекция

Чтобы что-то изобрести, нужны двое. Один находит варианты, другой отбирает, обнаруживает, что является важным в той массе, которую представил первый. В том, что мы называем гением, значительно меньшая доля от первого, чем от второго, отбирающего нужное из того, что разложено перед ним.

Поль Валери (цитировано в [Hadamard 1945])

Помимо прямых уроков это обсуждение ведет к более тонким следствиям.

Простые уроки звучали неоднократно: не слишком полагаться на документ с требованиями, не доверять грамматическим критериям.

Менее очевидный урок вытекает из обзора ложных тревог. Суть его в том, что нужен не только критерий для поиска классов, но и критерий для **отбраковки (rejecting)** кандидатов. Концепция может показаться вначале обнадеживающей, а в результате анализа она отбраковывается. Примеров подобных ситуаций в данной книге предостаточно (см. [лекцию 5](#)).

В книгах по ОО-анализу и проектированию, которые мне довелось читать, довольно мало рассуждений по этому вопросу. Это удивительно, поскольку в практике консультирования ОО-проектов, особенно в командах новичков, я обнаруживал, что исключение плохих идей не менее важно, чем нахождение хороших.

Это может быть даже более важно. Как правило, идей по поводу классов (обычно предлагаемых в виде объектов) хватало с избытком. Проблемой было поставить плотину на пути этого потока. Хотя некоторые важные классы пропускались, значительное большее количество отвергалось по результатам анализа.

Так что следует расширить рамки названия, вынесенного в заголовок этой лекции. Термин "Как найти классы?" означает две вещи: поиск абстракций, подходящих на роль кандидатов в классы, и исключение из них неадекватных задаче нашей системы. Эти две задачи не следует рассматривать как последовательные, - они постоянно перемешиваются. Подобно садовнику, ОО-разработчик должен постоянно высаживать новые растения и выпальывать плохие.

Принцип Выявления класса

Выявление класса - это двойственный процесс: генерирование кандидатов, их отбраковка.

Остаток этой лекции посвящен изучению составляющих этого процесса.

Сигналы опасности

Наш поиск предпочтительнее начать с процесса отбраковки кандидатов. Давайте рассмотрим типичные признаки, указывающие на плохой выбор. Поскольку проектирование не является строго формализованной дисциплиной, не следует рассматривать эти признаки как **доказательство** - в конкретной ситуации, несмотря

на признаки, предложенное решение может быть вполне законным. В терминах предыдущей лекции эти признаки не являются "абсолютно отрицательными", но "рекомендательно отрицательными" - сигналами опасности, предупреждающими о присутствии подозрительных образцов, требующих тщательного рассмотрения.

Большое Заблуждение

Большинство из сигналов опасности, обсуждаемых ниже, указывают на общую и наиболее опасную ошибку, одновременно и наиболее очевидную - проектирование класса, которого нет.

При ОО-конструировании модули строятся вокруг типов объектов, а не функций. В этом ключ к преимуществам, открываемым при расширяемости системы и ее повторном использовании. Но новички склонны попадать в наиболее очевидную ловушку, называя классом то, что в действительности является функцией (подпрограммой). Записав модуль в виде `class... feature ... end`, еще не означает появления настоящего класса, это просто программа, скрывающаяся под маской класса.

Этого Большого Заблуждения (Grand Mistake) достаточно просто избежать, как только оно осознано. Проверка ситуации обычна: следует убедиться, что каждый класс соответствует осмысленной абстракции данных. Из этого следует, что всегда есть риск, что модуль, представленный как кандидат в классы, и носящий одежду класса, на самом деле является нелегальным иммигрантом, не заслуживающим гражданства в обществе ОО-модулей.

Мой класс выполняет...

В формальных и неформальных обсуждениях архитектуры проекта часто задается вопрос о роли некоторого класса. И часто можно слышать в ответ: "**Этот класс печатает результаты**" или "**Класс разбирает вход**" - варианты общего ответа "**Этот класс делает...**".

Такой ответ обычно указывает на изъяны в проекте. Класс не должен делать одну вещь, он должен предлагать несколько служб в виде компонентов над объектами некоторого типа. Если же он выполняет одну работу, то, скорее всего, имеет место "Большое Заблуждение".

Вполне вероятно, что ошибка не в самом классе, а способе его описания - использовании операционной фразеологии. Но все-таки в этой ситуации лучше провести проверку класса.

Императивные имена

Предположим, что в процессе проектирования появились классы с такими именами, как PARSE (**РАЗОБРАТЬ**) или PRINT (**ПЕЧАТАТЬ**) - глагол в императивной форме. Это должно насторожить, не делает ли класс одну вещь и, следовательно, не должен быть классом.

Возможно, вы найдете, что с классом все в порядке, но тогда имя его выбрано неудачно. Вот "абсолютно положительное" правило:

Правило Имен класса

Имя класса всегда должно быть либо:

- существительным, возможно квалифицированным;
- прилагательным (только для отложенных классов, описывающих структурное свойство).

Хотя подобно любым другим правилам, относящимся к стилю, это дело соглашения, оно помогает поддерживать принцип: каждый класс представляет абстракцию данных.

Первая форма - существительные - покрывает большинство важнейших случаев. Существительное может появляться само по себе, например TREE, или с квалифицирующими словами - LINKED_LIST, квалифицируемое прилагательным, LINE_DELETION, квалифицируемое другим существительным.

Вторая форма возникает в специфических случаях - классах, описывающих структурное свойство, как, например, библиотечный класс COMPARABLE, описывающий объекты с заданным отношением порядка. Такие классы должны быть отложены, их имена (в английском и французском языках) часто заканчиваются на ABLE. Так, в системе, учитывающей ранжирование игроков в теннис, класс PLAYER может быть наследником класса COMPARABLE. В таксономии видов наследования эта схема классифицируется как **структурное наследование** (см. [лекцию 6](#)).

Единственный случай, который может показаться исключением из правила, задает командные классы, так как они введены в шаблоне проектирования undo-redo, покрывающем абстракции действий. Но даже и здесь

можно следовать правилу, задавая имена командных классов текстового редактора в виде: LINE_DELETION и WORD_CHANGE, а не DELETE_LINE и REPLACE_WORD (Удаление_Строка, а не Удалить_Строчку).

Английский язык предоставляет большую гибкость, чем многие другие языки, где грамматическая категория - это скорее дело веры, чем факта, и почти каждый глагол может быть и существительным. При использовании английского языка в программных именах легче придерживаться этого правила и строить короткие имена. Вы можете назвать класс IMPORT, рассматривая это имя как существительное, а не как глагол. В других языках вам пришлось бы строить более тяжеловесное имя, нечто вроде IMPORTATION. Но здесь не должно быть надувательства: класс IMPORT должен покрывать абстракцию данных - "объекты, подлежащие импорту", а не быть командным классом, задающим единственную операцию импорта.

Заметьте разницу между Правилом Имен Класса и подходом "подчеркивания существительных", обсуждаемым в начале лекции. При подчеркивании формальный грамматический критерий применяется к неформальному тексту - документу с требованиями, потому ценность его сомнительна. Наше же Правило Имен применяет тот же критерий к формальному тексту.

Однопрограммные классы

Типичным симптомом Большого Заблуждения является эффективный класс, содержащий одну, часто весьма важную, экспортируемую подпрограмму, возможно вызывающую несколько внутренних подпрограмм. Такой класс - вероятно, результат функциональной, а не ОО-декомпозиции.

Исключением являются объекты, вполне законно представляющие абстрактные действия, например команды интерактивной системы (см. [лекцию 3](#)), или то, что в не объектном подходе представляло бы функцию, передаваемую в качестве аргумента другой функции. Но примеры, приведенные в предыдущих обсуждениях, достаточно ясно показывают, что даже в этих случаях у класса может быть несколько полезных компонентов. Так для класса, задающего подынтегральную функцию, может существовать не только компонент item, возвращающий значение функции. Другие компоненты этого класса могут задавать максимум и минимум функции на некотором интервале, ее производную. Даже если класс не содержит этих компонентов, знание того, что они могут появиться позднее, заставляет нас считать, что мы имеем дело с настоящей абстракцией.

При применении однопрограммного правила следует рассматривать все компоненты класса: те, которые введены в самом классе, и те, что введены в родительских классах. Нет ничего ошибочного, когда в тексте класса содержится описание одной экспортируемой подпрограммы, если это простое расширение вполне осмысленной абстракции, определенной его предками. Это может, однако, указывать на случай **таксомании (taxomania)**, изучаемый позже как часть методологии наследования (см. [лекцию 6](#)).

Преждевременная классификация

Упомянув таксонамию, следует отметить еще одну общую ошибку новичков - преждевременное построение иерархии классов.

Наследование занимает центральное место в ОО-методе, так что хорошая структура наследования, или более аккуратно, хорошая модульная структура, включающая отношения наследования и вложенности (клиентские), - основа качества проектирования. Но наследование уместно только для хорошо понимаемых абстракций. Когда они только разыскиваются, то думать о наследовании может быть рано.

Единственным четким исключением является ситуация, в которой для области приложения разработана и широко применяется таксономия, как это имеет место в некоторых областях науки. Тогда соответствующие абстракции будут появляться вместе со структурой наследования.

В других случаях к созданию иерархии наследования следует приступить после появления основных абстракций. Конечно, в результате этих усилий может потребоваться пересмотр ранее введенных абстракций; задачи выявления классов и структуры наследования взаимно питают друг друга. Если на ранних стадиях процесса проектирования некоторые разработчики фокусируются на проблемах классификации, когда родительские классы еще не до конца поняты, то, вероятно, речь идет о попытках запрячь телегу впереди лошади.

Мне приходилось видеть людей, начинавших с создания классов SAN_FRANCISCO и HOUSTON наследников класса CITY, когда нужно было промоделировать ситуацию с одним классом CITY и несколькими его экземплярами - объектами периода выполнения.

Классы без команд

Иногда можно обнаружить классы, вообще не имеющие команд или допускающие только запросы (доступ к объектам только в режиме чтения), но не команды (процедуры, модифицирующие объекты). Такие классы

являются эквивалентами записей языка Pascal или структур Cobol и С. Такие классы могут появиться из-за ошибок проектирования, которые могут быть двух видов, а, следовательно, нуждаются в некотором исследовании.

Прежде всего, рассмотрим три случая, когда такой класс **не является** результатом неподходящего проектирования:

- Он может представлять объекты, полученные из внешнего мира, не подлежащие изменениям в ПО. Это могут быть данные датчиков от органов системы управления прибора, пакеты, передаваемые в сеть, структуры С, которых ОО-система не должна касаться.
- Некоторые классы не предназначены для прямого использования - они могут инкапсулировать константы или выступают в качестве родителей других классов. Такое **льготное наследование (facility inheritance)** будет изучаться при обсуждении методологии наследования (см. [лекцию 6](#)).
- Наконец, класс может быть аппликативным - описывающим объекты, не подлежащие модификации. Это означает, что у класса есть только функции, создающие новые объекты. Например, операция сложения в классах INTEGER, REAL и DOUBLE следует математическим традициям - она не модифицирует значение, но, получив x и y, вырабатывает значение: x + y. В спецификации АТД такие функции характеризуются как командные функции.

Во всех этих случаях абстракции обнаруживаются довольно просто, так что не трудно идентифицировать оставшиеся два случая, которые реально могут указывать на дефекты проектирования.

Теперь вернемся к подозрительным случаям. В первом из них появление класса оправдано, команды классу нужны - проектировщик просто забыл обеспечить механизм модификации объектов. Простая техника **контрольной ведомости (checklist)** позволяет избежать подобных ошибок (см. [лекцию 5](#)).

Во втором случае существование класса не оправдано. Он не является настоящей абстракцией данных, представляет пассивную информацию, которая может быть задана структурой, подобной списку или массиву, добавленной в виде атрибута к какому-либо классу. Этот случай встречается, когда разработчик пишет класс, исходя из системы, ранее написанной на Pascal или Ada, отображая запись в класс. Но не все типы записей представляют независимые абстракции данных.

Следует внимательно исследовать подобную ситуацию, чтобы понять, имеем ли мы дело с настоящим классом, теперь или в будущем. Если ответ неясен, то может быть лучше оставить класс, даже с риском неоправданных расходов. Класс влечет некоторую потерю производительности, поскольку приводит к динамическому созданию объектов, для них может потребоваться больше памяти, чем в случае простого массива. Но, если класс действительно понадобится, и он не был введен своевременно, то позднейшие затраты на адаптацию могут быть велики.

Подобная история имела место при разработке ISE компилятора. Для внутренних потребностей идентификации классов решено было использовать целые числа, а не специальные объекты. Все это хорошо работало несколько лет. Но потом схема идентификации усложнилась, в частности пришлось перенумеровывать классы при слиянии нескольких систем в одну. Так что пришлось вводить класс CLASS_IDENTIFIER и заменять экземплярами этого класса прежние целые. Это потребовало усилий больше, чем хотелось бы.

Смешение абстракций

Еще один признак несовершенного проектирования состоит в том, что в одном классе смешиваются две или более абстракций.

В ранней версии библиотеки NeXT текстовый класс обеспечивал полное визуальное редактирование текста. Пользователи жаловались, что, хотя класс полезен, но очень велик. Большой размер класса - это симптом, истинная причина состояла в том, что были слиты две абстракции - собственно редактирование строк и визуализация текста. Класс был разделен на два класса: NSAttributedString, определяющий механизм обработки строк, и NSTextView, занимающийся интерфейсом.

Мэйлор Пейдж Джонс ([Page-Jones 1995]) использует термин **соразвитие (connascence)**, означающий совместное рождение и совместный рост. Для классов соразвитие означает отношение, существующее между двумя тесно связанными компонентами, когда изменение одного влечет одновременное изменение другого. Как он указывает, следует минимизировать соразвитие между классами библиотеки, но компоненты, появляющиеся внутри данного класса, все должны быть связаны одной и той же четко определенной абстракцией.

Этот факт заслуживает отдельного методологического правила, сформулированного в "положительной" форме:

Принцип Согласованности класса

Все компоненты класса должны принадлежать одной, хорошо определенной абстракции.

Идеальный класс

Этот обзор возможных ошибок по контрасту проявляет характерные черты идеального класса. Вот его некоторые типичные свойства:

- Имеется четко ассоциированная с классом абстракция данных (абстрактная машина).
- Имя класса является существительным или прилагательным, адекватно характеризующим абстракцию.
- Класс представляет множество возможных объектов в период выполнения - его экземпляров. Некоторые классы во время выполнения могут иметь только один экземпляр, что тоже приемлемо.
- Для нахождения свойств экземпляра доступны запросы.
- Для изменения состояния экземпляра доступны команды. В некоторых случаях у класса нет команд, но есть функции, создающие новые объекты, что тоже приемлемо.
- Абстрактные свойства могут быть установлены неформально или формально, что предпочтительнее. Они устанавливают, как результаты различных запросов связаны друг с другом (инвариант класса), при каких условиях компоненты класса применимы (предусловия), как выполнение команд сказывается на запросах (постусловия).

Этот список описывает множество неформальных целей, не являясь строгим правилом. Легитимный класс может обладать лишь одним из перечисленных свойств. Большинство из примеров, играющих важную роль в этой книге, - начиная от классов LIST и QUEUE до BUFFER, ACCOUNT, COMMAND, STATE, INTEGER, FIGURE, POLYGON и многих других, - обладают всеми этими свойствами.

Общие эвристики для поиска классов

Давайте теперь обратимся к положительной части нашего обсуждения - практическим эвристикам поиска классов.

Категории классов

Можно выделить три категории классов: классы анализа, классы проектирования, классы реализации. Это деление не является ни абсолютным, ни строгим (например, кто-то может привести аргументы в поддержку того, что отложенный класс LIST принадлежит к любой из трех категорий), но такое деление удобно как общее руководство.

Классы анализа описывают абстракцию данных, непосредственно выводимую из модели внешней системы. Типичными примерами являются классы PLANE в системе управления полетами, PARAGRAPH в системе обработки документов, PART в системе управления запасами.

Классы реализации описывают абстракцию данных, введенную, исходя из внутренних потребностей алгоритмов системы, например LINKED_LIST или ARRAY.

Классы проектирования описывают архитектурный выбор. Примерами могут служить класс COMMAND в системе, реализующей откаты, класс STATE в системе, управляемой панелями. Подобно классам реализации, классы проектирования принадлежат **пространству решений**, в то время как классы анализа принадлежат пространству проблемной области. Но подобно классам анализа и в отличие от классов реализации они описывают концепции высокого уровня.

Когда мы научимся получать классы всех трех категорий, мы обнаружим, что наиболее трудно идентифицировать классы проектирования, требующие архитектурной интуиции. Заметьте, сложность в выявлении этих классов не означает, что их трудно **построить**. Сложностью построения скорее отличаются классы реализации, если только мы не используем готовую библиотеку.

Внешние объекты: нахождение классов анализа

Давайте начнем с классов анализа, моделирующих внешние объекты.

Мы используем ПО для получения ответов на некоторые вопросы о внешнем мире, для взаимодействия с этим миром, для создания новых сущностей этого мира. В каждом случае ПО должно основываться на некоторой модели мира, на законах физики или биологии в научных программах, на синтаксисе и семантике языка программирования при построении компилятора, налоговых постановлений в системе расчета налогов.

В нашем разговоре мы избегаем термина "реальный мир", вводящего в заблуждение, поскольку ПО не менее реально, чем что-либо другое. Миры, которыми интересуются при создании ПО, зачастую искусственны, как, например, миры математика. Мы должны говорить о внешнем мире в противовес внутреннему миру ПО.

Любая программная система основывается на операционной модели некоторых аспектов внешнего мира. Операционной, поскольку используется для генерирования практических результатов, а иногда и для создания обратной связи, возвращая результаты во внешний мир. Модель, потому что любая полезная система должна следовать определенной интерпретации некоторых феноменов этого мира.

Эта точка зрения наиболее явно проявляется в такой области как **моделирование (simulation)**. Неслучайно, что первый ОО-язык программирования Simula 67 вырос из Simula 1 - языка моделирования дискретных событий. Хотя Simula 67 является универсальным языком общего назначения, он сохранил имя своего предшественника и включил множество мощных примитивов моделирования. В семидесятые годы моделирование являлось принципиальной областью приложения объектной технологии. Привлекательность ОО-идей для моделирования легко понять - создавать структуру программной системы, моделирующей поведение множества внешних объектов, проще всего при прямом отображении этих объектов в программные компоненты.

В широком смысле построение любой программной системы является моделированием. Исходя из операционной модели, ОО-конструирование ПО использует в качестве первых абстракций некоторые типы, непосредственно выводимые из анализа объектов в непрограммном смысле этого термина, объектов внешнего мира: датчиков, устройств, самолетов, служащих, банковских счетов, интегрируемых функций.

Эти примеры рисуют лишь часть общей картины. Как заметили Валден и Нерсон ([Walden 1995]) в представлении метода В.О.Н: "Класс, описывающий автомобиль, не более осязаем, чем тот, который моделирует удовлетворенность служащих своей работой"

Следует всегда иметь в виду этот комментарий при поиске внешних классов - они могут быть довольно абстрактными: SENIORITY_RULE в парламентской системе голосования, MARKET_TENDENCY в рыночной системе могут быть также реальны, как SENATOR и STOCK_EXCHANGE. Улыбка Чеширского Кота - такой же объект, как и сам Чеширский Кот.

Будучи материальными или абстрактными, внешние классы, используемые специалистами, всегда дают хороший шанс породить полезные внутренние классы. Но ключом остается абстракция. Хотя и желательно добиваться соответствия классов анализа концепциям проблемной области, но не эта близость делает класс удачным. Первая версия нашей системы, управляемой панелями, драматично это показала, - она была прекрасной моделью, построенной по образцу внешней системы, но оказалась ужасной с позиций инженерии программ. Хороший внешний класс должен базироваться на абстрактных концепциях проблемной области, характеризуемых внешними свойствами долговременной значимости.

Для ОО-разработчиков предварительно существующие абстракции являются драгоценными - они дают некоторые из фундаментальных классов системы, но, отметим еще раз, являются объектами для прополки.

Нахождение классов реализации

Классы реализации описывают структуры, создаваемые разработчиками по внутренним соображениям, диктуемым реализацией системы. Хотя в литературе по программной инженерии часто принято принижать роль реализации, отводя ей роль кодирования, разработчики хорошо знают, реализация требует немалого интеллекта, и на нее приходится большая часть усилий по разработке системы.

Плохая новость - классы реализации трудно **строить**. Хорошая новость - их легко **выявить**. Для них может существовать хорошая библиотека, допускающая повторное использование. По крайней мере, есть хорошая литература по этой тематике. Курс "Алгоритмы и структуры данных", иногда известный как CS 2, является необходимым компонентом образования в области информатики. Хотя большинство из существующих учебников явно не используют ОО-подход, многие следуют стилю АТД. Преобразование в классы в этом случае довольно прямолинейно и естественно.

В настоящее время некоторые учебники начали применять ОО-подход при изложении традиционных тем CS 2.

Каждый разработчик, независимо от того, проходил ли он этот курс, должен держать на своей книжной полке учебники по структурам данных и алгоритмам и обращаться к ним довольно часто. Легко ошибиться и выбрать неверное представление или неэффективный алгоритм, например, применить односвязный список к последовательной структуре, где по алгоритму требуется проходы в обоих направлениях, или использовать массив для структуры, растущей или сжимающейся непредсказуемым образом. Заметьте, в вопросах реализации по-прежнему правит АТД-подход - структуры данных и их представление следуют из служб, предлагаемых клиенту.

Помимо учебников и опыта лучшим вариантом для классов реализации являются библиотеки повторного использования.

Отложенные классы реализации

В традиционных учебниках естественно описываются эффективные (полностью реализованные) классы. На практике ценность большинства классов реализации, особенно, когда предполагается их повторное использование, связана с таксономией - структурой наследования, включающей отложенные классы. Например, различные реализации очереди могут быть потомками отложенного класса QUEUE, описывающего абстрактные концепции.

"Отложенный класс реализации" - это не нелепица (охутогон). Классы, подобные QUEUE, хотя и абстрактны, но помогают построить таксономию с согласованными вариациями структур реализации, отводя каждому классу точное место в общей схеме.

В одной из своих книг ([М 1993]) я описал "Линнеевскую" таксономию фундаментальных структур информатики, в основе которой лежат отложенные классы, характеризующие принципиальные виды структур данных, используемых при разработке ПО.

Нахождение классов проектирования

Классы проектирования представляют архитектурные абстракции, помогающие создавать элегантные расширяемые программные структуры. Хорошими примерами являются классы: STATE, APPLICATION, COMMAND, HISTORY_LIST, итератор и контроллер. Мы увидим и другие полезные идеи в последующих лекциях, такие как активные структуры данных и описатели ("handles").

Хотя, как отмечалось, нет уверенного способа найти классы проектирования, дадим несколько полезных советов - это лучше, чем ничего.

- Многие классы проектирования были изобретены уже до нас. Так что чтение книг и статей, описывающих решения проблем проектирования, может дать много плодотворных идей. Например, книга "Объектно-ориентированные приложения" ([М 1993]) содержит лекции, написанные ведущими разработчиками различных промышленных проектов. Приводятся точные и детальные архитектурные решения, полезные в таких областях, как телекоммуникации, автоматизированное проектирование, искусственный интеллект, и других проблемных областях.
- Книга Гаммы ([Gamma 1995]) посвящена образцам проектирования, за ней последовали другие подобные книги.
- Многие полезные классы проектирования лучше понимаются как "машины", чем как "объекты" в общем смысле этого слова.
- Как и в случае классов реализации, повторное использование лучше, чем изобретение. Можно надеяться, что текущие образцы перестанут быть просто идеями и превратятся в непосредственно используемые библиотечные классы.

Другие источники классов

Несколько эвристик доказали свою полезность в битвах за правильные абстракции.

Предыдущие разработки

Совет искать то, что доступно, применим не только к библиотечным классам. Когда вы пишете приложения, вы аккумулируете классы, облегчающие при правильном проектировании дальнейшие разработки.

Не все повторно используемое ПО таким и рождалось. Зачастую первая версия класса рождается для удовлетворения текущих потребностей. Однако затем, через некоторое время после разработки выясняется, что стоит затратить усилия на то, чтобы сделать класс общим и более устойчивым, улучшить документацию, добавить утверждения. Это отличается от ситуации, когда класс создается с самого начала как повторно используемый, но не менее плодотворно. Включенные в реальную систему классы проходят первую проверку на повторное использование, называемую **используемость**, - они служат, по крайней мере, одной полезной цели.

Адаптация через наследование

При обнаружении потенциально полезного класса иногда обнаруживается, что он не в полной мере отвечает потребностям и требует адаптации.

Если у класса нет дефектов, требующих их устранения в оригиналe, то обычно предпочтительнее оставить класс в целости и сохранности, заботясь о его клиентах в полном соответствии с принципом Открыт-Закрыт. Вместо этого можно использовать наследование и переопределение, настраивающее класс (потомка) на новые потребности.

Эта техника (см. [лекцию 10](#)), которая еще будет изучаться в деталях под именем **вариационное наследование (variation inheritance)**, предполагает, что новый класс задает вариант той же абстракции, что и оригинал. При подходящем использовании она представляет наиболее значительный вклад в Метод, позволяя разрешить проблему **reuse-redo** - сочетание повторного использования с расширяемостью.

Оценивание кандидатов декомпозиции

Критиковать проще, чем создавать. Один из способов обучения проектированию состоит в анализе существующих проектов. В частности, когда некоторое множество классов предлагается для решения определенной проблемы, следует проанализировать их в соответствии с критериями и принципами модульности, представленными в [лекции 3](#) курса "Основы объектно-ориентированного программирования", - составляют ли они автономные, согласованные модули со строго ограниченными каналами коммуникации? Часто обнаруживаются модули, тесно связанные, взаимодействующие с большим числом модулей, имеющие длинные списки аргументов, - все это указывает на ошибки проектирования, устранение которых ведет к построению лучшего решения.

Важный критерий исследовался (см. [лекцию 2](#)) в примере системы, управляемой панелями, - потоки данных. Мы видели тогда, как важно при рассмотрении структуры класса кандидата анализировать потоки объектов, передаваемых как аргументы в последовательных вызовах. Если, как это было с понятием состояния, обнаруживается, что некоторый элемент информации передается многим модулям, то это определенный признак того, что пропущена важная абстракция данных. Такой анализ привел нас к необходимости введения класса STATE, важного источника абстракции.

Конечно, предпочтительнее найти правильные классы с самого начала. При позднем, апостериорном обнаружении следует найти время на анализ причин, почему важная абстракция была пропущена, чтобы извлечь уроки на будущее.

Найдки других подходов

Пример анализа потока данных в исходящей структуре иллюстрирует идею выявления класса при рассмотрении необъектной декомпозиции. Это полезно в двух непересекающихся случаях:

- Может существовать не ОО-система, выполняющая свою часть работы. Тогда разумно провести ее анализ с позиций классов. Иногда, вместо работающей системы, используются результаты анализа или проектирования, выполненного другими, старыми методами.
- Некоторые из разработчиков могут иметь большой опыт работы в создании не объектных систем и, как следствие, вначале проектируют систему в терминах других концепций, затем реализуют ее в виде классов.

Вот примеры этого процесса, начиная с языков программирования и заканчивая методами анализа и проектирования.

Программы Fortran включают обычно один или несколько **общих блоков (common blocks)** - данных, разделяемых многими подпрограммами. Зачастую за общими блоками стоят очень важные абстракции данных. Более точно, хорошие Fortran программисты знают, что в общий блок следует включать те переменные и массивы, которые соответствуют тесно связанным понятиям, и в этом случае за этим стоит шанс создать класс на основе общего блока. К сожалению, это не универсальная практика, - в начале этой книги упоминалось о "мусорном" общем блоке, куда сваливают все подряд. В этом случае анализ должен проводиться куда более тщательно для обнаружения подходящих абстракций.

Программы Pascal и С используют записи, известные в С как структуры. Они могут также соответствовать классам при условии выявления операций над данными записей. Если это не так, то запись будет представлена атрибутами некоторого класса.

Структуры Cobol и его секции данных (Data Division) помогают идентифицировать важные типы данных.

При рассмотрении моделей, основанных на понятиях "сущность-отношение", сущности ("entities") часто служат основой для построения классов.

При проектировании потоков данных (dataflow) немногое может быть непосредственно использовано для ОО-целей, но иногда "хранилища" (stores) могут приводить к нужным абстракциям.

Файлы

"Хранилища" несут общую полезную идею. Иногда большая часть информации традиционных систем находится не в их программном тексте, а в структуре используемых файлов.

Для всякого с опытом Unix эта идея достаточно ясна: основная документация содержит описание не столько

специфических команд, а описание ключевых файлов и их форматов: `passwd` для паролей, `printcap` для свойств принтера, `termcap` или `terminfo` для свойств терминала. Эти файлы можно характеризовать как абстракции данных без абстракции - документированные на совершенно конкретном уровне ("Каждый вход в `printcap` файле описывает принтер и представляет строку, состоящую из полей, разделенных символом двоеточия" и т. д.). Файлы задают важные типы данных, доступные через хорошо определенные примитивы с ассоциированными свойствами и условиями использования. При переходе к ОО-подходу такие файлы должны играть центральную роль.

Подобное наблюдение применимо ко многим программам, использующим файлы. Однажды я консультировал менеджера программной системы, убежденного, что его система - коллекция Fortran программ - не может использоваться в целях ОО-декомпозиции. Когда он описывал, что делают его программы, он упоминал о нескольких файлах, обеспечивающих взаимодействие между программами. Я начал задавать вопросы об этих файлах, но он полагал, что они не имеют отношения к делу, считая их неважными. Я настаивал, и из объяснений стало понятно, что файлы описывают сложные структуры данных, охватывающие основные понятия программы. Урок ясен: с осознанием важности файлов пришло понимание, что они должны играть центральную роль в ОО-архитектуре, а бывшие ключевые элементы стали играть вспомогательную роль компонентов классов.

Использование ситуаций

Ивар Якобсон ([Jacobson 1992]) пропагандирует использование ситуаций для выявления классов. Ситуации, называемые также **сценариями (scenario)** или **трассами**, описываются как

полный набор событий, инициированных пользователем или системой, и взаимодействий между пользователем и системой.

В телефонной системе ситуация, например "вызов, инициированный заказчиком", приводит к последовательности событий: заказчик поднимает трубку телефона, системе посыпается сигнал идентификации, система вырабатывает гудок и так далее.

Использование ситуаций не самый лучший способ нахождения классов. Здесь возникает несколько рискованных моментов:

- В сценариях предполагается упорядоченность. Это несовместимо с объектной технологией (см. [лекцию 7](#) курса "Основы объектно-ориентированного программирования"). Не следует основываться на порядке действий, поскольку порядок в первую очередь подвержен изменениям. Не следует фокусироваться на свойствах в форме: "система выполняет а, затем б"; вместо этого следует задавать вопрос: "Какие операции доступны для экземпляров абстракции А, и каковы ограничения на эти операции?" По-настоящему фундаментальные свойства, отражающие последовательность выполнения операций, задаются ограничениями высокого уровня. Утверждение, что для стека число операций `pop` не должно превосходить число операций `push`, можно выразить в более абстрактной форме, используя пред и постусловия этих операций. Менее абстрактные свойства порядка вообще не должны учитываться на этапе анализа. При работе со сценариями возможность таких ошибок велика.
- Сценарии также предполагают фокусирование на пользовательском видении операций. Но система пока еще не существует. Может существовать ее предыдущая версия, но, если бы она была полностью удовлетворительной, то не возникла бы потребность в ее переписывании. Ваша задача состоит в том, чтобы предложить лучший сценарий. Есть много примеров неудач, связанных с рабским повторением существующих процедур.
- Использование сценариев предпочитают при функциональном подходе, основанном на процессах (действиях). Сохраняется опасность, что под маской классов скрывается традиционная форма функционального проектирования. Этот подход противоположен ОО-декомпозиции, сфокусированной на абстракции данных. Использование нескольких сценариев исключает одну главную программу, но и здесь начальной точкой остается вопрос, что делает система, в отличие от объектного подхода, где важнее, кто это делает. Дисгармония неизбежна.

Практические следствия очевидны:

Принцип использования сценариев

За исключением очень опытных команд разработчиков (имеющих опыт создания нескольких систем, содержащих несколько тысяч классов в чистом ОО-языке) не следует основываться на сценариях как средстве ОО-анализа и проектирования.

Из этого принципа не следует, что сценарии являются бесполезной концепцией. Они являются потенциально значимыми, но их роль в ОО-конструировании ПО неверно понимается. Они являются не средством анализа, а инструментом **проверки правильности (validation)**. Если, как и должно быть, в вашей команде разработчиков имеется независимая группа тестирования, то сценарии будут являться их инструментом, позволяющим проверить, способна ли система корректно выполнять различные сценарии, как они виделись

пользователям. В некоторых случаях может оказаться, что система поддерживает другой сценарий, но дающий тот же или лучший результат. Это, конечно же, вполне приемлемо.

Еще одно применение сценариев связано с заключительными аспектами реализации - система может включать специальные программы для запуска типичных сценариев. Такие программы зачастую задают некоторый вид абстрактного поведения, описывая общую схему, которая может быть переопределена различными способами. В книге [Jacobson 1992] вводится понятие абстрактного сценария, что в объектной терминологии называется классом поведения.

В этих двух ролях - механизме проверки и управления реализацией - использование сценариев дает несомненную пользу. Но в объектной технологии они не являются полезным средством анализа или проектирования. Разработчикам следует концентрироваться на абстракциях, а не на том, в каком порядке выполняются операции.

KOC (CRC) карты

Для полноты картины упомянем идею, рассматриваемую иногда как метод нахождения классов. Карты KOC (**Класс, Ответственность, Сотрудничество**) или CRC (**Class, Responsibility, Collaboration**) являются бумажными карточками, используемыми разработчиками при обсуждении потенциальных классов, в терминах их ответственостей и взаимодействия. Идея проста, отличается дешевизной - набор карточек дешевле рабочей станции с CASE инструментарием. Но его технический вклад в процесс проектирования неясен.

Повторное использование

Не изобретать классы, а получить их из библиотеки - вот простейший и наиболее продуктивный способ нахождения классов. Помимо прочего гарантируется проверка работоспособности, основанная на опыте предыдущих пользователей библиотеки.

Подход снизу вверх

Начиная с этапа анализа, следует применять разработку снизу вверх. Подход, сосредоточенный исключительно на документе с требованиями и запросами пользователей (как это делается в case-технологии) приводит к системам, требующим больших затрат и не учитывающим важного понимания сути, достигнутой в предыдущих проектах. Одной из задач команды разработчиков является, начиная с фазы рассмотрения требований к системе, учет того, что уже доступно, как существующие классы могут помочь в новой разработке. В ряде случаев это приводит к пересмотру требований.

Довольно часто, когда мы говорим о нахождении классов, подразумевается их **изобретение (devising)**. В объектной технологии с ростом качества библиотек и осознания идей повторного использования приобретает смысл именно **поиск (finding)** классов.

Сказка о поиске классов

Жил да был в стране ООП молодой человек, которому страстно хотелось узнать секрет нахождения классов. Он расспрашивал всех местных мастеров, но никто из них не знал этой тайны.

Будучи подвергнутым публичной епитимии Схемом-Блоком - аббатом Священного Порядка Стрелок и Пузырей, - он решил, что тому известна истина и настал конец его поискам. С трудом он отыскал пещеру Схема, вошел в нее и увидел Схема, погруженного в поиски вечного различия между Классами и Объектами. Осознав, что не здесь обретается истина, без единого вопроса он покинул пещеру и отправился в дальнейшие странствия.

Однажды он нечаянно подслушал разговор двух людей, один из которых занимался вталкиванием какого-то груза в тележку, а другой немедленно его выталкивал. Они шептались о неком старце, знающем секрет классов. Молодой человек решил найти этого великого Мастера. Многое дорог он прошел, на многие горы он поднимался, многие реки пересекал, пока не нашел убежище Мастера. Поиски продолжались так долго, что он давно уже перестал быть молодым человеком. Но, как всем пилигримам, ему еще предстояло пройти очищение в течение тридцати трех месяцев, прежде чем он был допущен к объекту своих поисков.

Наконец, в один из темных зимних дней, когда снег покрыл все окружающие горные вершины, он вошел в комнату Мастера. С бьющимся сердцем, пересохшим от волнения голосом он задал свой сакральный вопрос: "Мастер, как мне найти классы?"

Мудрец склонил свою голову и ответил медленно и спокойно: "Возвращайся назад, откуда пришел. Классы уже найдены".

Оглушенный, он и не заметил, как слуги Мастера выводили его прочь. "Мастер", - теперь он почти кричал, - "пожалуйста, еще только один вопрос. Как называется эта история?" Старый Учитель покачал головой: "Разве

ты еще не понял? Это сказка о повторном использовании".

Метод получения классов

Мало-помалу идеи, обсуждаемые в этой лекции, в совокупности дают то, что не слишком претенциозно можно называть методом получения классов при конструировании ПО (напомним, метод - это способ породить, воспитать, проложить дорогу, создать нечто стоящее).

Идентификация класса требует двух неразрывно связанных видов деятельности - выявления множества кандидатов в классы и отбора среди них действительно нужных. В двух следующих таблицах подводятся итоги.

Прежде всего начнем с источников классов.

Таблица 4.1. Источники возможных классов

Источник идей	Что ищется
Существующие библиотеки	<ul style="list-style-type: none">Классы, отвечающие потребностям приложения.Классы, описывающие концепции, релевантные приложению.
Документ требований	<ul style="list-style-type: none">Часто встречающиеся термины.Термины, заданные явными определениями.Термины, не определенные точно, но считающиеся само собой разумеющимися.(Грамматические категории следует игнорировать.)
Обсуждения с заказчиками и будущими пользователями	<ul style="list-style-type: none">Важные абстракции проблемной области.Специфический жаргон проблемной области.Помнить, что классы, приходящие из "внешнего мира", могут описывать как материальные, так и концептуальные объекты.
Документация (руководства пользователя) для других систем в той же проблемной области, например от конкурентов	<ul style="list-style-type: none">Важные абстракции проблемной области.Специфический жаргон проблемной области.Полезные абстракции проектирования.
Не ОО-системы и их описания	<ul style="list-style-type: none">Элементы данных, передаваемые в виде аргументов компонентам ПО.Разделяемые данные (Common блоки FORTRAN).Важные файлы.Секции данных (COBOL).Типы записей (Pascal, C, C++).Сущности при ER-моделировании.
Обсуждения с опытными проектировщиками	<ul style="list-style-type: none">Классы проектирования, успешно используемые в предыдущих разработках.
Литература по алгоритмам и структурам данных	<ul style="list-style-type: none">Известные структуры данных, поддержанные эффективными алгоритмами.
Литература по ОО-проектированию	<ul style="list-style-type: none">Применимые образцы проектирования.

Рассмотрим теперь критерии, позволяющие более внимательно исследовать классы и, возможно, отвергнуть часть из них.

Таблица 4.2. Причины отбраковки кандидатов в классы

Сигналы опасности	Причина подозрительности
Класс с вербальным именем (инфinitив или императив)	Может быть простой подпрограммой, а не классом
Полностью эффективный класс с одной экспортруемой подпрограммой	Может быть простой подпрограммой, а не классом.

Класс, описанный как "выполняющий нечто"	Может не быть подходящей абстракцией.
Класс без подпрограмм	Может быть важной частью информации, но не АТД. Может быть АТД, для которого просто забыли указать операции.
Класс, введенный без компонентов или с небольшим их числом (но наследующий компоненты своих родителей)	Может быть результатом "таксомании".
Класс, покрывающий несколько абстракций	Должен быть разделен на несколько классов, по одному на каждую абстракцию.

Ключевые концепции

- Идентификация классов - одна из принципиальных задач ОО-конструирования ПО.
- Идентификация классов - двойственный процесс - предложение кандидатов и их отбор. Нужно уметь находить потенциальных кандидатов и уметь отсеивать неподходящих.
- Идентификация классов - это идентификация подходящих абстракций в моделируемой области пространстве решений.
- "Подчеркивание существительных в документе требований" - это не подходящая техника для обнаружения потенциальных классов, так как ее результаты зависят от стиля написания документа. Она может приводить как к появлению лишних кандидатов, так и к пропуску нужных.
- Классы разделяются на три группы. Классы анализа связаны с концепциями моделируемого внешнего мира. Классы проектирования описывают архитектурные решения. Классы реализации описывают структуры данных и алгоритмы.
- Классы проектирования обычно требуют наибольшей изобретательности.
- При проектировании внешних классов помните, что внешние объекты включают концепции наряду с материальными предметами.
- Применяйте критерий абстракции данных всякий раз, когда нужно решить, представляет ли данное понятие настоящий класс.
- Классы реализации включают как эффективные, так и отложенные классы, описывающие абстрактные категории.
- Наследование обеспечивает повторное использование с одновременной адаптацией к изменившимся условиям.
- Способ получения классов состоит в оценке кандидатов и поиске необнаруженных абстракций, в частности путем анализа межмодульных передач данных.
- Использование Case-технологии или сценариев может быть полезно как средство проверки правильности и как руководство на заключительных этапах реализации, но не должно использоваться на этапах анализа и проектирования.
- Лучшим источником классов являются библиотеки повторного использования.

Библиографические замечания

Советы по использованию существительных документа требований в качестве начальной точки нахождения классов стали популярными благодаря Гради Бучу [Booch 1986], который заимствовал эту идею из ранней статьи Эббота [Abbot ACM, 26, 1983]. Дальнейшие рекомендации появились в [Wirfs-Brock 1990].

Статья по формальным спецификациям [M 1985a] анализировала проблемы, связанные с естественным языком документа требований. В ней изучались описания, широко используемые в литературе по верификации программ, и была введена таксономия возникающих дефектов: шум, двусмысленность, противоречие, излишняя спецификация, ссылки вперед. В ней обсуждалось, как формальные спецификации могут справиться с некоторыми из проблем.

[Walden 1995] рассмотрел полезные рекомендации для идентификации классов.

В приложении В [Page-Jones 1995] перечислены "симптомы" отбраковки кандидатов, уведомляющие проектировщиков об опасных сигналах, подобно тем, что были рассмотрены в данной лекции.

[Ong 1993] описывает инструментарий преобразования не ОО-программ, главным образом Fortran программ в ОО-форму. Преобразование является полуавтоматическим. Автор описывает некоторые из эвристик идентификации классов, согласующиеся с данной лекцией, в частности COMMON-блоки.

Simula 1 описана в [Dahl 1966]. Более подробное описание языка Simula дается в [лекции 17](#).

Книги по типовым структурам данных являются надежным источником классов реализации, включая известный трехтомник Кнута [Knuth 1981, Knuth 1973], [Aho 1974, Aho 1983].

[Gore 1996] представляет фундаментальные структуры данных и алгоритмы полностью в ОО-манере.

Источники классов проектирования приведены в [Gamma 1995], где даны образцы проектирования, и в [M1994], содержащей библиотеку классов и обсуждающей в деталях понятия "класса описателя" и "класса итератора". В книге [Krief 1996] представлена модель Smalltalk MVC.

Упражнения

У4.1 Floors как integers

Определите класс FLOOR как наследника INTEGER, ограничив применимые операции.

У4.2 Инспектирование объектов

Даниел Холберт и Патрик О-Брайен обсуждали проблему, возникающую при проектировании окружения разработки ПО:

Рассмотрим свойство `inspector`, используемое для отображения информации об объекте в окне отладки. Для разных объектов нужны разные инспекторы. Например, информация о точке может быть выведена в простом формате, а о большом идвумерном массиве может потребовать вертикального и горизонтального скроллинга.

Прежде всего следует решить, где описать поведение инспектора - в классе, связанном с инспектируемым объектом, или в отдельном классе?

Отвечая на это вопрос, рассмотрите все за и против каждого варианта. Заметьте, могут оказаться полезными результаты обсуждения, приведенные в последующих лекциях, посвященных наследованию.

Основы объектно-ориентированного проектирования

5. Лекция: Принципы проектирования класса

Опытные разработчики ПО знают, что одной из наиболее критичных проблем является проблема проектирования интерфейсов модуля. В больших и длительных проектах многие обсуждения и недоразумения связаны со спецификациями интерфейса модулей: "Но я полагал, что вы передаете мне уже нормализованные данные...", "Зачем Вы делаете эту обработку, я ведь уже об этом позаботился?.." и так далее. Если от объектной технологии ожидать только одного преимущества, то следовало бы указать на проектирование интерфейсов. В самом начале мы определяли ОО-технологию как архитектурную технику производства программных систем, построенных из модулей, согласованных по интерфейсу. Теперь у нас есть техническая база, позволяющая пользоваться наилучшими ОО-механизмами для построения модулей с привлекательными интерфейсами. Поскольку успех класса определяется тем, насколько он привлекателен для своих клиентов, то наше внимание будет уделяться не только в внутренней реализации, а тому, насколько прост его интерфейс, легок в обучении, прост для запоминания, способен выдержать проверку временем и изменениями. В задачу нашего исследования входит ряд важных вопросов: должны ли функции допускать побочные эффекты, много ли аргументов должно быть у компонентов, чем отличаются связанные понятия операнда и опции (option), следует ли сосредоточиваться на размерах класса, как сделать абстрактные структуры активными, роль выборочного экспорта, как документировать класс, что делать в особых случаях? Из нашего обсуждения будет ясно, что проектировщик класса должен как опытный мастер довести свое изделие до совершенства, полируя до блеска, делая его привлекательным для клиентов, насколько это только возможно. Этот дух рассмотрения классов как тщательно сработанных инженерных изделий, совершенных по замыслу, исполнению и всегда остающимися таковыми, является в ее объемлющим качеством хорошо определенной объектной технологии. По очевидным причинам он особенно проявляется при конструировании библиотек классов. Оригинальные конструктивные разработки вначале проверяются на машинах, участвующих в гонках Formula 1, прежде чем они станут достоянием автомобильной промышленности. Доказав применимость при разработке успешной библиотеки повторно используемых компонентов, рассматриваемые идеи обеспечивают преимущество и любому ОО-ПО независимо от того, предполагалось ли его повторное использование с самого начала.

Побочные эффекты в функциях

Первый вопрос, исследованием которого мы займемся, оказывает глубокое влияние на стиль нашего проектирования. Законно ли для функций - подпрограмм, возвращающих результат, - иметь еще и побочный эффект, то есть изменять нечто в их окружении?

Наш ответ - нет. Но почему? Обоснование требует понимания роли побочных эффектов, осознания различий между "хорошим" и "плохим" побочным эффектом. Рассмотрим этот вопрос в свете наших знаний о классах - их происхождения от АТД, понятия абстрактной функции и роли инварианта класса.

Команды и запросы

Напомним используемую терминологию. Компоненты, характеризующие класс разделяются на **команды и запросы**. Команды модифицируют объекты, а запросы возвращают информацию о них. Команды реализуются процедурами, а запрос может быть реализован либо атрибутом - тогда в момент запроса возвращается значение соответствующего поля экземпляра класса, либо функцией - тогда происходит вычисление значения по алгоритму, заданному функцией. Процедуры и функции называются подпрограммами.

В определении запроса не сказано, могут ли изменяться объекты в момент запроса. Для команд ответ очевиден - да, поскольку в этом и состоит их назначение. Для запросов вопрос имеет смысл только в случае их реализации функциями, поскольку доступ к атрибуту ничего не меняет. Изменение объектов, выполняемое функцией, называется ее **побочным эффектом (side effect)**. Функция с побочным эффектом помимо основной роли - возвращения ответа на запрос, меняя объект, играет одновременно и дополнительную роль, которая часто является фактически основной. Но следует ли допускать побочные эффекты?

Формы побочного эффекта

Определим, какие конструкции могут приводить к побочным эффектам. Операциями, изменяющими объекты, являются: присваивание `a := b`, попытка присваивания `a = b`, инструкция создания `create a`. Если цель `a` является атрибутом, то выполнение операции присвоит новое значение его полю для объекта, соответствующего цели текущего вызова подпрограммы.

Нас будут интересовать только такие присваивания, в которых `a` является атрибутом; если же `a` - это локальная сущность, то его значение используется только в момент выполнения подпрограммы и не имеет постоянного эффекта, если `a` - это `Result`, присваивание вычисляет результат функции, но не действует на объекты.

Заметим, что, применяя принципы скрытия информации, мы при проектировании ОО-нотации тщательно избегали любых косвенных форм модификации объектов. В частности, синтаксис исключает присваивания в форме `obj.attr := b`, чья цель должна быть достигнута через вызов `obj.set_attr (b)`, где процедура `set_attr (x:...)` выполняет присваивание атрибуту `attr := x` (см. [лекцию 7](#) курса "Основы объектно-ориентированного программирования").

Присваивание атрибуту, ставшее причиной побочного эффекта, может находиться в самой функции или встроено глубже - в другой подпрограмме, вызываемой функцией. Вот полное определение:

Определение: конкретный побочный эффект

Функция производит конкретный побочный эффект, если ее тело содержит:

- присваивание, попытку присваивания или инструкцию создания, чьей целью является атрибут;
- вызов процедуры.

Термин "конкретный" будет пояснен ниже. В последующем определении мы второе предложение сформулируем как "вызов подпрограммы, создающей (рекурсивно) конкретный побочный эффект". Определение побочного эффекта будет расширено и не будет, как теперь, относиться только к функциям. Но выше приведенное определение на практике предпочтительнее, хотя по разным причинам его можно считать либо слишком строгим, либо слишком слабым:

- Определение кажется слишком строгим, поскольку любой вызов процедуры рассматривается как создающий побочный эффект, в то время как можно написать процедуру, ничего не меняющую в мире объектов. Такие процедуры могут менять нечто в окружении: печатать страницу, посыпать сообщения в сеть, управлять рукой робота. Мы будем рассматривать это как своего рода побочный эффект, хотя программные объекты при этом не меняются.
- Определение кажется слишком слабым, поскольку оно игнорирует случай функции f ,зывающей функцию g с побочным эффектом. Соглашение состоит в том, что в этом случае сама f считается свободной от побочного эффекта. Это допустимо, поскольку правило, которое будет выработано в процессе нашего рассмотрения, будет запрещать все побочные эффекты определенного вида, так что нет необходимости в независимой сертификации каждой функции.

Преимущество этого соглашения в том, что для определения статуса побочного эффекта достаточно анализировать тело только самой функции. Достаточно тривиально, имея анализатор языка, встроить простой инструментарий, который для каждой функции независимо определял бы, обладает ли она конкретным побочным эффектом в соответствии с данным определением

Ссылочная прозрачность

Почему нас волнуют побочные эффекты функций? Ведь в природе ПО заложено изменение вещей в процессе выполнения.

Если позволить функциям, подобно командам, изменять объекты, то мы потеряем многие из их простых математических свойств. Как отмечалось при обсуждении АТД (см. [лекцию 6](#) курса "Основы объектно-ориентированного программирования"), математики знают, что их операции над объектами не меняют объектов (Вычисление $|2^{1/2}|$ не меняет числа 2). Эта неизменяемость является основным отличием мира математики и мира компьютерных вычислений.

Некоторые подходы в программировании стремятся к этой неизменяемости - Lisp в его так называемой "чистой" форме, языки функционального программирования, например язык FP, предложенный Бэкусом, другие аппликативные языки. Но в практической разработке ПО изменения объектов являются основой вычислений.

Неизменяемость объектов имеет важное практическое следствие, известное как **ссылочная прозрачность** (**referential transparency**) и определяемое следующим образом:

Определение: ссылочная прозрачность

Выражение e является ссылочно-прозрачным, если возможно заменить любое его подвыражение эквивалентным значением без изменения значения e .

Если x имеет значение 3, мы можем использовать x вместо 3, и наоборот, в любом ссылочно-прозрачном выражении. (Только академики Лапуты из "Путешествий Гулливера" Свифта игнорировали ссылочную прозрачность, - они всегда носили с собой вещи, предъявляя их при каждом упоминании.) Ссылочную прозрачность называют также "заменой равного равным".

При наличии функций с побочным эффектом ссылочная прозрачность исчезает. Предположим, что класс содержит атрибут и функцию:

```
attr: INTEGER
sneaky: INTEGER is do attr := attr + 1 end
```

Значение $sneaky$ при ее вызове всегда 0; но 0 и $sneaky$ не являются взаимозаменяемыми, например:

```
attr := 0; if attr /= 0 then print ("Нечто странное!") end
```

ничего не будет печатать, но напечатает "Нечто странное!" при замене 0 на $sneaky$.

Поддержка ссылочной прозрачности в выражениях важна, поскольку позволяет строить выводы на основе программного текста. Одна из центральных проблем конструирования ПО четко сформулирована Э. Дейкстрой ([Dijkstra 1968]). Она состоит в сложности динамического поведения (миллионы различных вычислений даже для простых программ), порождаемого статическим текстом программы. Поэтому крайне важно сохранить проверенную форму вывода, обеспечиваемую математикой. Потеря ссылочной прозрачности означает и потерю основных свойств, которые настолько укоренились в нашем сознании и практике, что мы и не осознаем этого. Например, $n + n$ не эквивалентно $2^* n$, если n задано функцией, подобной $sneaky$:

```
n: INTEGER is do attr := attr + 1; Result := attr end
```

Если attr инициализировать нулем, то $2^* n$ возвратит 2, в то время как $n + n$ вернет 3.

Функции без побочных эффектов можно рассматривать в программных текстах как термы в обычном математическом смысле. Мы будем поддерживать четкое различие команд, изменяющих объекты, но не возвращающих результатов, и запросов, обеспечивающих информацией об объектах, но не изменяющих их.

Это же правило неформально можно выразить так: **"задание вопроса не меняет ответ"**.

Объекты как машины

Следующий принцип выражает этот запрет в более точной форме:

Принцип: Разделение Команд и Запросов

Функции не должны обладать абстрактным побочным эффектом.

Заметьте, к этому моменту мы определили только понятие конкретного побочного эффекта, но пока можно игнорировать разницу между абстрактным и конкретным побочными эффектами.

Только командам (процедурам) будет разрешено обладать побочным эффектом. Фактически мы не только допускаем, но и ожидаем изменения объектов командами, что и отличает императивный подход от аппликативного, полностью свободного от побочных эффектов.

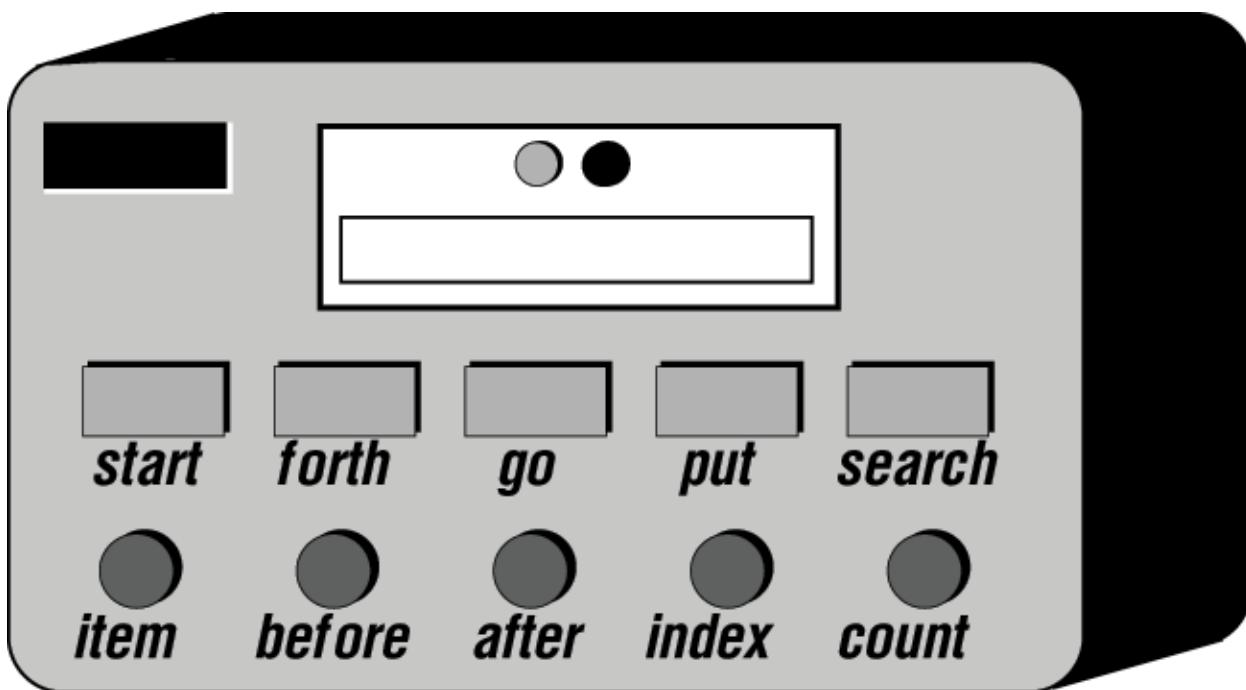


Рис. 5.1. Объект list как list-машина

Из этого обсуждения следует, что объекты можно рассматривать как машины с ненаблюдаемым внутренним состоянием и двумя видами кнопок - командными, изображенными на рисунке в виде прямоугольников, и кнопками запросов, отображаемыми кружками. Метафору "машины", как обычно, следует принимать с осторожностью.

При нажатии командной кнопки машина изменяет состояние, она начинает гудеть, щелкать и работает, пока не придет в новое стабильное состояние. Увидеть состояние (открыть машину) невозможно, но можно нажать кнопку с запросом. Состояние при этом не изменится, но в ответ появится сообщение, показанное в окне дисплея в верхней части нашей машины. Для запросов булевского типа предусмотрены две специальные кнопки над окном сообщения: одна из них загорается, когда запрос имеет значение `true`, другая - `false`. Многократно нажимая кнопки запросов, мы всегда будем получать одинаковые ответы, если в промежутке не нажимать командные кнопки (задание вопроса не меняет ответ).

Команды, как и запросы, могут иметь аргументы. В нашей машине для их ввода предназначен слот, показанный слева вверху.

Наш рисунок основан на примере объекта `list`, интерфейс которого описан в предыдущих лекциях и будет еще обсуждаться подробнее в данной лекции. Команды включают `start` (курсор передвигается к первому элементу), `forth` (продвижение курсора к следующей позиции), `search` (передвижение курсора к следующему вхождению элемента, введенного в верхний левый слот). Запросы включают `item` (показ на дисплее панели значения элемента в позиции курсора), `index` (показ текущей позиции курсора). Заметьте разницу между понятием "курсора", связанного с внутренним состоянием и, следовательно, напрямую не наблюдаемым, и понятиями `item` или `index`, более абстрактными, задающими официально экспортную информацию о состоянии.

Функции, создающие объекты

Следует ли рассматривать создание объекта как побочный эффект? Ответ - да, если целью создания является атрибут `a`, то инструкция `create a` изменяет значение поля объекта. Ответ - нет, если целью является локальная сущность подпрограммы. Но что если целью является результат самой функции - `create Result` или в общей форме `create Result.make (. . .)`? Такая инструкция не должна рассматриваться как побочный эффект, она не меняет объектов и не нарушает ссылочной прозрачности.

С позиций математика можно полагать, что все интересующие нас объекты в прошлом, настоящем и будущем уже описаны в Великой Книге Объектов и инструкция создания просто получает один из этих готовых объектов, ничего не меняя. Так что вполне законно и допустимо, чтобы функция создавала, инициализировала и возвращала такие объекты.

Эти же рассуждения применимы и для второй формы создания объектов - процедуры `make`, которая тоже не создает побочного эффекта, а возвращает уже созданный объект.

Чистый стиль для интерфейса класса

Из принципа Разделения Команд и Запросов следует стиль проектирования, вырабатывающий простой, понятный при чтении программный текст, способствующий надежности, расширяемости и повторному использованию.

Как вы могли заметить, этот стиль отличается от доминирующей сегодня практики, в частности от стиля программирования на языке С, предрасположенного к побочным эффектам. Игнорирование разницы между действием и значением - не просто свойство общего С-стиля (иногда кажется, что С-программисты не в силах противостоять искушению, получая значение, что-нибудь не изменить при этом). Все это глубоко встроено в язык, в такие его конструкции, как `x++`, означающую возвращение значения `x`, а затем его увеличение на 1; нимало не смущающую конструкцию `++x`, увеличивающую `x` до возвращения значения; Эти конструкции сокращают несколько нажатий клавиш: `y = x++` эквивалентно `y = x; x := x+1`. Целая цивилизация фактически построена на побочном эффекте.

Было бы глупо полагать бездумным стиль побочных эффектов. Его широкое распространение говорит о том, что многие находят его удобным, чем частично объясняется успех языка С и его потомков. Но то, что было привлекательным в прошлом веке, когда популяция программистов возрастала каждые несколько лет, когда важнее было сделать работу, не задумываясь о ее долговременном качестве, - не может подходить инженерии программ двадцать первого столетия. Мы хотим, чтобы ПО совершенствовалось вместе с нами, чтобы оно было понятным, управляемым, повторно используемым, и ему можно было бы доверять. Принцип Разделения Команд и Запросов является одним из требуемых условий достижения этих целей.

Строгое разделение команд и запросов при запрете побочных эффектов в функциях особенно важно при построении больших систем, где ключом успеха является сохранение полного контроля над каждым межмодульным взаимодействием.

Если вы пользовались противоположным стилем, то на первых порах ограничение может показаться довольно строгим. Но, получив практику, я думаю, вы быстро осознаете его преимущества.

В предыдущих лекциях этот принцип Разделения применялся повсюду. Вспомните, в наших примерах интерфейс для всех стеков включал процедуру `remove`, описывающую операцию выталкивания (удаление элемента из вершины стека), и функцию `item`, возвращающую элемент вершины. Первая является командой, вторая - запросом. При других подходах обычно вводят подпрограмму (функцию) `pop`, удаляющую элемент из стека и возвращающую его в качестве результата. Этот пример, надеюсь, ясно показывает выигрыш в ясности и простоте, получаемый при четком разделении двух аспектов.

Другие следствия принципа могут показаться более тревожными. При чтении ввода многие пользуются функциями, подобными `getint`, - имя взято из С, но ее эквиваленты имеются во многих языках. Эта функция читает очередной элемент из входного потока и возвращает его значение, очевидно, она обладает побочным эффектом:

- если дважды вызвать `getint ()`, то будут получены два разных ответа;
- вызовы `getint () + getint ()` и `2 * getint ()` дают разные результаты (если сверхусердный "оптимизирующий" компилятор посчитает первое выражение эквивалентным второму, то вы пошлете его автору разгневанный отчет об ошибке, и будете правы).

Другими словами, мы потеряли преимущества ссылочной прозрачности - рассмотрение программных функций как их математических аналогов с кристально ясным взглядом на то, как можно строить выражения из функций и что означают эти выражения.

Принцип Разделения возвращает ссылочную прозрачность. Это означает, что мы будем отделять процедуру, передвигающую курсор к следующему элементу, и запрос, возвращающий значение элемента, на который указывает курсор. Пусть `input` имеет тип `FILE`; инструкция чтения очередного целого из файла `input` будет выглядеть примерно так:

```
input.advance  
n := input.last_integer
```

Вызвав `last_integer` десять раз подряд, в отличие от `getint`, вы десять раз получите один и тот же результат. Вначале это может показаться непривычным, но, вкусив простоту и ясность такого подхода, вам уже не захочется возвращаться к побочному эффекту.

В этом примере, как и в случае `x++`, традиционная форма явно выигрывает у ОО-формы, если считать, что целью является уменьшение числа нажатий клавиш. Объектная технология вообще не обязательно является оптимальной на микроуровне (игра, в которой выигрывают языки типа APL или современные языки сценариев типа PERL). Выигрыш достигается на уровне глобальной структуры за счет повторного использования, за счет таких механизмов, как универсальность (параметризованные классы), за счет автоматической сборки мусора, благодаря утверждениям. Все это позволяет уменьшить общий размер текста системы намного больше, чем уменьшение числа символов в отдельной строчке. Мудрость локальной экономии зачастую оборачивается глобальной глупостью.

Генераторы псевдослучайных чисел: упражнение

В защиту функций с побочным эффектом приводят пример генератора псевдослучайных чисел, возвращающего при каждом вызове случайное число из последовательности, обладающей определенными статистическими свойствами. Последовательность инициализируется вызовом в форме:

```
random_seed (seed)
```

Здесь `seed` задается клиентом, что позволяет при необходимости получать одну и ту же последовательность чисел. Каждое очередное число последовательности возвращается при вызове функции:

```
xx := next_random ()
```

Но и здесь нет причин делать исключение и не ввести дихотомию команда/запрос. Забудем о том, что мы видели выше и начнем все с чистого листа. Как описать генерирование случайных чисел в ОО-контексте?

Как всегда, в объектной технологии зададимся вопросом - зачастую первым и единственным:

Что является абстракцией данных?

Соответствующей абстракцией здесь не является "генерирование случайного числа" или "генератор случайных чисел" - обе они функциональны по своей природе, фокусируясь на том, **что делает система**, а не на том, **кто это делает**.

Рассуждая дальше, рассмотрим в качестве кандидата понятие "случайное число", но и оно все же не является правильным ответом. Вспомним, что абстракция данных должна сопровождаться командами и запросами, довольно трудно придумать, что можно делать с одним случайным числом.

Понятие "случайное число" приводит к тупику. При изучении общих правил выявления классов уже говорилось, что ключевой шаг состоит в отсеве кандидатов. И опять-таки мы видим, что не все многообещающие существительные документа требований ведут к нужным классам. Можно не сомневаться, что данный термин обязательно встретится в любом документе, описывающем рассматриваемую проблему.

Случайное число не имеет смысла само по себе, оно должно рассматриваться в связи со своими предшественниками в генерируемой последовательности.

Стоп - появился термин последовательность, или, более точно, последовательность псевдослучайных чисел. Это и есть разыскиваемая абстракция! Она вполне законна и напоминает рассмотренный ранее список с курсором, только является бесконечной. Ее свойства включают:

- команды: `make` - инициализация некоторым начальным значением `seed`; `forth` - передвинуть курсор к следующему элементу последовательности;
- запросы: `item` - возвращает элемент в позиции курсора.

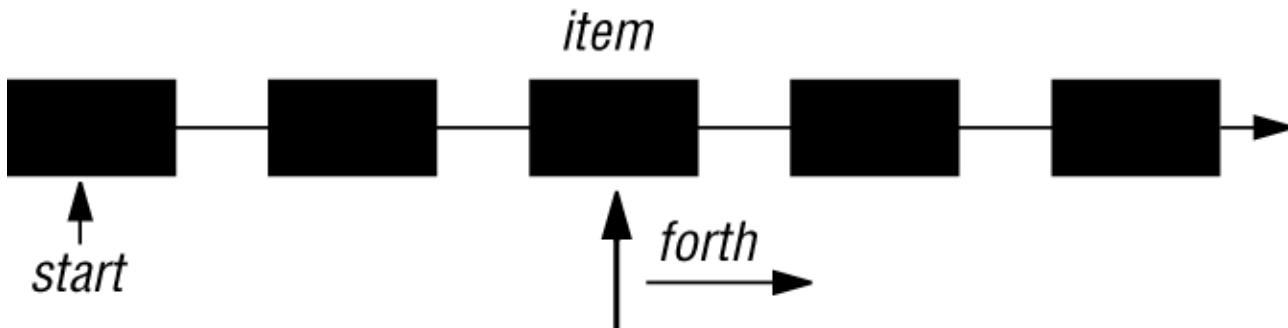


Рис. 5.2. Бесконечный список как машина

Для получения новой последовательности `rand` клиенты будут использовать `create rand.make (seed)`, для получения следующего значения - `rand.forth`, для получения текущего значения - `xx := rand.item`.

Как видите, нет ничего специфического в интерфейсе последовательности случайных чисел за исключением аргумента

seed в процедуре создания. Добавив процедуру start, устанавливающую курсор на первом элементе (которую процедура make может вызывать при создании последовательности), мы получаем каркас отложенного класса COUNTABLE_SEQUENCE, описывающего произвольную бесконечную последовательность. На его основе можно построить, например, последовательность простых чисел, определив класс PRIMES - наследника COUNTABLE_SEQUENCE, чьи последовательные элементы являются простыми числами. Другой пример - последовательность чисел Фибоначчи.

Эти примеры противоречат часто встречающемуся заблуждению, что на компьютерах нельзя представлять бесконечные структуры. АТД дает ключ к их построению - структура полностью определяется applicativeными операциями, число которых конечно (здесь их три - start, forth, item) плюс любые дополнительные компоненты, добавляемые при желании. Конечно, любое выполнение будет всегда создавать только конечное число элементов этой бесконечной структуры.

Класс COUNTABLE_SEQUENCE и его потомки, такие как PRIMES, являются частью универсальной иерархии ([М 1994]) информатики.

Абстрактное состояние, конкретное состояние

Из дискуссии о ссылочной прозрачности, казалось бы, следует желательность запрета конкретного побочного эффекта у функций. Такое правило имело то преимущество, что его можно было бы встроить непосредственно в язык, так как компилятор может легко обнаруживать наличие конкретного побочного эффекта у функций.

К сожалению, это неприемлемое ограничение. Принцип Разделения Команд и Запросов запрещает только **абстрактные** побочные эффекты, к объяснению которых мы и переходим. Дело в том, что некоторые конкретные побочные эффекты не только безвредны, но и полезны. Есть два таких вида.

Первая категория включает функции, модифицирующие состояние по ходу выполнения. Они изменяют видимые компоненты, но, заканчивая свою работу, все приводят в порядок, восстанавливая исходное состояние. Рассмотрим в качестве примера класс, описывающий целочисленный список с курсором и функцию, вычисляющую максимальный элемент списка:

```
max is
    -- Максимальное значение элементов списка
    require
        not empty
    local
        original_index: INTEGER
    do
        original_index := index
        from
            start; Result := item
        until is_last loop
            forth; Result := Result.max (item)
        end
        go (original_index)
    end
```

Для прохода по списку алгоритму необходимо перемещать курсор поочередно ко всем элементам, так что функция, вызывающая такие процедуры, как start, forth и go, полна побочными эффектами, но, начиная свою работу с курсором в позиции original_index, она и заканчивает свою работу в этой же позиции, благодаря вызову процедуры go. Но ни один компилятор в мире не может обнаруживать, что подобные побочные эффекты только кажущиеся, а не реальные.

Побочные эффекты второго приемлемого типа могут реально изменять состояние объектов, воздействуя на невидимые клиентам свойства. Для более глубокого понимания концепции полезно вернуться к обсуждению понятий абстрактной функции и инвариантов реализации, рассматриваемых в [лекции 11](#) курса "Основы объектно-ориентированного программирования", в частности стоит взглянуть на рисунки, соответствующие этим понятиям.

Мы видели, что программный (конкретный) объект является реализацией абстрактного объекта и что два конкретных объекта могут быть реализациями одного и того же абстрактного объекта. Например, два различных представления стека могут задавать один и тот же стек. Конкретные стеки могут использовать массивы с маркером вершины count и одинаковыми элементами ниже count. Но они могут быть массивами разной размерности и иметь разные элементы, расположенные за count. С точки зрения математика каждый конкретный объект принадлежит области определения абстрактной функции a, и мы можем иметь $c1 \neq c2$ хотя $a(c1) = a(c2)$.

Для нас это означает, что функция, модифицирующая конкретный объект, безвредна, если соответствующий абстрактный объект при этом не изменился. Предположим, например, что функция над стеками содержит операцию:

```
representation.put (some_value, count + 1)
```

(с гарантией, что емкость массива, по меньшей мере, равна count + 1). Тогда побочный эффект затронет область выше той, что отведена стеку, и в этом нет ничего плохого.

Конкретный побочный эффект, изменяющий конкретное состояние объекта с, является абстрактным побочным эффектом, если он также изменяет абстрактное состояние, другими словами, изменяет значение `a(s)` (другое определение, непосредственно используемое, появится чуть позже). Если побочный эффект не затрагивает абстрактного состояния - он безвреден.

Функции, производящие безвредный конкретный эффект, при рассмотрении объекта как машины соответствуют кнопкам запросов, изменяющих внутреннее состояние. Эти изменения, однако, не затрагивают ответы. Например, машина может выключать энергию в перерыве между обращениями. Подобные изменения легитимны.

ОО-подход хорош тем, что допускает "умные" реализации, допускающие изменения состояния "за сценой". Ниже мы увидим разумный и полезный пример применения этой техники.

Так как не для каждого класса определение основывается на полностью специфицированном АТД, то необходимо рабочее определение абстрактного побочного эффекта. Сделать это нетрудно. На практике АТД определяется интерфейсом, предлагаемым классом своим клиентам (отраженным, например, в краткой форме класса). Побочный эффект будет действовать на абстрактный объект, если он изменяет результат какого-либо из запросов, доступных клиентам. Вот определение:

Определение: абстрактный побочный эффект

Абстрактным побочным эффектом является такой конкретный эффект, который может изменить значение неsekретного запроса.

Это и есть то понятие, которое используется в Принципе Разделения - принципе, запрещающем абстрактные побочные эффекты в функциях.

Определение ссылается на "несекретные", а не на экспортируемые запросы. Причина в том, что между статусами "секретный" (закрытый) и "экспортируемый" допускается статус выборочно экспортируемых запросов. Как только запрос "несекретный" - экспортируемый какому-либо из клиентов за исключением `NONE`, - мы полагаем, что изменение его результата является абстрактным побочным эффектом.

Стратегия

Абстрактные побочные эффекты, в отличие от конкретных, не могут легко обнаруживаться компиляторами. В частности, недостаточно проверить, что сама функция сохраняет значения всех неsekретных атрибутов - эффект может быть непрямым и зависит от вызовов других процедур и функций. Другая причина в том, что, как в примере та x некоторые побочные эффекты могут в конечном итоге исчезать. Многие из компиляторов способны выдавать предупреждение, если функция модифицирует экспортируемый атрибут.

Поэтому Принцип Разделения Команд и Запросов является методологическим предписанием, а не языковым ограничением. Это не снижает его важности.

Каждый ОО-разработчик должен применять этот принцип без исключения. Я следую ему многие годы и не пишу функций с побочным эффектом. Наша фирма ISE применяет его во всех своих продуктах. Конечно, для тех, где используется язык C, этот принцип нельзя выдержать полностью, но и там мы применяем его всюду, где можно. Он помогает нам добиваться лучших результатов - в инструментарии и библиотеках, допускающих повторное использование, при расширениях и масштабировании.

Возражения

Важно рассмотреть два общих возражения, приводимых при отказе от стиля, свободного от побочных эффектов.

Первое связано с обработкой ошибок. Часто функция с побочным эффектом в действительности является процедурой, а ее результат задает статус ошибок, которые могли возникнуть при работе процедуры. Но есть лучшие способы справиться с этой проблемой. Соответствующая ОО-техника позволяет клиенту после выполнения операции сделать запрос о ее статусе, представленном соответствующим атрибутом, как в следующем примере:

```
target.some_operation (...)  
how_did_it_go := target.status
```

Заметьте, техника возвращения функцией статуса в качестве результата немного хромает. Она позволяет преобразовать процедуру в функцию, но хуже работает, когда подпрограмма сама является функцией, - что тогда делать с ее результатом? Возникают проблемы, когда статус задается не одним индикатором; в таких случаях нужно возвращать структуру, что близко к приведенной схеме, либо использовать глобальные переменные, что приводит к новым проблемам, особенно для больших систем, где многие модули могут включать состояние ошибки.

Второе возражение связано с общим недоразумением, например, считается, что интерфейс списка с курсором несовместим с параллельным доступом к объектам. Эта вера широко распространена (где бы я не читал лекции - в Санта-Барбаре, Сиэтле, Сингапуре, Санкт-Петербурге - всегда найдется человек, задающий подобный вопрос).

Недоразумение связано с тем, что в параллельном контексте имеется некоторая операция `get` доступа к буферу - параллельному аналогу очереди. Такая функция выполняется без прерываний и в нашей терминологии реализует как

вызов `item`, так и `remove`. Элемент возвращается в качестве результата функции, а удаление из буфера является побочным эффектом. Но использование подобных примеров в качестве аргументов в защиту функций `get`-стиля смешивает два понятия. Что нам действительно необходимо в параллельном контексте - это способ, дающий клиенту исключительный доступ к заготовленному элементу для выполнения некоторых операций. Имея такой механизм, можно защитить клиента, когда он выполняет последовательно операции:

```
x := buffer.item; buffer.remove
```

где гарантируется, что элемент, полученный при выполнении первой операции, будет действительно удален второй операцией. Такой механизм необходим вне зависимости от того, допускаем ли мы побочный эффект в функциях. Он, например, может понадобиться при выполнении операций удаления двух соседних элементов:

```
buffer.remove; buffer.remove
```

Гарантирование того, что удаляются два соседних элемента, никак не связано с побочными эффектами функций.

Позже в этой книге (см. [лекцию 12](#)) вопрос о параллельности будет подробно изучаться, там мы рассмотрим простой и элегантный подход к распределенным вычислениям, полностью совместимый с Принципом Разделения Команд и Запросов, который фактически поможет нам в достижении цели.

Законные побочные эффекты: пример

Закончим обсуждение побочных эффектов рассмотрением законного побочного эффекта - функции, не меняющей абстрактного состояния, но изменяющей конкретное состояние, и по вполне разумным причинам. Этот пример достаточно представителен и представляет некоторый шаблон проектирования.

Рассмотрим реализацию комплексных чисел. Как и в случае с точками, обсуждаемом в предыдущих лекциях, возможны два представления - декартово (с координатами x и y) и полярное (с расстоянием r и углом θ). Какое из них выбрать? Простого ответа нет. Если, как обычно, обратиться к АТД, то разные применимые операции - сложение, вычитание, умножение и деление - и запросы для получения значений x , y , r и θ эффективно выполняются для разных представлений (декартово представление лучше для сложения и умножения, полярное - для умножений и делений).

Можно было бы позволить клиенту решать, какое выбрать представление, но это делает классы трудными в использовании и нарушает принцип скрытия информации от клиента, которому нет дела до представления.

Альтернативой является одновременное хранение двух представлений. Но это приводит к издержкам производительности. Предположим, что клиенту требуются только операции умножения и деления. В этом случае операции используют только полярное представление, но мы бы каждый раз вычисляли бы x и y , выполняя бесполезные и дорогие вычисления.

Лучшее решение состоит в отказе от априорного выбора, выполняя выбор при необходимости. Мы практически ничего не проигрываем по памяти - все атрибуты нам все равно нужны, к ним добавятся только два булевых атрибута, указывающих на выбор текущего представления, но это позволит избежать лишних вычислений.

Пусть наш класс включает следующие операции:

```
class COMPLEX feature
    ... Объявления компонентов:
        infix "+", infix "-", infix "*", infix "/",
        add, subtract, multiply, divide,
        x, y, rho, theta, ...
end
```

Запросы x , y , ρ и θ представляют экспортруемые функции, возвращающие вещественные значения. Они всегда определены (исключая θ для комплексного числа 0). Помимо инфиксных функций "+" и других предполагаем процедуру `add` и другие. Вызов: $z1 + z2$ дает новое комплексное число, вызов $z1.add(z2)$ изменяет $z1$. На практике могут понадобиться только функции или только процедуры.

Наш класс включает следующие секретные (закрытые) атрибуты:

```
cartesian_ready: BOOLEAN
polar_ready: BOOLEAN
private_x, private_y, private_rho, private_theta: REAL
```

Не все четыре вещественных атрибута необходимы постоянно, фактически только два являются текущими. Более точно, следующий инвариант реализации должен быть включен в класс:

```
invariant
    cartesian_ready or polar_ready
    polar_ready implies (0 <= private_theta and private_theta <= Two_pi)
        -- cartesian_ready implies (private_x and private_y являются текущими)
        -- polar_ready implies (private_rho and private_theta являются текущими)
```

Последние два предложения выражены неформально в форме комментария.

В каждый момент по крайней мере одно из представлений является текущим. Любая из операций, запрашиваемая клиентом, должна использовать наиболее подходящее для нее представление, что может потребовать вычислений, если представление не является текущим. В качестве побочного эффекта операции другое представление перестает быть текущим.

Две закрытые процедуры доступны для проведения изменений представления:

```
prepare_cartesian is
    -- Сделать доступным декартово представление
    do
        if not cartesian_ready then
            check polar_ready end
            -- Поскольку инвариант требует, чтобы одно
            -- из двух представлений было текущим
            private_x := private_rho * cos (private_theta)
            private_y := private_rho * sin (private_theta)
            cartesian_ready := True
            -- Здесь cartesian_ready и polar_ready равны true:
            -- Оба представления являются текущими
        end
    ensure
        cartesian_ready
    end
prepare_polar is
    -- Сделать доступным полярное представление
    do
        if not polar_ready then
            check cartesian_ready end
            private_rho := sqrt (private_x ^ 2 + private_y ^ 2)
            private_theta := atan2 (private_y, private_x)
            polar_ready := True
            -- Здесь cartesian_ready и polar_ready равны true:
            -- Оба представления являются текущими
        end
    ensure
        polar_ready
    end
```

Функции `cos`, `sin`, `sqrt` и `atan2` берутся из стандартной математической библиотеки, `atan2(y, x)` вычисляет `arctangent(y/x)`.

Нам также нужны процедуры создания - `make_cartesian` и `make_polar`:

```
make_cartesian (a, b: REAL) is
    -- Инициализация: abscissa a, ordinate b
    do
        private_x := a; private_y := b
        cartesian_ready := True; polar_ready := False
    ensure
        cartesian_ready; not polar_ready
    end
```

и симметрично для `make_polar`.

Экспортируемые операции пишутся просто, начнем, например, с процедуры, имеющей варианты в зависимости от операции:

```
add (other: COMPLEX) is
    -- Добавляет значение other
    do
        prepare_cartesian; polar_ready := False
        private_x := x + other.x; private_y = y + other.y
    ensure
        x = old x + other.x; y = old y + other.y
        cartesian_ready; not polar_ready
    end
```

Заметьте, в постусловии важно использовать `x` и `y`, а не `private_x` и `private_y`, которые могут не быть текущими перед вызовом.

```
divide (z: COMPLEX) is
    -- Divide by z.
```

```

require
  z.rho /= 0
  -- Численное выражение дает более реалистичное предусловие
do
  prepare_polar; cartesian_ready := False
  private_rho := rho / other.rho
  private_theta = (theta - other.theta) \\\ Two_pi
  -- \\\ - остаток от деления
ensure
  rho = old rho / other.rho
  theta = (old theta - other.theta) \\\ Two_pi
  polar_ready; not cartesian_ready
end

```

Аналогично для вычитания и умножения - subtract и multiply. (Предусловие и постусловие могут быть слегка адаптированы для учета особенностей операций с плавающей точкой.) Варианты функций следуют тому же образцу:

```

infix "+" (other: COMPLEX): COMPLEX is
  -- Сумма текущего числа и other
do
  create Result.make_cartesian (x + other.x, y + other.y)
ensure
  Result.x = x + other.x; Result.y = y + other.y
  Result.cartesian_ready
end
infix "/" (z: COMPLEX): COMPLEX is
  -- Частное от деления текущего комплексного числа на z
require
  z.rho /= 0
do
  create Result.make_polar (rho / other.rho, (theta - other.theta) \\\ Two_pi)
ensure
  Result.rho = rho / other.rho
  Result.theta = (old theta - other.theta) \\\ Two_pi
  Result.polar_ready
end

```

Аналогично для infix "-" и infix "**".

Обратите внимание на последние предложения в постусловиях этих функций - cartesian_ready и polar_ready должны экспортироваться самому классу, появляясь в предложениях в форме feature {COMPLEX}; они не экспортируются никакому другому классу.

Но где здесь побочные эффекты? В последних двух функциях они непосредственно не видны. Все дело в x, y, rho и theta - они являются хитроумными создателями побочных эффектов. Вычисление x или y приведет к изменению представления (вызовется prepare_cartesian), если не подготовлено декартово представление. Все симметрично для rho и theta. Вот примеры для x и theta:

```

x: REAL is
  -- Abscissa
do
  prepare_cartesian; Result := private_x
end
theta: REAL is
  -- Angle
do
  prepare_polar; Result := private_theta
end

```

Функции y и rho подобны. Все эти функции вызывают процедуру, которая может включить изменение состояния. В отличие от add и его собратьев, однако, они не делают предыдущее представление неверным, когда вычисляется новое представление. Например, если x вызывается в состоянии с ложным значением cartesian_ready, оба представления (все четыре вещественных атрибута) станут текущими. Все это потому, что функциям разрешается производить побочные эффекты только на конкретных объектах, но не на ассоциированных абстрактных объектах. Выразим это свойство более формально: вычисление z.x или другой функции может изменять конкретный объект, связанный с z, скажем от c₁ до c₂, но всегда с гарантией того, что

$$a(c_1) = a(c_2)$$

где a - абстрактная функция. Объекты c₁ и c₂ могут быть различными, но они представляют один и тот же математический объект - комплексное число.

Такие побочные эффекты безвредны. Они действуют только на секретные атрибуты и, следовательно, не могут быть обнаружены клиентами.

ОО-подход поощряет такие гибкие, адаптирующиеся схемы, выбирающие наилучшее представление, соответствующее потребностям текущего момента. Пока реализация действует на конкретное состояние, не затрагивая абстрактного, ее функции не нарушают Принципа Разделения и не создают угрозу ссылочной прозрачности.

Много ли аргументов должно быть у компонента?

Пытаясь сделать классы, особенно повторно используемые, простыми в использовании, следует особое внимание обратить на число аргументов у компонентов. Как мы увидим, объектная технология вырабатывает стиль интерфейса компонентов, радикально отличающийся от традиционных подходов, в частности характеризуемый небольшим числом аргументов.

Важность числа аргументов

Когда разработка базируется на классе поставщика, день изо дня приходится обращаться к его компонентам. Простота их интерфейса определяет простоту использования класса. Влияют и другие факторы, в частности, непротиворечивость соглашений, но в конечном счете над всем доминирует простой численный критерий: как много аргументов имеют компоненты. Чем больше аргументов, тем труднее их запомнить.

Это же верно и для библиотечных классов. Критерий успеха прост, после того как потенциальный пользователь библиотеки поймет, что представляет собой класс, он будет его использовать, если изучение компонентов не потребует большого времени и постоянного обращения к документации. Помимо других факторов важную положительную роль играет короткий список аргументов.

Анализируя типичную библиотеку подпрограмм, часто можно встретить программы с большим числом аргументов. Вот пример программы интегрирования с прекрасным алгоритмом, но с традиционным интерфейсом (предупреждаю, это не ОО-интерфейс!).

```
nonlinear_ode
  (equation_count: in INTEGER;
   epsilon: in out DOUBLE;
   func: procedure (eq_count: INTEGER; a: DOUBLE; eps: DOUBLE;
   b: ARRAY [DOUBLE]; cm: pointer Libtype)
   left_count, coupled_count: in INTEGER;
   ...)
[И так далее. Всего 19 аргументов, включающих:
 - 4 in out значения;
 - 3 массива, используемы как входные и выходные;
 - 6 функций, каждая имеющая 6 - 7 аргументов, из которых
 2 или 3 являются массивами!]
```

Так как нашей целью является не критика конкретной библиотеки, а выяснение разницы между ОО и традиционными интерфейсами, то имена программы и аргументов изменены, а синтаксис адаптирован.

Некоторые свойства делают эту процедуру особенно сложной в использовании:

- Большинство аргументов имеют статус **in out**, означающий необходимость их инициализации перед вызовом и обновление их значений в процессе работы программы. Например, аргумент `epsilon` указывает на входе, требуется ли продолжение функций (да, если меньше 0, если между 0 и 1, то продолжение требуется, если `epsilon < vprecision` и т. д.). На выходе аргумент представляет оценку приращения.
- Многие из аргументов как самой процедуры, так и функций, являющихся ее аргументами, заданы массивами, служащими для передачи информации в процедуру и обратно.
- Некоторые аргументы служат для спецификации большого числа возможностей по обработке ошибок (прервать обработку, записывать сообщения в файл, продолжать в любых ситуациях...)

Хотя высококачественные библиотеки численных методов вычислений существуют и применяются многие годы, все же они не столь широко распространены в научном мире, как это следовало. Сложность их интерфейсов, в частности большое число аргументов, иллюстрируемое `nonlinear_ode`, во многом является этому причиной.

Часть этой сложности, несомненно, связана со сложностью самой проблемы. Но все можно сделать лучше. ОО-библиотека для численных вычислений - Math ([Dubois 1997]) - предлагает совсем другой подход, согласованный с концепциями объектной технологии и принципами этой книги. Как ранее упоминалось, эта библиотека служит примером использования объектной технологии для упаковки старого программного обеспечения - ее ядром является не ОО-библиотека. Было бы абсурдно не использовать хорошо зарекомендовавшие себя алгоритмы, и прекрасно, когда им придается современный интерфейс, привлекательный для клиентов. Базисная подпрограмма `nonlinear_ode` имеет в ней форму:

```
solve
  -- Решить проблему, записав ответ в x и у
```

У нее теперь вообще нет аргументов! Просто создается экземпляр класса `GENERAL_BOUNDARY_VALUE_PROBLEM`,

представляющий требуемую задачу, устанавливаются его свойства, отличные от значений, принятых по умолчанию. При этом могут вызываться подходящие процедуры, присоединенные к объекту, решающему проблему. Затем вызывается метод `solve` для этого объекта. Атрибуты класса `x` и у дают возможность анализа ответа.

Таким образом, применение ОО-техники дает существенный эффект по сокращению числа аргументов. Измерения, сделанные для библиотек ISE, показывают, что среднее число аргументов находится в пределах от 0,4 для базовых библиотек Base до 0,7 для графической библиотеки Vision. Для корректного сравнения с не ОО-библиотеками следует добавлять единицу, поскольку в объектном случае мы учитываем два аргумента в вызове `x.f(a, b)` против трех в необъектной программе `f(x, a, b)`. Но все равно сравнение явно в пользу объектной технологии, так как число аргументов, как мы видели, в необъектном случае достигает 19 аргументов и часто имеет значения 5, 10 или 15.

Эти цифры сами по себе не являются целью и, конечно, не являются индикатором качества. Но они в значительной степени являются результатом глубокого принципа проектирования, к рассмотрению которого мы переходим.

Операнды и необязательные параметры (опции)

Аргументы подпрограммы могут быть одного из двух возможных видов: **операнды и опции**.

Для понимания разницы рассмотрим пример класса DOCUMENT и его процедуру печати `print`. Предположим - просто для конкретизации изложения, - что печать основана на Postscript. Типичный вызов иллюстрирует возможный интерфейс, не совместимый с ниже излагаемыми принципами. Вот пример:

```
my_document.print (printer_name, paper_size, color_or_not,  
                    postscript_level, print_resolution)
```

Какие из пяти аргументов являются обязательными? Если не задать, например, Postscript уровень, то по умолчанию используется наиболее доступное значение, это же касается и остальных аргументов, включая и имя принтера.

Этот пример иллюстрирует разницу между операндами и опциями:

Определение: операнд и опция

Аргумент является операндом, если он представляет объект, с которым оперирует программа.

Аргумент является опцией, если он задает режим выполнения операции.

Это определение носит общий характер и оставляет место для неопределенности. Существуют два прямых критерия:

Как отличать опции от операндов

- Аргумент является опцией, если предполагается, что клиент может не поддерживать его значение, для него может быть установлено разумное значение по умолчанию.
- При эволюции класса аргументы имеют тенденцию оставаться неизменными, а опции могут добавляться или удаляться.

В соответствии с первым критерием аргументы `print` являются опциями. Заметьте, однако, что цель вызова - неявный аргумент `my_document`, как и все цели, должна быть операндом. Если не сказать, какой документ следует печатать, никто не сделает за вас этот выбор.

Второй критерий менее очевиден, так как требует некоторого предвидения, но он отражает то, что является предметом наших забот, начиная с первой лекции. Класс не является неизменным продуктом - он может изменяться в процессе жизненного цикла. Некоторые его свойства меняются чаще, чем другие. Операнды - это долго живущая информация, добавление или удаление операнда является изменением, затрагивающим сущность класса. Опции, с другой стороны, могут появляться и исчезать. Например, нетрудно понять, что поддержка цвета при печати могла появиться не сразу. Это типично для опций.

Принцип

Определение операндов и опций **дает** правило для аргументов:

Принцип операндов

Аргументы подпрограмм должны быть только операндами, но не опциями.

Два случая ослабления правила, не рассматриваемые как исключения, упоминаются ниже.

В стиле, продвигаемом этим принципом, опции к операциям устанавливаются не при вызове операции, а при вызове специальных процедур, задачей которых является установка опций:

```
my_document.set_printing_size ("A4")
```

```
my_document.set_color  
my_document.print
```

-- Совсем нет аргументов

Будучи однажды установленной, опция действует, пока целевой объект не изменит установку при новом вызове. В отсутствие любого вызова соответствующей процедуры или явной установки в момент создания объекта действует значение опции, устанавливаемой по умолчанию.

Для любого типа, отличного от Boolean, процедуры, устанавливающие опцию, имеют ровно один аргумент соответствующего типа, как это проиллюстрировано при вызове `set_printing_size`. Стандартное имя для таких процедур имеет форму `set_property_name`. Заметьте, аргументы таких процедур сами удовлетворяют Принципу Операнда. Так, например, аргумент, задающий размер страницы, является опцией для процедуры `print`, но операндом для установочной процедуры `set_printing_size`.

Для булевых процедур та же техника приводила бы к аргументу, принимающему всего два значения - `True` or `False`. Оказывается, что пользователи часто забывают, какая из двух возможностей соответствует `True`, поэтому лучше использовать пару процедур с удобными именами в форме `set_property_name` и `set_no_property_name`, например, `set_color` и `set_no_color`, во втором случае можно предложить и другой вариант `set_black_and_white`.

Применение Принципа Операндов дает несколько преимуществ:

- Необходимо указывать только то, что отличается от установок по умолчанию.
- Новички не обязаны изучать все, они могут игнорировать специальные свойства, оставляя их профессионалам.
- При более глубоком изучении класса осваиваются новые свойства, но помнить нужно только то, что используется.
- Вероятно, наиболее важно то, что эта техника сохраняет расширяемость и отвечает Принципу Открыт-Закрыт. При добавлении новых опций нет необходимости изменять интерфейс подпрограммы и, следовательно, нарушать работу существующих клиентов. Если значение по умолчанию соответствует прежним неявным установкам, существующие клиенты не должны вносить никаких изменений.

Рассмотрим возможные возражения Принципу Операндов. Мы не избавляемся от сложности, а только переносим ее глубже: вместо вызова аргументов приходится вызывать специальные процедуры. Это не совсем точно. Вызовы нужны только для тех опций, для которых мы явно хотим установить значения, отличные от значений по умолчанию.

Заметьте также, часто одно и то же значение опции успешно работает для многих вызовов. Использование аргументов заставляло бы задавать значение при каждом вызове, наша техника позволяет установить его один раз.

Некоторые языки поддерживают необязательные аргументы, дающие преимущества Принципа Операндов, но лишь частично. Сравнение двух подходов является предметом упражнения, но уже сейчас заметим, что, если значение опции многократно используется и отличается от значения по умолчанию, то придется задавать его при каждом вызове.

Преимущества, обеспечиваемые Принципом Операндов

Комментарии, сделанные в свое время при рассмотрении Принципа Разделения Команд и Запросов, относятся и к Принципу Операндов. Они противоречат доминирующей сегодня практике и некоторые читатели, несомненно, будут на первых порах отвергать их. Но я могу рекомендовать их без всякого зазрения совести, поскольку применяю их многие годы и получаю в результате большие выгоды. Они приводят к простому, элегантному стилю, способствуя ясности и расширяемости.

Этот стиль вскоре становится естественным для разработчиков. (Мы сделали его частью стандарта в ISE.) Вы создаете требуемые объекты, устанавливаете любые значения, отличающиеся от принятых по умолчанию, затем применяете нужные операции. Эта схема была показана на примере `solve` в библиотеке `Math`. Она, конечно, предпочтительнее передачи 19 аргументов.

Исключения из Принципа Операндов?

Принцип Операндов универсально применим. Но два специальных случая, не являясь настоящими исключениями, требуют некоторой адаптации.

Во-первых, можно получить преимущества от существования множества процедур создания. Класс поддерживает разные способы инициализации объектов, вызывая `create` `x.make_specific(argument, ...)`, где `make_specific` - соответствующая процедура создания. Можно ослабить Принцип Операндов для таких процедур, облегчая задачу клиенту, предлагая различные способы установки значений, отличные от значений по умолчанию. Однако имеют место два ограничения:

- помните, что, как всегда, процедура создания должна обеспечить выполнение инварианта класса;
- множество процедур создания должно включать минимальную процедуру (называемую `make` в рекомендованном стиле), не включающую опций в качестве аргументов и устанавливающую значения опций по умолчанию.

Другой случай ослабления Принципа Операндов следует из последнего наблюдения. Можно заметить, что некоторые операции часто используют установки опций, соответствующие некоторому стандартному образцу, например:

```
my_document.set_printing_size ("...")  
my_document.set_printer_name ("...")  
my_document.print
```

В таком случае может быть удобнее во имя инкапсуляции и повторного использования, а также в согласии с Принципом Списка Требований, изучаемом далее, обеспечить для удобства клиентов специальную процедуру:

```
print_with_size_and_printer (printer_name: STRING; size: SIZE_SPECIFICATION)
```

Это предполагает, конечно, что основная минимальная программа (`print` в нашем примере) остается доступной и что новая программа является дополнением, упрощая задачу клиента в тех случаях, когда она действительно часто встречается.

В действительности речь не идет о нарушении принципа, так как аргументы действительно требуются по природе решаемой задачи, так что здесь они являются не опциями, а операндами.

Контрольный перечень

Принцип Операндов, заставляющий уделять опциям должное внимание, подсказывает технику, помогающую получить правильный класс. Для каждого класса перечислите все поддерживаемые опции и создайте таблицу, содержащую одну строку для каждой опции. Эта техника иллюстрируется на классе DOCUMENT следующей таблицей, представленной одной строкой:

Таблица 5.1. Описание опций класса

Option	Initialized	Queried	Set
Paper size	default:A4 (international) make_LTR: LTR (US)	size	set_size set_LTR set_A4

Столбцы таблицы последовательно перечисляют: назначение опции, как она инициализируется различными процедурами создания, как она доступна клиентам, как она может устанавливать различные значения. Тем самым задается полезный контрольный перечень для часто встречающихся дефектов:

- **Initialized** помогает обнаружить ошибочную инициализацию, особенно в случае умолчаний. (Если, например, по умолчанию хотим установить цветную печать, то значение опции `Black_and_white_only` должно быть установлено как `false`.)
- **Queried** помогает обнаружить ошибки доступа. Заметьте, программа, получающая объект, может изменять опции в своих собственных целях, но затем восстанавливать их начальное состояние. Это возможно, если разрешен запрос начального состояния.
- **Set** помогает обнаружить пропущенные опции установочных процедур.

Ни одно из правил, предлагаемых здесь, не является абсолютным. Но они применимы в большинстве случаев, так что важно проверить, что входы таблицы отвечают ожидаемому поведению класса. Таблица может также помочь в документировании класса.

Размер класса: Подход списка требований

Мы твердили, как параноики, что нужно ограничивать внешний размер компонента, измеряемый числом его аргументов, поскольку он во многом определяет простоту использования и, следовательно, качество интерфейса. В меньшей степени мы заботились о внутреннем размере, измеряющем, например, числом инструкций, поскольку он зависит от сложности алгоритма. Как показывает практика, при хорошем ОО-проектировании тела подпрограмм, как правило, имеют также небольшие размеры.

Следует ли подобным образом сосредоточиться на рассмотрении размеров класса как целого? Уверенного ответа здесь нет.

Определение размера класса

Следует определить, как измерять размер класса. Можно посчитать общее число строк или, что более разумно, число определений и инструкций, в меньшей степени зависящих от текстуальных предпочтений автора. В последнем случае простой анализатор языка справится с этой задачей. Хотя это и интересно для некоторых приложений, эти измерения отражают позицию поставщика класса. Если же нас интересует, как много функциональности предоставляет класс своим клиентам, то подходящим критерием является, скорее, число его компонентов.

Все же остаются два вопроса:

- Скрытие информации: следует ли учитывать все компоненты (**внутренний** размер) или только экспортруемые (**внешний** размер)?
- Наследование: следует ли учитывать только непосредственные компоненты, введенные в самом классе, -

непосредственный (immediate) размер - или считать все компоненты, включая наследованные от всех предков, - **плоский (flat)** размер, связанный с понятием плоской формы класса. Возможно, следует считать только непосредственные компоненты, присоединяя к ним компоненты, модифицируемые в классе через переопределение и эффективизацию, не учитывая переименования, которое не влияет на **возрастающий (incremental)** размер?

Различные комбинации могут представлять интерес. Для нашего обсуждения наиболее важны два измерения - **внешнее и возрастающее**. Внешний размер означает, что мы смотрим на класс с позиций клиента, и нас мало волнует, что там делается внутри в собственных интересах класса. Возрастающий размер позволяет сосредоточиться на ценностях, добавляемых классом. При этом игнорируется важная наследуемая часть функциональности, но иначе одни и те же компоненты учитывались бы многократно, как у класса, так и у всех его потомков.

Поддержка согласованности

Некоторые авторы, в том числе Поль Джонсон ([Johnson 1995]) настаивают на ограничении размеров класса:

Проектировщики класса часто пытаются включить в него много компонентов. В результате, в интерфейсе появляется несколько обще используемых компонентов и довольно много странных подпрограмм. Совсем плохо, когда список компонентов велик.

Из опыта ISE следует другая точка зрения. Мы полагаем, что сам по себе размер класса не создает проблем. Хотя большинство классов относительно невелико (от нескольких компонентов до дюжины), встречаются и большие классы (от 60 до 80 компонентов, а иногда и больше), и с ними не возникает никаких особых проблем, если они хорошо спроектированы.

Этот опыт привел нас к подходу, называемому **список требований (shopping list)**. Реализация не ухудшается при добавлении в класс компонентов, связанных с ним концептуально. Если вы колебитесь, включать ли в класс экспортруемый компонент, то вас не должен беспокоить размер класса. Единственный критерий, подлежащий учету, - это согласованность компонента с остальными членами класса. Этот критерий отражен в следующей рекомендации:

Совет: Список Требований

При рассмотрении добавления нового экспортруемого компонента следите за следующими правилами:

- **S1** Компонент должен соответствовать абстракции данных, задающей класс.
- **S2** Он должен быть совместимым с другими компонентами класса.
- **S3** Он не должен дублировать цель другого компонента класса.
- **S4** Он должен поддерживать инвариант класса.

Первые два требования связаны с Принципом Согласованности (см. [лекцию 4](#)), устанавливающим, что все компоненты класса должны относиться к одной хорошо идентифицируемой абстракции. В качестве контрпримера рассматривался строковый класс `string` из библиотеки NEXTSTEP, покрывающий фактически две абстракции, разделенный, в конечном итоге, на два класса. Проблемой здесь был не размер исходного класса, а качество проектирования.

Интересно отметить, что тот же пример - класс `string` - один из самых больших классов в библиотеках ISE, был подвергнут критике из-за своих размеров Полем Джонсоном. Но реакция пользователей библиотеки в течение многих лет была противоположной, - они просили добавления свойств. Класс, хотя и обладал богатой функциональностью, но был прост в использовании, поскольку все компоненты четко применяли одну и ту же абстракцию - строку символов, над которой по ее природе определено много операций, начиная от извлечения подстроки и замены до конкатенации и глобальной подстановки.

Класс `STRING` показывает, что большой не означает сложный. Некоторые абстракции по своей природе обладают многими компонентами. Процитирую Валдена и Нерсона ([Walden 1995]):

Класс, обрабатывающий документ, содержит 100 различных операций, позволяющих выполнять различные вариации со шрифтом, <...> фактически имеет дело с небольшим числом лежащих в его основе концепций, довольно простых и легко усваиваемых. Простота выбора надлежащей операции поддерживается прекрасно уорганизованным руководством.

В этом случае разделение класса, вероятнее всего, ухудшило, а не улучшило бы простоту использования класса.

Чрезвычайно минималистская точка зрения состоит в том, что класс должен содержать только атомарные компоненты, - не выражаемые в терминах уже введенных операций. Это бы привело бы к исключению некоторых фундаментальных схем, успешных ОО-конструкций, в частности **классов поведения (behavior classes)**, в которых эффективный компонент описывается через другие низкоуровневые компоненты класса, большинство которых являются отложенными.

Минимализм приведет к запрету двух теоретически избыточных, но практически важных дополнительных компонентов. Обратимся к классу `COMPLEX`, описывающему комплексные числа, разработанному в этой лекции. Для арифметических операций некоторым клиентам могут понадобиться функциональные версии:

```
infix "+", infix "-", infix "*", infix "/"
```

Вычисление выражения `z1 + z2` создаст новый объект, представляющий сумму `z1` и `z2`, аналогично для других функций. Другие клиенты, или тот же клиент в другой ситуации предпочтет процедурную версию операции, где вызов `z1.add(z2)` обновляет объект `z1`. Теоретически избыточно включать и функции и процедуры в класс, поскольку они взаимно заменимы. На практике удобно иметь обе версии по меньшей мере по трем причинам: удобства, эффективности, повторного использования.

Запреты и послабления

В последнем примере два множества компонентов, будучи теоретически избыточными, практически являются различными. Конечно же, не следует вводить новый компонент, если есть старый, выполняющий аналогичную работу, о чем говорит предложение **S3** в рекомендациях Списка Требований. Это предложение более требовательное, чем может показаться с первого взгляда. В частности:

- Предположим, вы хотите изменить порядок аргументов в подпрограмме для совместимости с другими подпрограммами. Но вас сдерживает совместимость с уже существующим ПО. Решение не может состоять в том, чтобы иметь оба компонента с тем же статусом, - это противоречило бы рекомендации **S3**. Вместо этого следует использовать библиотечный механизм эволюции **obsolete**, который чуть позже будет описан в этой лекции.
- Аналогично следует поступать для обеспечения значений аргументов, требуемых некоторой программе. Не следует предоставлять две версии, одну со специальными аргументами, а другую - общую, основанную на умолчаниях, как обсуждалось ранее в этой лекции. Сделайте один интерфейс официальным, а другой обеспечьте через механизм **obsolete**.
- Если вы колебаетесь в выборе имени компонента, следует почти всегда сопротивляться попытке сделать имена синонимами. В библиотеке ISE единственное исключение сделано для фундаментальных компонентов, имеющих инфиксное имя и идентификатор, например доступ к массиву может осуществляться двояко: `my_array.item(some_index)` или `my_array @ some_index`. Каждая форма предпочтительнее в определенном контексте. Но это редкая ситуация. Как правило, проектировщик должен выбрать имя, не перекладывая ответственность на клиентов.

Как вы заметили, наша политика в результате представляет смесь из запретов и послаблений. Она смягчена, поскольку допускает компоненты, не являющиеся основными. Но она достаточно строга, поскольку определяет жесткие условия для компонента. Компоненты класса могут покрывать столько потребностей, сколько это необходимо, но они должны покрывать только релевантные потребности, и каждой из них должен соответствовать ровно один компонент.

Политика Списка Требований возможна только потому, что мы следуем систематической политике сохранения минимальности языка. Минималистская позиция в проектировании языка - небольшое число чрезвычайно мощных конструкций и никакой избыточности - позволяет разработчикам класса не быть минималистами. Каждый разработчик должен знать язык, поскольку язык минимален, то разработчик знает о нем все. Классы используются только клиентами и они могут пропустить то, что они не используют.

Следует связать Рекомендации Списка Требований с предшествующей дискуссией о размере компонентов. Трудности использования класса определяются не числом его компонентов, а их индивидуальной сложностью. Более точно, размер класса является некоторой проблемой лишь на первых порах. После завершения этапа освоения разработчик будет постоянно иметь дело с компонентами, скорее всего, подмножеством компонентов. Размер компонента становится приоритетным, а размер класса перестает быть таковым. Не следует пользоваться численными критериями типа: "никакой класс не должен иметь более n строк или m компонентов", - разделение класса на такой основе может лишь сделать его **более** трудным в использовании.

Урок для разработчиков класса, вытекающий из Рекомендаций Списка Требований: следует заботиться о качестве класса, в частности о его концептуальной целостности и размере его компонентов, но не о размере самого класса.

Активные структуры данных

Примеры этой и предыдущих лекций часто используют понятие списка или последовательности, характеризуемой в каждый момент "позицией курсора", указывающей точку доступа, вставки и удаления. Такой вид структуры данных широко применим и заслуживает детального рассмотрения.

Для понимания достоинств такого подхода полезно начать с общего рассмотрения и оценки его ограничений.

Представление связного списка

Обсуждение будет основываться на примере списков. Хотя результаты не зависят от выбора реализации, необходимо иметь некоторое представление, позволяющее описывать алгоритмы и иллюстрировать проблемы. Будем использовать популярный выбор - односвязный линейный список. Наша общечелевая библиотека должна иметь классы со списковыми структурами и среди них класс `LINKED_LIST`.

Вот некоторые сведения о связных списках, применимые ко всем стилям интерфейса, обсуждаемым далее, - с курсором и без курсора.

Связный список является полезным представлением последовательной структуры с эффективно реализуемыми

операциями вставки и удаления элементов. Элементы хранятся в отдельных ячейках, называемых звеньями (**linkables**). Каждое звено содержит значение и ссылку на следующий элемент списка:

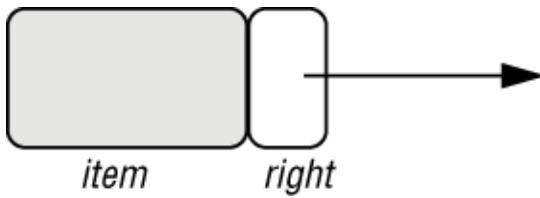


Рис. 5.3. Элемент списка - звено (linkable)

Соответствующий класс должен быть универсальным (синонимы: родовым, параметризованным), так как структура должна быть применима к спискам с элементами любого типа. Значение звена, заданное компонентом *item*, имеет тип *G* - родовой параметр. Оно может быть непосредственно встроено, если фактический родовой параметр развернутого типа, например, для списка целых, или быть ссылкой в общем случае. Другой атрибут *right* типа *LINKABLE[G]* всегда представляет ссылку.

Сам список задается отдельной ячейкой - заголовком, содержащим ссылку *first_element* на первое звено, и, возможно, дополнительной информацией, например *count* - текущим числом элементов списка. Вот как выглядит связный список символов:

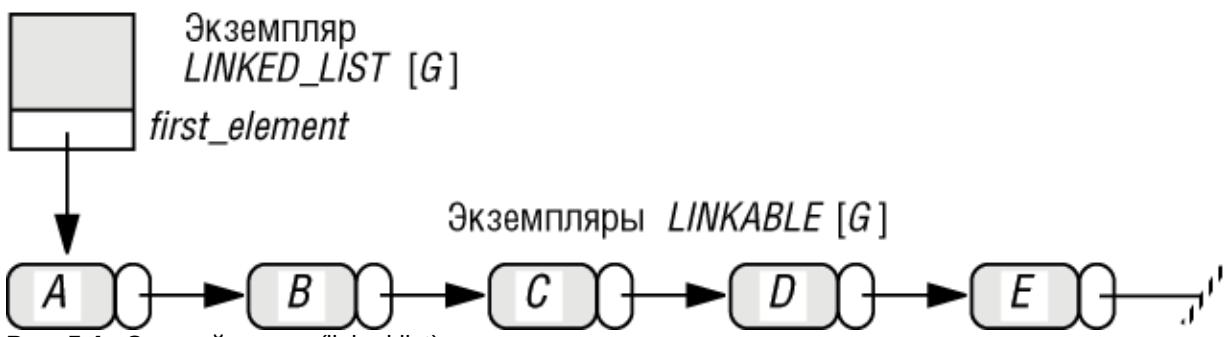


Рис. 5.4. Связный список (linked list)

Это представление позволяет быстро выполнять операции вставки и удаления, если есть ссылка, указывающая на звено слева от цели операции. Достаточно выполнить несколько манипуляций над ссылками, как показано на следующем рисунке:

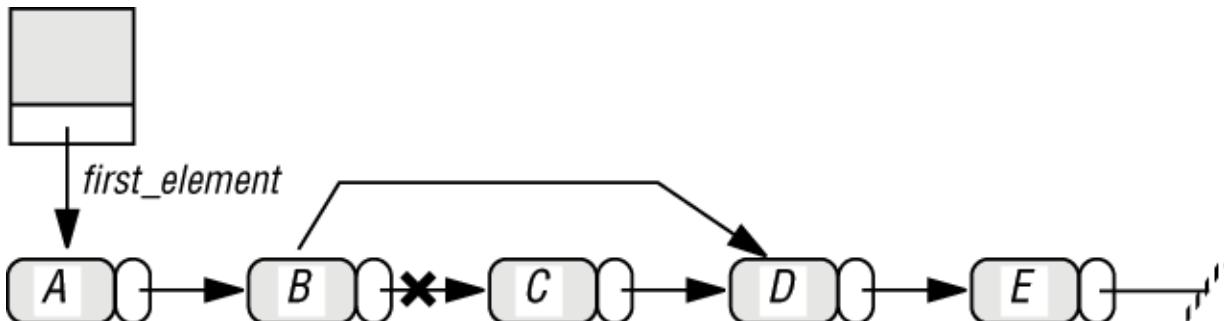


Рис. 5.5. Удаление в связном списке

Но, с другой стороны, списковое представление не очень подходит для таких операций как поиск элемента по его значению или позиции, поскольку они требуют прохода по списку. Представление массивом, по контрасту, хорошо подходит для получения нужного элемента по индексу (позиции), но не подходит для операций вставки и удаления. Существует много других представлений, некоторые из которых объединяют преимущества обоих миров. Связный список остается одной из наиболее употребительных реализаций, предлагая эффективную технику для приложений, где большинство операций связано со вставкой и удалением и почти не требуется случайный доступ.

Технический момент: рисунок не фиксирует в деталях атрибуты *LINKED_LIST* кроме *first_element*, показывая просто затененную область. Хотя можно обойтись *first_element*, классы ниже включают атрибут *count*. Этот запрос может быть функцией, но неэффективно при каждом проходе по списку подсчитывать число элементов. Конечно, при использовании атрибута каждая операция вставки и удаления должна обновлять его значение. Здесь применим Принцип Унифицированного Доступа - можно менять реализацию, не нанося вреда клиентам класса.

Пассивные классы

Ясно, что нам нужны два класса: *LINKED_LIST* для списка (более точно, заголовка списка), *LINKABLE* для элементов списка - звеньев. Оба они являются универсальными.

Понятие *LINKABLE* является основой реализации, но не столь важно для большинства клиентов. Следует позаботиться об интерфейсе, обеспечивающем модули клиентов нужными примитивами, но не следует беспокоить их такими деталями реализации как представление элементов в звене списка. Атрибуты, соответствующие рисунку, появятся

как:

```
indexing
  description: "Звенья, используемые в связном списке"
  note: "Частичная версия, только атрибуты"
class
  LINKABLE1 [G]
feature {LINKED_LIST}
  item: G
    -- Значение звена
  right: LINKABLE [G]
    -- Правый сосед
end
```

Тип `right` можно было бы задавать как `like Current`, но предпочтительнее на этом этапе сохранить больше свободы в переопределении, поскольку пока непонятно, что может потребовать изменений у потомков `LINKABLE`.

Для получения настоящего класса следует добавить подпрограммы. Что допустимо для клиентов при работе со звеньями? Они могут изменять поля `item` и `right`. Можно также ожидать, что многие из клиентов захотят при создании звена инициализировать его значение, что требует процедуры создания. Вот подходящая версия класса:

```
indexing
  description: "Звенья, используемые в связном списке"
class LINKABLE [G] creation
  make
feature {LINKED_LIST}
  item: G
    -- Значение звена
  right: LINKABLE [G]
    -- Правый сосед
  make (initial: G) is
    -- Инициализация item значением initial
    do put (initial) end
  put (new: G) is
    -- Замена значения на new
    do item := new end
  put_right (other: LINKABLE [G]) is
    -- Поместить other справа от текущего звена
    do right := other end
end
```

Для краткости в классе опущены очевидные постусловия процедуры (такие как `ensure item = initial` для `make`). Предусловий здесь нет.

Ну, вот и все о `LINKABLE`. Теперь рассмотрим сам связный список, внутренне доступный через заголовок. Рассмотрим его экспортируемые компоненты: запрос на получение числа элементов (`count`), пуст ли список (`empty`), значение элемента по индексу `i(item)`, вставка нового элемента в определенную позицию (`put`), изменение значения `i`-го элемента (`replace`), поиск элемента с заданным значением (`occurrence`). Нам также понадобится запрос, возвращающий ссылку на первый элемент (`void`, если список пуст), который не должен экспортirоваться.

Вот набросок первой версии. Некоторые тела подпрограмм опущены.

```
indexing
  description: "Односвязный список"
  note: "Первая версия, пассивная"
class
  LINKED_LIST1 [G]
feature -- Access
  count: G
  empty: BOOLEAN is
    -- Пуст ли список?
  do
    Result := (count = 0)
  ensure
    empty_if_no_element: Result = (count = 0)
  end
  item (i: INTEGER): G is
    -- Значение i-го элемента
  require
    1 <= i; i <= count
  local
    elem: LINKABLE [G]; j: INTEGER
  do
```

```

        from
            j := 1; elem := first_element
        invariant j <= i; elem /= Void variant i - j until
            j = i
        loop
            j := j + 1; elem := elem.right
        end
        Result := elem.item
    end
occurrence (v: G): INTEGER is
    -- Позиция первого элемента со значением v (0, если нет)
    do ... end
feature -- Element change
put (v: G; i: INTEGER) is
    -- Вставка нового элемента со значением v,
    -- так что он становится i-м элементом
    require
        1 <= i; i <= count + 1
    local
        previous, new: LINKABLE [G]; j: INTEGER
    do
        -- Создание нового элемента
        create new.make (v)
        if i = 1 then
            -- Вставка в голову списка
            new.put (first_element); first_element := new
        else
            from
                j := 1; previous := first_element
            invariant
                j >= 1; j <= i - 1; previous /= Void
                -- previous - это j-й элемент списка
            variant
                i - j - 1
            until j = i - 1 loop
                j := j + 1; previous := previous.right
            end
            -- Вставить после previous
            previous.put_right (new)
            new.put_right (previous.right)
        end
        count := count + 1
    ensure
        one_more: count = old count + 1
        not_empty: not empty
        inserted: item (i) = v
        -- For 1 <= j < i,
        -- элемент с индексом j не изменил свое значение
        -- For i < j <= count,
        -- элемент с индексом j изменил свое значение
        -- на то, которое элемент с индексом j - 1
        -- имел перед вызовом
    end
replace (i: INTEGER; v: G) is
    -- Заменить на v значение i-го элемента
    require
        1 <= i; i <= count
    do
        ...
    ensure
        replaced: item (i) = v
    end
feature -- Removal
prune (i: INTEGER) is
    -- Удалить i-й элемент
    require
        1 <= i; i <= count
    do
        ...
    ensure
        one_less: count = old count - 1
    end

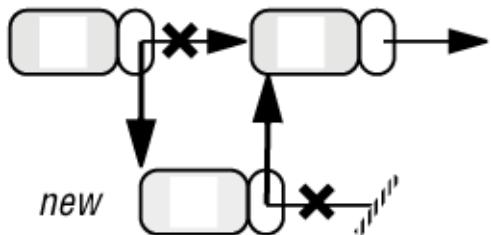
```

```

... другие компоненты ...
feature {LINKED_LIST} -- Implementation
    first_element: LINKABLE [G]
invariant
    empty_definition: empty = (count = 0)
    empty_iff_no_first_element: empty = (first_element = Void)
end

```

previous



Это хорошая идея попытаться самому закончить определение occurrence, replace и prune в этой первой версии. Убедитесь при этом, что поддерживается истинность инварианта.

Инкапсуляция и утверждения

До рассмотрения лучших версий несколько комментариев к первой попытке.

Класс LINKED_LIST1 показывает, что даже на совершенно простых структурах манипуляции со ссылками - это некий вид трюков, особенно в сочетании с циклами. Использование утверждений помогает в достижении корректности (смотри процедуру put и инвариант), но явная трудность операций этого типа является сильным аргументом в пользу их инкапсуляции раз и навсегда в повторно используемых модулях, как рекомендуется ОО-подходом.

Обратите внимание на применение Принципа Унифицированного Доступа; хотя count это атрибут и empty это функция, клиентам нет необходимости знать такие детали. Они защищены от возможных последующих изменений в реализации.

Утверждения для put полны, но из-за ограничений языка утверждений они не полностью формальны. Аналогично подробные предусловия следует добавить в другие подпрограммы.

Критика интерфейса класса

Удобно ли использование LINKED_LIST1? Давайте оценим наш проект.

Беспокоящий аспект - существенная избыточность: item и put содержат почти идентичные циклы. Похожий код следует добавить в процедуры, оставленные читателю (occurrence, replace, remove). Все же не представляется возможным вынести за скобки общую часть. Не слишком многообещающее начало.

Это внутренняя проблема реализации, - отсутствие повторно используемого внутреннего кода. Но это указывает на более серьезный изъян - плохо спроектированный интерфейс класса.

Рассмотрим процедуру occurrence. Она возвращает индекс элемента, найденного в списке, или 0 в случае его отсутствия. Недостаток ее в том, что она дает только первое вхождение. Что, если клиент захочет получить последующие вхождения? Но есть и более серьезная трудность. Клиент, выполнивший поиск, может захотеть изменить значение найденного элемента или удалить его. Но любая из этих операций требует повторного прохода по списку!

Проектируя компонент библиотеки общеселевого использования, нельзя допустить такую неэффективность. Любые потери производительности в повторно используемых решениях неприемлемы, в противном случае разработчики просто откажутся платить, обрекая на провал идею повторного использования.

Простые, напрашивающиеся решения

Как справиться с неэффективностью? На ум приходят два возможных решения:

- Пусть occurrence возвращает не целое, а ссылку на звено LINKABLE, где впервые появилось искомое значение, или void при неуспешном поиске. Тогда клиент имеет прямой указатель на нужный ему элемент и может выполнить требуемые операции без прохода по списку. (Например, использовать put класса LINKABLE для изменения элемента; при удалении нужна еще ссылка на предыдущий элемент.)
- Можно было бы обеспечить дополнительные примитивы, реализующие различные комбинации: поиск и удаление, поиск и замена.

Первое решение, однако, противоречит всей идее инкапсуляции данных, - клиенты должны манипулировать

внутренним представлением, со всеми вытекающими отсюда угрозами. Понятие звена является внутренним, а мы хотим, чтобы программист клиентского модуля думал в терминах списка и его значений, а не в терминах звеньев и указателей. В противном случае теряется смысл абстракции данных.

Второе решение мы пытались реализовать в ранней версии ISE. Для вставки элемента непосредственно перед вхождением известного значения клиент вместо вызова search вызывал:

```
insert_before_by_value (v: G; v1: G) is
    -- Вставка нового элемента со значением v перед первым
    -- вхождением элемента со значением v1 или в конец списка,
    -- когда нет вхождений
do
    ...
end
```

Это решение сохраняет скрытым внутреннее представление, устранивая неэффективность первой версии.

Но вскоре мы осознали, на какой скользкий путь мы встали. Рассмотрим все потенциально полезные варианты: search_and_replace, insert_before_by_value, insert_after_by_value, insert_after_by_position, insert_after_by_position, delete_before_by_value, insert_at_end_if_absent и так далее.

Это затрагивает жизненно важные вопросы проектирования библиотек. Написание общечелевого, повторно используемого ПО является трудной задачей, и нет гарантии, что с первого раза все хорошо получится. Конечно, хотелось бы, чтобы проект при его повторных использований следовал горизонтальной линии, показанной на [рис. 5.6](#). Но так не бывает, нужно быть готовым добавлять в классы новые компоненты для удовлетворения потребностей, возникающих у новых пользователей. Но этот процесс должен быть сходящимся. К сожалению, наше решение соответствует графику, показанному пунктиром на [рис. 5.6](#).

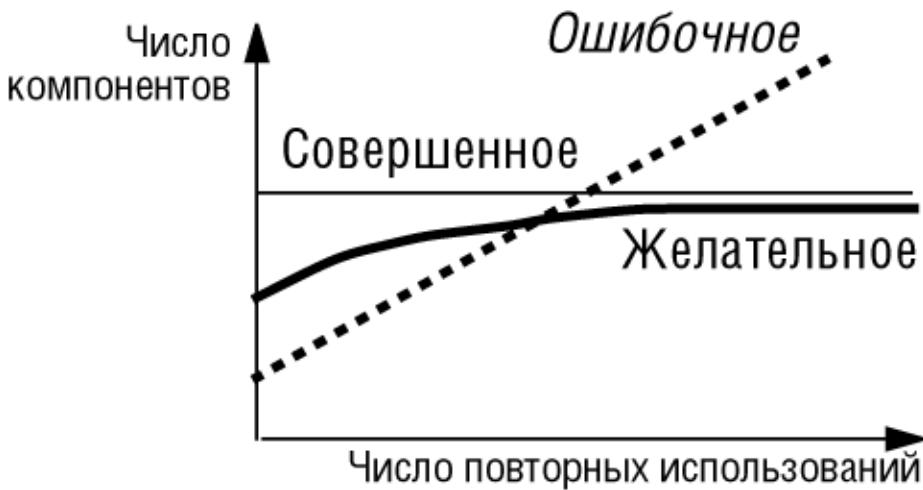


Рис. 5.6. Эволюция библиотечного класса

Введение состояния

К счастью, есть выход. Для его нахождения следует обратиться, как и положено, к абстрактному типу данных.

До сих пор список рассматривался как пассивное хранилище информации. Для обеспечения клиентов лучшим сервисом, список должен стать активным, "запоминая" точку выполнения последней операции.

Как отмечалось в этой лекции, без колебаний можно рассматривать объекты как машины с внутренним состоянием и вводить как команды, изменяющие состояние, так и запросы на этом состоянии. В первом решении список уже имел состояние, определяемое его содержимым и модифицируемое командами, такими как put and remove; но, добавив еще несколько компонентов, мы получим лучший интерфейс, делающий класс более простым и эффективным.

Помимо содержимого списка состояние теперь будет включать текущую активную позицию или курсор, интерфейс теперь будет позволять явным образом передвигать курсор.

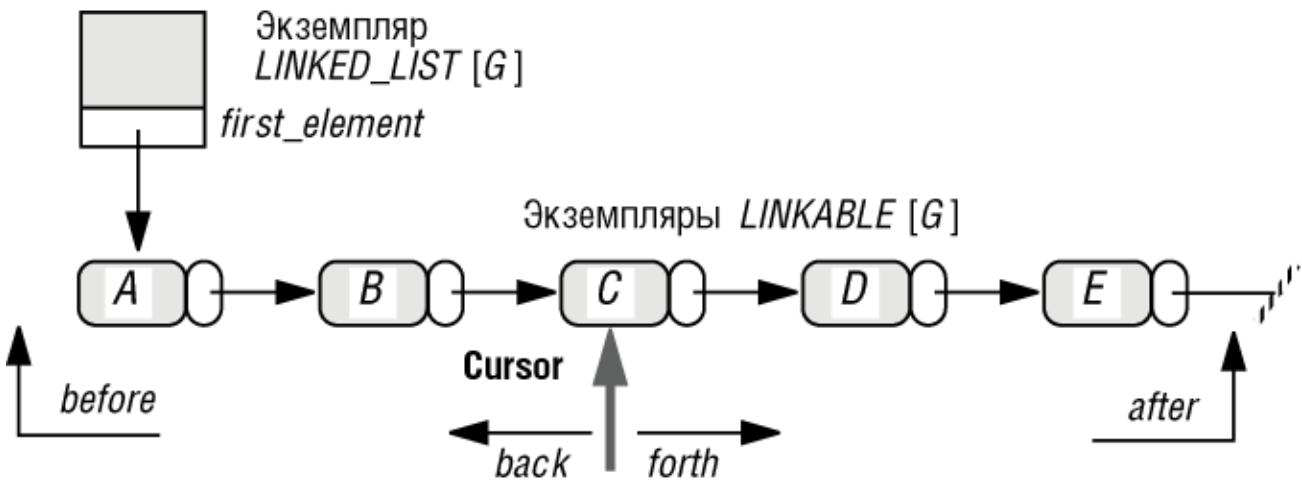


Рис. 5.7. Список с курсором

Мы полагаем, что курсор может указывать на элемент списка, если таковые имеются, или быть в позиции слева от первого - в этом случае булев запрос `before` будет возвращать `true`, или справа от последнего - тогда `after` принимает значение `true`.

Примером команды, передвигающей курсор, является процедура `search`, заменяющая функцию `occurrence`. Вызов `l.search(v)` передвинет курсор к первому элементу со значением `v` справа от текущей позиции курсора, или передвинет его в позицию `after`, если таких элементов нет. Заметьте, помимо прочего, это решает проблему множественных вхождений, - просто повторяйте поиск столько раз подряд, сколь это необходимо. Для симметрии можно также иметь `search_back`.

Основные команды манипулирования курсором:

- `start` и `finish`, передвигающие курсор в первую и последнюю позицию, если они определены;
- `forth` и `back`, передвигающие курсор в следующую и предыдущую позицию;
- `go(i)`, передвигающие курсор в позицию `i`.

Кроме `before` и `after` запросы о позиции курсора включают `index`, целочисленный номер позиции, начинающийся с 1 для первой позиции, а также булевы запросы `is_first` и `is_last`.

Процедуры построения и модификации списка - вставка, удаление, замена - становятся проще, поскольку они не должны заботиться о позиции, а будут действовать на элемент, заданный курсором. Все циклы исчезают! Например, `remove` не будет теперь вызываться как `l.remove(i)`, а просто `l.remove`. Необходимо установить точные и согласованные условия того, что случится с курсором при выполнении каждой операции:

- `Remove`, без аргументов, удаляет элемент в позиции курсора и перемещает курсор к правому соседу, так что значение `index` не изменится в результате.
- `Put_right(v: G)` вставляет элемент со значением `v` справа от курсора, не передвигая его, так что значение `index` не изменится.
- `Put_left(v: G)` вставляет элемент со значением `v` слева от курсора, не передвигая его, увеличивая значение `index` на единицу.
- `Replace(v: G)` изменяет значение элемента в позиции курсора. Значение этого элемента может быть получено функцией `item`, не имеющей аргументов, которая может быть реализована атрибутом.

Поддержка согласованности: инвариант реализации

При построении класса для фундаментальной структуры данных следует тщательно позаботиться обо всех деталях. Утверждения здесь обязательны. Без них велика вероятность пропуска важных деталей. Например:

- Является ли вызов `start` допустимым, если список пуст; если да, то каков эффект вызова?
- Что случится с курсором после `remove`, если курсор был в последней позиции? Неформально мы к этому подготовились, позволяя курсору передвигаться на одну позицию левее и правее списка. Но нам нужны более точные утверждения, недвусмысльно описывающие все случаи.

Ответы на вопросы первого вида будут даны в виде предусловий и постусловий.

Для таких свойств, как допустимые позиции курсора, следует использовать предложения, устанавливающие инвариант реализации. Напомним, он отражает согласованность представления, задающего класс - визави АТД. В данном случае он включает свойство:

```
0 <= index; index <= count + 1
```

Что можно сказать о пустом списке? Необходима симметрия по отношению к левому и правому. Одно решение, принятное в ранних версиях библиотеки, состояло в том, что пустой список это тот единственный случай, когда `before`

и after оба истинны. Это работает, но приводит в алгоритмах к частой проверке тестов в форме: if after and not empty. Это привело нас к концептуальному изменению точки зрения и введению в список двух специальных элементов - **стражей (sentinel)**, изображаемых на рисунке в виде специальных картинок.

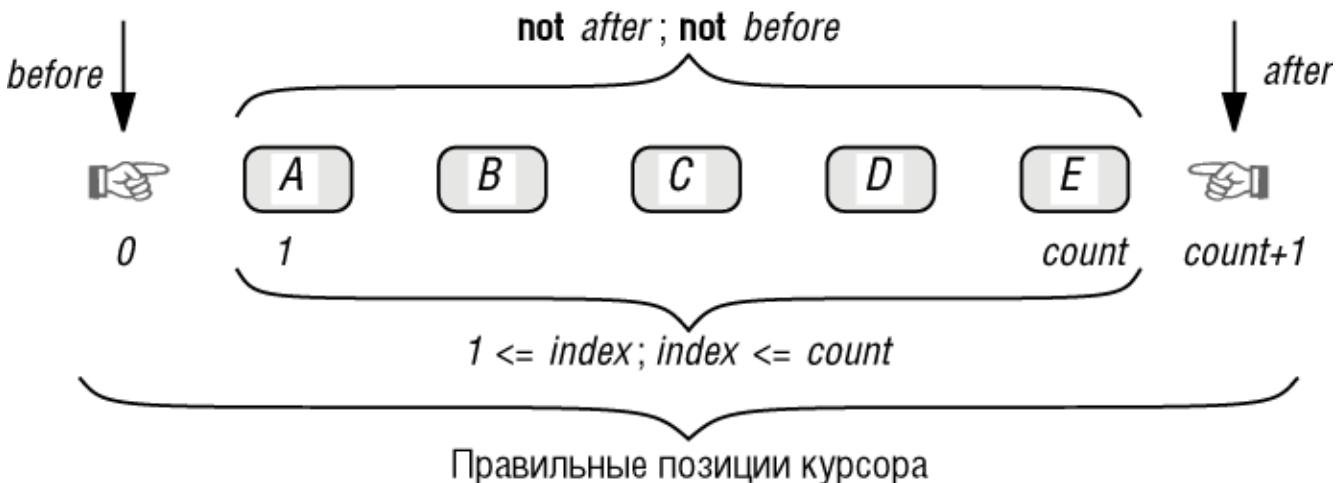


Рис. 5.8. Список со стражами

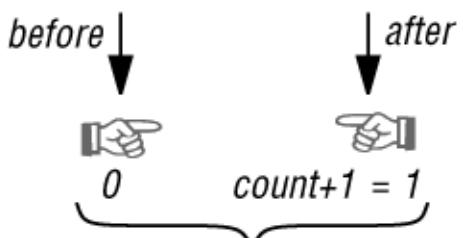
Стражи помогают нам разобраться в структуре, но нет причин хранить их в представлении. Реализация рассматривает возможность хранения только левого стража, но не правого, можно также использовать реализацию совсем без стражей, хотя и продолжающую соответствовать концептуальной модели, представленной на предыдущем рисунке.

Часто хочется установить, что некоторый индекс соответствует позиции какого-либо элемента списка. Для этого можно предложить соответствующий запрос:

```
on_item (i: INTEGER): BOOLEAN is
    -- Есть ли элемент в позиции i?
    do
        Result := ((i >= 1) and (i <= count))
    ensure
        within_bounds: Result = ((i >= 1) and (i <= count))
        no_elements_if_empty: Result implies (not empty)
    end
```

Для установления того, что есть элемент списка в позиции курсора, можно определить запрос readable, чье значение определялось бы как on_item (index). Это хороший пример Принципа Списка Требований, поскольку readable концептуально избыточен, то минималистская позиция отвергала бы его, в то же время его включение обеспечивало бы клиентов лучшей абстракцией, освобождая их от необходимости запоминания того, что точно означает индекс на уровне реализации.

Инвариант устанавливает истинность утверждения: not (after and before). В граничном случае пустого списка картина выглядит так:



Правильные позиции курсора

Рис. 5.9. Пустой список со стражами

Итак, пустой список имеет два возможных состояния: empty and before и empty and after, соответствующие двум позициям курсора на рисунке. Это кажется странным, но не имеет неприятных последствий и на практике предпочтительнее прежнего соглашения empty = (before and after), сохраняя справедливость empty implies (before or after).

Отметим два общих урока: важность, как во многих математических и физических задачах, проверки граничных случаев, важность утверждений, выражающих точные свойства проекта. Вот некоторые принципиальные предложения, включенные в инвариант:

```
0 <= index; index <= count + 1
before = (index = 0); after = (index = count + 1)
is_first = ((not empty) and (index = 1));
is_last = ((not empty) and (index = count))
empty = (count = 0)
```

```
-- Три следующих предложения являются теоремами,
-- выводимыми из предыдущих утверждений:
empty implies (before or after)
not (before and after)
empty implies ((not is_first) and (not is_last))
```

Этот пример иллюстрирует общее наблюдение, что написание инварианта - лучший способ прийти к настоящему пониманию особенностей класса. Утверждения инварианта применимы ко всем реализациям последовательных списков, они будут дополнены еще несколькими, отражающими специфику выбора связного представления.

Последние три предложения инварианта выводимы из предыдущих (см. У5.6). Для инвариантов минимализм не требуется, часто полезно включать дополнительные предложения, если они устанавливают важные, нетривиальные свойства. Как мы видели (см. [лекцию 6](#) курса "Основы объектно-ориентированного программирования"), АТД и, как следствие, реализация класса является теорией, в данном случае теорией связных списков. Утверждения инварианта выражают аксиомы теории, но любая полезная теория имеет также и интересные теоремы.

Конечно, если вы хотите проверять инварианты в период выполнения, то рассматривать дополнительные предложения имеет смысл, если вы не доверяете обоснованности теории. Но это делается только на этапах разработки и отладки. В производственной системе нет причин наблюдения за инвариантами.

С точки зрения клиента

Этот проект обеспечивает простой и элегантный интерфейс реализации связных списков. Операции, такие как "поиск, а затем вставка", используют два последовательных вызова, хотя и без существенной потери эффективности:

```
l: LINKED_LIST [INTEGER]; m, n: INTEGER
...
l.search (m)
if not after then l.put_right (n) end
```

Вызов `search (m)` передвинет курсор к следующему вхождению `m` после текущей позиции курсора или `after`, если таковой нет. (Здесь предполагается, что курсор изначально установлен на первом элементе, если нет, то клиент должен прежде выполнить `l.start()`.)

Для удаления третьего вхождения некоторого элемента клиент выполнит:

```
l.start; l.search (m); l.search (m); l.search (m)
if not after then l.remove end
```

Для вставки элемента в позицию `i`:

```
l.go (i); l.put_left (i)
```

и так далее. Мы получили простое и ясное использование интерфейса, сделав явным внутреннее состояние, обеспечив клиента подходящими командами и запросами об этом состоянии.

Взгляд изнутри

Новое решение упрощает реализацию, также как и улучшает интерфейс. Более важно, дав каждой подпрограмме простую спецификацию, мы смогли сконцентрироваться только на одной задаче. Это позволило избавиться от ненужной избыточности, в частности от лишних циклов. Процедуры вставки и удаления занимаются теперь своей задачей и им не нужно выполнять проход по списку. Ответственность за позиционирование курсора теперь лежит на других подпрограммах (`back`, `forth`, `go`, `search`), только некоторым из которых нужны циклы (`go` и `search`).

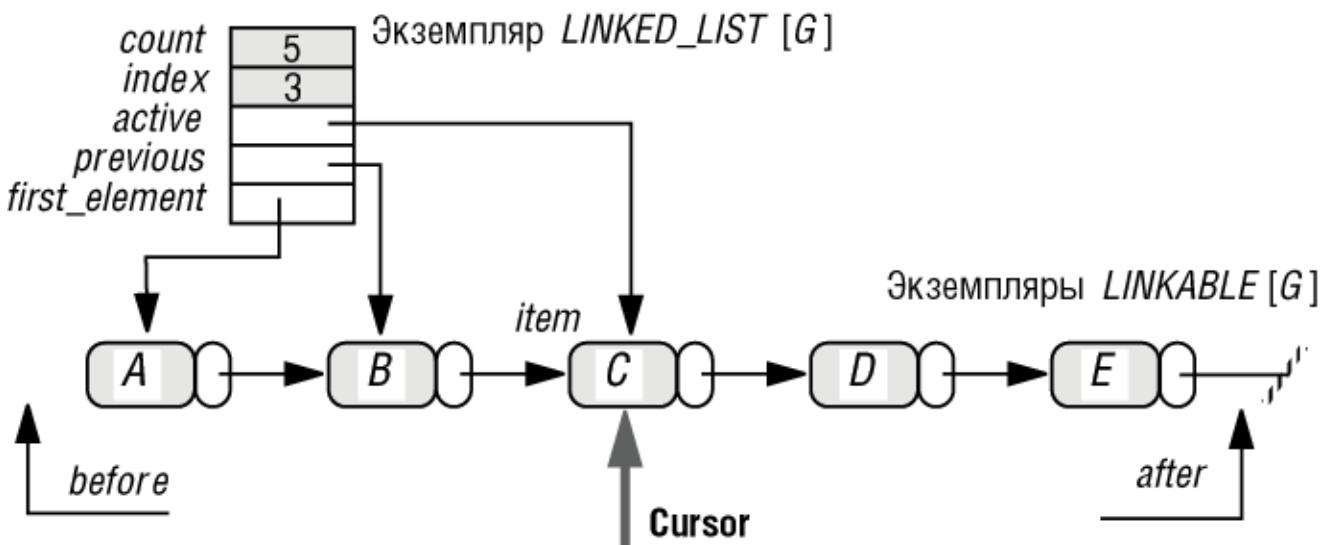


Рис. 5.10. Представление списка с курсором (первый вариант)

В заголовке списка наряду со ссылкой на первый элемент `first_element` полезно хранить еще две ссылки на элемент в позиции курсора `active` и предшествующий ему элемент - `previous`. Это позволит эффективно выполнять вставку и удаление.

Клиенты могут узнать, каково состояние списка, имея доступ к открытым целочисленным атрибутам `count` и `index` и булевым запросам: `before`, `after`, `is_first`, `is_last`, `item`. Вот две типичные функции:

```
after: BOOLEAN is
    -- Находится ли курсор за списком?
    do Result := (index = count + 1) end
is_first: BOOLEAN is
    -- Установлен ли курсор на первом элементе?
    do Result := (index = 1) end
```

(Напишите самостоятельно функции `before` и `is_last`.) Для функции `after` высказывание "Стоит ли курсор справа от последнего элемента?" не совсем корректно, так как `after` может быть истинным, даже если в списке совсем нет элементов. Комментарии к заголовкам следует писать так, чтобы они были ясными; лаконичность и аккуратность - сестры таланта (см. [лекцию 8](#)).

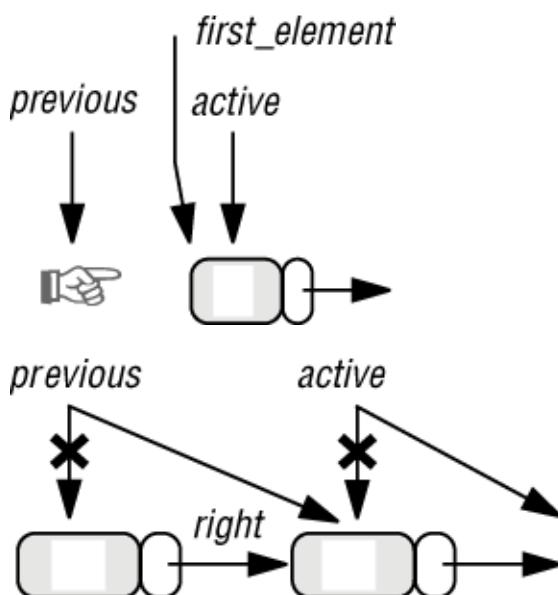
Запрос `item` возвращает элемент в позиции курсора, если таковой имеется:

```
item: G is
    -- Элемент в позиции курсора
    require
        readable: readable
    do
        Result := active.item
    end
```

Напоминаю, `readable` указывает, установлен ли курсор на элементе списка (`index` между 1 и `count`). Также заметьте, `item` в `active.item` ссылается на атрибут в `LINKABLE`, а не на функцию из самого `LINKED_LIST`.

Рассмотрим теперь основные команды манипулирования курсором. Обращаться с ними нужно довольно деликатно, в утешение можно заметить, что лишь небольшая их часть - `start`, `forth`, `put_right`, `put_left` и `remove`, - выполняет нетривиальные операции над ссылками. Давайте начнем с команд `start` и `forth`. Процедура `start` должна работать как с пустым, так и с не пустым списком. Для пустого списка соглашение состоит в том, что `start` передвигает курсор ко второму стражу.

```
start1 is
    -- Передвигает курсор к первой позиции.
    -- (Предварительная версия.)
do
    index := 1
    previous := Void
    active := first_element
ensure
    moved_to_first: index = 1
    empty_convention: empty implies after
end
forth1 is
    -- Передвигает курсор к следующей позиции.
    -- (Предварительная версия.)
require
    not_after: not after
do
    index := index + 1
    if before then
        active := first_element; previous := Void
    else
        check active /= Void end
        previous := active; active := active.right
    end
ensure
    moved_by_one: index = old index + 1
end
```



Пора остановиться! Все становится слишком сложным и неэффективным. Производительность процедуры *forth* является критической, поскольку типично она используется клиентом в цикле *from start until after loop ...; forth end*. Можно ли избавиться от теста?

Можно, если всерьез рассматривать левого стражи и всегда создавать его одновременно с созданием списка. (Процедура создания *make* для *LINKED_LIST* остается в качестве упражнения.) Заменим *first_element* ссылкой *zeroth_element* на левого стража:

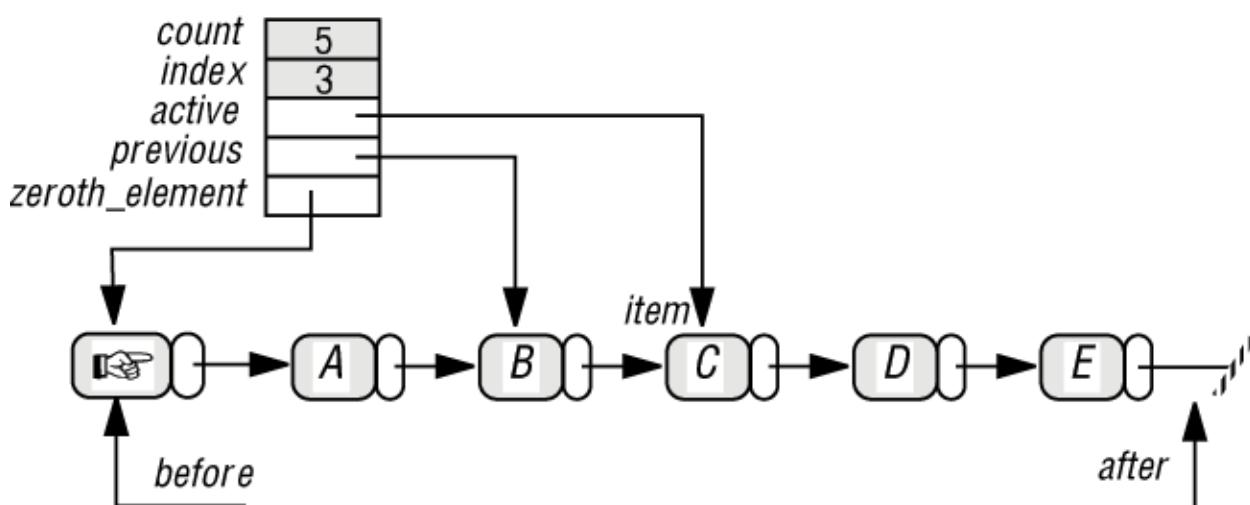


Рис. 5.11. Представление списка с курсором (пересмотренная версия)

Свойства *zeroth_element* /= Void и *previous* /= Void будут теперь частью инварианта (следует, конечно, убедиться, что процедура создания обеспечивает его выполнение). Они весьма ценные, поскольку позволяют избавиться от многих повторяемых проверок.

Процедура *forth*, запускаемая после обновленной процедуры *start*, теперь проще и быстрее (без проверок!):

```

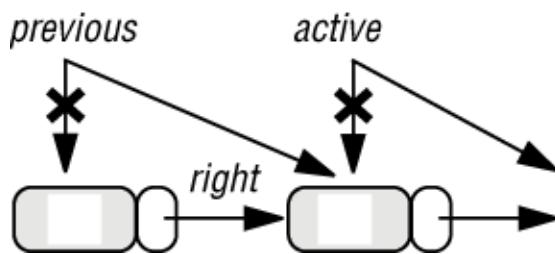
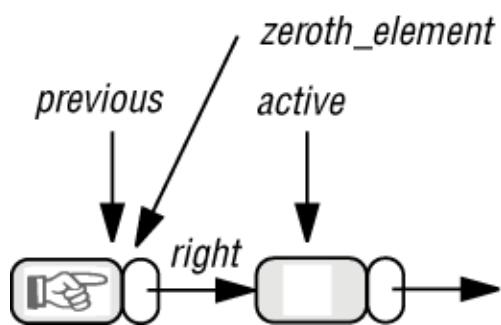
start is
    -- Передвигает курсор к первой позиции
do
    index := 1
    previous := zeroth_element
    active := previous.right
ensure
    moved_to_first: index = 1
    empty_convention: empty implies after
    previous_is_zeroth: previous = zeroth_element
end
forth is
    -- Передвинуть курсор к следующей позиции.
    -- (Версия пересмотрена в интересах эффективности. Без тестов!)
require
    not_after: not after
do
    index := index + 1

```

```

    previous := active
    active := active.right
ensure
    moved_by_one: index = old_index + 1
end

```

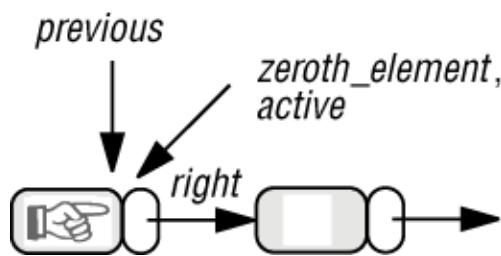


Удобно определить go_before, устанавливающую курсор на левом страже:

```

go_before is
    -- Передвигает курсор к before
do
    index := 0
    previous := zeroth_element
    active := zeroth_element
ensure
    before: before
    previous_is_zeroth: previous = zeroth_element
    previous_is_active: active = previous
end

```



Процедура go определяется в терминах go_before и forth:

```

go (i: INTEGER) is
    -- Передвигает курсор к i-й позиции
require
    not_offleft: i >= 0
    not_offright: i <= count + 1
do
    from
        if i < index then go_before end
    invariant index <= i variant i - index until index = i loop
        forth
    end
ensure
    moved_there: index = i
end

```

Мы старательно избегали проходов по списку. Процедуре go, единственной из рассмотренных, необходим цикл. Для симметрии следует добавить finish, перемещающую курсор к последней позиции, реализуемую вызовом go (count + 1).

Хотя и нет настоящей независимости, удобно (Принцип Списка Требований) экспортовать go_before. Тогда для

симметрии следует включить и go_after, выполняющую go (count + 1), и экспортировать ее.

Также для симметрии добавлена процедура back, содержащая цикл, подобный go:

```
back is
    -- Передвинуть курсор к предыдущей позиции
    require
        not_before: not before
    do
        check index - 1 >= 0 end
        go (index - 1)
    ensure
        index = old index - 1
    end
```

Приятно иметь симметрию между back и forth, однако в ней таится угроза, поскольку клиент может беззаботно вызывать back, не думая, что ее реализация содержит цикл, в котором index - 1 раз вызывается forth. Если работа с левой частью списка проводится от случая к случаю, то односторонний список является подходящим, если же одинаково часто необходимо обращаться к элементам слева и справа от текущего, то необходимо перейти к двунаправленному списку. Соответствующий класс может быть построен как наследник LINKED_LIST (наследование используется корректно, так как двунаправленный список одновременно является и односторонним). Создание такого списка оставлено в качестве упражнения (см. У5.7). Следует его выполнить, если хотите достигнуть полного понимания концепций.

Ранние утверждения в инварианте не зависели от реализации. Добавим теперь утверждения, описывающие особенности реализации:

```
empty = (zeroth_element.right = Void)
zeroth_element /= Void; previous /= Void
(active = Void) = after; (active = previous) = before
(not before) implies (previous.right = active)
(previous = zeroth_element) = (before or is_first)
is_last = ((active /= Void) and then (active.right = Void))
```

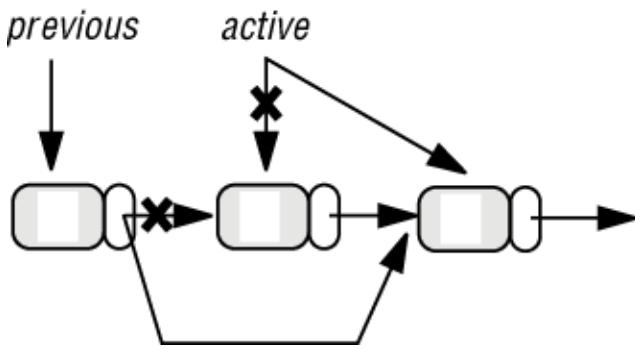
Большинство из запросов реализуются непосредственно - before возвращает булево значение (index = 0) и after - (index = count + 1). Элемент в позиции курсорадается:

```
item: G is
    -- Значение элемента в позиции курсора
    require
        readable: readable
    do
        Result := active.item
    end
```

Процедура search подобна go и оставлена читателю. Следует также написать процедуру i_th (i: INTEGER), возвращающую элемент в позиции i. Следует позаботиться об отсутствии абстрактного побочного эффекта, допуская конкретный побочный эффект.

Последняя категория компонентов включает процедуры вставки и удаления:

```
remove is
-- Удаляет элемент в позиции курсора и передвигает курсор к правому соседу.
-- (Если нет правого соседа, то становится истинным after).
require
    readable: readable
do
    active := active.right
    previous.put_right (active)
    count := count - 1
ensure
    same_index: index = old index
    one_less_element: count = old count - 1
    empty_implies_after: empty implies after
end
```



Процедура выглядит тривиальной, но это благодаря технике левого стража как физического объекта, что позволяет избежать тестов в форме `previous /= Void` и `first_element /= Void`. Стоит рассмотреть более сложное и менее эффективное тело процедуры, полученное без этого упрощения. Внимание: отвергнутая версия!

```

active := active.right
if previous /= Void then previous.sput_right (active) end
count := count - 1
if count = 0 then
    first_element := Void
elseif index = 1 then
    first_element := active
-- Иначе first_element не изменяется
end

```

Утверждения помогают понять намерения и избежать ошибок. Следует поупражняться в овладении этой техникой, написав процедуры `put_left` и `put_right`.

АТД и абстрактные машины

Понятие активной структуры данных широко применимо и согласуется с ранее введенными принципами, в частности с Принципом Разделения Команд и Запросов. Явно вводя состояние структуры данных, зачастую приходим к простому интерфейсу документа.

Кто-то может высказать опасение, что в результате структура становится менее абстрактной, но это не тот случай. Абстракция не означает пассивность. Теория АТД говорит, что объекты становятся известными через описания применимых операций и свойств, но это не предполагает их рассмотрение как хранилищ данных. Введение состояния и операций над ним фактически обогащает спецификацию АТД, добавляя функции и свойства. Состояние является чистой абстракцией, всегда непосредственно доступной благодаря командам и запросам.

Взгляд на объекты как на машину с состояниями соответствует тому, что АТД становятся более **императивными**, но не менее абстрактными.

Отделение состояния

Возможно развитие предыдущей техники. До сих пор курсор был лишь концепцией, реализованной атрибутами `previous`, `active` и `index`, но не был одним из классов. Можно определить класс `CURSOR`, имеющий потомков, вид которых зависит от структуры курсора. Тогда мы можем отделить атрибуты, задающие содержимое списка (`zeroth_element`, `count`), от атрибутов, связанных с перемещением по списку, хранящихся в объекте курсора.

Хотя мы не будем доводить до логического конца эту идею, отметим ее полезность для параллельного доступа. Если некоторым клиентам нужно разделять доступ к структуре данных, то каждый из них мог бы иметь свой собственный курсор.

Слияние списка и стражей

(Этот раздел описывает улучшенную оптимизацию и может быть опущен при первом чтении.)

Пример связного списка со стражами может быть улучшен благодаря еще одной оптимизации, которая реально используется в последней версии библиотек ISE. Мы бегло ее рассмотрим, поскольку она достаточно специфична и не носит общего характера. Такие оптимизации, тщательно выполняемые, должны осуществляться только для широко используемых компонентов повторного использования. Другими словами, они не для домашних разработок.

Можно ли получить преимущества от стражей без соответствующих потерь памяти? При рассмотрении их концепции отмечалось, что их можно рассматривать фиктивно, но тогда мы потеряли критически важную оптимизацию, позволившую нам написать тело forth следующим образом:

```

index := index + 1
previous := active
active := active.right

```

без дорогих проверок предыдущей версии. Мы избежали тестов, будучи уверенными, что `active` не равно `Void`, когда список не находится в состоянии `after`. Это следствие утверждения инварианта (`active = Void`) = `after`; верного потому, что у нас есть настоящее звено - страж, доступный как `active`, даже если список пуст.

Для других программ, отличных от `forth`, оптимизация не столь существенна. Но `forth`, как отмечалось, - наущная потребность, "хлеб и масло" клиентов, обрабатывающих списки. Из-за последовательной природы списков типичное использование имеет вид:

```
from your_list.start until your_list.after loop ...; your_list.forth end
```

Нет ничего необычного, если при построении профиля, измеряя, как выполняются вычисления, вы обнаружите, что большая доля времени приходится на работу `forth`. Так что стоило заплатить за ее оптимизацию, поскольку она обеспечивает кардинальное улучшение производительности, будучи свободной от тестов.

За выигрыш во времени мы заплатили, как обычно, проигрышем в памяти, - теперь каждый список имеет дополнительный элемент, не хранящий информации. Это кажется проблемой лишь в случае большого числа коротких списков, иначе относительные потери несущественны. Но могут возникнуть более серьезные проблемы:

- Во многих случаях, как упоминалось, могут понадобиться двунаправленные списки, полностью симметричные с элементами класса `BT_LINKABLE`, имеющие ссылки на левого и правого соседа. Класс `TWO_WAY_LIST` (который, кстати, может быть написан как дважды наследуемый от `LINKED_LIST`, основываясь на технике дублируемого наследования) будет нуждаться как в левом, так и правом страже.
- Связные деревья (см. [лекцию 15](#) курса "Основы объектно-ориентированного программирования") представляют более серьезную проблему. Практически важным является класс `TWO_WAY_TREE`, задающий удобное представление деревьев с двумя ссылками (на родителя и потомка). Построенный на идеях, описываемых при представлении множественного наследования, класс объединяет понятия узла и дерева, так что он является наследником `TWO_WAY_LIST` и `BT_LINKABLE`. Но тогда каждый узел является списком, может быть двунаправленным и хранить обоих стражей.

Хотя во втором случае есть и другие способы решения проблемы - переобъявление структуры наследования - давайте попробуем получить лучшее из возможного.

Для нахождения решения зададимся неожиданным вопросом.

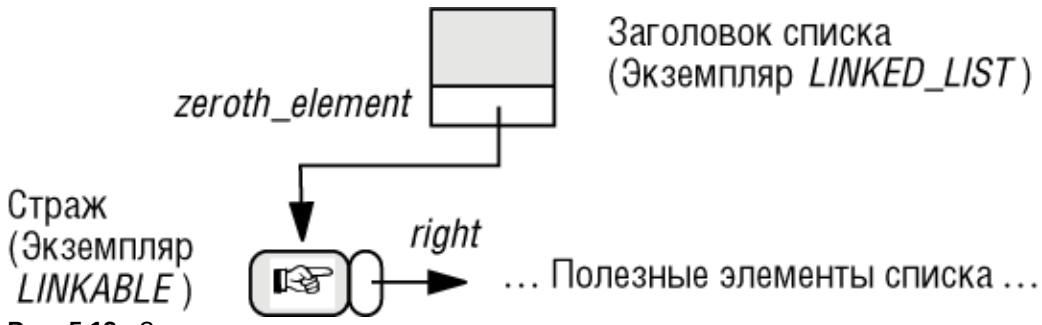


Рис. 5.12. Заголовок и страж

В нашей структуре нужны ли действительно **два** объекта, занимающиеся учетом? По настоящему полезная информация находится в фактических элементах списка, не показанных на рисунке. Для управления ими мы добавили заголовок списка и страж - двух стражей для двунаправленного списка. Для длинных списков можно игнорировать эту раздутую структуру учета, подобно большой компании, содержащей много сотрудников среднего звена во времена процветания, но в тяжелые времена, объединяющую функции управления.

Можем ли мы заставить заголовок списка играть роль стража? Оказывается, можем. Все, что имеет `LINKABLE` - это поле `item` и ссылку `right`. Для стража необходима только ссылка, указывающая на первый элемент, так что, если поместить ее в заголовок, то она будет играть ту же роль, как когда она называлась `first_element` в первом варианте со стражами. Проблема, конечно, была в том, что `first_element` мог иметь значение `void` для пустого списка, что принуждало во все алгоритмы встраивать тесты в форме `if before then ...`. Мы точно не хотим возвращаться назад к этой ситуации. Но можем использовать концептуальную модель, показанную на рисунке, избавленную от стража

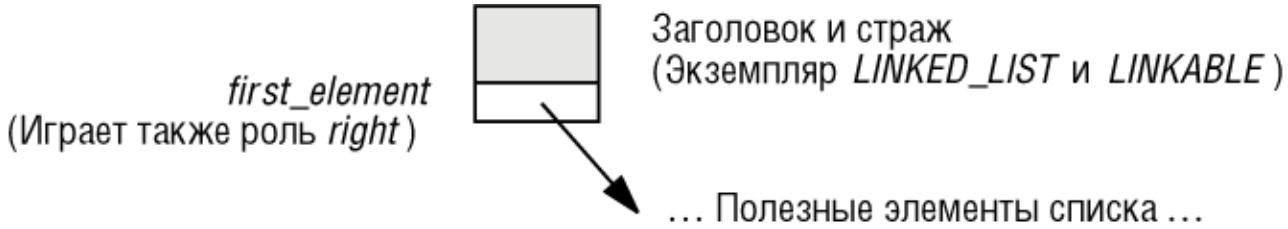


Рис. 5.13. Заголовок как страж (на непустом списке)

Концептуально решение является тем же самым, что и прошлое, с заменой `zeroth_element` ссылкой на сам заголовок списка. Для представления того, что ранее было `zeroth_element.right`, теперь используется

`first_element`.



Рис. 5.14. Заголовок как страж (на пустом списке)

На пустом списке следует присоединить `first_element` к самому заголовку. Таким образом, `first_element` никогда не будет void - критическая цель, сохраняющая ситуацию простой. Осталось лишь выполнить правильные замены.

Сохраним все желательные утверждения инварианта предыдущей версии со стражами:

```
previous /= Void  
(active = Void) = after; (active = previous) = before  
(not before) implies (previous.right = active)  
is_last = ((active /= Void) and then (active.right = Void))
```

Предложения, включающие ранее `zeroth_element`:

```
zeroth_element /= Void  
empty = (zeroth_element.right = Void)  
(previous = zeroth_element) = (before or is_first)
```

теперь будут иметь вид:

```
first_element /= Void  
empty = (first_element = Current)  
(previous = Current) = (before or is_first)
```

Но чтобы все получилось так просто, необходимо (нас ждут опасные вещи, поэтому пристегните ремни) сделать `LINKED_LIST` наследником `LINKABLE`:

```
class LINKED_LIST [G] inherit  
    LINKABLE [G]  
        rename right as first_element, put_right as set_first_element end  
...Остальное в классе остается как ранее с выше показанной заменой zeroth_element...
```

Не нонсенс ли это - позволить `LINKED_LIST` быть наследником `LINKABLE`? Совсем нет! Вся идея в том, чтобы слить воедино два понятия заголовка списка и стража, другими словами рассматривать заголовок списка как элемент списка. Поэтому мы имеем прекрасный пример отношения "is-a" ("является") при наследовании. Мы решили рассматривать каждый `LINKED_LIST` как `LINKABLE`, поэтому наследование вполне подходит. Заметьте, отношение "быть клиентом" даже не участвует в соревновании - не только потому, что оно не подходит по сути, но оно добавляет лишние поля к нашим объектам!

Убедитесь, что ваши ремни безопасности все еще застегнуты, - мы начинаем рассматривать, что происходит в наследуемой структуре. Класс `BI_LINKABLE` дважды наследован от `LINKABLE`. Класс `TWO_WAY_LIST` наследован от `LINKED_LIST` (один раз или, возможно, дважды в зависимости от выбранной техники наследования) и, в соответствии с рассматриваемой техникой, от `BI_LINKABLE`. Со всем этим повторным наследованием каждый может подумать, что вещи вышли из-под контроля и наша структура содержит все виды ненужных полей. Но нет, правила разделения и репликации при дублируемом наследовании позволяют нам получить то, что мы хотим.

Последний шаг - класс `TWO_WAY_TREE`, по разумным причинам наследуемый от `TWO_WAY_LIST` и `BI_LINKABLE`. Достаточно для небольшого сердечного приступа, но нет - все прекрасно сложилось в нужном порядке и в нужном месте. Мы получили все необходимые компоненты, ненужных компонентов нет, концептуально все стражи на месте, так что `forth`, `back` и все связанные с ними циклы выполняются быстро, как это требуется, и стражи совсем не занимают памяти.

Это схема реально применяется ко всем действующим классам библиотеки Base. Прежде чем закончить наш полет, еще несколько наблюдений:

- Ни при каких обстоятельствах не следует выполнять работу такого вида, включающую трюки с манипуляцией структурой данных, без использования преимуществ, обеспечиваемых утверждениями. Просто невозможно обеспечить правильную работу, не установив точные утверждения инварианта и проверки совместимости.
- Механизмы дублируемого наследования являются основой (см. [лекцию 15](#) курса "Основы объектно-ориентированного программирования"). Без методов, введенных нотацией этой книги, позволяющих дублируемым потомкам добиваться разделения или покомпонентной репликации, основываясь на простом критерии имен, невозможно эффективно описать любую ситуацию с серьезным использованием дублируемого наследования.
- Повторим наиболее важный комментарий: такие оптимизации, требующие крайней осторожности, имеют смысл только в общечелевых библиотеках, предназначенных для широкого повторного использования. В обычных

ситуациях они стоят слишком дорого. Это обсуждение включено с целью дать читателю почувствовать, какие усилия требуются для разработки профессиональных компонентов от начала и до конца. К счастью, большинство разработчиков не должны прилагать столько усилий в своей работе.

Выборочный экспорт

Отношение между классами LINKABLE и LINKED_LIST иллюстрируют важность поддержки у компонента более двух статусов экспорта - открытого (общедоступного) и закрытого (секретного).

Класс LINKABLE не должен делать свои компоненты - item, right, make, put, put_right - общедоступными, так как большинство клиентов не должно влезать во внутренности звеньев, они должны использовать только связные списки. Но их нельзя делать секретными, поскольку это спрятало бы их от LINKED_LIST, для которого они и предназначены, так как вызовы active.right, основа операций forth и других подпрограмм LINKED_LIST, были бы тогда невозможны.

Выборочный экспорт обеспечивает решение, позволяя LINKABLE отбирать то множество классов, которому и только которому экспортируются эти компоненты:

```
class
  LINKABLE [G]
feature {LINKED_LIST}
  item: G
  right: LINKABLE [G]
  -- и т.д.
end
```

Напомним, это делает эти компоненты доступными для всех потомков LINKED_LIST, что является непременным условием, если им нужно переопределить некоторые из наследуемых подпрограмм или добавить свои собственные.

Иногда, как мы видели в предыдущих лекциях, класс должен экспортировать компонент выборочно самому себе. Например, BI_LINKABLE, наследник LINKABLE, описывающий двунаправленный список с полем left, включает утверждение инварианта в форме:

```
(left /= Void) implies (left.right = Current)
```

Это требует, чтобы right было объявлено в предложении feature { ... Другие классы ..., BI_LINKABLE}; в противном случае вызов left.right будет неверным.

Предложения выборочного экспорта существенны, когда группе связанных классов, таким как LINKABLE и LINKED_LIST, взаимно необходимы для их реализации компоненты друг друга, хотя эти компоненты остаются закрытыми и не должны быть доступными для других классов.

Напоминание: при обсуждении в предыдущих лекциях отмечалось, что выборочный экспорт является ключевым требованием для децентрализации архитектуры ОО-ПО.

Как справляться с особыми ситуациями

Наша следующая тема проектирования интерфейса связана с проблемой, возникающей перед каждым разработчиком: как управлять случаями, отклоняющимися от нормальных, ожидаемых схем?

Вне зависимости от причин возникновения ошибок - по вине пользователя системы, операционного окружения, сбоев аппаратуры, некорректных исходных данных или некорректного поведения других модулей - специальные случаи являются головной болью разработчиков. Необходимость учета всех возможных ситуаций - серьезное препятствие в постоянной битве со сложностью ПО.

Эта проблема оказывает серьезное влияние на проектирование интерфейса модулей. Если бы эти заботы были сняты с разработчика, то можно было бы писать ясные, элегантные алгоритмы для нормальных случаев и полагаться на внешние механизмы, берущие на себя заботу в остальных ситуациях. Много надежд возлагалось на механизм исключений. В языке Ada, например, можно написать нечто такое:

```
if some_abnormal_situation_detected then
  raise some_exception;
end;
"далее - нормальная обработка"
```

Выполнение инструкции raise прервёт выполнение текущей программы и передаст управление "обработчику события", написанному в модуле,зывающем программу. Но это всего лишь управляющая структура, а не метод, позволяющий справиться с ненормальными ситуациями. В конечном счете придется решать, что делать в той или иной ситуации: возможно ли ее исправить? Если да, то как это сделать, и что делать потом, как вернуть управление системе? Если нет, то как лучшим способом, быстро и элегантно завершить выполнение?

Мы видели в [лекции 12](#) курса "Основы объектно-ориентированного программирования", что механизм дисциплинированных исключений полностью соответствует ОО-подходу, в частности согласуется с Принципом

Проектирования по Контракту. Но не во всех специальных случаях обоснованно обращаться к исключениям. Техника проектирования, которой мы сейчас займемся, на первый взгляд, покажется менее выразительной, может характеризоваться как "техника низкого уровня" ("low-tech"). Но она чрезвычайно мощная и подходит ко многим возможным практическим ситуациям. После ее изучения дадим обзор тех ситуаций, где использование исключений остается непременным.

Априорная схема

Вероятно, наиболее важный критерий, позволяющий справляться с особыми случаями на уровне интерфейса модуля - это спецификация. Если вы точно знаете, какие входы готов принять каждый программный элемент и какие гарантии он дает на выходе, то половина битвы уже выиграна.

Эта идея была глубоко разработана в [лекции 11](#) курса "Основы объектно-ориентированного программирования", где изучалось Проектирование по Контракту. В частности, мы видели, что, противореча общепринятой мудрости, надежность не достигается включением возможных проверок. Ответственность четко разделяется, каждый класс - клиент или поставщик - несет свою долю ответственности.

Включение ограничений в предусловие подпрограммы означает, что за их выполнение отвечает клиент. Предусловие выражает те требования, которые необходимы, чтобы операцию можно было выполнить.

```
operation (x...) is
  require
    precondition (x)
  do
    ... Код, работающий только при условии выполнения предусловия...
  end
```

Предусловие должно быть полным, когда это возможно, гарантируя, что любой удовлетворяющий ему вызов успешно закончится. В этом случае у клиента есть два способа работы. Один - явная проверка условия перед вызовом операции:

```
if precondition (y) then
  operation (y)
else
  ... Подходящие альтернативные действия...
end
```

(Для краткости этот пример использует неквалифицированный вызов, но, конечно же, большинство вызовов будут квалифицированными в форме: `z.operation (y)`.) Чтобы избежать теста `if...then...else`, следует убедиться, что из контекста следует выполнение предусловия:

```
...Некоторые инструкции, которые, среди прочего, гарантируют выполнение предусловия...
  check precondition (y) end
operation (y)
```

Желательно в этих случаях включать инструкцию `check`, дающую два преимущества: для читателя программного текста становится ясным, что предусловие не забыто, в случае же, если вывод о выполнении предусловия был ошибочным, при включенном мониторинге утверждений облегчается отладка. (Если вы забыли детали инструкции `check`, обратитесь к [лекции 11](#) курса "Основы объектно-ориентированного программирования".)

Такое использование предусловий, обеспечиваемое клиентом до вызова - либо путем явной проверки, либо как следствие выполнения других инструкций, - может быть названо априорной схемой: клиента просят выполнить некие мероприятия во избежание любых ошибок.

Препятствия на пути априорной схемы

Из-за простоты и ясности априорная схема, в принципе, идеальна. По трем причинам она не является универсально применимой:

- **A1** По соображениям эффективности непрактично в некоторых случаях проверять предусловия перед вызовом.
- **A2** Ограничения языка утверждений приводят к тому, что некоторые утверждения не могут быть выражены формально.
- **A3** Наконец, некоторые условия успешного выполнения зависят от внешних событий и не являются утверждениями.

Примером случая **A1** является решатель линейных уравнений. Функция, дающая решение системы линейных уравнений в форме $a \cdot x = b$, где a - матрица, x и b - векторы, может быть взята из соответствующего библиотечного класса `MATRIX`:

```
inverse (b: VECTOR): VECTOR
```

Решение системы находится так: $x := a.inverse(b)$. Единственное решение системы существует, только если матрица не "сингулярна". (Сингулярность здесь означает линейную зависимость уравнений, признаком которой

является равенство 0 определителя матрицы.) Можно было бы ввести проверку на сингулярность в предусловие `inverse`, требуя, чтобы вызовы клиента имели вид:

```
if a.singular then
    ...Подходящие действия для сингулярной матрицы...
else
    x := a.inverse (b)
end
```

Эта техника работает, но она неэффективна, поскольку определение сингулярности, по сути, дается тем же алгоритмом, что и нахождение решения системы. Так что одну и ту же работу придется выполнять дважды - сплошное расточительство.

Примеры **A2** включают случаи, когда предусловие представляет глобальное свойство всей структуры данных и не может быть выражено кванторами, например, граф не содержит циклов или список отсортирован. Наша нотация не поддерживает такие утверждения. Как отмечалось, в таких утверждениях мы можем использовать функции, но это может возвращать нас к случаю **A1** - вычисление функций в предусловиях может дорого стоить, столько же, как и решение задачи.

Наконец, ограничение **A3** возникает, когда невозможно проверить применимость операции без попытки выполнить ее, поскольку она взаимодействует с внешним миром - пользователем системы, линиями связи и так далее.

Апостериорная схема

Когда не работает априорная схема, иногда возможна простая апостериорная схема. Идея состоит в том, чтобы раньше выполнить операцию, а затем определить, как она прошла. Идея работает, если неудачи при выполнении операции не имеют печальных последствий прерывания вычислений.

Примером может служить по-прежнему решение системы линейных уравнений. При апостериорной схеме клиентский код может выглядеть так:

```
a.invert (b)
if a.inverted then
    x := a.inverse
else
    ... Подходящие действия для сингулярной матрицы...
end
```

Функция `inverse` заменена процедурой `invert`, более аккуратным именем которой было бы `attempt_to_invert`. При вызове процедуры вычисляется атрибут `inverted`, истинный или ложный в зависимости от того, найдено ли решение. В случае успеха решение становится доступным через атрибут `inverse`. (Инвариант класса может быть задан в виде: `inverted = (inverse /= Void)`.)

При таком подходе любая функция, выполнение которой может давать ошибку, преобразуется в процедуру, вычисляющую атрибут, характеризующий ошибку и атрибут, задающий результат, если он получен. Для экономии памяти вместо атрибута можно использовать однократную функцию (см. [лекцию 18](#) курса "Основы объектно-ориентированного программирования").

Это также работает и для операций, связанных с внешним миром. Например, функцию чтения входных данных "read" лучше представить процедурой, осуществляющей попытку чтения с двумя атрибутами - одним булевым, указывающим была ли операция успешной, и другим, дающим результат ввода в случае успеха.

Эта техника, как можно заметить, полностью согласуется с Принципом Разделения Команд и Запросов. Функция, которая может давать ошибку при выполнении, не должна представлять результат как побочный эффект. Лучше преобразовать ее в процедуру (команду) и иметь два запроса к атрибутам, вычисляемым командой. Все согласуется и с идеей представления объектов как машин, чье состояние изменяется командами и доступно через запросы.

Пример с функциями ввода типичен для случаев, когда эта схема дает преимущества. Большинство функций чтения, поддерживаемых языками программирования или встроенными библиотеками, имеют форму "next integer", "next string", требуя от клиента предоставления корректного формата данных. Неизбежно они приводят к ошибкам, когда ожидания не совпадают с реальностью. Предлагаемые процедуры чтения могут осуществлять попытку ввода без всяких предусловий, а затем уведомлять о ситуации, используя запросы, доступные клиенту.

Этот пример наглядно показывает правило, относящееся к "работе над ошибками": лучше избегать ошибок, чем исправлять их последствия.

Роль механизма исключений

Предыдущее обсуждение показало, что разбор случаев является основой для того, чтобы справиться с особыми ситуациями. Хотя априорная схема не всегда практична, часто можно проверять успешность результата после его получения.

Остаются, однако, ситуации, когда обе схемы не являются адекватными. Возможны три таких категории:

- Некоторые исключительные события - при вычислениях, запросах памяти - могут приводить к отказам аппаратуры или операционной системы, возбуждая исключения, и, если наша программная система их не перехватывает, то они приводят к вынужденному прерыванию ее выполнения. Зачастую это неприемлемо, особенно в жизненно важных системах, например медицинских.
- Некоторые особые ситуации, хотя и не обнаруживаемые предусловием, должны диагностироваться как можно раньше: операция не должна завершаться (для апостериорной проверки), поскольку это может привести к катастрофическим последствиям - нарушить целостность базы данных, подвергнуть опасности человеческую жизнь, как, например, в системах управления роботом.
- Наконец, разработчик может пожелать включить некую форму защиты от катастрофических последствий любых оставшихся ошибок в системе, поэтому использует механизм исключений для придания системе устойчивости.

В таких ситуациях механизм обработки исключений необходим, его детали рассмотрены в [лекции 12](#) курса "Основы объектно-ориентированного программирования".

Эволюция классов. Устаревшие классы

Мы пытаемся сделать наши классы совершенными. Все приемы, аккумулированные в этом обсуждении, направлены на эту цель - недостижимую, конечно, но полезную, как всякое стремление к идеалу.

К сожалению, (николько не собираясь обидеть читателя) все мы не являемся примером совершенства. Что делать, если после нескольких месяцев, а может быть, и лет работы, мы осознаем, что интерфейс класса мог бы быть спроектирован лучше? Не самая приятная дилемма, которую предстоит разрешить:

- В интересах текущих пользователей: это означает продолжать жить с устаревшим дизайном, чьи неприятные эффекты будут по прошествии времени становиться все более тяжкими. В индустрии это называется **восходящей совместимостью (upward compatibility)**. Совместимость, как много преступлений совершается во имя твоё, как писал Виктор Гюго (правда, говоря о свободе).

В соответствие с фольклором Unix одно из наиболее неприятных соглашений в инструментарии Make обеспокоило нескольких новых пользователей, обнаруживших его нездолго после выхода первой версии инструментария. Так как исправление вело к изменению языка, а неудобство показалось не слишком серьезным, то было принято решение оставить все как есть, дабы не тревожить сообщество пользователей. Следует сказать, что сообщество пользователей Make включало тогда одну или две дюжины людей из Bell Laboratories.

- В интересах будущих пользователей: приходится причинять вред нынешним пользователям, чей единственный грех в том, что они слишком рано доверились вам.

Иногда - но только иногда - есть другой выход. Мы вводим в нашу нотацию концепцию устаревших компонентов (obsolete features) или устаревших классов (obsolete classes). Вот пример подобной подпрограммы:

```
enter (i: INTEGER; v: G) is
    obsolete "Используйте put (value, index)"
    require
        correct_index (i)
    do
        put (v, i)
    ensure
        entry (i) = v
    end
```

Это реальный пример, хотя и неиспользуемый в настоящее время. Ранее при эволюции библиотек Base мы пришли к пониманию необходимости замены некоторых имен и соглашений (тогда еще принципы стиля, изложенные в [лекции 8](#), не были сформулированы). Предполагалось изменить имя `put` на `enter` и `item` на `entry` и, что еще хуже, изменить порядок следования аргументов для совместимости с компонентами других классов в библиотеке.

Приведенное выше объявление сглаживало эволюцию. Обратите внимание, как старый компонент `enter` получает новую реализацию, основанную на новом компоненте `put`. Следует использовать эту схему, когда компонент становится устаревшим для избежания двух конкурирующих реализаций с риском потери надежности и расширяемости.

Каковы следствия того, что компонента объявляется устаревшей? На практике они незначительны. Инструментарий окружения должен обнаружить это свойство и вывести соответствующее предупреждение, когда клиентская система использует класс. Компилятор, в частности, выведет сообщение, включающее строку, следующую за ключевым словом `obsolete`, такую как "Используйте `put (value, index)`" в нашем примере. Это все. Компонент, с другой стороны, продолжает нормально использоваться.

Подобный синтаксис позволяет объявить целый класс устаревшим.

Разработчикам клиентов это открывает дорогу к миграции. Сказав им, что компонент будет удален, вы поощряете их адаптацию, но не приставляете нож к горлу. Если изменение обосновано, как и должно быть, пользователи не должны обновлять свою часть сразу же, что было бы неприемлемо, но при переходе к новой версии они могут внести все изменения непосредственно. Дав им время, получите готовность принять нововведения.

На практике период миграции должен быть ограничен. При выпуске очередной версии (через месяцы или через год) следует удалить все устаревшие классы и компоненты. В противном случае предупреждения об устарелости никто не будет принимать всерьез. Вот почему упоминавшийся пример с устаревшими именами `enter` и `entry` уже не является текущим. Но в свое время он сыграл свою положительную роль, сделав счастливым не одного разработчика.

Старение компонентов и классов решает одну специфическую проблему. Когда в проекте обнаруживается изъян, единственно разумный подход его коррекции состоит в приложении усилий, помогающих пользователям совершить переход. Ни переопределение в потомках, ни объявление проекта устаревшим не заменят необходимости исправления обнаруженных ошибок существующего ПО. Но старение - это прекрасный механизм в ситуациях, когда существующий проект, отличный во всех отношениях, перестает соответствовать современности. Хотя в старом проекте нет серьезных пороков, но сейчас все можно сделать лучше: упростить интерфейс, обеспечить лучшую согласованность с остальным ПО, обеспечить взаимодействие с другими продуктами, применить лучшие соглашения по наименованию. В таких ситуациях объявление классов и компонентов устаревшими - прекрасный способ защитить вложения текущих пользователей, пока они не пройдут свою часть пути к светлому будущему.

Документирование класса и системы

Владея совершенными методами проектирования интерфейса классов, можно построить множество великолепных классов. Для достижения успеха, которого они заслуживают, необходима еще хорошая документация. Мы уже видели основное средство документирования - краткую форму - и ее вариант - плоскую краткую форму. Давайте обобщим их использование и исследуем дополняющие механизмы, работающие со всей системой, а не с отдельными классами.

Упоминание краткой формы в этом обсуждении будет также относиться и к плоской краткой форме. Разница между ними, как помните, состоит в том, что плоская краткая форма учитывает и наследуемые компоненты, в то время как просто краткая форма ссылается только на компоненты, непосредственно объявленные в классе. В большинстве случаев авторам клиентских модулей необходима плоская краткая форма.

Показ интерфейса

Краткая форма непосредственно применяет правило Скрытия Информации, удаляя закрытую от клиента информацию, включающую:

- любой незэкспортируемый компонент и все, что с ним делается (например, утверждения, относящиеся к компоненту);
- любую реализацию подпрограммы, заданную предложением `do ...`

То, что остается, представляет абстрактную информацию о классе, обеспечивая авторов клиентских модулей - настоящих и будущих - описанием, независимым от реализации, необходимым для эффективного использования класса.

Напомним, что основной целью является абстракция, а не защита. Мы не намереваемся скрыть от авторов клиентов закрытые свойства классов, мы желаем лишь освободить их от лишней информации. Отделяя функции от реализации, скрытие реализации должно рассматриваться как помочь авторам клиентов, а не как помеха в их работе.

Краткая форма избегает техники, поддерживаемой в отсутствие утверждений такими языками, как Ada, Modula-2 and Java, написание раздельных и частично избыточных частей реализации и интерфейса. Такое разделение всегда чревато ошибками при эволюции классов. Как это всегда бывает в программной инженерии, повторение ведет к несогласованности. Вместо этого все помещается в один класс, а специальный инструментарий извлекает оттуда абстрактную информацию.

В начале этой книги был введен принцип, согласно которому ПО должно быть самодокументированным, насколько это возможно. Фундаментальную роль в этом играет здравый выбор утверждений. Просмотрев примеры этой лекции и построив, хотя бы мысленно, краткие формы, можно получить достаточно явные свидетельства этому.

Чтобы краткая форма давала наилучшие результаты, следует при написании классов всегда применять следующий принцип:

Принцип Документации

Пытайтесь писать программный текст так, чтобы он включал все элементы, необходимые для документирования, обнаруживаемые автоматически на разных уровнях абстракции.

Это простая трансляция общего Принципа Самодокументирования в практическое правило, которое следует применять ежедневно и ежечасно в своей работе. В частности, крайне важны:

- хорошо спроектированные предусловия, постусловия и инварианты;
- тщательный выбор имен как для классов, так и для их компонентов;
- информативное индексирование предложений программного текста.

Лекция 8, посвященная стилю, подробно рассмотрит два последних пункта.

Документирование на уровне системы

Инструментарий **short** и **flat-short**, разработанный в соответствии с правилами этой книги (утверждения, Проектирование по Контракту, скрытие информации, четкие соглашения именования, заголовочные комментарии и так далее) применяет принцип Документации на уровне модуля. Но есть необходимость для документации более высокого уровня - документации на уровне всей системы или одной из ее подсистем. Но здесь текстуального вывода, хотя и необходимого, явно недостаточно. Для того чтобы охватить организацию возможно сложной системы, полезно графическое описание.

Инструментарий Case из окружения ISE, основанный на концепциях BON (Business Object Notation), обеспечивает такое видение. На [рис.5.15](#) показана сессия, предназначенная для реверс-инженерии (reverse-engineering) библиотек Base.

Хотя детализация инструментария выходит за рамки этой книги (см. [М 1995с]), можно отметить, что это средство поддерживает анализ больших систем, позволяет изменять масштаб диаграммы, поддерживает возможность фокусироваться на кластерах (подсистемах), позволяет объединять графические снимки с текстовой информацией о подсистемах.

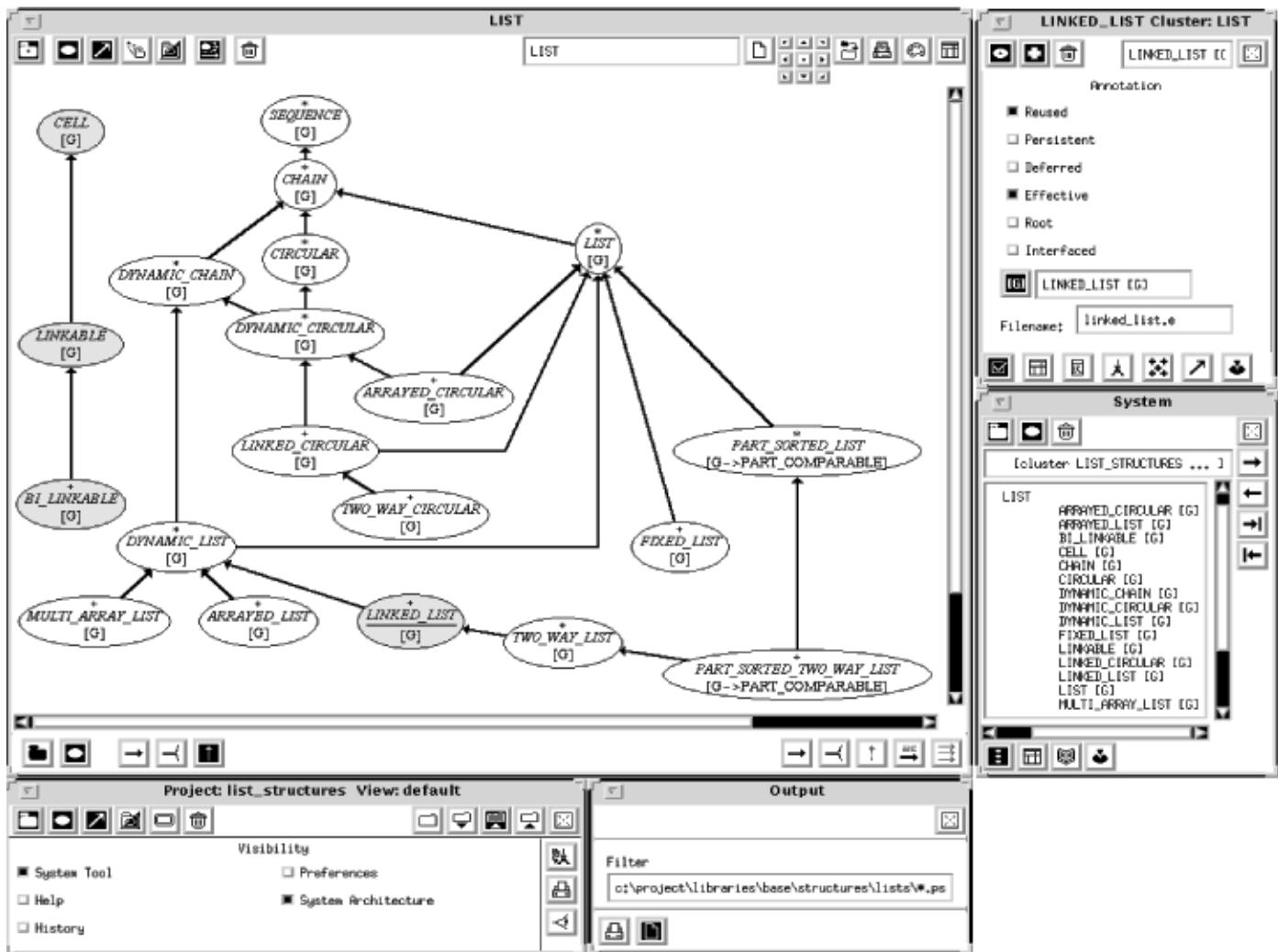


Рис. 5.15. Диаграмма архитектуры системы

Все эти средства, являясь приложением Принципа Документирования, приближают нас к идеалу Самодокументирования. Все это достигается благодаря тщательно спроектированной нотации и современному окружению.

Ключевые концепции

- Класс должен быть известен своим интерфейсом, специфицируя предлагаемые службы независимо от их реализации.
- Разработчики класса должны стараться предоставить простые согласованные интерфейсы.
- Одна из ключевых проблем при проектировании класса состоит в правильном разделении экспортруемых и закрытых компонентов.
- Проектирование повторно используемых модулей не является первоочередной задачей - интерфейс должен стабилизироваться после некоторого периода использования. Если этого не происходит, то это свидетельствует об изъяне в проекте. Механизм устаревших классов и компонентов делает возможным сладить переход к лучшей версии проекта.
- Зачастую полезно рассматривать некоторые структуры данных как активные машины с внутренним состоянием, запоминаемым между вызовами компонентов.
- Правильное использование утверждений - предусловий, постусловий, инвариантов - является основой

документирования интерфейсов.

- Для разбора особых случаев лучше применять стандартные управляющие структуры, применяя либо априорную, либо апостериорную схему. Механизм дисциплинированных исключений остается необходимым в тех случаях, когда выполнение должно быть прервано из-за потенциальных угроз, связанных с некорректным выполнением операции.

Библиографические замечания

В работах Парнаса ([Parnas 1972], [Parnas 1972a]) вводится много похожих идей по проектированию интерфейса.

Различие операндов и опций и результирующий принцип взяты из работы [M1982a].

Понятие активной структуры данных поддерживается в некоторых языках программирования управляющими абстракциями, называемыми итераторами. Итератор представляет механизм, определенный совместно со структурой данных, описывающий, как применять некую операцию к каждому элементу структуры. Например, итератор, ассоциированный со списком, описывает циклический механизм прохода по списку, применяя данную операцию к каждому элементу списка, итератор дерева задает некую стратегию обхода дерева. В работах [Liskov 1981], [Liskov 1986] содержится подробное обсуждение концепции итераторов, доступных в языке программирования CLU. В объектной технологии итераторы могут быть определены в классах, не встраивая их в виде конструкций языка программирования [M 1994a].

Пример адаптивной реализации комплексных чисел взят из [M1979], где он описан на Simula.

Грамотное программирование (Literate programming) [Knuth 1984] утверждает, как и данная лекция, что программы должны содержать свою документацию. Его концепции, однако, существенно отличаются от концепций объектной технологии. Одно из упражнений предлагает сравнить два подхода.

Статьи Джеймса Мак-Кима и Ричарда Билака [Bielak 1993], [McKim 1992a] дают полезные советы по проектированию интерфейса, основанные на понятии Проектирования по Контракту.

Упражнения

У5.1 Функция с побочным эффектом

Пример управления памятью на уровне компонентов (см. [лекцию 9](#) курса "Основы объектно-ориентированного программирования") для связных списков имеет функцию `fresh`, вызывающую процедуру `getframe` для стеков, следовательно, имеющую побочный эффект. Обсудите, является ли это приемлемым.

У5.2 Операнды и опции

Исследуйте класс или доступную библиотеку и определите, какие аргументы подпрограмм являются операндами и какие - опциями.

У5.3 Возможные аргументы

Некоторые языки, такие как Ada, предполагают в подпрограммах возможные аргументы, каждый с ассоциированным ключевым именем. Если ключевое имя не включено, аргумент может быть установлен по умолчанию. Обсудите, какие преимущества принципа Операндов эта техника поддерживает, а какие определенно нарушаются.

У5.4 Число элементов как функция

Адаптируйте определение класса `LINKED_LIST [G]` так, чтобы `count` стал функцией, а не атрибутом, оставив неизменным интерфейс класса.

У5.5 Поиск в связных списках

Напишите процедуру `search (x: G)` для класса `LINKED_LIST`, разыскивающую следующее вхождение `x`.

У5.6 Теоремы в инварианте

Докажите истинность трех утверждений из первой части инварианта класса `LINKED_LIST`, отмеченных как теоремы (см. [лекцию 5](#)).

У5.7 Двунаправленные списки

Напишите класс, задающий двунаправленные списки с интерфейсом `LINKED_LIST`, но более эффективной реализацией таких операций, как `back`, `go` и `finish`.

У5.8 Альтернативный проект связного списка

Предложите вариант класса для связного списка, использующий соглашение о том, что для пустого списка одновременно выполняются `after` и `before`. (В первом издании книги использовался этот прием.) Оцените оба подхода.

У5.9 Вставка в связный список

Глядя на `remove`, напишите процедуры `put_left` и `put_right` для вставки элементов слева и справа от позиции курсора.

У5.10 Циклические списки

Объясните, почему класс `LINKED_LIST` не может использоваться для циклических списков. (Подсказка: покажите, что утверждения будут нарушаться.) Напишите класс `CIRCULAR_LINKED`, реализующий циклические списки.

У5.11 Функции ввода, свободные от побочных эффектов

Спроектируйте класс, описывающий входные файлы с операциями ввода без любых функций с побочным эффектом. Достаточно написать только интерфейс класса без предложений `do`, но с заголовками подпрограмм и всеми подходящими утверждениями.

У5.12 Документация

Обсудите, расширив и переопределив, принцип Самодокументирования и его различные разработки в этой книге, рассматривая различные виды документации. Проанализируйте, какие стили документации подходят при определенных обстоятельствах и различных уровнях абстракции.

У5.13 Самодокументированное ПО

Подход к самодокументированию, защищаемый в этой книге, предполагает лаконичность и не поддерживает долгих объяснений проектных решений. Стиль "грамотного программирования" Кнута комбинирует методы программирования и текстовой обработки так, чтобы полная документация проекта и его история содержались в одном документе. Метод основан на классической парадигме разработки проекта сверху вниз. Отталкиваясь от работы Кнута, обсудите, как его метод может быть транспортирован при объектном подходе.

Основы объектно-ориентированного проектирования

6. Лекция: Используйте наследование правильно

Изучение технических деталей наследования и связанных механизмов в лекциях 7-18 курса "Основы объектно-ориентированного программирования" не означает еще автоматического владения всеми методологическими следствиями. Из всех проблем ОО-технологии ни одна не вызывает столько обсуждений и вопросов о том, как и когда использовать наследование. В этой лекции мы продвинемся в понимании смысла наследования не ради теории, а ради наилучшего применения наследования в наших проектах. В частности, мы попытаемся понять, чем наследование отличается от другого межмодульного отношения, его брата и соперника, - отношения встраивания, чаще всего называемого клиентским отношением. Мы исследуем, когда следует использовать одно, когда - другое, а когда оба отношения являются приемлемыми. Мы установим основной критерий использования наследования, по пути выяснив, в каких случаях его использование ошибочно. Мы сможем создать классификацию различных случаев легитимного использования, часть из которых принимается безоговорочно (наследование подтипов), другие - более спорны. На этом пути мы попытаемся освоить опыт таксономии или систематики, привнесенный из других научных дисциплин.

Как не следует использовать наследование

Для выработки методологического принципа часто полезно - как показано во многих обсуждениях этой книги - вначале понять, как **не следует делать** вещи. Понимание того, "что такое плохо", позволяет осознать, "что такое хорошо". Если постоянно тепло, то грушевое дерево не зацветет, ему необходима встряска зимним морозом - тогда оно расцветет весной.

Вот и встряска для нас, любезно предоставленная широко известным во всем мире вузовским учебником, выдержавшим 4 издания, по которому программной инженерии учатся многие студенты. Вот начало текста по поводу множественного наследования:

Множественное наследование позволяет нескольким объектам выступать в роли базовых и поддерживается во многих языках (ссылка на первое издание этой книги [M1988]).

Помимо неудачного использования "объектов", вместо классов начало кажется весьма подозрительным. Цитата продолжается:

Характеристики нескольких различных классов объектов

(классы, уже хорошо!)

могут комбинироваться, создавая новый объект.

(Нет, опять неудача.) Далее следует пример множественного наследования:

например, пусть мы имеем класс объектов CAR, инкапсулирующий информацию об автомобиле, и класс PERSON, инкапсулирующий информацию о человеке. Мы можем использовать их для определения

(неужели оправдаются наши наихудшие подозрения?)

нового класса CAR-OWNER, комбинирующего атрибуты CAR и PERSON.

(Они оправдались.) Нас приглашают рассматривать каждый объект CAR-OWNER не только как персону, но и как автомобиль. Для каждого, кто изучал наследование даже на элементарном уровне, это станет сюрпризом.

Несомненно, вы понимаете, что второе отношение является клиентским, а не наследованием, владелец автомобиля **является (is)** персоной, но **имеет (has)** автомобиль.

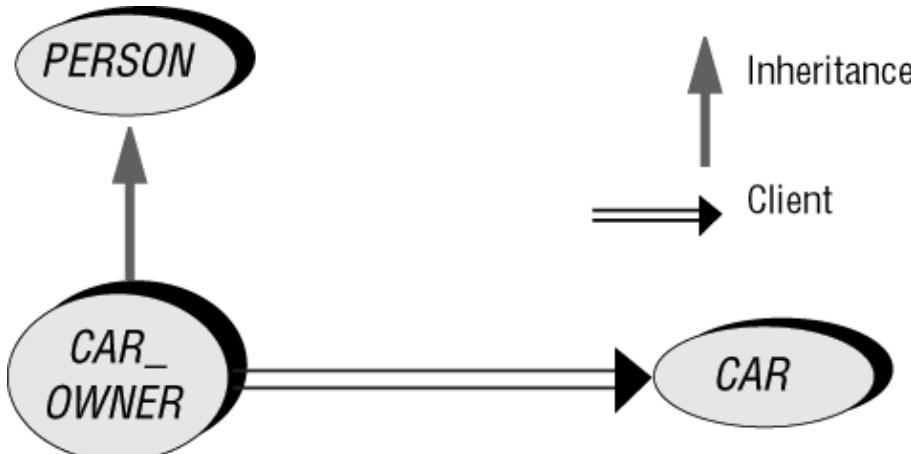


Рис. 6.1. Походящая модель

В формальной записи:

```
class CAR_OWNER inherit  
    PERSON
```

```
feature
    my_car: CAR
    ...
end
```

В цитируемом тексте обе связи используют отношение наследования. Наиболее интересный пассаж в этом обсуждении следует далее, когда автор советует читателям рассматривать наследование с осторожностью:

Адаптация через наследование имеет тенденцию приводить к избыточной функциональности, что может делать компоненты неэффективными и громоздкими.

И в самом деле, громоздкими - подумаем о владельце машины с крышей, мотором, карбюратором, не говоря уже о четырех колесах и запаске. Эта картина возникла, возможно, под влиянием образчика австралийского юмора, где владелец машины выглядит так, как если бы он **являлся** своим автомобилем.



Рис. 6.2. Рисунок Джекфа Хокинга (голова его похожа на его же авто с открытыми дверцами)

Наследование не является тривиальной концепцией, так что мы можем забыть и простить автора процитированного отрывка, но сам пример имеет важную практическую пользу - он помог нам стать немного умнее и напомнил базисное правило наследования:

Правило: Наследование "Is-a" (**является**)

Не делайте класс В наследником класса А, если нельзя привести аргументы в защиту того, что каждый экземпляр В **является** также экземпляром А.

Другими словами, мы должны быть способными убеждать, что каждый В **is an A** (отсюда имя: "is-a").

Вопреки первому впечатлению, это слабое, а не строгое правило, и вот почему:

- Обратите внимание на фразу "привести аргументы". Мы не требуем **доказательства** того, что каждый В всегда **является** А. В большинстве случаев мы оставляем пространство для дискуссии. Верно ли, что каждый "сберегательный счет" (savings account) является "текущим счетом" (checking account)? Здесь нет абсолютного ответа - все зависит от политики банка и вашего анализа свойств различных видов счетов. Возможно, вы решите сделать класс SAVINGS_ACCOUNT наследником BANK_ACCOUNT или поместить его где-либо еще в структуре наследования. Разумные люди могут все же не согласиться с результатом. Это нестращно, важно лишь, чтобы был случай, для которого ваши аргументы способны устоять. В нашем контрпримере: нет ситуации, при которой аргументы в пользу того, что CAR_OWNER является CAR, могли бы устоять.
- Наш взгляд на то, что означает отношение "**является**", будет довольно либеральным. Он не будет, например, препятствовать **наследованию реализации** - форме наследования, многими считающейся подозрительной.

Эти наблюдения показывают как полезность, так и ограниченность правила "Is-a". Оно полезно как **отрицательное** правило, позволяя обнаружить и отвергнуть неподходящее использование наследования. Но как положительное правило оно недостаточно - не все, что проходит тест, заданный правилом, является подходящим случаем наследования.

Покупать или наследовать

Основное правило выбора между двумя возможными межмодульными отношениями - клиентом и наследованием - обманчиво просто: клиент **имеет**, наследование **является**. Почему же тогда выбор столь непрост?

Иметь и быть (To have and to be)

Причина в том, что **иметь** не всегда означает **быть**, но во многих случаях **быть** означает **иметь**.

Нет, это не дешевая попытка экзистенциалистской философии - это отражение трудностей системного моделирования. Иллюстрацией первой половины высказывания опять-таки может служить наш пример: владелец автомобиля **имеет** машину, но нет никаких причин утверждать, что он **является** машиной.

Что можно сказать об обратной ситуации? Рассмотрим простое предложение о двух объектах из обычной программистской жизни:

Каждый инженер-программист является инженером. [A]

Очевидно, это хороший пример отношения **является**. Кажется, трудно думать по-другому - здесь ясно видно, что мы имеем дело со случаем быть, а не иметь. Но перефразируем утверждение:

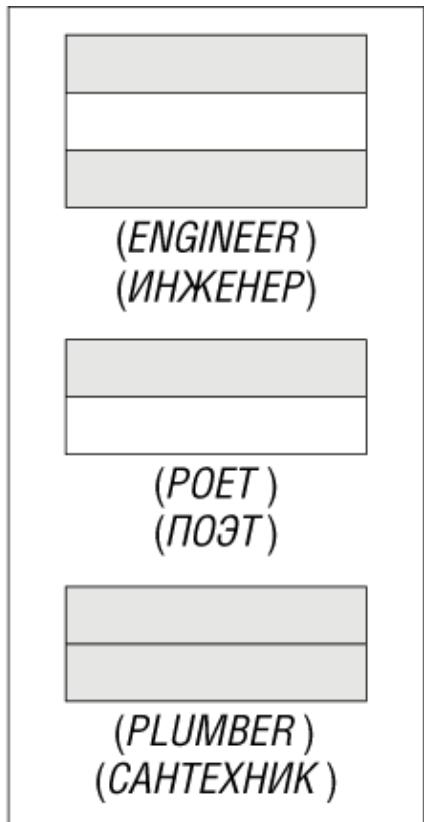
В каждом инженере-программисте заключена частица инженера. [B]

Представим его теперь так:

Каждый инженер-программист имеет инженерную составляющую. [C]

Трюкчество - да, но все же [C] в основе не отличается от исходного высказывания [A]! Что отсюда следует: слегка изменив точку зрения, можно представить свойство **является** как **имеет**.

Рассмотрим структуру нашего объекта, как это делалось в предыдущих лекциях:



(SOFTWARE_ENGINEER)
(ИНЖЕНЕР-ПРОГРАММИСТ)

Рис. 6.3. Объект "инженер-программист" как агрегат

Экземпляр SOFTWARE_ENGINEER показывает различные аспекты деятельности инженера-программиста. Вместо представления типа этого объекта как развернутого, можно рассматривать представление в терминах ссылок:

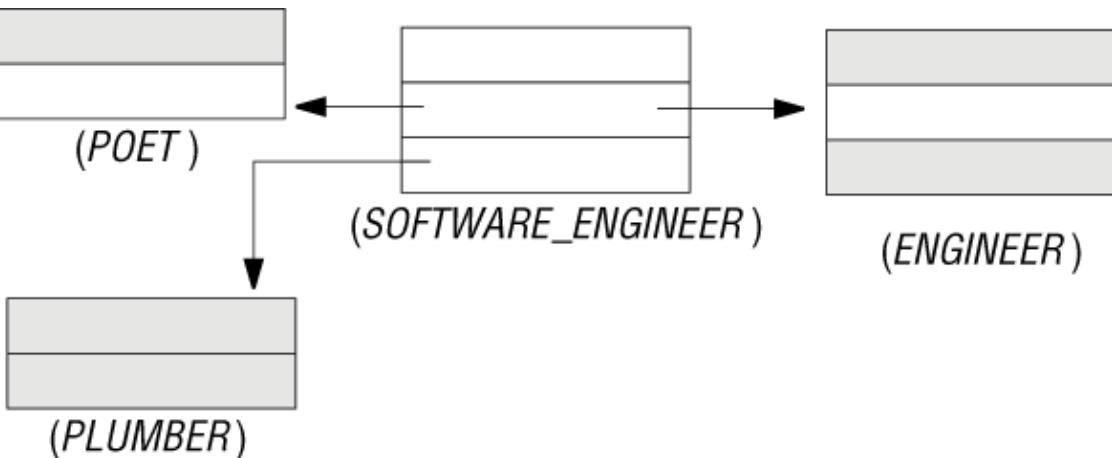


Рис. 6.4. Другое возможное представление

Рассматривайте оба представления как способы визуализации ситуации, ничего более. Оба они исходят, однако, из отношения клиента **имеет**, интерпретации, в которой каждый инженер-программист **несет в себе** инженера как одну из своих ипостасей, что полностью согласуется с названием профессии. Одновременно в нем может быть сидит частица поэта и (или) сантехника. Подобные наблюдения могут быть сделаны для любого похожего отношения "**является**".

Вот почему проблема выбора между клиентом и наследованием не тривиальна - когда отношение "является" законно, то справедлив переход к отношению "иметь".

Обратное неверно. Это наблюдение предохраняет от простых ошибок, очевидно для всякого, кто понимает базисные концепции и, вероятно, объяснимо даже для авторов учебника. Но когда применимо отношение "является", то у него сразу же появляется соперник. Так что два компетентных специалиста могут не прийти к одному решению: один выберет наследование, другой предпочитет клиентское отношение.

К счастью, существуют два критерия, помогающих в таких спорах. Иногда они могут не приводить к единственному решению. Но в большинстве практических случаев они без всяких колебаний указывают, какое из отношений является правильным.

Один из этих критериев предпочитает наследование, другой - клиента.

Правило изменений

Первое наблюдение состоит в том, что клиентское отношение обычно допускает изменения, а наследование - нет. Сейчас мы должны с осторожностью обходиться с глаголами "**быть**" и "**иметь**", помогающими нам до сих пор характеризовать природу двух отношений между программными модулями. Правила для программ, как всегда, более точные, чем их двойники из обычного мира.

Одним из определяющих свойств наследования является то, что это отношение между классами, а не между объектами. Мы интерпретировали свойство "Класс В наследует от класса А" как "каждый объект В является объектом А". Следует помнить, что это свойство не в силах изменить никакой объект - только класс может достичь такого результата. Свойство характеризует ПО, но не его отдельное выполнение.

Для отношения клиента ограничения слабее. Если объект типа В имеет компонент типа А (либо подобъект, либо ссылку) вполне возможно изменить этот компонент - ограничением служит лишь система типов.

Заданное отношение между объектами может быть результатом как отношения наследования, так и клиентского отношения между классами. Важно различать, допускаются изменения или нет. Например, наша воображаемая структура объекта могла быть результатом отношения наследования между соответствующими классами:

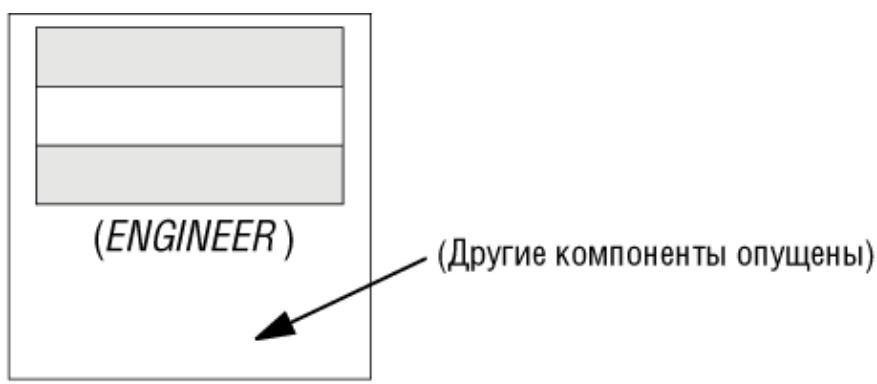


Рис. 6.5. Объект и подобъект

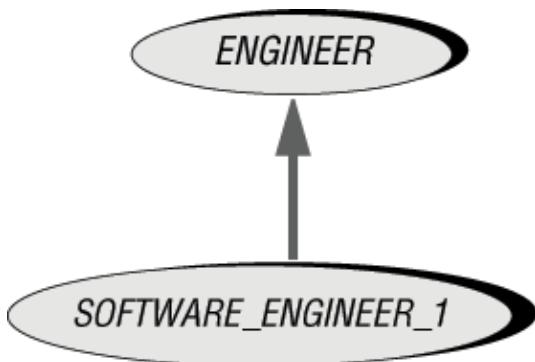
```
class SOFTWARE_ENGINEER_1 inherit
```

ENGINEER

feature

...

end

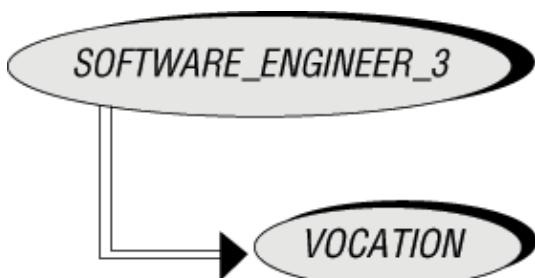


Она могла быть точно так же получена через отношение клиента:

```
class SOFTWARE_ENGINEER_2 feature
    the_engineer_in_me: ENGINEER
    ...
end
```

Фактически оно могло быть и таким:

```
class SOFTWARE_ENGINEER_3 feature
    the_truly_important_part_of_me: VOCATION
    ...
end
```



Для удовлетворения ограничений системы типов класс ENGINEER должен быть потомком класса VOCATION.

Строго говоря, последние два варианта представляют слегка отличную ситуацию. Если предположить, что ни один из заданных классов не является развернутым, то вместо подобъектов в последних двух случаях объекты "software engineer" будут содержать ссылки на объекты "engineer", как показано на [рис.6.4](#). Введение ссылок, однако, не оказывается на сути нашего обсуждения.

Поскольку отношение наследования задается между классами, то, приняв первое определение класса, динамически будет невозможно изменить отношение между объектами: инженер всегда останется инженером.

Но для других двух определений модификация возможна: процедура класса "software engineer" может присвоить новое значение полю соответствующего объекта (полю `the_engineer_in_me` или `the_truly_important_part_of_me`). В случае класса SOFTWARE_ENGINEER_2 новое значение должно быть типа ENGINEER или совместимого с ним; для класса SOFTWARE_ENGINEER_3 оно может быть любого типа, совместимого с VOCATION (**Профессия**). Такая программа способна моделировать инженера-программиста, который после многих лет притязаний стать настоящим инженером, наконец, покончил с этой составляющей своей личности и решил стать поэтом или сантехником. ("Не надо оваций. Графа Монте-Кристо из меня не вышло. Придется переквалифицироваться в управдомы".)

Это приводит к нашему первому критерию:

Правило изменений

Не используйте наследование для описания отношения, воспринимаемого как "**является**", если компоненты соответствующего объекта могут изменять тип в период выполнения.

Используйте наследование только при условии, что отношение между объектами постоянно. В других случаях используйте отношение клиента.

По настоящему интересный случай имеет место для SOFTWARE_ENGINEER_3. Для SOFTWARE_ENGINEER_2 можно заменить инженерный компонент на другой, но того же инженерного типа. Но для SOFTWARE_ENGINEER_3 класс VOCATION может быть более высокого уровня, вероятнее всего, отложенным, так что атрибут может (благодаря полиморфизму) представлять объекты многих возможных типов, согласованных с VOCATION.

Это также означает, что, хотя решение использует клиента как первичное отношение, но на практике в своей окончательной форме оно часто использует дополнительное отношение наследования. Это станет особенно ясно, когда мы приедем к понятию описателя (handle).

Правило полиморфизма

Займемся теперь критерием, требующим наследования и исключающим клиента. Этот критерий прост: он основан на полиморфизме. При изучении наследования мы видели, что для объявления в форме:

x: C

х обозначает в период выполнения (предполагая, что класс С не является развернутым) полиморфную ссылку. Другими словами, х может быть присоединен как к прямому экземпляру С, так и к экземпляру потомков С. Это свойство представляет ключевой вклад в мощность и гибкость ОО-метода, особенно из-за следствий - возможности определения полиморфных структур данных, подобных LIST [C], которые могут содержать экземпляры любого из потомков С.

В нашем примере это означает, что, выбрав решение SOFTWARE_ENGINEER_1 - форму, в которой класс является наследником ENGINEER, клиент может объявить сущность:

eng: ENGINEER

Эта сущность в период выполнения может быть присоединена к объекту типа SOFTWARE_ENGINEER_1. Можно иметь список инженеров, базу данных, включающую инженеров-механиков, инженеров-химиков наряду с программистами.

Методологическое напоминание: использование слов, не относящихся к программе, облегчает понимание концепций, но это нужно делать с осторожностью, особенно для антропологических примеров. Объекты нашего интереса являются программными объектами, поэтому, когда мы говорим "a software engineer", то это фактически означает экземпляр класса SOFTWARE_ENGINEER_1.

Такие полиморфные эффекты требуют наследования: в случае SOFTWARE_ENGINEER_2 или SOFTWARE_ENGINEER_3 сущности или структуры данных типа ENGINEER не могут непосредственно означать объекты "software engineer".

Обобщая это наблюдение, характерное не только для этого примера, приходим к правилу, дополняющему правило изменений:

Правило полиморфизма

Наследование подходит для описания отношения, воспринимаемого как "**является**", если для сущностей может возникнуть потребность присоединения к объектам различных типов.

Резюме

Хотя оно и не вводит новых концепций, следующее правило удобно как итог обсуждения критериев, высказывающихся за и против наследования.

Выбор между клиентом и наследованием

При решении, как выразить зависимость между классами В и А, применяйте следующие критерии:

- **CI1** Если каждый экземпляр В изначально имеет компонент типа А, но этот компонент в период выполнения может нуждаться в замене объектом другого типа, сделайте В клиентом А.
- **CI2** Если необходимо, чтобы сущности типа А обозначали объекты типа В или в полиморфных структурах, содержащих объекты типа А, некоторые могли быть типа В, сделайте В наследником А.

Приложение: техника описателей

Приведем пример, использующий предшествующее правило. Он приводит к широко применимому образцу проектирования - **описателям (handles)**.

Первый проект библиотеки Vision для платформенно-независимой графики столкнулся с общей проблемой, как

учитывать зависимость от платформы. Первое решение использовало множественное наследование следующим образом: типичный класс, задающий например окна, имел двух родителей - одного, описывающего общие свойства, не зависящие от платформы, другого, учитывающего специфику данной платформы.

```
class WINDOW inherit
    GENERAL_WINDOW
    PLATFORM_WINDOW
feature
    ...
end
```

Класс GENERAL_WINDOW и ему подобные, такие как GENERAL_BUTTON, являются отложенными: они выражают все, что может быть сказано о соответствующих графических объектах и применимых операциях без ссылки на особенности графической платформы. Классы, такие как PLATFORM_WINDOW, обеспечивают связь с графической платформой, такой как Windows, OS/2 Presentation-Manager или Unix Motif; они дают доступ к механизмам, специфическим для данной платформы (встраиваемым в библиотеки, такие как WEL или MEL).

Класс, такой как WINDOW, будет комбинировать свойства родителей, реализуя отложенные компоненты GENERAL_WINDOW механизмами, обеспечиваемыми PLATFORM_WINDOW.

Класс PLATFORM_WINDOW (как и другие подобные классы) должен присутствовать в нескольких вариантах - по одному на каждую платформу. Эти идентично именуемые классы будут храниться в различных каталогах; инструментарий Ace при компиляции выберет подходящий.

Это решение работает, но его недостаток в том, что понятие WINDOW становится тесно связанным с выбранной платформой. Перефразируя недавний комментарий о наследовании, можно сказать: окно, став однажды окном Motif, всегда им и останется. Это не слишком печально, поскольку трудно вообразить, что однажды, достигнув почтенного возраста, окно Unix вдруг решит стать окном OS/2. Картина становится менее абсурдной при расширении определения платформы - при включении форматов, таких как Postscript или HTML; графический объект может изменять представление, становясь то документом печати, то Web-документом.

Попытаемся выразить тесную связь между GUI-объектами и поддерживающим инструментарием, используя вместо наследования клиентское отношение. Наследственная связь останется между WINDOW и GENERAL_WINDOW, но зависимость от платформы будет представлена клиентской связью с классом TOOLKIT, представляющим необходимый инструментарий. Как это выглядит, показано на [рис. 6.6](#):

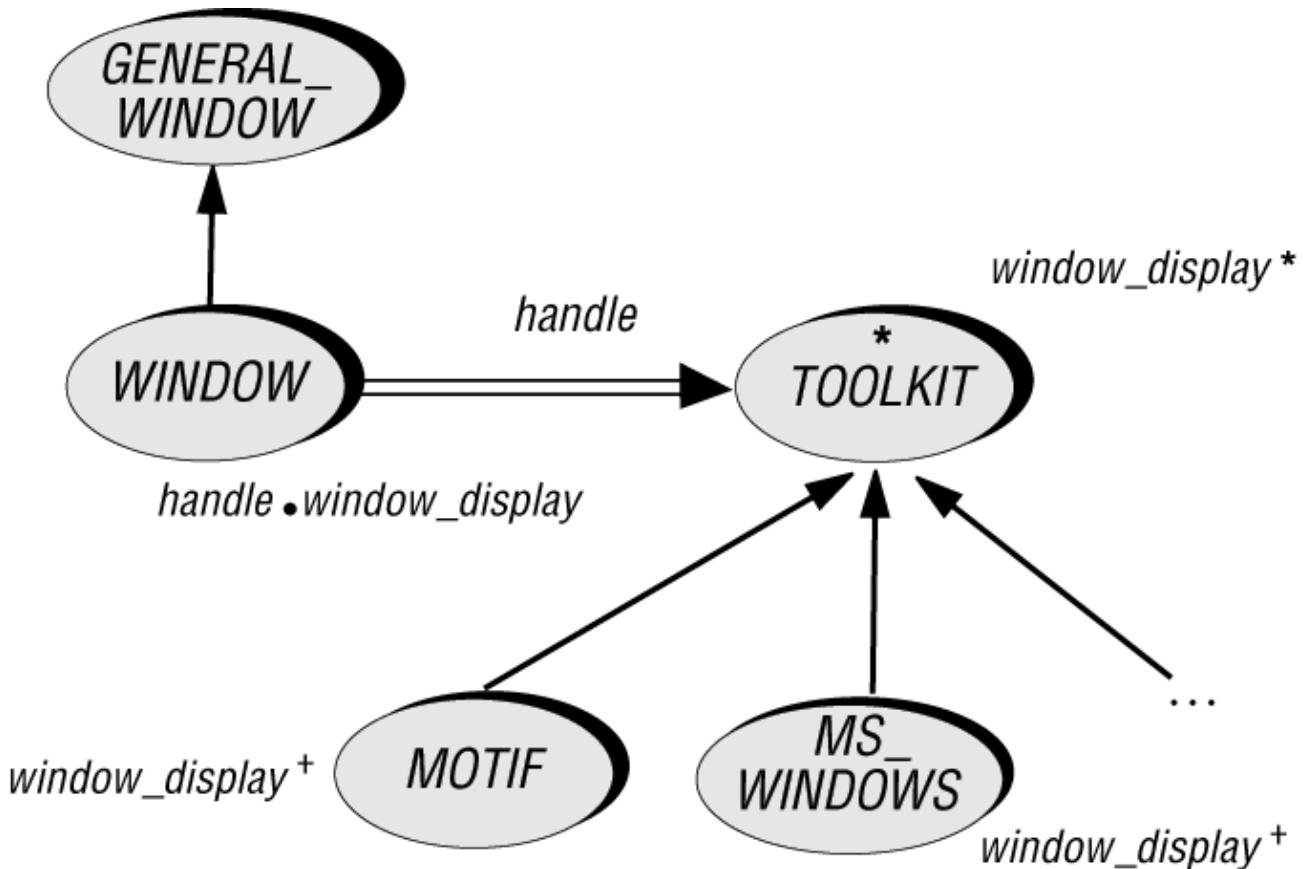


Рис. 6.6. Комбинация отношений наследования и клиента

Интересный аспект этого решения в том, что понятие инструментария (toolkit) становится полноценной абстракцией, представляющей отложенный класс TOOLKIT. Каждый специфический инструментарий, такой как MOTIF или MS_WINDOWS представляется эффективным потомком класса TOOLKIT.

Вот как это работает. Каждый класс, описывающий графические объекты, такие как WINDOW, имеет атрибут,

обеспечивающий доступ к соответствующей платформе:

```
handle: TOOLKIT
```

Так появляется поле для каждого экземпляра класса. Описатель может быть изменен:

```
set_handle (new: TOOLKIT) is
    -- Создать новый описатель new для этого объекта
do
    handle := new
end
```

Типичная операция, наследуемая от GENERAL_WINDOW в отложенной форме, реализуется через вызовы платформенного механизма:

```
display is
    -- Выводит окно на экран
do
    handle.window_display (Current)
end
```

Через описатель графический объект запрашивает платформу, требуя выполнить нужную операцию. Компонент, такой как window_display, в классе TOOLKIT является отложенным, но реализуется его различными потомками, такими как MOTIF.

Заметьте, было бы неверным, глядя на этот пример, прийти к заключению: "Ага! Вот ситуация, при которой наследование было избыточным, и данная версия призвана избежать его". Начальная версия вовсе не была ошибочной, она работает довольно хорошо, но менее гибкая, чем вторая. И в основе второй версии лежит наследование, полиморфизм и динамическое связывание, комбинируемое с клиентским отношением. Без иерархии наследования с корнем TOOLKIT, полиморфной сущности handle и динамического связывания компонентов, таких как window_display, все бы это не работало. Вовсе не отвергая наследование, эта техника демонстрирует его более сложную форму.

Техника описателей широко применима к разработке библиотек, поддерживающих совместимость платформ. Помимо графической библиотеки VisiOn мы применяли ее к библиотеке баз данных Store, где понятие платформы связывается с основанными на SQL различными интерфейсами реляционных баз данных, таких как Oracle, Ingres, Sybase и ODBC.

Таксомания

Для каждой из категорий наследования, вводимых в этой лекции, наследник не тривиален - он либо переобъявляет (переопределяет или реализует) некоторые наследуемые компоненты, либо вводит собственные компоненты, либо делает добавления в инвариант класса. Конечно, он может делать все это одновременно. Результатом является следующее правило, фактически являющееся следствием правила Наследования, которое появится в этой лекции чуть позднее:

Правило Таксомании (ограничения таксономии)

Каждый наследник обязан ввести новый компонент, или переобъявить наследуемый компонент, или добавить предложение в инвариант класса.

Это правило призвано бороться с человеческой слабостью, свойственной новичкам, овладевшим ОО-методом, - с энтузиазмом они стараются применить таксономическое деление (отсюда и имя правила, как сокращение "мания таксономии"). В результате появляется сверхсложенная структура иерархии наследования. Таксономия и наследование являются способом, призванным помочь справиться со сложностью, но не порождать ее. Добавление бесполезных уровней классификации означает нанесение ущерба самому себе.

Как часто бывает в таких случаях, вернуться к правильному видению - и возвратить новичков на греческую землю - помогает обращение к АТД. Класс является реализацией АТД, частичной или полной. Различные классы, в частности родитель и его наследники, должны описывать различные АТД. Поскольку АТД полностью характеризуется применимыми компонентами и их свойствами, охватываемые утверждениями класса, новый класс должен изменять наследуемые компоненты, вводить новые компоненты и утверждения. Так как предусловие или постусловие можно изменить только при переопределении компонента, то последний случай означает добавление предложения инварианта класса (**наследование с ограничением (restriction inheritance)**) - одна из категорий в нашей таксономии).

Иногда можно найти оправдание случаю таксономии: не приносящий ничего нового класс вводится на том основании, что наследник описывает важный частный случай, а пока подстилается соломка, предполагая в будущем возможность внесения изменений. Это может быть особенно разумным, если такой класс существует в естественной иерархии, принятой в данной проблемной области. Но всегда к введению таких классов следует подходить с осторожностью, всячески сопротивляясь появлению классов без новых компонентов.

Вот один пример. Предположим, некоторая система или библиотека включает класс PERSON, и вы рассматриваете целесообразность введения его потомков - MALE и FEMALE. Оправдано ли это? Следует все тщательно взвесить. Система управления персоналиями, в которой пол играет роль, например учитывая материнство, предоставление

отпусков, может получить преимущество от введения таких классов. Но во многих других случаях никаких специфических характеристик эти классы могут не нести, например, в статистических исследованиях, где достаточно иметь поле, задающее пол персоны, имея единый класс PERSON и булев атрибут:

```
female: BOOLEAN
```

или

```
Female: INTEGER is unique  
Male: INTEGER is unique
```

Однако если есть шанс, что специфические свойства персон разного пола могут проявиться позднее, то, возможно, предпочтительнее ввести эти классы заранее.

О чём следует помнить, так это о принципе Единого Выбора. Мы научились не доверять явному перечислению вариантов, реализуемых константами `unique`, из опасения обнаружить в нашем ПО куски кода с условиями выбора в форме:

```
if female then  
  ...  
else  
  ...
```

или `inspect` инструкциями. В данном случае, однако, не стоит особенно беспокоиться:

- Критика этого стиля связана с тем, что добавление каждого нового варианта приводит к цепной реакции изменений во всей системе, но в подобных случаях можно быть уверенным, что новый пол не появится.
- Даже при фиксированном множестве вариантов стиль явного `if` менее эффективен, чем основанный на динамическом связывании вызовов `this_person.some_operation`, где `MALE` и `FEMALE` по-разному определяют `some_operation`. Но тогда, если необходимо разделять людей по полу, мы нарушаем предпосылки данного обсуждения - отсутствие специфических свойств. Если такие свойства существуют, наследование оправдано.

Последний комментарий сигнализирует о реальных трудностях. Простые случаи таксономии, когда без необходимости добавляются узлы в структуру наследования, диагностируются довольно просто (достаточно заметить отсутствие специфических свойств). Но что, если варианты должны иметь специфические свойства, в результате чего классификация конфликтует с другими критериями? Система управления персоналиями оправдывает появление класса `FEMALE_EMPLOYEE`, если специфические свойства пола сотрудника выделяют этот класс, подобно тому как другие свойства выделяют классы постоянных и временных служащих. Но тогда речь больше не идет о таксономии - возникает другая общая и тонкая проблема **многокритериальной классификации (multi-criteria classification)**, чье возможное решение обсуждается позже в этой лекции.

Использование наследования: таксономия таксономии

Мощь наследования - это следствие его универсальности. Правда и то, что временами оно наносит вред, заставляя многих авторов вводить ограничения на механизм. Понимая эти опасения, а иногда и разделяя их, отбросим случайные сомнения и страхи и научимся радоваться наследованию во всех его законных вариантах, к исследованию которых мы теперь и переходим.

Дадим обзор правильного использования наследования:

- наследование подтипов (`subtype inheritance`);
- наследование вида (`view inheritance`);
- наследование с ограничением (`restriction inheritance`);
- наследование с расширением (`extension inheritance`);
- наследование с функциональной вариацией (`functional variation inheritance`);
- наследование с вариацией типа (`type variation inheritance`);
- наследование с конкретизацией (`reification inheritance`);
- структурное наследование (`structure inheritance`);
- наследование реализации (`implementation inheritance`);
- льготное наследование (`facility inheritance`) с двумя специальными вариантами: наследование констант и абстрактной машины (черного ящика) (`constant inheritance` и `machine inheritance`).

Некоторые из этих категорий (подтипы, вид, реализация, конкретизация, льготы) приводят к специфическим проблемам, обсуждаемым в отдельных разделах.

Область действия правил

Относительно широкое рассмотрение наследования, предпринятое в этой книге, не означает, что "подходит все". Мы принимаем и фактически поддерживаем только некоторые формы наследования, часть из которых одобряется не всеми авторами. Конечно, есть много способов неверного использования наследования, вроде `CAR_OWNER`. Так что случаи наследования строго ограничены:

Правило Наследования

Каждое использование наследования должно принадлежать одной из допустимых категорий.

Это правило утверждает, что все типы наследования **известны** и что, если встречается ситуация, не покрываемая этими типами, то **не следует** применять наследование.

Под допустимыми категориями понимаются категории, рассматриваемые в этом разделе. И я надеюсь, что все имеющие смысл ситуации полностью покрываются этим рассмотрением. Но таксономия (введение классификации) может нуждаться в дальнейшем обдумывании. Я нашел немногое в литературе по этой теме, наиболее полезная ссылка на неопубликованные тезисы докторской диссертации [Girod 1991]. Так что вполне возможно, что в этой попытке классификации пропущены некоторые категории. Но правило говорит, что, если вы рассматриваете возможное применение наследования, не укладывающееся в предложенную схему, то следует серьезно подумать, скорее всего, применять его не следует. Если же по здравому размышлению вы решите применить наследование, то это стоит рассматривать как новый вклад в классификацию.

Мы уже видели следствие правила Наследования - правило Таксомании, устанавливающее необходимость введения собственного вклада для класса наследника. Это непосредственно следует из того, что каждая легитимная форма наследования, детализируемая ниже, требует от наследника выполнения по крайней мере одной из ранее перечисленных операций.

Правило Наследования не запрещает наследственные связи, принадлежащие более чем к одной категории. Однако такая практика не рекомендуется.

Правило Упрощения Наследования

Следует предпочитать наследование, принадлежащее ровно одной допустимой категории.

Это не абсолютное правило; оно относится к рекомендательным положительным правилам. Оно вводится в интересах простоты и ясности: всякий раз, когда вводится наследственная связь между двумя классами, неявно применяются методологические принципы, в особенности при решении вопроса выбора одного из применимых вариантов. Простота структуры уменьшает вероятность ошибки проектирования или создания хаоса, усложняющего использование и сопровождение.

Конечно, возможна ситуация, при которой одна наследственная связь служит двум различным целям в нашей классификации. Но такие случаи составляют меньшинство.

К сожалению, я не знаю простого критерия, недвусмысленно говорящего о корректности свертки нескольких категорий в одну наследственную связь. Отсюда и происходит рекомендательный характер правила. Автор разработки, основываясь на ясном понимании методологии наследования, должен принимать решение в каждом спорном случае.

Ошибочное использование

Прежде чем рассмотреть правильные случаи, еще раз поговорим об ошибках. Ошибаться - в природе человека, нельзя надеяться на полноту классификации возможных ошибок, но несколько общих ошибок идентифицируются просто.

Первая типичная ошибка связана с путаницей отношений "**has**" и "**is**". Класс CAR_OWNER служит примером - экстремальным, но не уникальным. Мне доводилось слышать и видеть и другие подобные примеры, такие как APPLE_PIE, наследуемый от APPLE и от PIE, или (упоминаемый Adele Goldberg) ROSE_TREE, наследуемый от ROSE и от TREE.

Другим типичным примером является **таксомания**, в котором простое булево свойство, такое как пол персоны (или свойство с несколькими фиксированными значениями, такое как цвет светофора), используется как критерий наследования, хотя нет важных вариантов компонентов, зависящих от свойства.

Третьей типичной ошибкой является **наследование по расчету (convenience inheritance)**, при котором разработчик видит некоторые полезные компоненты класса и создает наследника просто для того, чтобы использовать эти компоненты. Заметьте, использование "наследования реализации" или "наследование компонентов класса" являются допустимыми формами наследования, изучаемыми позже в этой лекции. Ошибка в том, что класс используется как родитель **без подходящего отношения is-a** между соответствующими абстракциями, а в некоторых случаях вообще без адекватной абстракции.

Общая таксономия

С этого момента речь пойдет о правильном использовании наследования. Список включает двенадцать различных категорий, для удобства сгруппированных в три семейства:

Правильное использование наследования



Рис. 6.7. Классификация допустимых категорий наследования

Классификация основана на том наблюдении, что любая программная система отражает как внешнюю модель, так и связь с реалиями в области программных приложений. В связи с этим будем различать:

- наследование модели, отражающее отношения "is-a" между абстракциями, характерными для самой модели;
- программное наследование, выражающее отношения между объектами программной системы, не имеющих очевидных двойников во внешней модели;
- наследование вариаций - специальный случай, относящийся как к моделям, так и программному наследованию, служащий для описания вариаций семейства классов.

Эти три общие категории облегчают понимание, но наиболее важные свойства задаются терминальными категориями.

Так как классификация сама по себе является таксономией, можно из любопытства задаться вопросом, как применить к ней самой идентифицируемые категории. Это является темой упражнения У6.2.

Следующие далее определения используют имя А для родительского класса и В для наследника:

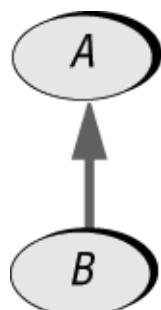


Рис. 6.8. Соглашение именования при определении категорий наследования

Каждое из определений будет устанавливать, в каких случаях А и В могут быть отложенными, а когда - эффективными. Обсуждение завершается таблицей, содержащей сводку применимых категорий для каждой комбинации отложенных и

эффективных классов.

Наследование подтипов

Начнем с наиболее очевидной формы модельного наследования. При моделировании внешней системы часто возникает ситуация, при которой категория внешних объектов естественно разделяется на непересекающиеся подкатегории. Например, замкнутые фигуры можно разделить на многоугольники и эллипсы. Вот формальное определение:

Определение: наследование подтипов

Наследование подтипов применимо, если А и В представляют некоторые множества A' и B' внешних объектов, так что B' является подмножеством A' , и множество, моделируемое любым другим подтипов, наследуемым от А, не пересекается с B' . Класс А должен быть отложенным.

A' может быть множеством замкнутых фигур, B' - множеством многоугольников, А и В - соответствующие классы. В большинстве практических случаев "внешняя система" не принадлежит миру программ, например, определяет некоторые аспекты деятельности компании (внешними объектами являются специальные и депозитные счета) или часть внешнего мира (с планетами и звездами).

Наследование подтипов является формой наследования ближайшей к иерархической таксономии в ботанике, зоологии и других естественных науках.

(ПОЗВОНОЧНЫЕ ← МЛЕКОПИТАЮЩИЕ и подобные примеры).

Мы настаиваем, что родитель А должен быть отложенным, поскольку он описывает не полностью специфицированное множество объектов, в то время как наследник В может быть как эффективным, так и отложенным. Следующие две категории рассматривают ситуации, где А может быть эффективным классом.

В одном из следующих разделов эта категория наследования будет рассмотрена детальнее, поскольку она не столь уж проста, как может показаться с первого взгляда.

Наследование с ограничением

Определение: Наследование с Ограничением

Наследование с ограничением применимо, если экземпляры В являются экземплярами А, удовлетворяющими некоторому ограничению, выраженному, если это возможно, как часть инварианта В, не включенного в инвариант А. Любой компонент, введенный в В, должен быть логическим следствием добавленного ограничения. А и В должны быть оба отложенными или оба эффективными.

Типичным примером является: Прямоугольник ← Квадрат.

Ограничением является утверждение: сторона1 = сторона2 (включается в инвариант класса Квадрат).

Многие математические примеры подпадают под эту категорию.

Последняя часть определения позволяет избежать смешения этой формы наследования с другими, такими как наследование с расширением, допускающим появление полностью новых компонентов у наследника. Здесь в интересах простоты предпочтительно ограничивать новые компоненты теми, что непосредственно следуют из добавленного ограничения. Например, у класса CIRCLE может появиться новый компонент radius, отсутствующий у родительского класса ELLIPSE.

Поскольку допускаются только концептуальные изменения класса А, добавляющие некоторые ограничения в класс В, то оба класса должны быть либо отложенными, либо эффективными.

Наследование с ограничением концептуально близко к наследованию подтипов; последующее обсуждение создания подтипов (subtyping) будет относиться к обеим категориям.

Наследование с расширением

Определение: Наследование с Расширением

Наследование с расширением применимо, когда В вводит компоненты, не представленные в А и не применимые к прямым экземплярам А. Класс А должен быть эффективным.

Присутствие обоих вариантов - расширения и сужения (ограничения) - является одним из парадоксов наследования. Как отмечалось при обсуждении наследования, расширение применяется к компонентам, в то время как ограничение (понимаемое как специализация) применяется к экземплярам. Но это не устраняет парадокс.

Проблема в том, что добавляемые компоненты обычно включают атрибуты. Так что при наивной интерпретации типа

(заданного классом) как множества его экземпляров отношение между классом и наследником (рассматриваемых как множества) "быть подмножеством" становится полностью ошибочным. Рассмотрим пример:

```
class A feature a1: INTEGER end
class B inherit
    A
feature
    b1: REAL
end
```

Рассмотрим каждый экземпляр класса А как одноэлементное множество (которое можно записать как $\langle n \rangle$, где n целое), а каждый экземпляр В - как пару, содержащую целое и вещественное (например, пару $\langle 1, -2.5 \rangle$). Множество пар МВ не является подмножеством одноэлементного множества МА. Верно обратное, отношение "быть подмножеством" имеет место в обратном направлении, поскольку существует отображение один-к-одному между МА и множеством всех пар, имеющих данный второй элемент.

Обнаружение того факта, что отношение "быть подмножеством" не выполняется, делает наследование расширением довольно подозрительным. Например, в ранней версии уважаемой ОО-библиотеки (не от ISE) класс RECTANGLE был наследником SQUARE, в отличие от изучаемого нами способа. Причина простая: класс SQUARE имеет атрибут side; класс RECTANGLE наследует его, добавляя новый компонент other_side. Этот проект был подвергнут критике, он был пересмотрен с обращением наследования.

Но не следует исключать наследование с расширением как общую категорию. У нее есть эквивалент в математике, где специализация некоторого понятия происходит путем добавления новых операций. Такое происходит довольно часто и считается необходимым. Типичным примером является понятие кольца, представляющее специализацию понятия **группы**. В группе задана некоторая операция, назовем ее +, обладающая рядом свойств. Кольцо является группой, потому имеет ту же операцию + с теми же свойствами. Но в кольце добавляется новая операция, скажем, *, со своими собственными свойствами. По сути это не отличается от введения нового атрибута классом наследником.

Соответствующая схема используется и при разработке ОО-ПО. Конечно, класс SQUARE должен быть наследником RECTANGLE, а не наоборот, но можно предложить легитимные примеры. Класс MOVING_POINT (в приложениях кинематики) может наследовать от чисто графического класса POINT и добавлять компонент speed, описывающую величину и направление скорости. Другой пример, в текстовом процессоре класс CHAPTER может наследовать от DOCUMENT, добавляя специфические свойства - текущую позицию лекции в книге и процедуру ее сохранения.

Подходящая математическая модель

(Читатели - не математики могут пропустить этот раздел.)

Для успокоения совести следует разрешить видимый парадокс, отмеченный выше (обнаружение того, что МВ не является подмножеством МА), так как мы хотим, чтобы некоторое отношение подмножества имело место между наследником и родителем. И это отношение реально существует, парадокс лишь показывает, что декартово произведение атрибутов не является подходящей моделью для моделирования класса. Рассмотрим класс:

```
class C feature
    c1: T1
    c2: T2
    c3: T3
end
```

Мы **не должны** выбирать в качестве математической модели С' - множества экземпляров С - декартово произведение $T_1 \times T_2 \times T_3$, где знак штрих ' указывает на рекурсивное использование модели множеств, приводящее к парадоксу (наряду с другими недостатками).

Вместо этого, следует рассматривать любой экземпляр как частичную функцию, отображающую множество возможных имен атрибутов ATTRIBUTE в множество возможных значений VALUE, со следующими свойствами:

- **A1** Функция определена для с1, с2 и с3.
- **A2** Множество VALUE (множество цели для функции) является супермножеством $T'_1 \cup T'_2 \cup T'_3$.
- **A3** Значения функции для с1 лежат в T'_1 и так далее.

Тогда, если вспомнить, что функция является специальным случаем отношения и что отношение является множеством пар (например, в ранее упоминаемом случае экземпляр класса А может быть промоделирован функцией $\{\langle a1, 25 \rangle\}$, а экземпляр класса В - $\{\langle a1, 1 \rangle, \langle b1, -2.5 \rangle\}$), мы получаем ожидаемое свойство - В' является подмножеством А'. Заметьте, здесь уже элементы обоих множеств являются парами и первая функция задает все возможные отображения второго атрибута.

Заметьте также, что принципиально важно установить свойство A1 как "Функция определена для...", но не в виде "Областью определения функции является...", что ограничивало бы область множеством {с1, с2, с3}, не позволяя потомкам добавлять свои собственные атрибуты. Как результат такого подхода, каждый программный объект моделируется неограниченным числом математических объектов.

Это обсуждение дает только схему математической модели. С деталями использования частичных функций для моделирования кортежей и общими математическими основами можно ознакомиться в [М 1990].

Наследование вариаций

(Читатели - не математики, добро пожаловать!) Переидем теперь ко второму семейству категорий - наследованию вариаций.

Определение: Наследование вариаций типа и функций

Наследование вариаций применяется, если В переопределяет некоторые компоненты А; А и В являются оба либо отложенными, либо эффективными. Класс В не должен вводить никаких новых компонентов за исключением тех, что непосредственно необходимы переопределаемым компонентам. Здесь рассматриваются два случая:

- Наследование вариаций функций: переопределения действуют на тела компонентов, но не на их сигнатуры.
- Наследование вариаций типа: все переопределения являются переопределениями сигнатур.

Наследование вариаций применимо, когда существующий класс А, задающий некоторую абстракцию, полезен сам по себе, но обнаруживается необходимость представления подобной, хотя и не идентичной абстракции, имеющей те же компоненты, но с отличиями в сигнатуре и реализации.

Определение требует, чтобы оба класса были эффективными (общий случай) или оба отложенными. Оно не рассматривает эффективизацию компонентов, когда речь идет о переходе от абстрактной формы к конкретной. Тесно связанной является рассматриваемая далее категория "отмена эффективизации", в которой некоторые эффективные компоненты становятся отложенными.

Из определения следует, что наследник не должен вводить новых компонентов за исключением непосредственно необходимых для переопределения. Этим проводится граница между наследованием расширением и наследованием вариаций.

При вариациях типа можно изменять только сигнатуры некоторых компонентов (число и типы аргументов и результата). Эта форма наследования подозрительна и часто является признаком таксомании. В законных случаях, однако, это может быть подготовкой для наследования расширением или реализацией. Примером наследования вариации типа могут быть наследники MALE_EMPLOYEE и FEMALE_EMPLOYEE.

Наследование вариации типа не является необходимым, когда начальная сигнатура использует закрепленные (like...) объявления. Например, в классе SEGMENT интерактивного пакета для рисования можно ввести функцию:

```
perpendicular: SEGMENT is
    -- Сегмент, повернутый на 90 градусов
    . . .
```

Затем определим наследника DOTTED_SEGMENT, дающего графическое представление пунктирными, а не непрерывными линиями. В этом классе perpendicular должен возвращать результат типа DOTTED_SEGMENT, так что необходимо переопределить тип. Этого бы не требовалось, если бы изначально результат объявлялся как like Current. Так что, будь у вас доступ к источнику и его автору, можно было бы предложить модифицировать оригинал, не нанося ущерба существующим клиентам. Но если нет возможности модифицировать оригинал или по ряду причин закрепленное объявление не подходит оригинал (вероятно, из-за потребностей других потомков), то возможность переопределить тип может стать палочкой-выручалочкой.

При наследовании **функциональной** вариации изменяются тела некоторых компонентов. Если, как обычно в таком случае, компонент уже эффективен, то это означает изменение реализации. Спецификация компонента, заданная утверждением, также может измениться. Также возможно, хотя и реже встречается, иметь функциональную вариацию между двумя отложенными классами, в этом случае будут меняться только утверждения. Это может повлечь изменения в некоторых функциях, отложенных или эффективных, связанных с утверждениями, или даже добавление новых компонентов.

Наследование функциональной вариации является прямым приложением принципа Открыт-Закрыт: мы хотим адаптировать существующий класс, не затрагивая оригинал (к коду которого мы можем и не иметь доступа) и его клиентов. Это может стать предметом злоупотреблений, некоей формой хакерства, перекручивая существующий класс, приспособливая его для других целей. Во всяком случае это будет организованное хакерство, позволяющее избежать угроз модификации существующего ПО. Но если есть доступ к оригинал, то предпочтительной может оказаться реорганизация иерархии наследования путем введения абстрактного класса, для которого как уже существующий класс А, так и новичок В будут его потомками или подходящими наследниками с равным статусом.

Отмена эффективизации

Определение: Наследование с Отменой эффективизации

Наследование с отменой эффективизации применимо, если В переопределяет некоторые из эффективных компонентов А, преобразуя их в отложенные компоненты.

Отмена эффективизации не является общим приемом и не должна им быть. Основная идея этого способа противоречит общему направлению, так как обычно ожидается конкретизация потомка В своего более абстрактного родителя А (как это имеет место в следующей рассматриваемой категории, для которой А является отложенным, а В эффективным или, по крайней мере, менее отложенным). По этой причине новичкам следует избегать отмены эффективизации. Но она может быть законной в двух случаях:

- При множественном наследовании сливаются компоненты, наследуемые от двух различных родителей. Если один из них отложенный, а другой эффективный, то сливание произойдет автоматически при условии совпадения имен (возможно после переименования), эффективная версия будет определять реализацию. Но если обе версии эффективны, то следует провести потерю эффективизации одной версии, отдавая предпочтение другой версии.
- Хотя абстракция соответствует потребностям, но повторно используемый класс **слишком конкретен** для наших целей. Отмена эффективизации позволит удалить нежеланную реализацию. Перед использованием этого решения следует рассмотреть альтернативу - реорганизовать иерархию наследования, сделав более конкретный класс наследником нового отложенного класса. По понятным причинам это не всегда возможно, например из-за отсутствия доступа к исходному коду. Отмена эффективизации может в таких случаях обеспечить полезную форму обобщения.

Для этой категории наследования класс В будет отложенным; А обычно эффективным, но может быть частично отложенным.

Наследование с конкретизацией

Перейдем теперь к третьей и последней группе категорий - программному наследованию.

Определение: Наследование с конкретизацией

Наследование с конкретизацией применимо, если А задает структуру данных общего вида, а В представляет ее частичную или полную реализацию. А является отложенным; В может быть эффективным или все еще отложенным, оставляя пространство для дальнейшей конкретизации собственными потомками.

Примером, используемым многократно в предыдущих лекциях, является отложенный класс TABLE, описывающий таблицы самой общей природы. Конкретизация ведет к потомкам SEQUENTIAL_TABLE и HASH_TABLE, все еще отложенным. Заключительная конкретизация SEQUENTIAL_TABLE приводит к эффективным классам ARRAYED_TABLE, LINKED_TABLE, FILE_TABLE.

Термин "конкретизация" (reification), введенный Георгом Лукасом, происходит от латинского слова, означающего "превращение в вещь". Он используется в спецификациях и методе разработки VDM.

Структурное наследование

Определение: структурное наследование

Структурное наследование применяется, если А, отложенный класс, представляет общее структурное свойство, а В, который может быть отложенным или эффективным, представляет некоторый тип объектов, обладающих этим свойством.

Обычно А представляет математическое свойство, которым может обладать некоторое множество объектов. Например, А может быть классом COMPARABLE, поставляемым с такими операциями, как infix "<" и infix ">=", представляющим объекты с заданным отношением полного порядка. Класс, которому необходимо отношение порядка, такой как STRING, становится наследником COMPARABLE.

Наследование таким способом от нескольких родителей является обычным приемом. Например, класс INTEGER в библиотеке Kernel наследует от COMPARABLE и класса NUMERIC (с такими компонентами, как infix "+" и infix "*"), задающего арифметические свойства. (Класс NUMERIC более точно представляет математическое понятие кольца.)

В чем разница между конкретизацией и структурным наследованием? При конкретизации В представляет то же понятие, что и А, отличаясь большей степенью реализации; при структурном наследовании В представляет собственную абстракцию, для которой А задает лишь один из аспектов, такой как порядок на объектах или наличие арифметических операций.

Валден и Нерсон заметили, что новички иногда верят, что они используют подобную форму наследования, подменяя фактически имеющее место отношение "is" вариантом схемы "car-owner" (AIRPLANE наследуется от VENTILATION_SYSTEM). Они указывают, что этой ошибки просто избежать благодаря абсолютному критерию, не оставляющему места для сомнений или двусмысленности:

При схеме наследования, хотя наследуемые свойства являются вторичными, они все же являются свойствами всего объекта, описываемого классом. Если мы делаем AIRPLANE наследником COMPARABLE, то отношение порядка применимо к каждому самолету как к целому, но свойства VENTILATION_SYSTEM не таковы. Компонент stop VENTILATION_SYSTEM не прекращает полет самолета.

Заключение в этом примере очевидно: AIRPLANE должен быть клиентом, а не наследником класса VENTILATION_SYSTEM.

Наследование реализации

Определение: Наследование Реализации

Наследование реализации применяется, если В получает от А множество компонентов (отличных от константных атрибутов и однократных функций), необходимых для реализации абстракции, связанной с В. Как А, так и В должны быть эффективными.

Наследование реализации в деталях обсуждается позднее в этой лекции. Общей ситуацией является "брак по расчету", основанный на множественном наследовании, где один из родителей обеспечивает спецификацию (наследование с конкретизацией), а другой - предоставляет реализацию (наследование реализации).

Случай наследования константных атрибутов и однократных функций покрывается следующим вариантом.

Льготное наследование

Льготное наследование является схемой, в которой родитель представляет коллекцию полезных компонентов, предназначенных для использования его потомками.

Определение: Льготное наследование

Льготное наследование применяется, если А существует единственно в целях обеспечения множества логически связанных компонентов, дающих преимущества его потомкам, таким как В. Двумя общими вариантами являются:

- Наследование констант, при котором компоненты А все являются константами или однократными функциями, описывающими разделяемые объекты.
- Наследование абстрактной машины, в котором компоненты А являются подпрограммами, рассматриваемыми как операции абстрактной машины.

Примером льготного наследования может служить класс EXCEPTIONS, класс, предоставляющий множество утилит, обеспечивающих доступ к механизму обработки исключений.

Иногда, как в примерах, которые появятся чуть позже, при льготном наследовании используется только один вариант - константы или абстрактная машина, но в других случаях, как для класса EXCEPTIONS, родительский класс предоставляет как константы (такие как коды исключений `Incorrect_inspect_value`), так и подпрограммы (такие как `trigger` для возбуждения исключения разработчика). Так как при нашем обсуждении категории наследования рассматриваются как непересекающиеся, то льготное наследование с двумя пересекающимися вариантами рассматривается как одна категория.

При наследовании констант как А, так и В являются эффективными. При наследовании абстрактной машины ситуация более гибкая, но В должно быть, по меньшей мере, столь же эффективно как и А.

В деталях льготное наследование еще будет обсуждаться в данной лекции.

Использование наследования с отложенными и эффективными классами

В следующей таблице обобщены правила, определяющие для каждой категории, должны ли родитель и его потомок быть эффективными или отложенными классами. "Вариация" покрывает случаи вариации типа и функциональной вариации. Элементы, помеченные символом *, появляются более чем в одном входе.

Таблица 6.1. Отложенные и эффективные наследники и их родители

Потомок	Родитель	
	Отложенный	Эффективный
Отложенный	Константы*	Расширение*
	Ограничение*	Потеря эффективизации*
	Структура*	
	Тип*	
	Потеря эффективизации*	
	Вариация*	

Вид	
Эффективный	Константы*
	Конкретизация
	Структура*
	Тип*
	Константы*
	Расширение*
	Реализация
	Ограничение*
	Вариация*

Один механизм, или несколько?

Заметьте, это обсуждение предполагает в качестве основы раннюю презентацию, определяющую смысл наследования (см. [лекцию 14](#) курса "Основы объектно-ориентированного программирования").

Разнообразие использования наследования, чему свидетельством является предшествующее рассмотрение, может создавать впечатление, что должны существовать разнообразные механизмы языка, покрывающие основополагающие понятия. В частности, ряд авторов предлагают разделение **наследование модуля**, в особенности как средства повторного использования существующих компонентов в новом модуле, и **наследования типа**, в частности механизма классификации типов.

Такое разделение нанесло бы больше вреда, чем принесло пользы. Вот несколько доводов.

Во-первых, сведение наследования к двум категориям не отражает всего разнообразия, вводимого нашей классификацией. Так как никто не будет отстаивать введение десяти различных языковых механизмов, то введение двух механизмов приводило бы к ограниченному результату.

Практическим следствием были бы бесполезные методологические обсуждения: предположим, вы хотите наследовать от класса итератора, такого как `LINEAR_ITERATOR`; следует ли использовать наследование модуля или наследование типа? Можно приводить аргументы в защиту одного и другого решения. Вклад этого предложения в критерий качества нашего ПО и скорость его создания будут фактически нулевыми.

В упражнении У6.8 требуется проанализировать наши категории, отнеся их либо к наследованию модуля, либо к наследованию типа.

Интересно задуматься и о тех последствиях в усложнении языка, к которым привело бы такое разделение. Наследование сопровождается некоторыми вспомогательными механизмами, большинство из которых необходимо обоим видам:

- **Переопределение** полезно как для подтипов (вспомните `RECTANGLE`, переопределяющий `perimeter` от `POLYGON`) и для расширения модуля (принцип Открыт-Закрыт требует при наследовании модуля сохранения гибкости изменений, без чего будет потеряно одно из главных преимуществ ОО-метода).
- **Переименование** полезно при наследовании модуля. Полагать его неподходящим при наследовании типа (см. [Breu 1995]) представляется серьезным ограничением. При моделировании внешней системы варианты некоторого понятия могут вводить специальную терминологию, которую желательно сохранить в ПО. Класс `STATE_INSTITUTIONS` в географической или выборной информационной системе может иметь потомка `LOUISIANA_INSTITUTIONS`, отражающего особенности политической структуры штата Луизиана, поэтому вполне ожидаемо желание потомка переименовать компонент `counties`, задающий список округов штатов, в `parishes` - имя, используемое для округа в данном штате.
- **Дублируемое наследование** может встретиться для любой из форм. Так как можно ожидать, что только наследование модуля сохранит полиморфную подстановку, то при наследовании типов тут же возникнет необходимость разбора случаев и предложения `select` со всеми недостатками при появлении новых случаев. Появляются и другие вопросы - когда разделять компоненты, а когда их дублировать.
- При введении в язык новых механизмов они взаимодействуют друг с другом и с другими механизмами языка. Должны ли мы защитить класс от совместного наследования и модуля, и типа? Если да, то будут возмущены разработчики, использующие класс двумя возможными способами; если нет, мы откроем ящик Пандоры, грозящий появлением множества проблем - конфликтов имен, переопределений и так далее.

Все это ради преимуществ пурристской точки зрения - ограниченной и спорной. Нет ничего плохого в защите спорной точки зрения, но следует быть крайне осторожным в нововведениях и учитывать их последствия для пользователей языка. И снова примером может служить Эдсгар Дейкстра в исследовании `goto`. Он не только в деталях объяснил все недостатки этой инструкции, основываясь на теории конструирования ПО и процесса его выполнения, но и показал, как можно без труда заменить этот механизм. В данном же случае убедительные аргументы не представлены, по крайней мере, я не увидел, почему "плохо" иметь единый механизм, покрывающий как наследование модулей, так и наследование типа.

Помимо общих возражений, основанных на предвзятых идеях о том, каково должно быть наследование, приводится лишь один серьезный аргумент против единого механизма наследования - сложность **статической проверки типов**, возникающая при этом подходе. Эта проблема полностью анализировалась в [лекции 17](#) курса "Основы объектно-ориентированного программирования", бремя ее решения возлагается на компиляторы, которые признают это бремя разумным, если этим существенно облегчается жизнь разработчиков.

Как покажет в конечном итоге это обсуждение, возможность иметь единый механизм для наследования модуля и типа является результатом **важнейшего решения** ОО-конструирования ПО: унификации концепций модуля и типа в едином понятии класса. Если мы принимаем классы как модули и как типы, то следует принять и наследование, аккумулирующее модули и типы.

Наследование подтипов и скрытие потомков

Первая категория наследования из нашего списка, вероятно, единственная, с которой согласится каждый, по меньшей мере, тот, кто принимает наследование: то, что мы можем назвать чистым наследованием подтипов (типов).

Большая часть данного обсуждения применима и к наследованию с ограничением, чье принципиальное отличие состоит в том, что от родителя не требуется быть отложенным классом.

Определение подтипа

Как указывалось во введении в наследование, мощь этой идеи частично происходит от интеграции механизма модуля с механизмом типов: определения нового типа как специального случая существующего типа, определение нового модуля как расширения уже существующего модуля. Большинство из спорных вопросов о наследовании идут от осознания конфликтов между этими двумя точками зрения. При наследовании подтипов такие вопросы не возникают, хотя, как мы увидим, это не означает, что все становится простым.

Наследование подтипов по своему образцу соответствует принципам таксономии в естественных и математических науках. Каждое позвоночное является животным, каждое млекопитающее является позвоночным, каждый слон является млекопитающим. Каждая группа (в математике) является моноидом, каждое кольцо является группой, каждое поле является кольцом. Подобными примерами, многие из которых мы видели в предыдущих лекциях, изобилует ОО-ПО:

- FIGURE ← CLOSED_FIGURE ← POLYGON ← QUADRANGLE ← RECTANGLE ← SQUARE
- DEVICE ← FILE ← TEXT_FILE
- SHIP ← LEISURE_SHIP ← SAILBOAT
- ACCOUNT ← SAVINGS_ACCOUNT ← FIXED_RATE_ACCOUNT

В любом из этих случаев четко идентифицируется множество объектов, описываемое родительским типом, и мы выделяем подмножество, характеризуемое некоторыми свойствами, которыми обладают не все объекты родителя. Например, текстовый файл является файлом со специальными свойствами, вытекающими из того, что элементы файла являются строками текста, что нехарактерно, например, для бинарных файлов.

Общее правило при наследовании подтипов состоит в том, что потомки задают непересекающиеся подмножества экземпляров. Ни одна из замкнутых фигур не является одновременно эллипсом и многоугольником.

Некоторые из примеров, такие как RECTANGLE ← SQUARE, возможно, включают эффективного родителя и потому представляют случаи наследования с ограничением.

Различные взгляды

Наследование подтипов кажется простым, когда существует четкий критерий классификации вариантов определенного понятия. Но иногда некоторые качества начинают конкурировать между собой. Даже в, казалось бы, совсем простом случае классификации многоугольников могут возникнуть сомнения: следует ли использовать для классификации число сторон, создавая такие классы, как TRIANGLE, QUADRANGLE etc., или следует разделять объекты на правильные многоугольники (EQUILATERAL_POLYGON, SQUARE и т. д.) и неправильные?

Несколько стратегий доступно для разрешения конфликтов. Они будут рассмотрены позднее в этой лекции, как часть обзора наследования.

Взгляд на подтипы

Тип - это не просто множество объектов. Он характеризуется также применимыми операциями (компонентами) и их семантическими свойствами (утверждениями: предусловиями, постусловиями, инвариантами). Мы предполагаем, что компоненты и утверждения наследника совместимы с концепцией подтипа, означая, что любой экземпляр наследника должен рассматриваться также как экземпляр родителя.

Правила, применяемые к утверждениям, поддерживают этот взгляд на подтипы:

- инвариант родителя автоматически является частью инварианта наследника, так что все ограничения, специфицированные для экземпляров родителя, применимы к экземплярам родителя;
- предусловие подпрограмм, возможно ослабленное, применимо к любому ее переопределению у потомка, так что любой вызов, удовлетворяющий требованиям для экземпляров родителя, будет также удовлетворять требованиям экземпляров наследника;
- постусловие подпрограмм, возможно, усиленное, применимо к любому ее переопределению у потомка, так что любое свойство на выходе подпрограммы, специфицированное для экземпляра родителя, будет также

выполняться экземплярами наследника.

Для компонентов ситуация более тонкая. С точки зрения на подтипа требуется, чтобы все операции, применимые к экземплярам родителя, должны быть применимы к экземплярам наследника. Внутренне это всегда верно: даже для класса ARRAYED_STACK, наследуемого от ARRAY, который, кажется, далек от наследования подтипов, компоненты ARRAY доступны наследнику и фактически являлись основой для реализаций свойств стека. Но в этом случае мы скрываем все эти компоненты ARRAY от клиентов наследника по вполне разумным причинам (мы не хотим, чтобы клиенты могли выполнять операции, зависящие от внутреннего представления, так как это бы нарушило интерфейс класса).

Для чистого наследования подтипов можно предложить более сильное правило: **каждый** компонент, применимый клиентом к экземплярам родительского класса, тем же клиентом может быть применен к экземплярам наследника. Другими словами, нет скрытия компонента потомком: если B наследует f от A, то статус экспорта f в B не менее широкий, чем в A. (Так что общеэкспортируемый компонент f таковым и остается, а выборочно экспортируемый может только расширить круг клиентов.)

Необходимость скрытия потомком

В совершенном мире можно было бы ввести правило, не допускающее скрытия потомком, но это не подходит для реального мира ПО. Наследование должно быть полезно даже для классов, написанных людьми, не обладающими совершенным предвидением, некоторые из включенных в класс компонентов могут не иметь смысла для потомков, написанных позднее и в другом контексте. Такие случаи можно назвать таксономией исключений (В другом контексте достаточно было бы одного слова "исключение", но не хочется пересечения с программистским понятием исключения, изучаемым в предыдущих лекциях.)

Следует ли отказываться от наследования привлекательного и полезного класса из-за таксономии исключений, другими словами, из-за того, что у него есть пара компонентов, не подходящих для вызова нашими клиентами. Это было бы неразумно. Следует просто спрятать эти компоненты, сделать их невидимыми для наших клиентов и продолжать свою работу.

Альтернативы обсуждались при рассмотрении основополагающего принципа Открыт-Закрыт и они не кажутся привлекательными:

- Можно было бы модифицировать оригинальный класс. Это повлекло бы к поломке уже работающих систем у всех клиентов класса - нет уж, увольте! И это не всегда возможно практически из-за недоступности кода.
- Можно было бы написать новую версию класса, если нам повезло, и мы располагаем исходным кодом. Этот подход противоположен всему ОО-подходу, он противоречит прежде всего повторному использованию.

Как избежать скрытия потомком

Прежде чем понять, когда и почему необходимо скрытие потомком, следует заметить, что чаще всего этого делать не стоит. Эта техника должна находиться в резерве главного командования. Когда есть полный контроль над структурой наследования на ранних этапах разработки системы, то **предусловия** дают лучший способ справиться с таксономией исключений.

Рассмотрим класс ELLIPSE. Эллипс имеет два фокуса, через которые можно провести прямую:

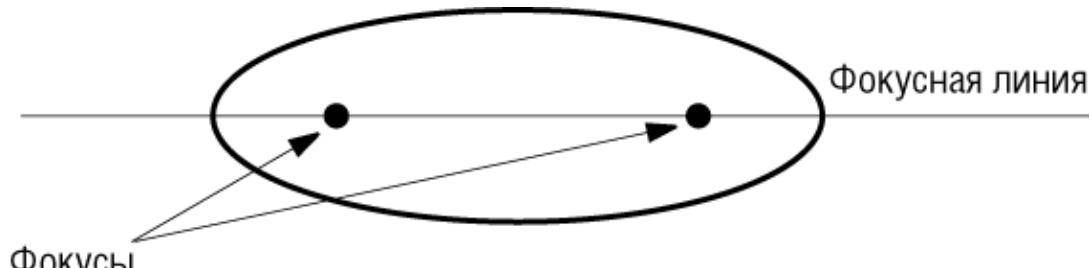


Рис. 6.9. Эллипс и фокусная линия

Класс ELLIPSE может соответственно иметь компонент focus_line.

Естественно, определить класс CIRCLE как наследника ELLIPSE: каждая окружность является эллипсом. Но для окружности два фокуса сливаются в одну точку - центр окружности, так что фокусная линия исчезает. (Вероятно, более корректно говорить о бесконечном множестве фокусных линий, любая прямая, проходящая через центр, может рассматриваться как фокусная линия, но на практике эффект будет тот же.)

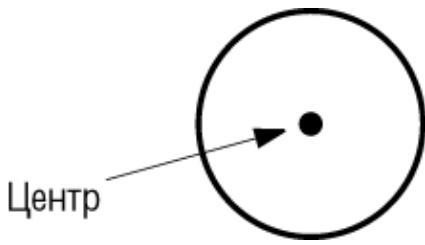


Рис. 6.10. Круг и его центр

Хороший ли это пример для скрытия потомком? Должен ли класс CIRCLE сделать компонент focus_line закрытым, как здесь:

```
class CIRCLE inherit
  ELLIPSE
    export {NONE} focus_line end
  ...

```

Вероятно, нет. В данном случае у разработчика родительского класса была вся необходимая информация для понимания того, что не все эллипсы имеют фокусную линию. Для компонентов, представляющих подпрограмму, следует ввести предусловие:

```
focus_line is
  -- Линия, проходящая через два фокуса
  require
    not equal (focus_1, focus_2)
  do
  ...
end
```

(Предусловие может быть абстрактным, использующим функцию distinct_focuses; с тем преимуществом, что класс CIRCLE может сам переопределить ее.)

Необходимость поддержки эллипсов без фокусной линии вытекает из анализа проблемы. Создать соответствующий класс с функцией focus_line, не имеющей предусловия, является ошибкой проекта. Переложить решение на скрытие потомком является попыткой скрытия ошибки. Как указывалось в конце обсуждения принципа Открыт-Закрыт, ошибочные решения должны фиксироваться, потомки не должны латать прорехи проекта.

Приложения скрытия потомком

Пример с фокусной линией типичен для таксономии исключений, возникающей в областях приложений, подобных математике, обладающих серьезной теорией с подробной классификацией, накопленной за долгое существование. В таком контексте рекомендуется использовать предусловия на этапе создания исходного компонента.

Но эта техника не всегда применима, особенно в тех областях человеческой деятельности, где трудно предвидеть все возможные исключения.

Рассмотрим иерархию с корневым классом MORTGAGE (**ЗАКЛАДНАЯ**). Потомки организуются в соответствии с различными критериями, такими как фиксированная или переменная ставка, деловая или персональная, любыми другими. Для простоты будем полагать, что речь идет о таксономии - чистом случае подтипов. Класс MORTGAGE имеет процедуру redeem (выплачивать долг по закладной), управляемой выплатами по закладной в некоторый период, предшествующий сроку оплаты.

Теперь предположим, что Конгресс в порыве великодушия (или под давлением лоббистов) ввел новую форму закладных, субсидируемых правительством, чьи преимущества одновременно предполагают запрет досрочных выплат. В иерархии классов найдется место для класса NEW_MORTGAGE; но что делать с процедурой redeem?

Можно было бы использовать технику предусловий, как в случае с focus_line. Но что, если банкиру никогда не приходилось иметь дело с закладными, по которым нельзя платить досрочно? Тогда, вероятно, процедура redeem не будет иметь предусловия.

Так что использование предусловия потребует модификации класса MORTGAGE со всеми вытекающими последствиями. Предположим, однако, что в данном случае проблем с модификацией не будет, и мы добавим в класс булеву функцию redeemable и предусловие к redeem:

```
require
  redeemable
```

Но тем самым мы изменили интерфейс класса. Все клиенты класса и их бесчисленные потомки мгновенно стали потенциально некорректными. Все их вызовы m.redeem (...) должны быть теперь переписаны как:

```
if m.redeemable then
  m.redeem (...)
```

```
else
    ... (Кто в мире мог предвидеть это?)...
end
```

Вначале это изменение не является неотложным, поскольку некорректность только потенциальная: существующую систему используют только существующие потомки MORTGAGE, так что никакого вреда результатам не будет. Но не зафиксировать их означает оставить бомбу с тикающим часовым механизмом - незащищенные вызовы подпрограммы с предусловием. Как только разработчику клиента придется в голову умная идея использования полиморфного присоединения источника типа NEW_MORTGAGE, то при опущенной проверке возникнет жучок. А компилятор не выдаст никакой диагностики.

Отсутствие предусловия в исходной версии redeem не является ошибкой проектирования. В период создания проекта каждая закладная допускала досрочные выплаты. Мы не можем требовать предусловий для каждого компонента, иначе до конца своих дней придется каждый вызов предварять тестом if.

Пример redeem типичен для таксономии исключений, в отличие от focus_line и других случаев совершенной классификации он не может использовать тщательно спроектированные предусловия. Ранее сделанные замечания полностью здесь справедливы, было бы абсурдно отказываться от наследования полезного класса из-за пары компонентов, не применимых в данном контексте. В таких ситуациях следует использовать скрытие потомком:

```
class NEW_MORTGAGE inherit
    MORTGAGE
        export {NONE} redeem end
    ...

```

Ни ошибок, ни аномалий не появится в существующем ПО. Если кто-то модифицирует класс клиента, добавив класс с новыми закладными:

```
m: MORTGAGE; nm: NEW_MORTGAGE
...
m := nm
...
m.redeem (...)
```

то вызов redeem станет кэтколлом (см. [лекцию 17](#) курса "Основы объектно-ориентированного программирования"), и потенциальная ошибка будет обнаружена статически механизмом, описанным в [лекции 17](#) курса "Основы объектно-ориентированного программирования" при обсуждении типизации.

Таксономии и их ограничения

Исключения таксономии не являются спецификой программистских примеров. В большей степени они характерны для естественных наук, где почти невозможно найти утверждение в форме "члены ABC phylum [или genus, species etc.] **характеризуются свойством XYZ**", которому не предшествовало бы "**большинство**", "**обычно**" или за которым не следовало бы "**за исключением нескольких случаев**". Это справедливо на всех уровнях иерархии, даже для наиболее фундаментальных категорий, для которых, казалось бы, существуют бесспорные критерии!

Если вы думаете, например, что просто отличать животных от растений, то ознакомьтесь с текстом, взятым из популярного учебника (курсив добавлен):

Отличие растений от животных

Есть несколько общих факторов, позволяющих отличать растения от животных, хотя есть многочисленные **исключения**.

Перемещение. **Большинство** животных свободно передвигаются, в то время как **редкие** растения могут перемещаться в окружающем их пространстве. **Большинство** растений имеют корни в почве или прикреплены к скалам, деревьям или **другим материалам**.

Пища. Зеленые растения, содержащие хлорофилл, сами производят еду для себя, **большинство** животных питаются растениями или поедают других животных.

Рост. Растения **обычно** растут от концов своих ветвей и корней и от внешних участков ствола в течение всей жизни. У животных рост **обычно** идет во всех частях их тела и прекращается при достижении зрелости.

Химическая регуляция. Для растений и для животных **общим** является наличие гормонов и других химикатов, регулирующих определенные процессы в организме, однако химический состав гормонов **отличается в** растительном и животном мирах.

Те же комментарии применимы к другим областям изучения. Это в полной степени относится и к области человеческой культуры - классификация естественных языков внесла свой вклад в разработку систематической таксономии.

Известный пример из зоологии, ставший уже клише, иллюстрирует таксономию исключений. (Помните, однако, что это только аналогия, а не программистский пример.) Птицы летают. (Класс BIRD должен иметь процедуру fly.) Страус - птица, но страус не летает. При создании наследника - класса OSTRICH - необходимо будет указать, что эта самая

большая из птиц не летает.

При классификации птиц можно было бы попытаться разделить их на две категории - летающие и не летающие. Но это конфликтовало бы с другими возможными и более важными критериями, применяемыми в классификации, ставшей стандартной.

Пример со страусом (OSTRICH) имеет интересный поворот. Хотя, к сожалению, они и не сознают этого, страусы фактически должны летать. Молодые поколения теряют этот наследственный навык из-за случайности эволюционной истории. Анатомически страусы являются самой совершенной аэродинамической машиной среди всех птиц. Это свойство, немного осложняет работу профессионального таксономиста, (хотя ее может облегчить его коллега - профессиональный таксидермист) не помешает классифицировать страусов в иерархии птиц.

В программистских терминах класс OSTRICH будет наследником BIRD, скрывая наследуемый компонент fly.

Использование скрытия потомком

Все наши усилия (по классификации) кажутся беспомощными на фоне множественности отношений живых существ, окружающих нас. Эта битва Человека и Природы во всей ее бесконечности описана величайшим ботаником Гете. Одно можно сказать с уверенностью, Человек в ней всегда будет побежден.

Анри Бэйлон

Практика разработки ПО и аналогии природного мира свидетельствуют, что даже при самом тщательном проектировании остаются исключения таксономии. Скрытие redeem класса NEW_MORTGAGE или fly из OSTRICH не является свидетельством небрежного проектирования или недостаточного предвидения, оно свидетельствует о реальной сложности иерархии наследования.

Такие исключения таксономии имеют прецеденты, насчитывающие столетние усилия интеллектуальных гигантов (включая Аристотеля, Линнея, Бюффона и Дарвина). Они сигнализируют о внутренних ограничениях человеческой способности познания мира. Связаны ли они с результатами, шокирующими научную мысль в двадцатом столетии - принципом неопределенности в физике, неразрешимыми проблемами в математике?

Все это предполагает, что скрытие потомком остается, хотя, как отмечалось, не должно часто использоваться. Для тех немногих случаев при разработке ПО, когда есть принципиальные препятствия в разработке совершенной иерархии типов, скрытие потомков является более чем удобным и спасительным средством.

Наследование реализации

Часто критикуемой, но полезной и концептуально правильной формой является наследственная связь между классом, описывающим реализацию абстрактной структуры данных, и классом, обеспечивающим реализацию.

Брак по расчету

При обсуждении множественного наследования мы видели пример брака по расчету, комбинирующего отложенный класс с механизмом его реализации. Примером был стек, основанный на массиве ARRAYED_STACK:

```
class ARRAYED_STACK [G] inherit
  STACK [G]
    redefine change_top end
  ARRAY [G]
    rename
      count as capacity, put as array_put
    export
      {NONE} all
    end
  feature
    ... Реализация отложенных программ STACK, таких как put, count, full...
    ... и переопределение change_top в терминах операций ARRAY ...
  end
```

Интересно сравнить представленную схему класса ARRAYED_STACK с классом STACK2 из предыдущих обсуждений (см. [лекцию 11](#) курса "Основы объектно-ориентированного программирования") - реализацию стека массивом, но без использования наследования. Заметьте, устранение необходимости быть клиентом ARRAY упрощает нотацию (предыдущая версия должна была использовать вызов в форме implementation.put, теперь можно писать просто put).

При наследовании все компоненты ARRAY были сделаны закрытыми. Это типично при браках по расчету: все компоненты родителя, обеспечивающего спецификацию, здесь STACK, экспортируются; все компоненты родителя, обеспечивающего реализацию, здесь ARRAY, скрываются. Это вынуждает клиентов класса ARRAYED_STACK использовать соответствующие экземпляры только через компоненты стека.

Это выглядит привлекательно, но правильно ли это?

Наследование реализации подвергается критике. Скрытие компонентов кажется нарушением отношения "is-a".

Это не так. У этого отношения есть разные формы. По своему поведению стек, основанный на массиве, ведет себя как стек. По внутреннему представлению он массив, и экземпляры ARRAYED_STACK отличаются от экземпляров ARRAY лишь обогащением за счет атрибута (count). Экземпляры, создаваемые по единому образцу, представляют достаточно строгую форму отношения "is-a". И дело не только в представлении: все компоненты ARRAY, такие как put (переименованный в array_put), infix "@" и count (переименованный capacity), доступны для ARRAYED_STACK, хотя и не экспортируются его клиентам. Классу они необходимы для реализации компонентов STACK.

Так что концептуально ничего ошибочного нет в наследовании в интересах реализации. Показательно сравнение с контрпримером, изучаемым в начале этой лекции, класс CAR_OWNER свидетельство непонимания концепции наследования, класс ARRAYED_STACK задает хорошо определенную форму отношения "is-a".

Один недостаток здесь есть: механизм наследования, позволяющий ограничить доступность экспорта наследуемых компонентов (другими словами, разрешение предложения export), делает более трудной статическую проверку типов. Но это трудности разработчиков компиляторов, а не разработчиков прикладного ПО.

Как это делается без наследования

Давайте проверим, как можно выполнить эту работу без использования наследования. Для нашего примера это было уже сделано в классе STACK2 из предыдущих лекций. Он имеет атрибут representation типа ARRAY [G] и процедуры стека, реализованные в следующем виде (утверждения опущены):

```
put (x: G) is
    -- Добавляет x на вершину
    require
        ...
    do
        count := count + 1
        representation.put (count, x)
    ensure
        ...
end
```

Каждая манипуляция с представлением требует вызова компонента ARRAY с representation как цели. Платой являются потери производительности: минимальные по памяти (атрибут representation), более серьезные по времени (связанные с representation, накладные расходы, добавляемые при вызове каждой операции).

Предположим, что проблемы эффективности можно игнорировать. Остается еще одна утомительная необходимость выписывать перед каждой операцией префикс "representation". Это придется делать для всех классов, реализующих различные структуры данных - стеки, списки, очереди, все, что реализуется массивами.

ОО-разработчики ненавидят утомительные, повторяющиеся операции. "Встроенное повторение" - вот наш девиз. Если некий образец повторно встречается в множестве классов, естественной и здоровой реакцией является попытка понять общую абстракцию и встроить ее в класс. Абстракция здесь нечто подобное "структуре данных, имеющей доступ к массиву и его операциям".

```
indexing
    description: "Объекты, имеющие доступ к массиву и его операциям"
class
    ARRAYED [G]
feature -- Access
    item (i: INTEGER): G is
        -- Элемент представления с индексом i
    require
        ...
    do
        Result := representation.item (i)
    ensure
        ...
end
feature -- Element change
    put (x: G; i: INTEGER) is
        -- Замена на x элемента с индексом i
    require
        ...
    do
        representation.put (x, i)
    ensure
```

```

    ...
end
feature {NONE} -- Implementation
  representation: ARRAY [G]
end

```

Компоненты `item` и `put` экспортаны. Так как `ARRAYED` описывает только внутренние свойства структуры данных, нет реальной необходимости в экспортаных компонентах. Так что тот, кто не согласен с самой идеей разрешения потомкам скрывать некоторые из экспортаных компонентов, может предпочесть сделать закрытыми все компоненты `ARRAYED`. По умолчанию они тогда будут скрытыми и у потомков.

При таком определении класса не вызывает споров, что классы, такие как `ARRAYED_STACK` или `ARRAYED_LIST`, становятся наследниками `ARRAYED`: они действительно описывают структуры на массивах. Эти классы могут теперь использовать `item` вместо `representation.item` и так далее; мы избавились от утомительного повторения.

Но минуточку! Если наследовать от `ARRAYED` представляется правильным, почему же нельзя непосредственно наследовать от `ARRAY`? Никакой выгода от введения еще одного слоя, надстроенного над `ARRAY`. Введение `ARRAYED` позволило убедить себя, что наследование реализации не используется, но по соображениям практики мы пошли на это, сделав систему более сложной и менее эффективной.

На самом деле нет никаких причин для введения класса `ARRAYED`. Прямое наследование реализации от классов, подобных `ARRAY`, проще и легитимнее.

Льготное наследование

При льготном наследовании мы еще менее щепетильны, чем при наследовании реализации. Чистый расчет руководит нами при вступлении в брак. Мы видим класс с полезными свойствами и хотим использовать его. Здесь нет ничего предсудительного, поскольку таково назначение класса.

Использование кодов символов

Библиотека Base Libraries включает класс `ASCII`:

```

indexing
  description:
    "Множество символов ASCII. %
    %Этот класс - предок всех классов, нуждающихся в его свойствах."
class ASCII feature -- Access
  Character_set_size: INTEGER is 128; Last_ascii: INTEGER is 127
  First_printable: INTEGER is 32; Last_printable: INTEGER is 126
  Letter_layout: INTEGER is 70
  Case_diff: INTEGER is 32
    -- Lower_a - Upper_a
  ...
  Ctrl_a: INTEGER is 1; Soh: INTEGER is 1
  Ctrl_b: INTEGER is 2; Stx: INTEGER is 2
  ...
  Blank: INTEGER is 32; Sp: INTEGER is 32
  Exclamation: INTEGER is 33; Doublequote: INTEGER is 34
  ...
  ...
  Upper_a: INTEGER is 65; Upper_b: INTEGER is 66
  ...
  Lower_a: INTEGER is 97; Lower_b: INTEGER is 98
  ... и т.д. ...
end

```

Этот класс является хранилищем множества константных атрибутов (всего 142 компонента), описывающих свойства множества ASCII.

Рассмотрим, например, лексический анализатор, ответственный за идентификацию лексем входного текста. Лексемами текста, написанного на некотором языке программирования, являются целые, идентификаторы, символы и так далее. Одному из классов системы, скажем, `TOKENIZER`, необходим доступ к кодам символов для их классификации на цифры, буквы и т. д. Такой класс воспользуется льготами и наследует эти коды от ASCII:

```

class TOKENIZER inherit ASCII feature
  ... Программы класса могут использовать компоненты Blank, Case_diff и другие...
end

```

К классам, подобным ASCII, относятся иногда неодобрительно, но прежде чем перейти к методологической дискуссии, взглянем на еще один пример льготного наследования.

Итераторы

Второй пример демонстрирует наследование программ общего вида, а не константных атрибутов.

Предположим, мы хотим обеспечить общий механизм, позволяющий просмотр всех элементов (итерирование) некоторой структуры данных, например линейных структур, таких как списки. "Итерирование" означает выполнение некоторой процедуры, скажем, *action*, на элементах этой структуры, просматриваемых в последовательном порядке. Нам хочется обеспечить несколько различных механизмов итерирования, включающих применение *action* ко всем элементам, удовлетворяющим условию, заданному булевой функцией *test*, ко всем элементам до появления первого, удовлетворяющего *test*, или до первого, не удовлетворяющего этой функции. Ну и так далее, вариантов можно придумать много. Система, использующая этот механизм, должна быть способна применять его к произвольным компонентам *action* и *test*.

С первого взгляда может показаться, что итерирующие компоненты должны принадлежать классам, описывающим соответствующие структуры данных, таким как LIST или SEQUENCE.

В упражнении У6.7 предлагается показать, что это неправильное решение.

Предпочтительнее ввести независимую иерархию итераторов, показанную на [рис. 6.11](#).

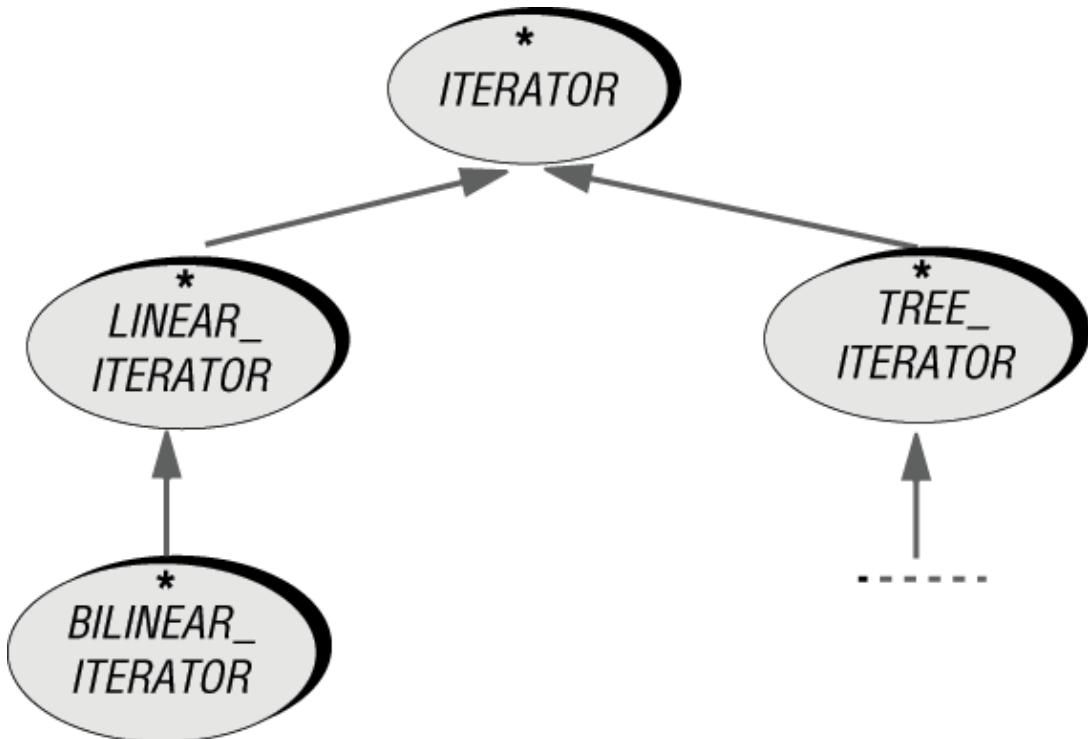


Рис. 6.11. Иерархия итераторов

Класс **LINEAR_ITERATOR**, один из наиболее интересных классов в этом обсуждении, выглядит так:

```
indexing
  description:
    "Объекты, допускающие итерирование на линейных структурах"
    names: iterators, iteration, linear_iterators, linear_iteration
  deferred class LINEAR_ITERATOR [G] inherit
    ITERATOR [G]
    redefine target end
  feature -- Access
    invariant_value: BOOLEAN is
      -- Свойство, сопровождающее итерацию (по умолчанию: true)
      do
        Result:= True
      end
    target: LINEAR [G]
    -- Структура, к которой будут применяться компоненты итерации
    test: BOOLEAN is
      -- Булево условие выбора применимых элементов
      deferred
    end
  feature - Basic operations
    action is
      -- Действие на выбранных элементах
      deferred
```

```

    end
do_if is
    -- Применить action в последовательности к каждому элементу
    --target, удовлетворяющему test.
    do
        from start invariant invariant_value until exhausted loop
            if test then action end
            forth
        end
    ensure then
        exhausted
    end
...
... И так далее: do_all, do_while, do_until и другие процедуры ...
end

```

Рассмотрим теперь класс, нуждающийся в выполнении некоторой операции над выбранными элементами списка специального типа. Например, это может быть командный класс в системе обработки текстов, нуждающийся в проверке всех абзацев документа, за исключением специально отформатированных (подобных абзацам с текстом программ). Тогда:

```

class JUSTIFIER inherit
    LINEAR_ITERATOR [PARAGRAPH]
    rename
        action as justify,
        test as justifiable,
        do_all as justify_all
    end
feature
    justify is
        do ... end
    justifiable is
        -- Подлежит ли абзац проверке?
        do
            Result := not preformatted
        end
    ...
end

```

Переименование облегчает понимание. Заметьте, нет необходимости в объявлении или повторном объявлении процедуры `justify_all` (бывшей `do_all`): будучи наследуемой, ожидаемая работа будет проделана эффективными версиями `action` и `test`.

Процедура `justify`, вместо того чтобы быть описанной в классе, может наследоваться от другого родителя. В этом случае множественного наследования будет выполняться операция объединения ("join"), эффективизирующая отложенную `action`, наследуемую от одного родителя под именем `justify` (здесь переименование существенно), с эффективной `justify`, наследуемой от другого родителя. Реально, это и есть брак по расчету.

Класс `LINEAR_ITERATOR` является замечательным примером **класса поведения (behavior class)**, рассматривая общее поведение и оставляя открытыми специфические компоненты, так чтобы его потомки могли подключить специальные варианты.

Формы льготного наследования

Два примера, `ASCII` и `LINEAR_ITERATOR`, демонстрируют два главных варианта льготного наследования:

- **наследование констант**, в котором принципиальным вкладом родителя являются константные атрибуты и разделяемые объекты;
- **наследование операций**, в котором вкладом являются подпрограммы.

Как отмечалось ранее, возможна комбинация этих вариантов в единой наследственной связи. Вот почему льготное наследование задается одной категорией, а не двумя.

Понимание льготного наследования

Некоторые рассматривают льготное наследование как злоупотребление механизмом наследования, как некоторую форму хакерства.

Главный вопрос, заслуживающий рассмотрения, связан не столько с наследованием, сколько с тем, как определены классы `ASCII` и `LINEAR_ITERATOR`. Как всегда, при рассмотрении проекта класса, следует спросить себя:

"Действительно ли мы описали значимую абстракцию данных - множество объектов, характеризуемых абстрактными свойствами?"

Для этих примеров ответ менее очевиден, чем для классов `RECTANGLE`, `BANK_ACCOUNT` или `LINKED_LIST`, но, по

сугубо, он тот же:

- Класс ASCII представляет абстракцию: "любой объект, имеющий доступ к свойствам множества ASCII".
- Класс LINEAR_ITERATOR представляет абстракцию: "любой объект, способный выполнять последовательные итерации на линейной структуре". Такой объект имеет тенденцию быть "абстрактной машиной", описанной в [лекции 5](#).

Как только эти абстракции принимаются, наследственные связи не вызывают никаких проблем: экземпляру TOKENIZER необходим "доступ к свойствам множества ASCII", а экземпляр JUSTIFIER способен "выполнять последовательные итерации на линейной структуре". Фактически можно было бы классифицировать такие примеры как наследование подтипов. Что отличает льготное наследование, так это природа родителей. Эти классы являются исходными, не использующими наследование. И класс приложения может предпочесть быть их клиентом, а не наследником. Это утяжеляет подход, особенно для класса ASCII:

```
charset: ASCII
...
create charset
```

При каждом использовании кода символа потребуется задавать целевой объект `charset.Lower_a`. Присоединяемый объект `charset` не играет никакой полезной роли. Те же комментарии справедливы и для класса LINEAR_ITERATOR. Но если классу необходимы несколько видов итерации, то тогда создание объектов-итераторов с собственными версиями `action` и `test` становится разумным.

Коль скоро мы хотим иметь объекты-итераторы, то нам нужны итераторные классы, и нет никаких причин отказывать им в праве вступления в клуб наследования.

Множественные критерии и наследование видов

Вероятно, наиболее трудная проблема наследования возникает при наличии альтернативных критериев классификации.

Классификация при множественных критериях

Традиционная классификация в естественных науках использует единственный критерий (возможно, объединяющий несколько качеств) на каждом уровне: позвоночные или беспозвоночные, ветви, обновляемые один или несколько раз в год, и тому подобное. Результатом будет то, что называется иерархией единственного наследования, чье главное преимущество - простота классификации. Конечно, возникают проблемы, поскольку природа определенно не пользуется единственным критерием. Это очевидно для всякого, кто когда-либо пытался провести классификацию, вооружившись книгой по ботанике с традиционной классификацией Линнея.

При разработке ПО, где единый критерий кажется ограничительным, мы можем использовать все приемы множественного и особенно дублирующего наследования, которыми мы овладели при изучении предыдущих лекций. Рассмотрим, например, класс EMPLOYEE в системе управления персоналом. Предположим также, что у нас есть два различных критерия классификации служащих:

- по типу контракта: временные или постоянные работники;
- по типу исполняемой работы: инженерная, административная, управленческая.

Оба эти критерия приводят к правильным классам-потомкам. При этом мы не впадаем в таксонамию, так как идентифицируемые классы, такие как TEMPORARY_EMPLOYEE по первому критерию и MANAGER по второму, действительно характеризуются специальными компонентами, не применимыми к другим категориям. Как же следует поступать?

В первой попытке введем все варианты на одном и том же уровне ([рис. 6.12](#)).

Для простоты на этой схеме имена классов сокращены. В реальной системе мы действуем более аккуратно и используем, как положено, полные имена, такие как PERMANENT_EMPLOYEE, ENGINEERING_EMPLOYEE и так далее.

Получившуюся иерархию наследования нельзя признать удовлетворительной, так как различные концепции представлены классами одного уровня.

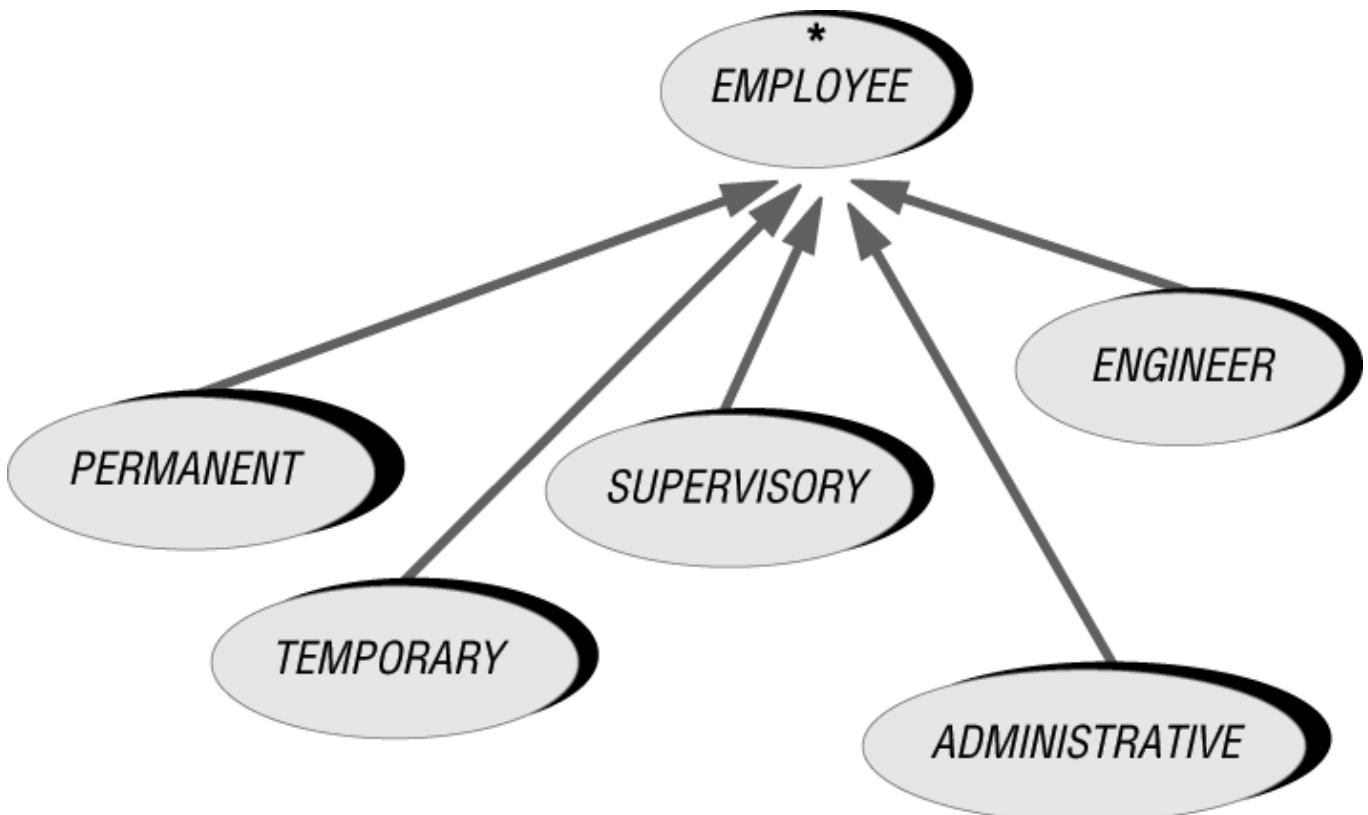


Рис. 6.12. Беспорядочная классификация

Наследование вида

Помня об идеях использования наследования для классификации, следует ввести промежуточный уровень, описывающий конкурирующие критерии классификации ([рис. 6.13](#)).

Появились два вида служащих. Заметьте, имя CONTRACT_EMPLOYEE не означает служащего, имеющего контракт, а служащего, характеризуемого контрактом (он может не иметь контракта!). Имя класса для другого вида означает "служащий, характеризуемый своей специальностью".

То, что эти имена кажутся неестественными, отражает определенную сложность, характерную для наследования видов. При наследовании подтипов мы встречались с правилом, устанавливающим, что экземпляры наследников принадлежат непересекающимся подмножествам множества, заданного родителем. Здесь это правило неприменимо. Постоянный служащий имеет специальность и может быть инженером. Такая классификация подходит для дублирующего наследования: некоторые потомки классов, показанных на рисунке, будут иметь в качестве предков CONTRACT_EMPLOYEE и SPECIALTY_EMPLOYEE не напрямую, но через наследование от классов PERMANENT и ENGINEER. Такие классы будут дублируемыми потомками EMPLOYEE.

Эта форма наследования может быть названа наследованием видов: различные наследники некоторого класса представляют не непересекающиеся подмножества его экземпляров, но различные способы классификации экземпляров родителя. Заметьте, это имеет смысл только при условии, что родитель и наследники являются отложенными классами, говоря другими словами, классами, описывающими общие категории, а не полностью специфицированные объекты. Наша первая попытка классификации EMPLOYEE по видам (та, у которой все потомки на одном уровне) нарушает это правило, вторая ему удовлетворяет.

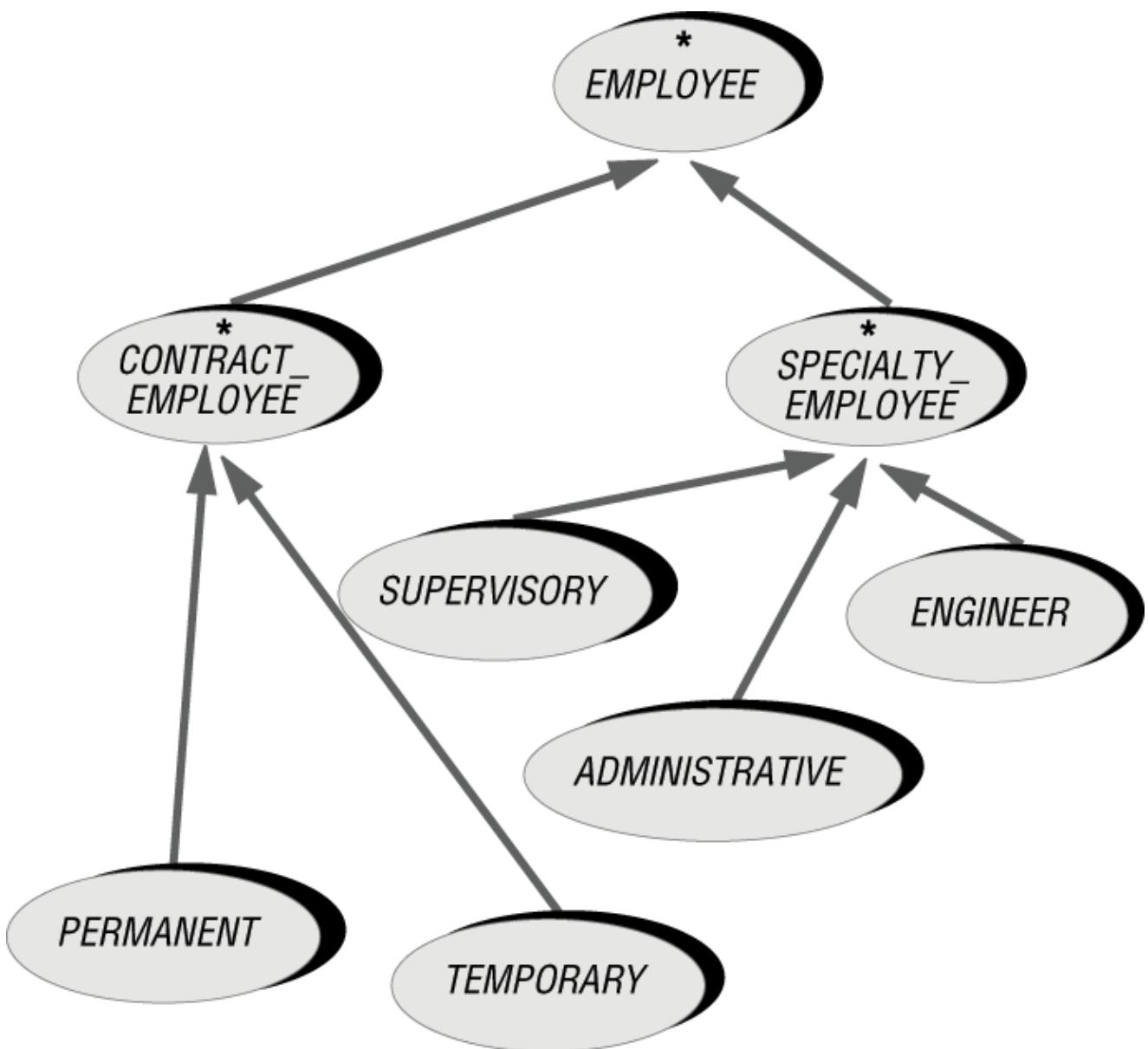


Рис. 6.13. Классификация, использующая виды

Подходит ли нам наследование видов?

Наследование видов не является общеприменимым и представляет собой объект для критики. Читатель может сам судить, стоит ли его использовать при решении возникающих у него проблем, но в любом случае необходимо разобрать все аргументы за и против.

Прежде всего должно быть ясно, что, подобно дублируемому наследованию, наследование видов **не является механизмом для новичков**. Правило осмотрительности, введенное для дублируемого наследования, справедливо и здесь: если ваш опыт разработки ОО-проектов измеряется несколькими месяцами, избегайте наследования типов.

Альтернативой наследованию типов служит выбор одного критерия в качестве первичного, он и будет руководить построением иерархии. Для учета других критериев следует использовать специальные компоненты класса. Стоит отметить, что современные зоологи и ботаники используют именно такой подход: их основной критерий классификации основан на реконструкции эволюционной истории, включающей деление на роды и виды. Значит ли это, что мы всегда имеем единый, бесспорный стандарт, руководящий нами при создании программистских таксономий?

Чтобы в нашем примере придерживаться единого критерия, мы могли бы принять решение, что тип работы служащего является более важным фактором, а статус контракта задать компонентом. Рассмотрим первую попытку введения в класс **EMPLOYEE** такого компонента:

```
is_permanent: BOOLEAN
```

Но такое решение накладывает серьезные ограничения. Расширяя возможности, приходим к варианту:

```

Permanent: INTEGER is unique
Temporary: INTEGER is unique
Contractor: INTEGER is unique
...
  
```

Но это означает, что мы сталкиваемся с явным перечислением, и лучшим подходом является введение класса `WORK_CONTRACT`, как правило, отложенного, имеющего потомков по числу видов контракта. Тогда мы сможем избежать явного разбора случаев в форме:

```
if is_permanent then ... else ... end
```

или

```
inspect
    contract_type
when Permanent then
    ...
when ...
    ...
end
```

Как неоднократно говорилось, разбор случаев приводит к ряду проблем при появлении новых вариантов и нарушает важные принципы непрерывности, единого выбора, открытость-закрытость и так далее. Вместо этого мы поставляем класс WORK_CONTRACT с отложенными компонентами, представляющими операции, зависящие от типа контракта, которые по-разному будут реализованы потомками. Большинству из этих компонентов будет необходим аргумент типа EMPLOYEE, представляющий служащего, к которому применяется операция, примерами операций могут быть *hire* (приглашение на работу) и *terminate* (увольнение).

Результирующая структура показана на [рис. 6.14](#).

Эта схема, как вы заметили, почти идентична образцу проектирования с **описателями**, изучаемому ранее в этой лекции.

Такая техника может использоваться вместо наследования видов. Это усложняет структуру из-за введения независимой иерархии, нового атрибута (здесь *contract*) и соответствующего клиентского отношения. Преимущество ее в том, что по поводу иерархии не возникает вопросов. В то же время при наследовании видов абстракции требуют больших пояснений (служащий, рассматриваемый с позиций его специальности или контракта).

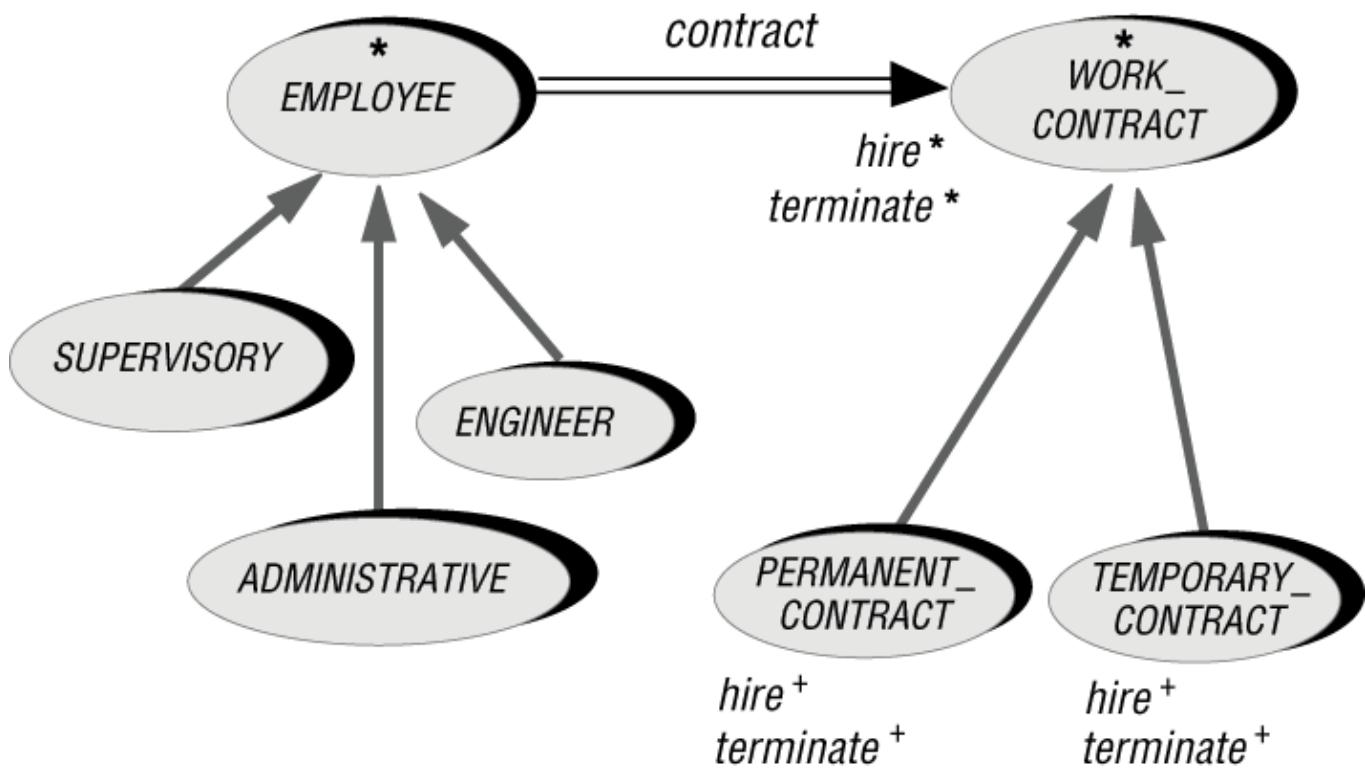


Рис. 6.14. Многокритериальная классификация с независимой иерархией, построенной для клиента

Критерии для наследования видов

Нет ничего необычного в рассмотрении наследования видов на ранних этапах анализа проблемной области, когда обсуждаются фундаментальные концепции и рассматриваются несколько равно привлекательных критерий классификации. В дальнейших исследованиях часто оказывается, что один из критериев начинает доминировать, выступая в качестве основы построения иерархической структуры. Тогда, как показывает наше обсуждение, следует отказаться от наследования типов в пользу построенной нами схемы.

Все же я нахожу наследование видов полезным при выполнении следующих трех условий:

- Различные критерии классификации одинаково важны, так что выбор одного в качестве основного представляется спорным.

- Многие возможные комбинации (такие как в примере: permanent supervisor, temporary engineer, permanent engineer и так далее) являются необходимыми.
- Рассматриваемые классы настолько важны, что стоит потратить время на разработку лучшей из возможных структур наследования. Чаще всего речь идет в таких случаях о библиотечных классах повторного использования.

Примером приложения, удовлетворяющего этим критериям, является библиотека Base с ее структурой иерархии на верхних уровнях, описанная в последней лекции этой книги. Классы, полученные в результате этих усилий, в деталях описаны в [М 1994а]. Они построены в традиции естественных наук с применением таксономических принципов систематической классификации основных программистских структур. Верхняя часть этой иерархии выглядит так:

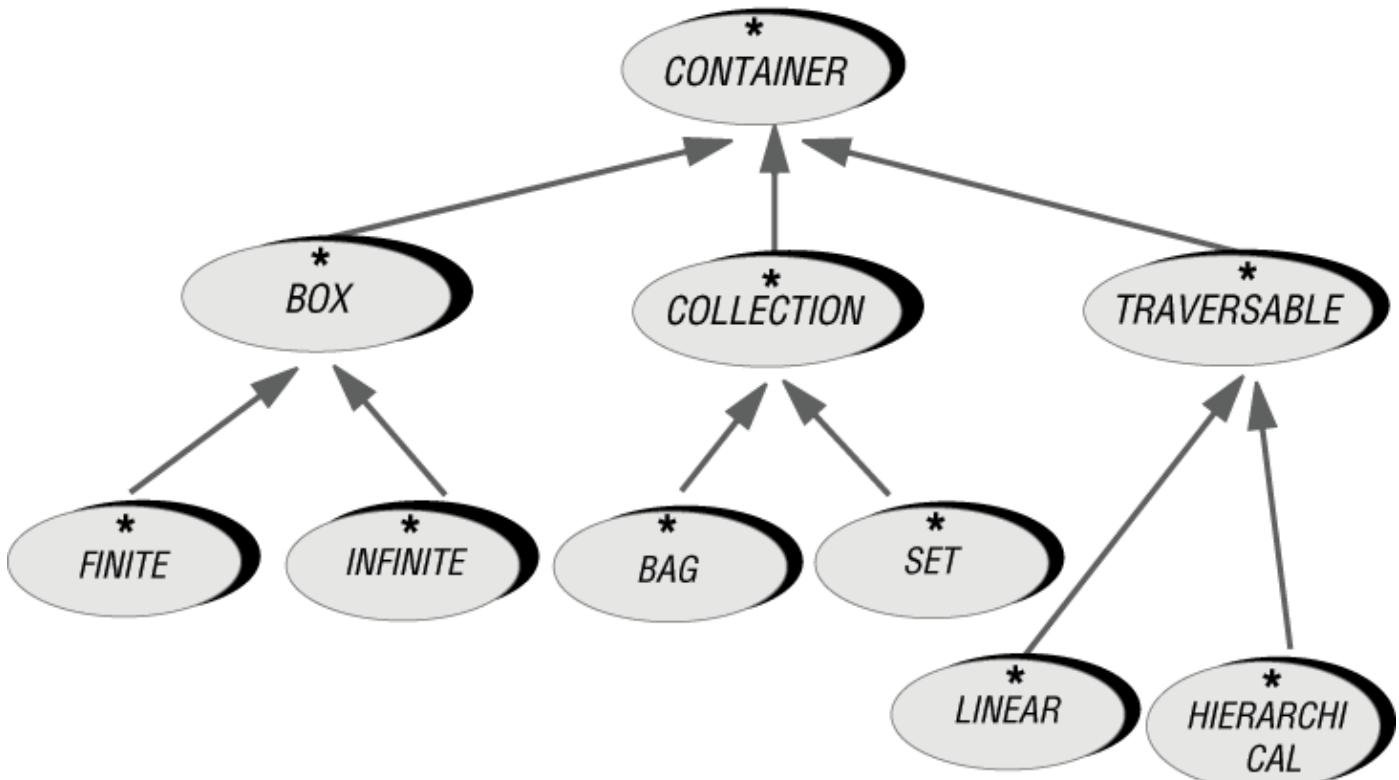


Рис. 6.15. Классификация, основанная на видах фундаментальных программистских структур

Классификация на первом уровне (BOX, COLLECTION, TRAVERSABLE) основана на типах; уровень ниже (и многие другие, не показанные на рисунке) задают классификацию подтипов. Структура контейнера характеризуется тремя различными критериями:

- COLLECTION определяет доступ к элементам. Класс SET позволяет определить сам факт присутствия элемента, в то время как BAG позволяет также посчитать число вхождений данного элемента. Дальнейшие уточнения включают такие абстракции доступа, как SEQUENCE (элементы доступны последовательно), STACK (элементы доступны в порядке, обратном их включению) и так далее.
- BOX определяет представление элементов. Варианты включают конечные и бесконечные структуры. Конечные структуры могут быть ограниченными и не ограниченными. Ограниченные структуры могут быть фиксированными или изменяемого размера.
- TRAVERSABLE определяет способы обхода структур.

Интересно отметить, что эта иерархия не начиналась, как иерархия видов. Начальная идея состояла в том, чтобы определить BOX, COLLECTION и TRAVERSABLE как несвязанные классы, каждый, задающий вершину своей независимой иерархии. Затем при описании реализации любой специальной структуры данных использовать множественное наследование с родителями из каждой иерархии. Например, связный список является конечным и неограниченным с последовательным доступом и линейным способом обхода.

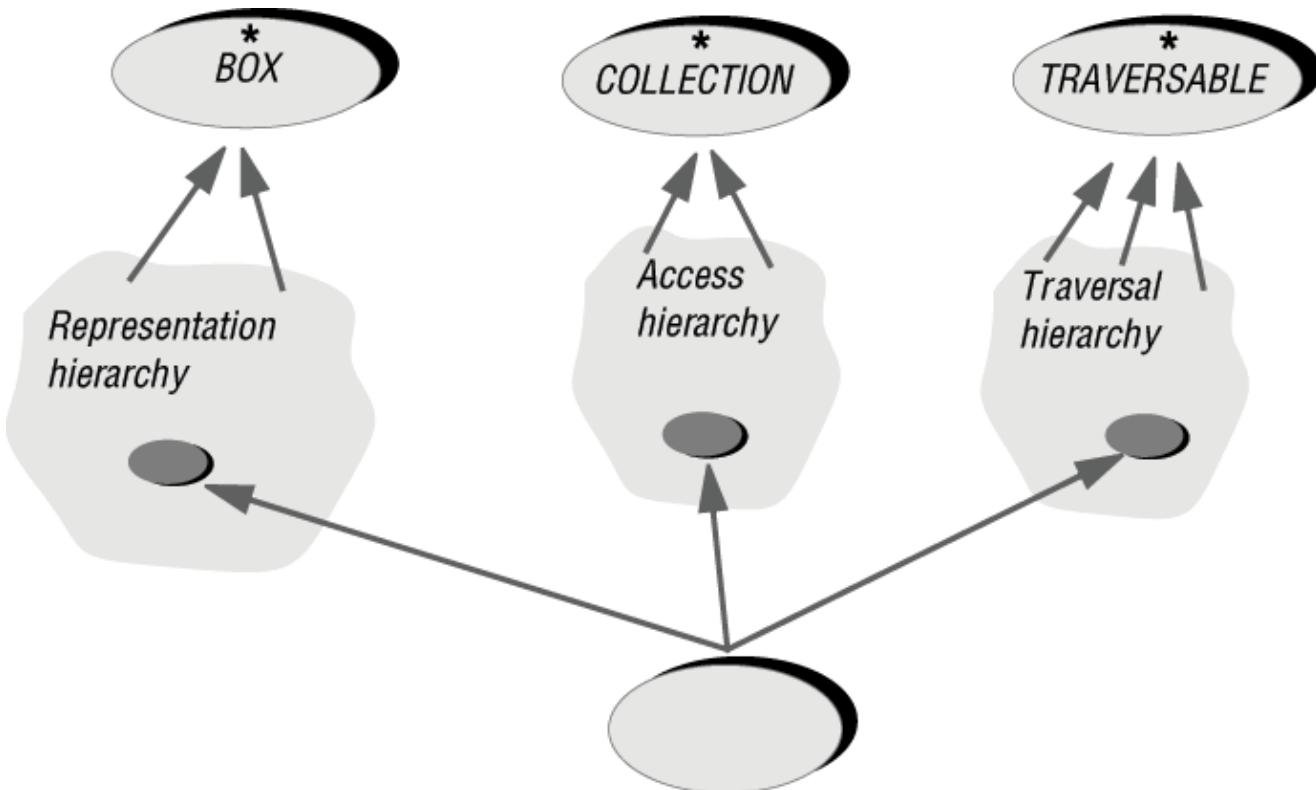


Рис. 6.16. Построение структуры данных комбинированием абстракций путем множественного наследования

Но затем мы осознали, что независимые семейства классов BOX, COLLECTION и TRAVERSABLE не лучший способ: им всем потребовались некоторые общие компоненты, в частности has (тест на проверку членства) и empty (тест на отсутствие элементов). Все это указывало на необходимость иметь общего родителя - CONTAINER, где эти общие свойства теперь и появляются. Следовательно, структура, изначально спроектированная как чистое множественное наследование с тремя непересекающимися иерархиями, превратилась в структуру с наследованием типов, приводящую к дублируемому наследованию.

Изначально трудно было сделать все сразу правильным, но со временем структура стала гибкой, стабильной и полезной. Она подтверждает заключение нашего обсуждения: наследование видов не для слабонервных. Когда оно применимо, то играет ключевую роль в сложных проблемных областях, где взаимодействуют многие критерии. Если усилия по ее созданию оправданы, как при создании фундаментальных библиотек повторно используемых компонентов, то их необходимо совершить.

Как разрабатываются структуры наследования

При чтении книги или учебной статьи по ОО-методу или при обнаружении библиотек классов с уже спроектированной иерархией наследования авторы не всегда говорят о том, как они пришли к конечному результату. Что же следует делать при проектировании собственных структур?

Специализация и абстракция

Произвольно или нет, но многие учебные презентации создают впечатление, что структуру наследования следует проектировать от наиболее общего (верхней ее части) к более специфическим частям (листьям). В частности, это происходит потому, что лучший способ **описать** существующую структуру - это идти от общего к частному, от фигур к замкнутым фигурам, затем к многоугольникам, прямоугольникам, квадратам. Но лучший способ описания структуры вовсе не означает, что он является и лучшим способом ее **создания**.

Подобный комментарий, сделанный Майклом Джексоном, упоминался при рассмотрении проектирования сверху вниз.

В идеальном мире, населенном совершенными людьми, мы бы сразу же обнаруживали правильные абстракции, выводили бы из них категории, их подкатегории и так далее. В реальном мире, однако, мы часто вначале обнаруживаем специальный случай и лишь потом открываем общую абстракцию.

Во многих ситуациях абстракция не является уникальной; как лучше обобщить некоторое понятие, зависит от того, что вы и ваши клиенты хотите сделать с этим понятием и его вариантами. Рассмотрим, например, понятие, неоднократно встречающееся в наших рассмотрениях, - точку в двумерном пространстве. Возможны по меньшей мере четыре обобщения:

- точки в пространстве произвольной размерности, приводящие к наследственной структуре, где братьями класса POINT будут классы POINT_3D и так далее;
- геометрические фигуры - другими классами структуры могут быть FIGURE, RECTANGLE, CIRCLE и так далее;
- многоугольники - с такими классами, как QUADRANGLE (четыре вершины), TRIANGLE (три вершины) и SEGMENT (две вершины), POINT является специальным случаем, имеющим ровно одну вершину;

- объекты, полностью определяемые двумя координатами - другими кандидатами являются комплексные числа и двумерные векторы COMPLEX и VECTOR_2D.

Интуитивно некоторые из этих обобщений кажутся более приемлемыми, чем другие, но невозможно со всей определенностью выбрать наилучшее. Ответ зависит от потребностей. Потому предусмотрительный и осторожный процесс, в котором абстракция создается с некоторым опозданием, чтобы точно убедиться в правильном выборе пути обобщения, может быть предпочтительнее скорых и быстрых решений, приводящих к непроверенной абстракции.

Произвольность классификации

Пример класса POINT типичен. Когда сталкиваешься с двумя конкурирующими классификациями из некоторого множества абстракций, то часто можно привести разумные аргументы в пользу одной из них. Значительно реже кто-то может утверждать, что данная структура является наилучшей из всех возможных.

Эта ситуация не является спецификой разработки ПО. Классификация Линнея не является универсально приемлемой или непреложной. У нее есть соперники, один из которых вместо традиционного эволюционного критерия использует другой, более индуктивный, основанный на DNA-анализе и приводящий к совершенно другим результатам. Есть зоологи, для которых умение (неумение) птиц летать является важным таксономическим признаком, но официальная классификация с этим не соглашается.

Индукция и дедукция

При проектировании семейства классов программной системы подходящим процессом является комбинация дедукции и индукции, специализации и обобщения. Иногда вы начинаете с абстракции, а затем выводите частные случаи, иногда вы обнаруживаете полезный класс, а затем осознаете существование более общей абстрактной концепции.

Если вы обнаруживаете абстракцию только после распознавания конкретного, **возможно, с вами все в порядке**. Вы просто используете нормальный подход к классификации. С приобретением опыта и появлением интуиции возрастет доля априорных решений. Но апостериорная составляющая всегда останется.

Разнообразие абстракции

Этот принцип атавизма - один из наиболее удивительных из всех атрибутов наследования.

Чарльз Дарвин

Две формы апостериорного конструирования родителя являются общими и полезными.

Абстрагирование представляет позднее обнаружение концепции высшего уровня. Вы находите класс B, покрывающий полезное понятие, но чей разработчик не обнаружил, что это фактически специальный случай общего понятия A, для которого оправдана наследственная связь:

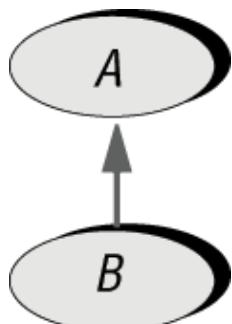


Рис. 6.17. Абстракция

То, что это понимание не пришло сразу, другими словами, то, что B был построен без учета A, - не является причиной отказа от наследования в этом случае. Сразу же при обнаружении необходимости A вы можете, а в большинстве случаев должны написать этот класс и адаптировать B как его наследника. Это не столь хорошо, как написать раньше A, но лучше, чем не написать вовсе.

Факторизация возникает в случае обнаружения того, что два класса E и F фактически представляют варианты одного и того же понятия:

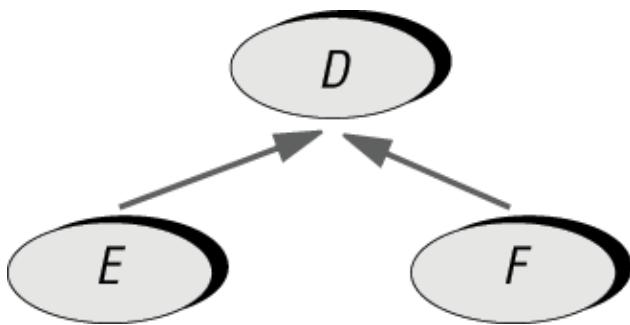


Рис. 6.18. Факторизация

Если вы с запозданием обнаружили эту общность, то шаг обобщения позволит добавить общий родительский класс D. Здесь снова предпочтительнее построить иерархию сразу же, но лучше позже, чем никогда.

Независимость клиента

Абстрагирование и факторизация могут во многих случаях выполняться без негативных последствий для существующих клиентов (приложение принципа Открыт-Закрыт). Это свойство является результатом использования скрытия информации. Рассмотрим снова предшествующие схематические случаи, но с типичным клиентским классом X, показанным на рисунке:

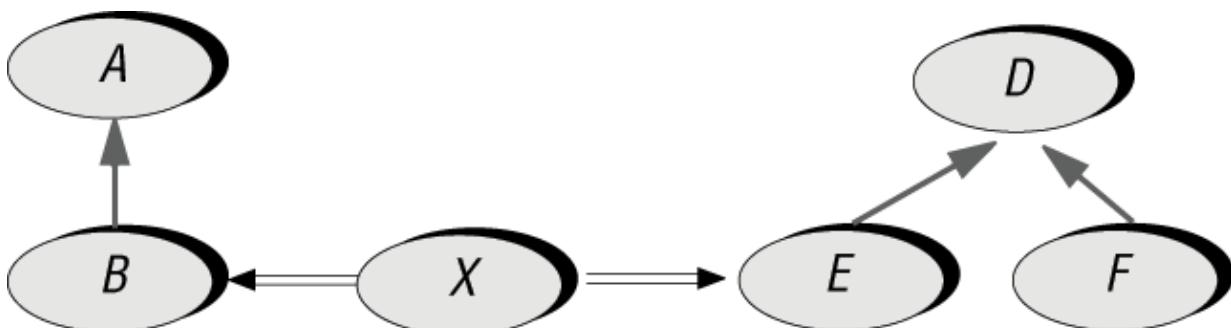


Рис. 6.19. Абстракция, факторизация и клиенты

Когда B абстрагируется в A, или компоненты E факторизуются с компонентами F в D, класс X, представляющий клиента В или Е (на рисунке он клиент обоих классов) в большинстве случаев не заметит никаких изменений. Включение класса в схему наследования не оказывает влияния на его клиентов, если они применяют компоненты класса на сущностях соответствующего типа. Другими словами, если X использует B и E как поставщиков по схеме:

```

b1: B; e1: E
...
b1.some_feature_of_B
...
e1.some_feature_of_E

```

то X не заметит, что B или E обрели родителей в результате абстрагирования или факторизации.

Совершенствование уровня абстракции

Абстрагирование и факторизация являются типичным процессом постоянных улучшений, характерных при успешном конструировании ОО-ПО. На основании своего опыта могу сказать, что это один из наиболее воодушевляющих аспектов практики метода: знание того, что при всей невозможности достичь совершенства с самого начала вам дается возможность улучшать ваш проект постоянно, пока он не будет удовлетворять каждого.

В группе разработчиков, правильно применяющих ОО-метод, это регулярное совершенствование уровня абстракции ПО и, как следствие, его качества, заметно ощущается членами команды и служит источником постоянной мотивации.

Итоговый обзор: используйте наследование правильно

Наследование никогда не перестанет удивлять нас своей мощью и универсальностью. В этой лекции мы пытались как можно лучше описать, что реально означает наследование и как можно его использовать для получения всех преимуществ. Перечислю несколько центральных выводов.

Прежде всего не следует бояться разнообразия способов использования наследования. Запрет множественного или льготного наследования не достигает никакой другой цели, кроме нанесения вреда самому себе. Механизмы должны помогать нам, используйте их правильно, но используйте их!

Наследование является, как правило, техникой **поставщика**. Это мощное оружие нашего арсенала в сражениях с противниками (в частности сложностью - безжалостным врагом разработчика).

Конечно, все ПО проектируется для клиентов, клиентские потребности управляют процессом. Множество классов

хорошо тогда, когда оно предоставляет замечательные службы клиентам: интерфейсы и ассоциированные с ними реализации, являющиеся полными, свободными от неприятных сюрпризов (таких как неожиданные потери производительности), просты в использовании, легки в обучении, запоминании, допускают расширяемость. Для достижения этих целей проектировщик свободен в использовании наследования и других ОО-методов любым способом, который ему нравится. Победителей не судят.

Напомню также, при проектировании структуры наследования целью является конструирование ПО, а не философия. Редко существует единственное, или лучшее решение. Для нас "лучшее" означает лучшее для целей некоторого класса клиентского приложения. Это особенно верно, когда мы уходим от таких областей как математика или компьютерные науки, где существуют широко применимые теории, и попадаем в мир деловых приложений.

В известной степени в этом есть некоторый комфорт. Натуралист, классифицирующий растения и животных, должен мыслить в абсолютных категориях. В программном мире эквивалентом является создание библиотек общеселевого назначения (фундаментальные структуры данных, графика, базы данных). Большой же частью ваши цели скромны, вам необходимо спроектировать хорошую иерархию, ту, которая удовлетворяет потребностям клиентов.

Заключительный урок этой лекции обобщает комментарий, сделанный при обсуждении льготного наследования: принципиальная трудность построения структуры классов не в наследовании, она в поиске абстракций. Если правильные абстракции идентифицированы, то структура наследования следует из них. Для поиска абстракций следует руководствоваться тем, чем мы руководствуемся на протяжении всей книги, - абстрактными типами данных.

Ключевые концепции

- Каждое использование наследования должно отражать некоторую форму отношения "is a" между двумя категориями объектов, либо из внешней моделируемой области, либо из мира самого ПО.
- Не используйте наследование для моделирования отношения "has"; это область клиентских отношений. (Помните о CAR_OWNER.)
- Когда наследование применимо, часто потенциально применим и клиент. Если соответствующая точка зрения может измениться, используйте клиентское отношение. Если предвидится полиморфное использование, используйте наследование.
- Не вводите промежуточных узлов наследования, если только они не задают хорошо определенные абстракции со своими специфическими компонентами.
- Данная классификация наследования основана на двенадцати видах, разделенных на три общие категории: наследование модели (описывающее отношения, существующие в моделируемой области), программное наследование (описывающее отношения, существующие в самом ПО) и наследование вариаций (для адаптации класса либо в модели, либо в ПО).
- Мощь наследования происходит из комбинации специализации типа и механизма расширения модуля. Было бы немудро использовать различные механизмы языка.
- Наследование реализаций и льготное наследование требуют особой тщательности, но представляют мощную технику, которую следует использовать на стороне поставщика.
- Наследование видов - это тонкая техника, включающая дублирующее наследование, позволяющее классифицировать типы объектов по нескольким конкурирующим критериям. Оно полезно при разработке профессиональных библиотек. Во многих случаях простая техника описателей предпочтительнее.
- Хотя и нет теоретического идеала, фактический процесс проектирования иерархии наследования идет в двух направлениях - от абстрактного к конкретному и наоборот.
- Наследование является главным образом техникой поставщика.

Библиографические замечания

Основным учебником по таксономии наследования является [Girod 1991]. Книга по ОО-методологии [Page-Jones 1995] - одна из немногих, где даются полезные методологические советы, включая ценные рекомендации, когда следует и когда не следует использовать наследование. Еще один полезный учебник Джона Мак-Грегора [McGregor 1992] исследует технику, названную в этой лекции наследованием видов.

В [Breu 1995] также содержатся полезные концепции, основанные на более ограничительном подходе к наследованию, чем в рассматриваемой лекции.

Техника, подобная "описателям" этой лекции, рассмотрена в [Gil 1994].

При подготовке этой статьи использовались комментарии нескольких биологов, ведущих доступные в Интернете Web-ресурсы по таксономии живых существ, в частности "Древо жизни" (<http://www.phylogeny.arizona.edu/tree/life.html>), учтиво предоставленные профессорами Дэвидом Мэдисоном, а для птиц Майклом Лореном. Помощь профессора Эдвина Эверхама из университета Редфорда была весьма полезной.

Руководства по теории классификации или систематике приведены в следующем разделе.

Приложение: история таксономии

Данное приложение является дополнительным материалом и не используется в остальной части курса. Изучение усилий по таксономии в других дисциплинах позволяет извлечь полезные уроки разработчикам ПО. Я надеюсь, что оно послужит толчком к дальнейшим исследованиям в этой интересной области, возможно, станет темой магистерской или докторской диссертации кандидата наук.

Мы прислушиваемся к мнениям биологов, звучащих с двух противоположных сторон: одних, отстаивающих априорную форму классификации, сверху вниз, дедуктивную и основанную на "естественном" порядке вещей, идущую к нам от кладистов, начиная с Линнея, других, отстаивающих эмпирическую, индуктивную точки зрения фенетиков снизу вверх, советующих нам наблюдать и собирать...

Вероятно, подобно эволюционным таксономистам, хотелось бы взять понемногу от тех и от других.

Упражнения

У6.1 Стек, основанный на массиве

Напишите полностью класс STACK и его потомка ARRAYED_STACK, набросок которого дан в этой лекции, используя технику "брата по расчету".

У6.2 Метатаксономия

Представьте себе, что введенная классификация форм наследования представляла бы собой иерархию наследования. Какие виды наследования были бы включены?

У6.3 Стеки Ханоя

(Это упражнение пришло из примера Филиппа Дрикса.)

Рассмотрим отложенный класс STACK с процедурой `put` для вталкивания элемента на вершину с предусловием, включающим булеву функцию `full` (которая также может быть названа `extendible`; когда вы ознакомитесь с упражнением, то заметите, что выбор имени может влиять на возможные решения).

Задача о Ханойских башнях, известная по многим учебникам как пример рекурсивной процедуры, идет от работы Эдварда Лукаса, Париж, 1883 г.

Рассмотрим теперь известную задачу о Ханойских башнях, где нужно перенести пирамиду (башню), составленную из отдельных дисков разного размера, с одного стержня на другой, используя третий, соблюдая правило: диск может быть переложен только на диск большего размера.

Можно ли определить класс HANOI_STACK, представляющий такие пирамиды, как наследника класса STACK? Если да, каким должен быть этот класс? Если нет, может ли HANOI_STACK как-то использовать STACK? Напишите класс полностью для различных возможных решений, обсудите все "за и против" каждого решения. Установите, какое из них предпочтительнее и объясните ваш выбор.

У6.4 Являются ли многоугольники списками?

Реализация нашего примера наследования класса POLYGON использовала атрибут связного списка `vertices` для представления числа сторон многоугольника. Следует ли вместо этого наследовать POLYGON от LINKED_LIST [POINT]?

У6.5 Наследование функциональной вариации

Приведите один или несколько примеров функциональной вариации. Для каждого из них обсудите, дают ли они законный образец принципа Открыт-Закрыт или являются примерами того, что называется "организованным хакерством".

У6.6 Примеры классификации

Для каждого из следующих случаев укажите, к какому виду наследования он относится:

- SEGMENT от OPEN FIGURE;
- COMPARABLE (объекты, поставляемые с отношением полного порядка), наследуемые от PART_COMPARABLE (объекты с отношением частичного порядка);
- некоторые классы EXCEPTIONS.

У6.7 Кому принадлежат итераторы?

Разумно ли компоненты итератора (`while_do` и ему подобные) включать в классы, описывающие структуры данных, которые они итерируют, такие как LIST? Рассмотрите следующие аргументы:

- простоту применения в процессе итерирования подпрограмм `action` и `test`, выбираемых приложением;
- расширяемость: возможность добавления новых схем итерирования;
- общность: выполнение ОО-принципов, в частности той идеи, что операции не существуют сами по себе, но связаны с некоторой абстракцией данных.

У6.8 Наследование типа и модуля

Предположим, мы разрабатываем язык с двумя типами наследования: расширением модуля и подтипами. К какому из этих типов следует отнести категории наследования, идентифицируемые в этой лекции?

У6.9 Наследование и полиморфизм

Из рассмотренных видов наследования этой лекции между родителем A и наследником B, для каких на практике характерно использование полиморфного присоединения, другими словами присваивание $x := y$ или соответствующая передача аргументов x типа A и у типа B?

Основы объектно-ориентированного проектирования

7. Лекция: Полезные приемы

Примеры ОО-проектирования, приведенные в предыдущих лекциях, иллюстрируют несколько характерных приемов. Хотя мы и не закончили наш обзор методологических проблем - нам предстоит еще рассмотрение правил стиля, концепций ОО-анализа, в вопросах обучения, процессов ПО, - сделаем паузу и кратко сформулируем то, что уже изучено. Это будет самая короткая лекция во всей книге. Она просто перечисляет плодотворные идеи, объединяя их в группы с предваряемыми ключевыми словами, напоминая о тех примерах, где мы впервые столкнулись с этими идеями.

Философия проектирования

Общая схема разработки

Разработка снизу вверх: постройте прочный базис, затем применяйте его к специальным случаям.

Бесшовность: применяйте согласованные приемы и инструментарий на этапах анализа, проектирования, разработки и сопровождения.

Обратимость: извлекайте пользу из уроков реализации и корректируйте функциональную спецификацию.

Обобщение: из специализированных классов создавайте повторно используемые классы. Абстрагирование, факторизация - дорога к общности.

Структура систем

Системы создаются только из классов.

Стиль разработки - снизу вверх. Начинайте с того, чем вы располагаете.

Пытайтесь сделать классы с самого начала настолько общими, насколько это возможно.

Пытайтесь сделать классы с самого начала настолько автономными, насколько это возможно.

Два отношения между классами: клиент (с вариантами "ссылающийся клиент" и "развернутый клиент"), наследование. Тесное соответствие с отношениями "has" и "is".

Используйте многослойную архитектуру для разделения абстрактного интерфейса и реализации для различных платформ: Vision, WEL/PEL/MEL.

Эволюция системы

Проектируйте с учетом изменений и повторного использования.

При улучшениях проекта вводите понятие устаревших (obsolete) компонентов и классов для облегчения перехода к новой версии.

Классы

Структура класса

Каждый класс должен соответствовать хорошо определенной абстракции данных.

Подход Списка Закупок: если компонент потенциально полезен и согласуется с абстракцией данных, добавьте его.

Классы, предоставляющие льготы: связанная группа полезных свойств (например множество констант).

Активные структуры данных (объекты как абстрактные машины).

Ключевым решением является задание статуса доступа компонентов: закрытых или экспортируемых.

Используйте выборочный экспорт для группы тесно связанных классов: LINKED_LIST, LINKABLE.

Обновление необъектного ПО: инкапсулируйте абстракции в классы (примером является библиотека Math).

Документация класса

Помещайте в класс настолько много информации, насколько это возможно.

Задавайте заголовочные комментарии тщательно и согласованно; они являются частью официального интерфейса.

Индексируйте классы.

Проектирование компонентов интерфейса

Принцип Разделения Команд и Запросов: функция не должна иметь абстрактного побочного эффекта (конкретный побочный эффект допустим).

В качестве аргументов используйте только операнды.

Установите статус, затем выполняйте операцию.

Для каждой команды, устанавливающей статус, обеспечьте запрос, возвращающий статус.

Для запросов без аргументов внешне не должна быть видима разница в их реализации - атрибутом или функцией.

Допускайте у объектов изменение представления по умолчанию в зависимости от результата запрашиваемой операции (примером является класс комплексных чисел).

Структуры с курсором (LIST, LINKED_LIST и многие другие).

Использование утверждений

Предусловие связывает клиента, постусловие - поставщика.

Делайте предусловие достаточно сильным, чтобы программа могла хорошо делать **свою** работу, - но не сильнее.

Два вида предложений инварианта: некоторые предложения идут от лежащей в основе абстракции данных, другие (**инвариант представления**) описывают согласованные свойства реализации. Используйте инвариант, чтобы выразить и улучшить ваше понимание отношений между различными составляющими класса, в частности атрибутами.

Для запросов без аргументов включайте абстрактные свойства в инвариант (даже если для функции свойство появляется в виде постусловия).

При повторном объявлении допустимо ослабление предусловий, позволяющее расширить область применения подпрограммы.

Для достижения эффекта усиления предусловия используйте абстрактные предусловия (основанные на булевых функциях) в оригинале.

Даже и без необходимости усиления абстрактные предусловия являются предпочтительными.

Любое предусловие должно обеспечиваться и проверяться клиентом перед вызовом компонента.

Не усердствуйте в усилении постусловий - оставьте возможность их усиления потомками (например, можно оставить одностороннюю импликацию **implies** вместо эквивалентности).

Как обращаться со специальными ситуациями

Априорная проверка: до выполнения операции проверяйте возможность ее применения.

Апостериорная проверка: выполните операцию, затем запросите атрибут для выяснения того, как она сработала.

Когда все рушится, используйте обработку исключений.

Организованный отказ: если в конце выполняется предложение **rescue**, не забудьте восстановить инвариант. Вызывающая программа получит также исключение.

Повторение выполнения: испробуйте другой алгоритм или (стратегия надежды) тот же повторно. Сохраните информацию в атрибутах или локальных сущностях, инициализируемых в момент вызова, но не при повторах **retry**.

Техника наследования

Повторные объявления

Переопределяя подпрограмму, используйте специфические алгоритмы для повышения эффективности: `perimeter` в `POLYGON`, `RECTANGLE`, `SQUARE`.

Переопределяйте функцию как атрибут: `balance` в `ACCOUNT`.

Делайте эффективным отложенный компонент родителя.

Объединяйте два или более компонентов через эффективизацию (все, кроме одного, должны быть отложенными, эффективный побеждает). Если нужно, то не переопределяйте некоторые из эффективных компонентов.

Два или более эффективных компонентов можно переопределить совместно.

Доступ к родительской версии при переопределении обеспечивает `precursor`.

Повторные объявления сохраняют семантику (правила утверждений).

Отложенные классы

Отложенные классы описывают категории высокого уровня.

Отложенные классы служат средством анализа и проектирования, описывая абстракции без ссылок на реализацию.

Классы поведения: задают общее поведение. Эффективные подпрограммы вызывают отложенные. Класс является частично отложенным, частично реализованным (охватывает частичный выбор реализации АТД).

Полиморфизм

Полиморфные структуры данных: наследование и универсальность позволяют комбинировать в нужных пропорциях подобие и вариации.

Описатели: благодаря полиморфным атрибутам задают компонент изменяемого типа.

Динамическое связывание: позволяет избежать явного разбора случаев.

Динамическое связывание на полиморфных структурах данных: применяется к каждому элементу структуры операцию, соответствующую элементу.

В точке единственного выбора полезна предварительно вычисленная структура данных с одним объектом каждого возможного типа (как в образце с откатами).

Формы наследования

Убедитесь, что все использования наследования принадлежат одной из категорий в таксономии.

Наследование подтипов.

Наследование, приводящее к расширению модуля.

Брак по расчету: реализация абстракции конкретной структурой.

Наследование с ограничением: добавление ограничения.

Наследование общечелевых механизмов благодаря классам, предоставляющим льготы.

Наследование функциональных вариаций: "организованное хакерство", принцип Открыт-Закрыт.

Наследование вариации типа: ковариантность.

Основы объектно-ориентированного проектирования

8. Лекция: Чувство стиля

Реализация ОО-метода требует обращать внимание на многие детали стиля, считающиеся пустяковыми для менее амбициозного подхода.

Дела косметические!

Хотя правила, представленные здесь, не столь фундаментальны, как принципы ОО-конструирования ПО, было бы глупо рассматривать их просто как "косметику". Хорошее ПО хорошо в **большом и в малом** - в архитектуре высокого уровня и в деталях низкого уровня. Качество деталей еще не гарантирует качества в целом, но небрежность в деталях верный признак более серьезных ошибок. (Если проект не выглядит красивым, то заказчики не поверят, что вы справились с по-настоящему трудным делом.) Серьезный инженерный процесс требует **все** делать правильно: великолепно и современно.

Так что не следует пренебрегать, казалось бы, такими пустяками как форматирование текста и выбор имен. Может показаться удивительным перейти, не снижая уровня внимания, от математических понятий достаточной полноты формальных спецификаций к тому, что символу "точка с запятой" должен предшествовать пробел. Объяснение простое: обе проблемы заслуживают внимания аналогично тому, как при создании качественного ПО следует уделять равное внимание проектированию и реализации.

Некоторые подсказки можно получить, исходя из понятия стиля в его литературном смысле. Хотя, говоря о хорошем произведении, на первом месте стоит способность автора создать соответствующую структуру и сюжет, никакой текст не будет успешным, пока в нем не все отработано: каждый абзац, каждое предложение, каждое слово.

Применение правил на практике

Можно проверять, выполняются ли правила стиля. Лучше, если они навязаны инструментарием и выполняются изначально. Однако инструментарий позволяет далеко не все, и нет замены тщательности в написании каждого участка ПО.

Часто программист откладывает применение правил стиля. Он пишет программный код как придется, полагая: "я почищу все позже, сейчас я даже не знаю, что из написанного мне пригодится". Это не наш путь. Если правило используется, не должно быть никакой задержки в его применении с первых шагов написания ПО, даже в отсутствие специальных средств поддержки. Всегда дороже последующие корректировки текста, чем его корректное написание с самого начала. Всегда есть риск, что на чистку не хватит времени или вы просто забудете о ней. Всякий, кто позже столкнется с вашей работой, потратит куда больше времени, чем это стоило бы вам при написании заголовочных комментариев, придумывании подходящих имен, применении нужного форматирования. Не забывайте: этим кто-то может быть вы сами.

Кратко и явно

Стиль ПО всегда колебался между краткостью и многословием. Двумя крайними примерами языков программирования являются, вероятно, APL и Cobol. Контраст между линейкой языков Fortran-C-C++ и традициями Algol-Pascal-Ada - не только в самих языках, но в стиле, который они проповедуют, - разителен.

Существенными для нас являются ясность и качество. Обе экстремальные формы противоречат этим целям. Зашифрованные С-программы, к несчастью, не ограничены известной дискуссией об "obfuscated" (затемненном, сбивающем с толку) С и C++. В равной степени почти столь же известные многословные выражения (DIVIDE DAYS BY 7 GIVING WEEKS) языка Cobol являются примером напрасной траты времени.

Стиль правил этой лекции представляет смесь ясности, характерной для Algol-подобных языков и краткости телеграфного стиля. Он никогда не скучится на нажатия клавиш, когда они по-настоящему способствуют пониманию программного текста. Например, одно из правил предписывает задание идентификаторов словами, а не аббревиатурами; было бы глупо экономить несколько букв, назвав компонент *disp* (двусмысленно), а не *display* (ясно и четко), или класс *ACCNT* (непроизносимое) вместо *ACCOUNT*. В данных ситуациях нет налога на число нажатий. Но в то же время, когда приходится исключать напрасную избыточность, правила безжалостны. Они ограничивают заголовки комментариев обязательными словами, освобождают от всех "the" и других подобных любезностей; они запрещают излишнюю квалификацию (подобную *account_balance* в классе *ACCOUNT*, где имени *balance* достаточно). Возвращаясь к домinantной теме, правила допускают группирование связанных составляющих сложной структуры в одной строке, например:

```
from i := 1 invariant i <= n until i = n loop.
```

Этой комбинации ясности и краткости следует добиваться в своих текстах. Раздутый размер текста, в

конечном счете, приводят к возрастанию сложности, но и не экономьте на размере, когда это необходимо для обеспечения ясности.

Если, подобно многим, вас интересует, будет ли текст ОО-реализации меньше, чем текст на языках C, Pascal, Ada или Fortran, то интересный ответ появится только на уровне большой системы или подсистемы. При записи основных алгоритмов, подобных быстрой сортировке Quicksort, или алгоритма Эвклида ОО-версия будет не меньше, чем на C, в большинстве случаев при следовании правилам стиля она будет больше, так как будет включать утверждения и подробную информацию о типах. Все же по опыту ISE на системах среднего размера мы иногда находили (не утверждаем, что это общее правило), что ОО-решение было в несколько раз короче. Почему? Дело не в краткости на микроуровне, результат объясняется широким применением архитектурных приемов ОО-метода:

- Универсальность - один из ключевых факторов. Мы обнаруживали в программах С один и тот же код, многократно повторяющийся для описания различных типов. С родовыми классами или с родовыми пакетами Ada вы избавляетесь от подобной избыточности. Огорчительно видеть, что Java, ОО-язык, основанный на C, не поддерживает универсальность.
- Наследование вносит фундаментальный вклад в сбор общности и удаление дублирования.
- Динамическое связывание заменяет многие сложные структуры разбора ситуаций, делая вызовы много короче.
- Утверждения и связанная с ними идея Проектирования по Контракту позволяет избегать избыточных проверок - принципиального источника раздувания текста.
- Механизм исключений позволяет избегать написания некоторого кода, связанного с обработкой ошибок.

Если вас заботят размеры кода, убедитесь, что вы позаботились об архитектурных аспектах. Следует быть краткими при выражении сути алгоритма, но не экономьте на нажатиях клавиш ценой ясности.

Роль соглашений

Большинство правил дает однозначное толкование без всяких вариантов. Исключения включают использование шрифтов, управляемое внешними обстоятельствами (что хорошо выглядит в книге, может быть не видимо на слайдах проектора), и точки с запятыми, для которых существуют две противоположные школы с весомыми аргументами.

Правила появились в результате многолетних наблюдений, дискуссий и тщательного анализа того, что работает и что работает менее хорошо. Даже при этом часть правил может показаться спорной и некоторые решения являются делом вкуса, так что разумные люди по ряду соображений могут с ними не согласиться. Если вы не принимаете какое-либо из рекомендуемых соглашений, вам следует определить свое собственное, пояснить его во всех деталях, явно документировать. Но тщательно все взвесьте, прежде чем принимать такое решение, - так очевидны преимущества универсального множества правил, систематически применяемых к тысячам классам в течение более десяти лет, известных и принятых многими людьми.

Многие из этих правил стиля первоначально разрабатывались для библиотек, а затем нашли свое место в разработке обычного ПО. В объектной технологии, конечно, все ПО разрабатывается в предположении, что, если оно и не предназначается для повторного использования, со временем оно **может** стать таковым, поэтому естественно с самого начала применять те же правила стиля.

Самоприменение

Подобно правилам проектирования, правила стиля применяются в примерах этой книги. Причины очевидны: каждый должен практиковать то, что он проповедует, не говоря о том, что правила способствуют ясности мысли и выражения при представлении ОО-метода.

Единственными исключениями являются некоторые отходы в форматировании программных текстов. Согласно правилам, программные тексты без колебаний следует располагать на многих строчках, требуя лишь, например, чтобы каждое предложение утверждения имело свою собственную метку. Строки компьютерного экрана не являются ресурсом, который нуждается в экономии. Полагается, что со временем будет произведена революция, и мы перейдем от стиля папирусных свитков к странично-структурированным книгам. Но данный текст определенно является книгой - постоянное применение правил форматирования программных текстов привело бы к неоправданному увеличению объема книги.

Эти случаи освобождения от обязательств распространяются лишь на несколько правил форматирования и будут специально отмечены ниже при представлении правил. Такие исключения разрешаются только для представлений на бумаге. Фактические программные тексты применяют правила буквально.

Дисциплина и творчество

Было бы ошибкой протестовать против правил этой лекции на том основании, что они ограничивают

творческую активность разработчиков. Согласованный стиль скорее помогает, чем препятствует творчеству, направляя его в нужное русло. Большая часть усилий при производстве ПО тратится на чтение уже существующих текстов. Индивидуальные предпочтения в стиле дают преимущества одному человеку, общие соглашения - помогают каждому.

В программистской литературе семидесятых годов пропагандировалась идея "**безлиного программирования**" ("egoless programming"): разрабатывать ПО так, чтобы личность автора в нем не ощущалась. Ее цель - возможность взаимозаменяемости разработчиков. Примененная к проектированию системы, цель эта становится явно нежелательной, даже если некоторые менеджеры страстно желают этого. Приведу отрывок из книги Барри Boehme, цитирующей эту идею: "**Программистский творческий инстинкт должен быть полностью затушеван в интересах общности и понятности**". Сам Boehme комментирует это так: "Давать программистам подобные советы, зная их повышенную мотивацию, - заведомо обрекать их на нервное расстройство".

Какого качества можно ожидать от ПО с **безликим** проектом и **безликим** выражением?

Более чем удивительно, но при разработке ПО почти полностью отсутствуют стандарты стиля. Нет другой дисциплины, которая называлась бы "инженерией", где был бы такой простор для персональных прихотей и капризов. Чтобы стать профессионалами, разработчики ПО должны контролировать сами себя и выработать свои стандарты.

Выбор правильных имен

Первое, что нуждается в регулировании, - это выбор имен. Имена компонентов следует строго контролировать, что всем принесет пользу.

Общие правила

Наиболее значимыми являются имена **классов и компонентов**, широко используемые в других классах.

Для этих имен используйте полные слова, но не аббревиатуры, если только последние не имеют широкого применения в проблемной области. В классе PART, описывающем детали в системе управления складом, назовите number, а не part, компонент (запрос), возвращающий номер детали. Печатание недорого стоит, сопровождение - чрезвычайно дорого. Аббревиатуры usa в Географической Информационной системе или copter в системе управления полетами вполне приемлемы, так как в данных областях приобрели статус независимых слов. Кроме того, некоторые сокращения используются годами и также приобрели независимый статус, такие как PART для PARTIAL, например в имени класса PART_COMPARABLE, описывающего объекты, поставляемые с частичным порядком.

При выборе имен целью является ясность. Без колебаний используйте несколько слов, объединенных пробелами, как в имени класса ANNUAL_RATE, или yearly_premium в имени компонента.

Хотя современные языки не ограничивают длину идентификаторов, и рассматривают все буквы как важные, длина имени должна оставаться разумной. Правила на этот счет для классов и компонентов различные. Имена классов вводятся только в ряде случаев - в заголовках класса, объявлениях типа, предложениях наследования и других. Имя класса должно полностью характеризовать соответствующую абстракцию данных, так что вполне допустимо такое имя класса - PRODUCT_QUANTITY_INDEX_EVALUATOR. Для компонентов достаточно двух слов, редко трех, соединенных подчеркиванием. В частности, не следует допускать **излишней квалификации** имени компонента. Если имя компонента чересчур длинно, то это, как правило, из-за излишней квалификации.

Правило: Составные имена компонентов

Не включайте в имя компонента имя базовой абстракции данных (служащей именем класса).

Компонент, задающий номер части в PART, должен называться просто number, а не part_number. Подобная сверхквалификация является типичной ошибкой новичков, скорее затуманивающей, чем проясняющей текст. Помните, каждое использование компонента однозначно определяет класс, например part1.number, где part1 должно быть объявлено типа PART или его потомка.

Для составных имен лучше избегать стиля, популяризируемого Smalltalk и используемого в библиотеках, таких как X Window System, объединяющих несколько слов вместе, начиная каждое внутренне слово с большой буквы, как в yearlyPremium. Вместо этого разделяйте компоненты подчеркиванием, как в yearly_premium. Использование внутренних больших букв конфликтует с соглашениями обычного языка и выглядит безобразно, оно приводит к трудно распознаваемому виду, следовательно, к ошибкам (сравните aLongAndRatherUnreadableIdentifier и an_even_longer_but_perfectly_clear_choice_of_name).

Иногда каждый экземпляр некоторого класса содержит поле, представляющее экземпляр другого класса. Это приводит к мысли использовать для имени атрибута имя класса. Например, вы определили класс RATE, а классу ACCOUNT потребовался один атрибут типа RATE, для которого кажется естественным использовать имя rate - в нижнем регистре, в соответствии с правилами, устанавливаемыми ниже. Хотя можно пытаться найти более специфическое имя, но приемлемо rate: RATE. Правила выбора идентификаторов допускают одинаковые имена компонента и класса. Нарушением стиля является добавление префикса the, как в the_rate, что только добавляет шумовую помеху.

Локальные сущности и аргументы подпрограмм

Акцент на ясные, хорошо произносимые имена сделан для компонентов и классов. Для локальных сущностей и аргументов подпрограмм, имеющих локальную область действия, нет необходимости в подобной выразительности. Имена, несущие слишком много смысла, могут даже ухудшить читабельность текста, придавая слишком большое значение вспомогательным элементам. (Им можно давать короткие однобуквенные имена, как, например, в процедуре класса TWO_WAY_LIST из библиотеки e Base)

```
move (i: INTEGER) is
    -- Поместить курсор в позицию i или after, если i слишком велико
    local
        c: CURSOR; counter: INTEGER; p: like FIRST_ELEMENT
    ...
remove is
    -- Удаляет текущий элемент; перемещает cursor к правому соседу
    -- (или after если он отсутствует).
    local
        succ, pred, removed: like first_element
    ...
```

Если бы succ и pred были бы компонентами, они бы назывались successor и predecessor. Принято использовать имя new для локальной сущности, представляющей новый объект, создаваемый программой, и имя other для аргумента, представляющего объект того же типа, что и текущий, как в объявлении для clone в GENERAL:

```
frozen clone (other: GENERAL): like other is...
```

Регистр

Регистр не важен в нашей нотации, поскольку было бы опасным позволять двум почти идентичным идентификаторам обозначать разные вещи. Но настоятельно рекомендуется в интересах читабельности и согласованности придерживаться следующих правил:

- Имена классов задаются буквами в верхнем регистре: POINT, LINKED_LIST, PRICING_MODEL. Это верно и для формальных родовых параметров, обычно начинающихся с G.
- Имена неконстантных атрибутов, подпрограмм, отличных от однократных, локальных сущностей и аргументов подпрограмм задаются полностью в нижнем регистре: balance, deposit, succ, i.
- Константные атрибуты начинаются с буквы в верхнем регистре, за которой следуют буквы нижнего регистра: Pi: INTEGER is 3.141598524; Welcome_message: STRING is "Welcome!". Это же правило применяется к уникальным значениям, представляющим константные цели.
- Те же соглашения применяются к однократным функциям, эквиваленту констант для небазисных типов: Error_window, Io. В нашем первом примере комплексное число i (мнимая единица) осталось в нижнем регистре для совместимости с математическими соглашениями.

Эти правила касаются имен, выбираемых разработчиком. Резервируемые слова нотации разделяются на две категории. **Ключевые слова**, такие как do и class, играют важную синтаксическую роль; они записаны в нижнем регистре полужирным шрифтом. Несколько зарезервированных слов не являются ключевыми, поскольку играют ассоциированную семантическую роль, они записываются курсивом с начальной буквой в верхнем регистре, подобно константам. К ним относятся: Current, Result, Precursor, True и False.

Грамматические категории

Точные правила управляют грамматическими категориями слов, используемых в идентификаторах. В некоторых языках эти правила могут применяться без колебаний, в английском, как ранее отмечалось, они обеспечивают большую гибкость.

Правило для имен классов уже приводилось: следует всегда использовать существительные, как в ACCOUNT, возможно квалифицированные, как в LONG_TERM_SAVINGS_ACCOUNT, за исключением случая отложенных

классов, описывающих структурные свойства, для которых могут использоваться прилагательные, как в NUMERIC или REDEEMABLE.

Имена подпрограмм должны отражать принцип Разделения Команд и Запросов:

- Процедуры (команды) должны быть глаголами в инфинитиве или повелительной форме, возможно, с дополнениями: `make`, `move`, `deposit`, `set_color`.
- Атрибуты и функции (запросы) никогда не должны использовать императив или инфинитив глаголов: никогда не называйте запрос `get_value`, назовите его просто `value`. Имена небулевых запросов должны быть существительными, такими как `number`, возможно, квалифицированными, как в `last_month_balance`. Булевые запросы должны использовать прилагательные, как в `full`. В английском возможна путаница между прилагательными и глаголами (`empty`, например, может значить "пусто ли это?" или "опустошить это!"). В связи с этим для булевых запросов часто применяется `is_` форма, как в `is_empty`.

Стандартные имена

Вы уже заметили многократное использование во всей книге нескольких базисных имен, таких как `put` и `item`. Они являются важной частью метода.

Большинству классов необходимы компоненты, представляющие операции нескольких базисных видов: вставка, замена, доступ к элементам структуры. Вместо придумывания специальных имен для этих и подобных операций в каждом классе, предпочтительно повсюду применять стандартную терминологию.

Вот каковы основные стандартные имена. Начнем с процедур создания: имя `make` рекомендуется для наиболее общей процедуры создания и имена вида `make_some_qualification`, например, `make_polar` и `make_cartesian` для классов `POINT` или `COMPLEX`.

Для команд наиболее общие имена приведены в таблице:

Таблица 8.1. Стандартные имена команд

Команда	Действие
<code>extend</code>	Добавить элемент
<code>replace</code>	Заменить элемент
<code>force</code>	Подобна команде <code>put</code> , но может работать для большего числа случаев. Например для массивов <code>put</code> имеет предусловие, требующее, чтобы индекс не выходил за границы, в то время как <code>force</code> не имеет предусловий и допускает выход за границы
<code>remove</code>	Удаляет (не специфицированный) элемент
<code>prune</code>	Удаляет специфицированный элемент
<code>wipe_out</code>	Удаляет все элементы

Для небулевых запросов (атрибутов или функций)

Таблица 8.2. Стандартные имена для не булевых запросов

Запрос	Действие
<code>item</code>	Базисный запрос для получения элемента: в классе <code>ARRAY</code> - элемент с заданным индексом; <code>STACK</code> - элемент вершины стека; <code>QUEUE</code> - "старейший" элемент и так далее
<code>infix "@"</code>	Синоним <code>item</code> в некоторых случаях, например в классе <code>ARRAY</code>
<code>count</code>	Число используемых элементов структуры
<code>capacity</code>	Физический размер, распределенный для ограниченной структуры, измеряемый числом потенциальных элементов. Инвариант должен включать <code>0 < count and count <= capacity</code>

Для булевых запросов стандартными именами являются:

Таблица 8.3. Стандартные имена булевых запросов

Запрос	Действие
<code>empty</code>	Содержит ли структура элементы?
<code>full</code>	Заполнена ли структура ограниченной емкости элементами? Обычно эквивалент <code>count = capacity</code>
<code>has</code>	Присутствует ли заданный элемент в структуре? (Базисный тест проверки членства)

<code>extendible</code>	Можно ли добавить элемент? (Может служить предусловием для <code>extend</code>)
<code>prunable</code>	Можно ли удалить элемент? (Может служить предусловием для <code>remove</code> и <code>prune</code>)
<code>readable</code>	Доступен ли элемент? (Может служить предусловием для <code>remove</code> и <code>item</code>)
<code>writable</code>	Можно ли изменить элемент? (Может служить предусловием для <code>extend</code> , <code>replace</code> , <code>put</code> и др.)

Преимущества согласованного именования

Схема имен, данная выше, вносит наиболее видимый вклад в характерный стиль конструирования ПО, разрабатываемый в соответствии с принципами этой книги.

Не зашли ли мы слишком далеко в борьбе за согласованность? Не получится ли, что в программах будут использоваться одни и те же имена, но с разной семантикой? Например, `item` для стека возвращает элемент вершины, а для массива - элемент с заданным индексом.

При систематическом подходе к ОО-конструированию, использованию статической типизации и Проектированию по Контракту этот страх не оправдан. Знакомясь с компонентом, автор клиента может полагаться на четыре вида свойств, представленных в краткой форме класса:

- **F1** Его имя.
- **F2** Его сигнатура (число и типы аргументов, если это процедура, тип результата для запросов).
- **F3** Предусловие и постусловие, если они заданы.
- **F4** Заголовочный комментарий.

Подпрограммы имеют, конечно же, тело, но предполагается, что тело не должно волновать клиента.

Три из этих элементов будут отличаться для вариантов базисных операций. Например, в краткой форме класса STACK можно найти компонент:

```
put (x: G)
    -- Втолкнуть x на вершину
require
    writable: not full
ensure
    not_empty: not empty
    pushed: item = x
```

В классе ARRAY появляется однофамилец:

```
put (x: G; i: INTEGER)
    -- Заменить на x значение элемента с индексом i
require
    not_too_small: i >= lower
    not_too_large: i <= upper
ensure
    replaced: item (i) = x
```

Сигнатуры различаются, предусловия, постусловия, заголовочные комментарии - все различно. Использование имени `put`, не создавая путаницы, обращает внимание читателя на общую роль этих процедур: они обе обеспечивают базисный механизм изменений.

Эта согласованность оказывается одним из наиболее привлекательных аспектов метода, в частности библиотек. Новые пользователи быстро привыкают к ней и при появлении нового класса, следующего стандартному стилю, принимают его как старого знакомого и могут сразу же сосредоточиться на нужных им компонентах.

Использование констант

Многие алгоритмы используют константы. Как отмечалось в одной из предыдущих лекций, у констант плохая слава из-за отвратительной практики изменения их значений. Следует предусмотреть меры против подобного непостоянства.

Манифестные и символические константы

Основное правило использования констант утверждает, что не следует явно полагаться на значения:

Принцип Символических констант

Не используйте манифестные (неименованные) константы в любых конструкциях, отличных от объявления символьических констант. Исключением являются нулевые элементы основных операций.

Манифестная константа задается явно своим значением, как, например, 50 (целочисленная константа) или "Cannot find file" (строковая константа). Принцип запрещает использование инструкций в форме:

```
population_array.make (1, 50)
```

или

```
print ("Cannot find file") -- Ниже смотри смягчающий комментарий
```

Вместо этого следует объявить соответствующий константный атрибут и в тела подпрограмм, где требуются значения, обозначать их именами атрибутов:

```
US_state_count: INTEGER is 50
file_not_found: STRING is "Cannot find file"
...
population_array.make (1, state_count)
...
print (file_not_found)
```

Преимущества очевидны: если появится новый штат или изменится сообщение, достаточно изменить только одно объявление.

Использование 1 наряду со state_count в первой инструкции не является нарушением принципа, так как он запрещает манифестные константы, **отличные от нулевых элементов**. Нулевыми элементами, допустимыми в манифестной форме, являются целые 0 и 1 (нулевые элементы сложения и умножения), вещественное число 0 . 0, нулевой символ, записываемый как '%0', пустая строка - """. Использование символьической константы One каждый раз, когда требуется сослаться на нижнюю границу массива (1 используется соглашением умолчания), свидетельствовало бы о педантичности, фактически вело бы к ухудшению читабельности.

В других обстоятельствах 1 может просто представлять системный параметр, имеющий сегодня одно значение, а завтра другое. Тогда следует объявить символьическую константу, как например Processor_count: INTEGER is 1 в многопроцессорной системе, использующей пока один процессор.

Принцип Символических Констант слишком строг в случае простых, однократно применяемых манифестных строк. Можно было бы усилить исключение, сформулировав его так: **"за исключением нулевых элементов основных операций и манифестных строковых констант, используемых однократно"**. В примерах этой книги используются такие константы. Такое ослабление правила приемлемо, но в долгосрочной перспективе лучше придерживаться правила в первоначальной форме, даже если это кажется педантичным. Одно из главных применений строковых констант - это вывод сообщений пользователю. Когда успешная система, выпущенная для национального рынка, выходит на международный, то с символическими константами переход на любой язык не представляет трудностей.

Где размещать объявления констант

Если число локальных константных атрибутов в классе становится большим, то, вероятно, имеет место нераспознанная абстракция данных - определенное понятие, характеризуемое рядом параметров.

Тогда желательно сгруппировать объявления констант, поместив их в отдельный класс, который может служить предком для любого класса, которому нужны константы. (Некоторые разработчики предпочитают в таких случаях использовать отношение клиента.) Примером является класс ASCII библиотеки Base.

Заголовочные комментарии и предложения индексации

Хотя формальные элементы класса несут достаточно подробную информацию, следует сопровождать класс неформальными пояснениями. Заголовочные комментарии к подпрограммам и предложения **feature**, дополненные предложениями indexing, задаваемыми для каждого класса, отвечают этой потребности.

Комментарии в заголовках: упражнение на сокращение

Подобно дорожным знакам на улицах Нью-Йорка, говорящим "Даже и не думайте здесь парковаться!", знаки на входе в отдел программистов должны предупреждать " Даже и не думайте написать подпрограмму без заголовочного комментария". Этот комментарий, следующий сразу за ключевым словом **is**, кратко формулирует цель программы; он сохраняется в краткой и плоской краткой форме класса:

```

distance_to_origin: REAL is
    -- Расстояние до точки (0, 0)
local
    origin: POINT
do
    create origin
    Result := distance (origin)
end

```

Обратите внимание на отступ: комментарий начинается на шаг правее тела подпрограммы.

Комментарий к заголовку должен быть информативным, кратким, ясным. Он имеет собственный стиль, которому будем обучаться на примере, начав вначале с несовершенного комментария, а затем улучшая его шаг за шагом. В классе CIRCLE к одному из запросов возможен такой комментарий:

```

tangent_from (p: POINT): LINE is
    -- Возвращает касательную линию к текущей
    -- окружности,
    -- проходящую через данную точку p,
    -- если эта точка лежит вне текущей окружности
require
    outside_circle: not has (p)
...

```

В стиле этого комментария много ошибок. Во-первых, он не должен начинаться "Возвращает ..." или "Вычисляет ... ", используя глагольные формы, поскольку это противоречит принципу Разделения Команд и Запросов. Имя, возвращаемое не булевым запросом, типично использует квалифицированное существительное. Поэтому лучше написать так:

```

-- Касательная линия к текущей окружности,
-- проходящая через данную точку p,
-- если эта точка лежит вне текущей окружности

```

Так как комментарий теперь не предложение, а просто квалифицированное имя, то точку в конце ставить не надо. Теперь следует избавиться от дополнительных слов (в английском особенно от **the**), не требуемых для понимания. Для комментариев желателен телеграфный стиль. (Помните, что читатели, любящие литературные красоты, могут выбрать для чтения романы Марселя Пруста.)

```

-- Касательная линия к текущей окружности,
-- проходящая через точку p,
-- если точка вне текущей окружности

```

Следующая ошибка содержится в последней строчке. Дело в том, что условие применимости подпрограммы - ее предусловие - **not has** (p), появится сразу после комментария в краткой форме, где оно выражено ясно и недвусмысленно. Поэтому нет необходимости в его перефразировке, что может привести только к путанице, а иногда и к ошибкам (типичная ситуация: предусловие в форме $x \geq 0$ с комментарием "применимо только для положительных x", а нужно "не отрицательных"); всегда есть риск при изменившемся предусловии забыть об изменении комментария. Наш пример теперь станет выглядеть так:

```
-- Касательная линия к текущей окружности из точки p
```

Еще одна ошибка состоит в использовании слов линия (line) и точка (point) при ссылках на результат и аргумент запроса: эта информация непосредственно следует из объявляемых типов LINE и POINT. Лучше использовать формальные объявления типов, которые появятся в краткой форме, чем сообщать эту информацию в неформальной форме комментария. Итак:

```
-- Касательная к текущей окружности из p
```

Наши ошибки состояли в излишнем дублировании информации - о типах, о требованиях предусловия. Из их анализа следует общее правило написания комментариев: исходите из того, что читатель компетентен в основах технологии, не включайте информацию, непосредственно доступную в краткой форме класса. Это, конечно, не означает, что никогда не следует указывать информацию о типах, например, в предыдущем примере Расстояние до точки (0,0) было бы двусмысленным без указания слова "точка" (point).

При необходимости сослаться на текущий экземпляр используйте фразы вида: текущая окружность, текущее число, вместо явной ссылки на сущность Current. Во многих случаях можно вообще избежать с упоминания текущего объекта, так как каждому программисту ясно, что компоненты при вызове применяются к текущему объекту. В данном примере наш заключительный комментарий выглядит так:

```
-- Касательная из р
```

На этом этапе осталось три слова, а начинали с трех строк из 18 длинных слов. Длина комментария сократилась примерно на 87%, мы можем считать, что упражнение на сокращение выполнено полностью, - сказать короче и яснее трудно.

Несколько общих замечаний. Отметим бесполезность в запросах фраз типа "Возвращает ...", других шумовых слов и фраз, которые следует избегать во всех подпрограммах: "Эта подпрограмма вычисляет (возвращает) ...", просто скажите, что делается. Вместо:

```
-- Эта программа записывает последний исходящий звонок
```

пишите

```
-- Записать исходящий звонок
```

Как показывает это пример, комментарий к командам (процедурам) должен быть в императивной или инфинитивной форме (в английском это одно и тоже). Он должен иметь стиль приказа и оканчиваться точкой. Для булевых запросов комментарий всегда должен быть в вопросительной форме и заканчиваться знаком вопроса:

```
has (v: G): BOOLEAN is
    -- Появляется ли v в списке?
    ...
```

Соглашение управляет использованием программных сущностей - атрибутов, аргументов, появляющихся в комментариях. При наборе текста они выделяются курсивом (о других соглашениях на шрифт смотри ниже). В исходных программных текстах они всегда должны заключаться в кавычки, так что оригинальный текст выглядит так:

```
-- Появляется ли 'v' в списке?
```

Инструментарий, генерирующий краткую форму класса, использует это соглашение для обнаружения ссылок на сущности.

Нужно следить за согласованностью. Если функция класса имеет комментарий: "Длина строки", в другой процедуре не должна идти речь о "ширине" строки: "Изменить ширину строки", когда речь идет об одном и том же свойстве строки.

Все эти рекомендации применимы к подпрограммам. Поскольку экспортруемые атрибуты внешне ничем не должны отличаться от функций без аргументов, то они тоже имеют комментарий, появляющийся с тем же отступом, что и у функций:

```
count: INTEGER
    -- Число студентов на курсе
```

Для закрытых атрибутов комментарии желательны, но требования к ним менее строгие.

Заголовочные комментарии предложений **feature**

Как вы помните, класс может иметь несколько предложений **feature**:

```
indexing
...
class LINKED_LIST [G] inherit ... creation
...
feature -- Initialization
    make is ...
feature -- Access
    item: G is ...
...
feature -- Status report
    before: BOOLEAN is ...
...
feature -- Status setting
...
feature -- Element change
    put_left (v: G) is ...
...
feature -- Removal
```

```

remove is ...
...
feature {NONE} -- Implementation
  first_element: LINKABLE [G].
...
end

```

Одна из целей введения нескольких предложений **feature** состоит в том, чтобы придать разным компонентам различный статус экспорта. Но в данном примере все компоненты за исключением последнего доступны всем клиентам. Другой целью введения нескольких предложений **feature**, демонстрируемой данным примером, является группировка компонентов по категориям. Комментарий, находящийся на той же строке, что и ключевое слово **feature**, характеризует категорию. Такие комментарии, подобно заголовочным комментариям подпрограмм, обнаруживаются инструментарием, таким как **short**, создающим документацию и краткую форму класса.

Восемнадцать категорий с соответствующими комментариями стандартизованы в библиотеках Base, так что каждый компонент (из примерно 2000) принадлежит одной из них. В этом примере показаны некоторые из наиболее важных категорий. **Status report** соответствует опциям (устанавливаются компоненты в категории **Status setting**, не включенной в этот пример). Закрытые и выборочно экспортируемые компоненты появляются в категории **Implementation**. Эти стандартные категории появляются всегда в одном и том же порядке, известном инструментарию (через список, редактируемый пользователем), и будут сохраняться или переустанавливаться при выводе документации. Внутри каждой категории инструментарий перечисляет компоненты в алфавитном порядке для простоты поиска.

Категории покрывают широкий спектр областей применения, хотя для специальных проблемных областей могут понадобиться собственные категории.

Предложения индексирования

Подобными заголовочным комментариям, но немного более формальными являются предложения индексирования, появляющиеся в начале каждого класса:

```

indexing
  description: "Последовательные списки в цепном представлении"
  names: "Sequence", "List"
  contents: GENERIC
  representation: chained
  date: "$Date: 96/10/20 12:21:03 $"
  revision: "$Revision: 2.4$"
...
class LINKED_LIST [G] inherit
...

```

Предложения индексирования строятся в соответствии с принципом Само-документирования аналогично встроенным утверждениям и заголовочным комментариям, позволяя включать в текст ПО возможную документацию. Для свойств, не появляющихся напрямую в программном тексте класса, можно включить индексирующие разделы в форме:

```
indexing_term: indexing_value, indexing_value, ...
```

где **indexing_term** является идентификатором, а каждое **indexing_value** является некоторым базисным элементом, таким как строка, целое и так далее. Идентификаторы разделов, имеющие альтернативные имена, позволяют потенциальным авторам клиентов отыскать нужный класс по именам (**names**), содержанию (**contents**), выбору представления (**representation**), информации об обновлениях (**revision**), информации об авторе и многому другому. В разделы включается все, что может облегчить понимание класса и поиск, использующий ключевые слова. Благодаря специальному инструментарию, поддерживающим повторное использование, облегчается задача разработчиков по поиску в библиотеках нужных им классов с нужными компонентами.

Как индексирующие термы, так и их значения могут быть произвольными, но возможности выбора фиксируются для каждого проекта. Множество стандартов, принятых в библиотеке Base, частично приведено в примере. Каждый класс должен иметь раздел **description**, значением которого **index_value** является строка, описывающая роль класса в терминах его экземпляров (**Последовательные списки...**, но не "этот класс описывает последовательные списки", или "последовательный список", или "понятие последовательного списка" и т. д.). Для наиболее важных классов в этой книге - но не в коротких примерах, предназначенных для специальных целей - раздел **description** включался в предложение **indexing**.

Не заголовочные комментарии

Предыдущие правила применяются к стандартизованным комментариям, появляющимся в определенных местах и играющих специальную роль в документировании класса.

Во всех способах разработки ПО существует необходимость в комментариях отдельных участков выполняемых алгоритмов, поясняющих суть работы.

Есть еще одно использование комментариев, часто используемое на практике, но редко упоминаемое в учебниках. Я говорю здесь о технике преобразования некоторого участка кода в комментарий либо потому, что он не работает, либо он еще просто не готов. Эта практика, очевидно, требует замены специальными механизмами. Она уже обогатила язык новой глагольной формой - "закомментировать" (**comment out**).

Каждый комментарий по уровню абстракции должен быть выше комментируемого примера. Известный контрпример: -- Увеличить *i* на 1 в инструкции *i := i + 1*. Здесь комментарий является перефразировкой кода и не несет полезной нагрузки.

Языки низкого уровня призывают к подробному комментированию. Каждую строку С следует комментировать, поскольку в современной разработке языку С отводится роль инкапсуляции машинно-ориентированных операций и выполнения функций уровня операционной системы, что по своей природе является неким видом трюкачества и потому требует пояснений. В ОО-разработках комментарии, не относящиеся к заголовкам, встречаются значительно реже, они остаются необходимыми для тонких мест разработки и тогда, когда предвидится возможное смешение понятий. В своих постоянных усилиях предотвратить появление ошибок, а не лечить их последствия, метод уменьшает необходимость в комментариях благодаря модульному стилю, выработке небольших, понятных подпрограмм, через механизм утверждений. Предусловия и постусловия, инварианты класса формально выражают семантику, инструкции **check** выражают ожидаемые свойства, которые должны выполняться в определенном состоянии. Этому способствуют и соглашения именования, введенные в этой лекции. Общий тезис: секрет в создании ясного, понятного ПО состоит не в постфактумном добавлении комментариев, но в производстве согласованной и стабильной структуры системы, правильной с самого начала.

Форматирование и презентация текста

Следующие правила определяют, как следует располагать программный текст на бумаге в реальной жизни или при моделировании этого процесса на дисплее компьютера. Более чем любые другие они взывают о "косметике" столь же важной для разработчиков ПО, как косметика от Кристиана Диора для ее покупательниц. Они играют немалую роль в том, как быстро ваш продукт будет понятен его пользователям - сопровождающим, повторно использующим, покупающим ваше ПО.

Форматирование

Рекомендуемое форматирование текста следует из общей синтаксической формы нотации, которую с некоторой натяжкой можно назвать "операторной грамматикой", когда текст класса представляет последовательность символов, разделенных на "операторы" и "операнды". Операторами являются фиксированные символы языка, такие как ключевые слова (*do*, например) или разделители (точка с запятой, запятая). Операндом является символ, выбираемый программистом (идентификатор, константа).

Основываясь на этом свойстве, форматирование текста следует **гребенчато-подобной структуре (comb-like structure)**, введенной в языке Ada. Идея состоит в том, что каждая синтаксически важная часть класса, такая как инструкция или выражение должна либо:

- размещаться на одной строке вместе с предшествующими и последующими операторами;
- либо с отступами размещаться на нескольких строках, организованных так, чтобы это правило выполнялось рекурсивно.

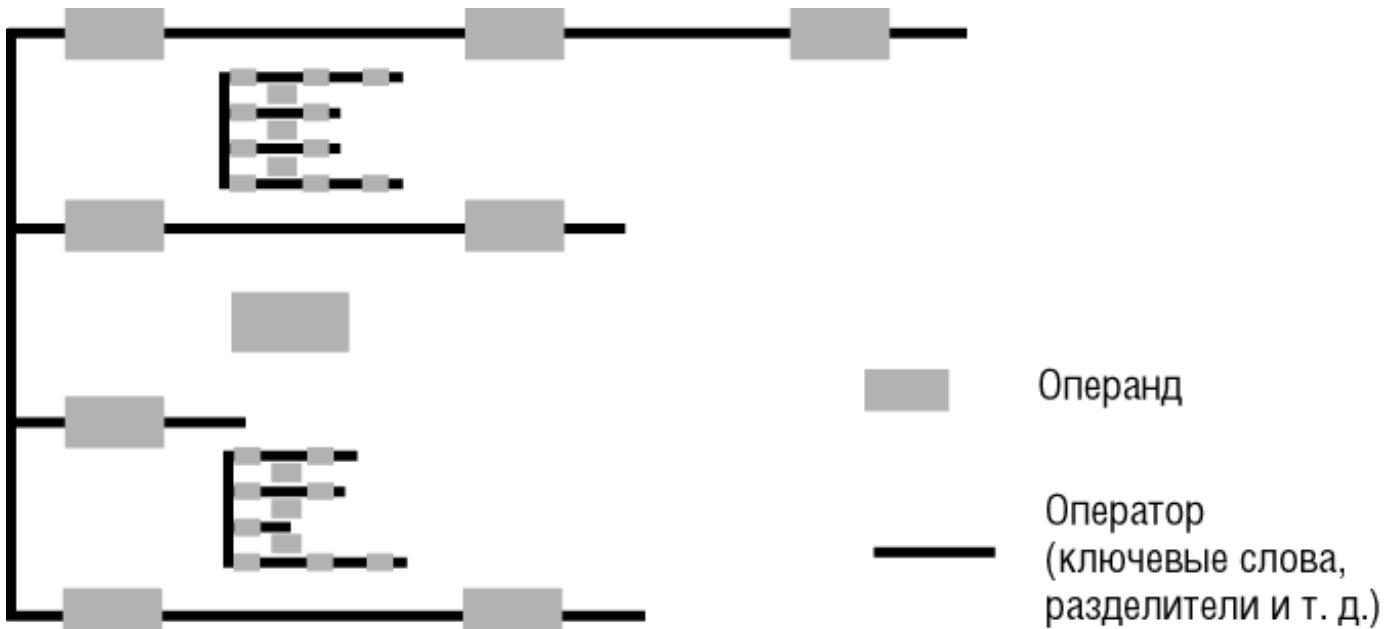


Рис. 8.1. Гребенчато-подобная структура организации программного текста

Каждая ветвь гребенки является последовательностью чередующихся операторов и операндов, обычно начинающихся и заканчивающихся оператором. В пространстве между двумя ветвями находится либо операнд, либо рекурсивно гребенчато-подобная структура.

Как пример, зависящий от размера его составляющих *a*, *b* и *c*, допустимы следующие формы представления инструкции выбора:

```
if c then a else b end
```

или

```
if
  c
then
  a
else
  b
end
```

или:

```
if c then
  a
else b end
```

Однако вы не можете использовать строку, содержащую просто *if* с или с *end*, так как они включают операнд вместе с чем-то еще, пропуская заканчивающий оператор в первом случае, а во втором - начинающий.

Подобным образом можно начать класс после предложения **indexing** так:

```
class C inherit          -- [1]
```

или

```
class C feature          -- [2]
```

или

```
class
  C
feature                  -- [3]
```

Нельзя писать

```
class C
feature                  -- [4]
```

поскольку первая строка нарушает правило.

Формы [1] и [2] используются в этой книге для небольших иллюстративных классов. Более практические классы имеют одно или несколько помеченных предложений **feature**, они в отсутствие предложения **inherit** должны использовать форму [3] (она предпочтительнее, чем форма [2]):

```
class
  С
feature -- Initialization
  ...
feature -- Access
  и т.д.
```

Высота и ширина

Подобно большинству современных языков, наша нотация не придает особого значения окончаниям строк за исключением строк, завершающихся комментарием. Две или более инструкций (объявления) могут располагаться на одной строке, разделенные в этом случае точками с запятой:

```
count := count + 1; forth
```

Этот стиль по ряду причин не очень популярен (многие инструментальные средства для оценки размера ПО используют **строки**, а не синтаксические единицы); большинство разработчиков предпочитают располагать не более одной инструкции на строку. Действительно, нежелательно упаковывать текст, но в ряде случаев удобно и разумно располагать ряд связанных инструкций на одной строке.

В этой области лучше полагаться на ваши предпочтения и хороший вкус. Если вы применяете внутристочное группирование, убедитесь, что оно остается умеренным и согласованным с внутренними отношениями между инструкциями. Принцип Точки с Запятой, который появится чуть позже, требует разделения таких инструкций точкой с запятой.

По очевидным причинам объема эта книга широко использует внутристочное группирование, согласованное со сделанными рекомендациями. Она также избегает расщепления многострочных инструкций на число строк, больше строго необходимого. Нужно лишь помнить, что в независимости от персонального вкуса следует соблюдать гребенчатую структуру.

Детали отступов

Гребенчатая структура использует отступы, для создания которых используется табуляция (но не пробелы!).

Вот какова иерархия отступов для основных видов конструкций, иллюстрируемых ниже следующим примером:

- Уровень 0: ключевые слова, вводящие первичные предложения класса. Они включают: **indexing** (начинающее предложение индексации), **class** (начинающее тело класса), **feature** (начинающее предложение **feature**, исключая случай, когда **feature** находится на той же строке, что и **class**), **invariant** (начинающее предложение инварианта) и заключительный **end** класса.
- Уровень 1: начало объявления компонента - **declaration**; разделы индексирования; предложения инварианта.
- Уровень 2: ключевые слова, начинающиеся последующими предложениями подпрограммы. Они включают: **require**, **local**, **do**, **once**, **ensure**, **rescue**, **end**.
- Уровень 3: Заголовочный комментарий подпрограмм и атрибутов; объявления локальных сущностей в подпрограмме; инструкции первого уровня.

Внутри тела программы может быть своя система отступов при гнездовании управляющих структур. Например, инструкция **if a then...** содержит две ветви, каждая с отступом. Эти ветви могут сами содержать инструкции цикла или выбора, приводящие к дальнейшему гнездованию. Еще раз заметим, что ОО-стиль этой книги приводит к простым подпрограммам, редко приводящим к высокому уровню гнездования.

Инструкция **check** задается с отступом. За ней, как правило, следует поясняющий комментарий, расположенный на следующем уровне справа от охраняемой инструкции.

```
indexing
  description: "Пример форматирования"
class EXAMPLE inherit
  MY_PARENT
    redefine f1, f2 end
```

```

MY_OTHER_PARENT
    rename
        g1 as old_g1, g2 as old_g2
    redefine
        g1
    select
        g2
    end
creation
    make
feature -- Initialization
    make is
        -- Сделать нечто
    require
        some_condition: correct (x)
    local
        my_entity: MY_TYPE
    do
        if a then
            b; c
        else
            other_routine
                check max2 > max1 + x ^ 2 end
                -- Из постусловия другой подпрограммы
                new_value := old_value / (max2 - max1)
        end
    end
feature -- Access
    my_attribute: SOME_TYPE
        -- Объяснение его роли (выровнено с комментарием для make)
        ... Объявления других компонентов и предложения feature ...
invariant
    upper_bound: x <= y
end --class Example

```

Пробелы

Белые пробелы (пробелы, табуляция, символы окончания строки) производят такой же эффект в программных текстах, как паузы в нотной записи.

Общее правило состоит в следовании, насколько это возможно, общепринятой практике обычного письменного языка. По умолчанию таковым языком является английский, хотя возможна адаптация правил к другим языкам.

Вот некоторые из следствий. Используйте пробелы:

- Перед открывающей скобкой, но не после: `f (x)` (но не `f(x)` в стиле C, или `f(x)`).
- После закрывающей скобки, если только следующим символом не является знак пунктуации, такой как точка или точка с запятой, но не перед скобкой. Следовательно: `proc1 (x); x := f1 (x) + f2 (y)`.
- После запятой, но не перед: `g (x, y, z)`.
- После двух тире, указывающих на начало комментария: `-- Комментарий`.

Аналогично, по умолчанию пробел ставится после, но не перед точкой с запятой:

`p1; p2 (x); p3 (y, z)`

Однако некоторые люди предпочитают французский стиль написания, согласно которому пробелы ставятся и до и после точки с запятой:

`p1 ; p2 (x) ; p3 (y, z)`

Выбирайте любой стиль, но применяйте его согласованно. (Эта книга использует английский стиль.) Английский и французский стили отличаются и для двоеточий. И здесь английский стиль предпочтительнее, как в `your_entity: YOUR_TYPE`.

Пробелы должны появляться до и после арифметических операций, как в `a + b`. (В этой книге из экономии пробелы могут опускаться, например в выражении `n+1`.)

Для точек нотация отходит от соглашений, принятых в естественном языке, поскольку точки используются в специальной конструкции, первоначально введенной в языке Simula. Как вы знаете, а . г означает: применить компонент г к объекту, присоединенному к а. Здесь не должно быть никаких пробелов ни до, ни после точки. В вещественных числах, таких как 3.14, используется обычная точка.

Приоритеты и скобки

Соглашения о приоритетах в нотации соответствуют традициям и принципу Наименьших Сюрпризов во избежание ошибок и двусмыслистостей.

Для ясности добавляйте скобки без колебаний; например, вы можете написать (а = (b + c)) implies (u /= v) несмотря на то, что смысл этого выражения не изменится, если все скобки будут опущены. В примерах этой книги зачастую расставлены "лишние" скобки, особенно в утверждениях, возможно, утяжеляя выражение, но избегая неопределенности.

Война вокруг точек с запятыми

С давних пор два равно известных клана живут в компьютерном мире и вражда между ними столь же ожесточена, как и в Вероне. Сепаратисты, наследники Algol и Pascal, сражаются за то, чтобы точка с запятой служила разделителем инструкций. Терминаллисты, объединившиеся под знаменами PL/I, C и Ada, хотят каждую инструкцию завершать точкой с запятой.

Пропагандистские машины обеих сторон приводят бесконечные аргументы в свою пользу. Культом Терминаллистов является единобразие: если каждая инструкция завершается одним и тем же маркером, никто и не посмеет задавать вопрос "должен ли я здесь ставить точку с запятой?" (ответ в языках Терминаллистов всегда - да, и всякого, кто нарушит предписание, ждет кара за измену). Они не хотят, чтобы нужно было удалять или добавлять этот символ при изменении местоположения инструкции, например, удаляя или внося ее в тело инструкции выбора.

Сепаратисты возносят хвалу элегантности их соглашения и его совместимости с математической практикой. Они рассматривают do instruction1; instruction2; instruction3 end как естественного родственника f (argument1, argument2, argument3). Кто в здравом уме, спрашивают они, предпочел бы писать f (argument1, argument2, argument3,) с ненужной заключительной запятой? Более того, они утверждают, что Терминаллисты фактически являются защитниками Компиляторщиков, жестоких людей, чьей единственной целью является обеспечение легкой жизни для разработчиков компиляторов, даже если это приведет к трудной жизни разработчиков приложений.

Сепаратисты должны постоянно бороться с инсинуациями, например, что их языки **не позволяют** лишних точек с запятой. Снова и снова они должны повторять истину: что каждый язык, заслуживающий этого имени, начиная с признанного патриарха этого племени, Algol 60, поддерживает понятие пустой инструкции, допускающей все виды написания:

```
a; b; c  
a; b; c;  
; a ;; b ;;; c;
```

Все строки здесь синтаксически правильны и эквивалентны. Они отличаются лишь наличием пустых инструкций в двух последних строках, которые любой уважающий себя компилятор спокойно удалит. Они указывают, насколько терпимее их соглашение, чем правило фанатичных соперников, когда каждая пропущенная точка с запятой является поводом для атак. Они же готовы принять столько точек с запятой, сколько Терминаллисты в силу привычки захотят добавить в тексты Сепаратистов.

Методы современной пропаганды нуждаются в научном обосновании и статистике. В 1975 году Терминаллисты провели исследование ошибок, для чего две группы программистов по 25 человек в каждой использовали языки, отличающиеся, среди прочего, соглашениями о точках с запятой. Его результаты широко цитировались и послужили оправданием терминаллистского соглашения в языке Ada. Из этих результатов следовало, что стиль Сепаратистов привел к десятикратному увеличению ошибок!

Взволнованные непрекращающейся вражеской пропагандой лидеры Сепаратистов обратились за помощью к автору настоящей книги, который, к счастью, вспомнил давно забытый принцип: **цитаты хороши, но лучше прочитать первоисточник**. Обратившись к оригинальной статье, он обнаружил, что язык Сепаратистов, используемый в сравнении, представлял мини-язык, предназначенный только для обучения студентов концепциям асинхронных процессов, в котором лишняя точка с запятой, как в begin a; b; end, рассматривалась как ошибка! Ни один реальный язык Сепаратистов, как отмечалось выше, такого правила не имеет. Из контекста статьи следовало также, что студенты, участвующие в эксперименте, имели предыдущий опыт работы с языком PL/I (Терминаллистов) и посему имели привычки ставить точки с запятыми повсюду. Так что результаты этой статьи не дают никаких оснований отдать предпочтение Терминаллизму в ущерб Сепаратизму.

Для некоторых других проблем, изучаемых в этой статье, таких пороков не замечено, так что она представляет интерес для людей, интересующихся проектированием языков программирования.

Все это показывает, что в таком чувствительном вопросе опасно принять точку зрения одной из сторон, особенно если хочешь сохранить друзей из обоих лагерей. Решение, принятое в нотации этой книги, радикально:

Правило Синтаксиса Точек с Запятой

Точки с запятой, как маркеры, ограничивающие инструкции, объявления и утверждения, возможны почти во всех позициях, где они могут появляться.

"Почти", поскольку в редких случаях, не встречающихся в этой книге, пропуск точки с запятой может стать причиной синтаксической неоднозначности.

Правило Синтаксиса Точек с запятой означает, что вы можете выбрать любой стиль:

- термиалист: каждую инструкцию, объявление или предложение утверждения заканчивать точкой с запятой;
- сепаратист: точки с запятой появляются между последовательными элементами, но не после последнего объявления компонента или локального предложения;
- умеренный Сепаратист: его стиль подобен стилю Сепаратистов, но он не беспокоится о лишних точках с запятой, появляющихся в результате привычки, или в результате перемещения элементов;
- минималист: вообще не ставит точек с запятой (исключая случаи, когда они требуются по принципу Стиля Точек с Запятой, приводимому ниже).

Это одна из тех областей, где предпочтительно позволять каждому пользователю следовать своему собственному стилю, поскольку его выбор не может стать причиной серьезных нарушений. Но при этом внутри одного класса или лучше внутри одной библиотеки классов следует придерживаться единого стиля, соблюдая следующий принцип:

Принцип Стиля Точки с Запятой

Если вы предпочитаете рассматривать точку с запятой как заключительную часть инструкции, (стиль Термиалиста), поступайте так для всех применимых элементов.

Если вы рассматриваете точку с запятой как разделитель, используйте ее только тогда, когда она синтаксически неизбежна или при разделении элементов, появляющихся на одной и той же строке.

Второе предложение управляет случаем двух или более элементов в одной строке:

```
found := found + 1; forth
```

Здесь точка с запятой должна всегда присутствовать. Ее пропуск будет ошибкой.

В этом обсуждении не дается совет, какой из четырех стилей является предпочтительным. Первое издание этой книги использовало стиль Сепаратистов. Но затем в обсуждениях с коллегами и опытными пользователями я обнаружил (помимо небольшой горстки Термиалистов) почти равное число Сепаратистов и Минималистов. Некоторые из Минималистов были весьма убедительными, в частности, университетский профессор, заявивший, что главная причина, по которой его студенты предпочитают данную нотацию, что в ней не обязательны точки с запятыми, - комментарий, который любой будущий проектировщик языка, при всех его грандиозных планах, должен найти поучительным или, по крайней мере, здравым.

Следует полагаться на свой вкус, пока он согласован и соответствует принципу Стиля Точки с Запятой. (Что же касается этой книги, то вначале скорее по привычке, чем по реальной привязанности я придерживался стиля Сепаратистов, но затем, наслушавшись призывов начать новую жизнь с приходом третьего тысячелетия и порвать со старыми привычками, я удалил все точки с запятыми в течение одной ночи сплошного разгула.)

Утверждения

Следует именовать утверждения для большей читабельности текста:

```
require  
  not_too_small: index >= lower
```

Это соглашение способствует созданию полезной информации при тестировании и отладке, поскольку, как вы

помните (см. [лекцию 11](#) курса "Основы объектно-ориентированного программирования"), метка утверждения включается в сообщение периода выполнения, создаваемое при нарушениях утверждений при включенном их мониторинге.

Это соглашение распространяется на утверждения, состоящих из нескольких предложений, расположенных на разных строках. В данной книге, опять-таки по соображениям экономии объема, метки опускаются, когда несколько утверждений располагаются на одной строке:

```
require
    index >= lower; index <= upper
```

При нормальных обстоятельствах лучше придерживаться официального правила и иметь по одному помеченному предложению утверждения на каждой строке текста.

Шрифты

При наборе программных текстов рекомендуются следующие соглашения, используемые в этой книге и связанных публикациях.

Основные правила

Используйте для программных элементов (имен классов, компонентов, сущностей и так далее) курсив. Это облегчает их включение в предложения обычного текста, как, например, "Можно видеть, что компонент `number` является запросом, а не атрибутом". (Слово `number` означает имя компонента, и вы не хотите, чтобы читатель мог подумать, что речь идет о числе компонентов!)

Ключевые слова, такие как `class`, `feature`, `invariant` и другие, набираются полужирным шрифтом (**boldface**).

Ключевые слова играют чисто синтаксическую роль: они не имеют собственной семантики. Как отмечалось ранее, есть несколько зарезервированных слов, не являющихся ключевыми, таких как `Current` и `Result`, обладающих семантикой выражений или сущностей. Они пишутся курсивом с начальным символом в верхнем регистре.

Следуя традициям математики, разделители - двоеточия, запятые, различные скобки и другие - всегда появляются прямыми (шрифтом *roman*), даже если они стоят после курсива¹¹. Подобно ключевым словам, они являются чисто синтаксическими элементами.

Текст комментария пишется прямым (*roman*) шрифтом. Имена программных элементов, в соответствии с ранее введенным правилом, даются в комментариях, курсивом. Например:

```
accelerate (s: SPEED; t: REAL) is
    -- Развить скорость s за максимум t секунд
...
set_number (n: INTEGER) is
    -- Сделать n новым значением number
...
```

В самих программных текстах, где невозможны вариации шрифта, такие вхождения формальных элементов в комментарии должны следовать соглашениям, уже упоминавшимся ранее: они появляются в одинарных кавычках

```
-- Сделать 'n' новым значением 'number'
```

(Заметьте, следует использовать разные символы для открывающей и закрывающей кавычки.) Инструментальные средства, обрабатывающие текст класса, такие как `short` и `flat`, знают об этом соглашении и при печати выводят закавыченные элементы курсивом.

Другие соглашения

Предыдущие соглашения о шрифтах хорошо работают для книг, статей, Web-страниц. В некоторых контекстах могут применяться другие подходы. Так при показе слайдов через проектор элементы, записанные курсивом, иногда и полужирным курсивом, не всегда читаются на экране.

В таких случаях я использую следующие соглашения:

- использую полужирный некурсивный шрифт для всего, что требует лучшего проектирования;
- выбираю достаточно широкий шрифт, такой как Bookman;

- вместо курсива использую цвет для распознавания различных элементов.

Цвет

Все формальные элементы появляются в цвете²⁾.

Библиографические замечания

Книга [Walden 1995] послужила источником примера, демонстрирующего, что подчеркивание в многословных идентификаторах separated_by_underscores предпочтительнее использования букв верхнего регистра internalUpperCase.

В [Gannon 1975] описан эксперимент по изучению влияния языковых предпочтений на уровень программистских ошибок.

Правила стандартного именования компонентов были впервые представлены в [M 1990b] и в деталях разработаны в [1994a].

Я получил важные комментарии от Рихарда Винера о студентах, оценивших возможность опускать точки с запятыми, и Кима Уолдена по поводу курсива и полужирного начертания.

Упражнения

У8.1 Стиль заголовочных комментариев

Перепишите следующий заголовочный комментарий в более подходящем стиле:

```
reorder (s: SUPPLIER; t: TIME) is
    -- Повторно заказывает текущую деталь у поставщика s,
    -- которую следует доставить до достижения срока t;
    -- эта программа работает только при условии,
    -- что срок поставки еще не истек
    require
        not_in_past: t >= Now
    ...
next_reorder_date: TIME is
    -- Выдает следующий срок, к которому текущая деталь
    -- должна быть повторно заказана
```

У8.2 Неоднозначность точки с запятой

Можете ли вы придумать случай, при котором пропуск точки с запятой между двумя инструкциями или утверждениями станет причиной синтаксической неоднозначности, или, по меньшей мере, создавал бы помехи наивному грамматическому разбору?

Подсказка: компонент может иметь в качестве цели выражение в скобках, как в (vector1 + vector2).count.

¹⁾ Это правило не всегда выдерживается в русском издании книги.

²⁾ В русском издании книги в отличие от оригинала, к сожалению, применяется черно-белая печать.

Основы объектно-ориентированного проектирования

9. Лекция: Объектно-ориентированный анализ

Направленный изначально на решение проблем реализации ОО-метод быстро распространился на весь жизненный цикл ПО. Особый интерес представляет приложение ОО-идей к моделированию как программных, так и непрограммных систем. Это применение объектной технологии для рассмотрения скорее проблем, чем решений, известно как объектно-ориентированный анализ. В последние годы появилось много книг по данной тематике, предложены специальные методы ОО-анализа. В разделе библиографии перечислены наиболее известные книги и ссылки на ресурсы Интернета для ряда популярных методов. Большинство концепций, представленных в предыдущих лекциях, имеет непосредственное отношение к ОО-анализу. В этой лекции мы рассмотрим особую роль объектного анализа и его отличия от других способов анализа. Для устранения несущественных различий необходимо отметить два терминологических момента. Во-первых, в качестве синонима понятия "анализ" используется термин системное моделирование или просто моделирование. Во-вторых, при обсуждении анализа в компьютерном сообществе существует тенденция использовать слово спецификация. В частности, специалисты направили значительные усилия на разработку методов и языков формальной спецификации, используя математические методы для системного моделирования. При единстве целей подходы могут отличаться. В последние годы два сообщества - разработчиков моделей и формальных спецификаций - стали обращать больше внимания на деятельность друг друга.

Цели анализа

Для понимания задач необходимо разобраться в роли анализа в разработке ПО и определить требования к методу анализа.

Задачи

Занимаясь анализом и создавая соответствующие документы, мы преследуем семь целей:

Цели проведения анализа

- A1 Понять проблему или проблемы, которые программная (или иная) система должна решить.
- A2 Задать значимые вопросы о проблеме и о системе.
- A3 Обеспечить основу для ответов на вопросы о специфических свойствах проблемы и системы.
- A4 Определить, что система должна делать.
- A5 Определить, что система не должна делать.
- A6 Убедиться, что система удовлетворит потребности ее пользователей и определить критерии ее приемки. Это особенно важно, когда система разработана по контракту для внешнего клиента.
- A7 Обеспечить основу для разработки системы.

Если анализ применяется к не программной системе или не связан с решением создания ПО, то существенными будут только цели A1, A2 и A3.

В случае ПО предполагается, что анализ следует за этапом **обоснования осуществимости (feasibility study)** проекта, на основании которого принято решение о разработке системы. Иногда эти этапы объединены в один, поскольку для определения возможности достижения удовлетворительного результата требуется глубокий анализ. Тогда необходимо добавить пункт A0, обеспечивающий принятие решения о разработке.

Безусловно связанные, перечисленные цели имеют отличия, что побуждает в дальнейшем искать дополняющие друг друга приемы. То, что хорошо для достижения одной цели, может быть неприемлемым для другой.

Цели A2 и A3 наименее полно освещены в литературе и заслуживают особого внимания. Одним из важнейших преимуществ анализа вне зависимости от конечного результата является то, что в процессе его задаются важные вопросы (A2). Какова максимально допустимая температура? Какие категории служащих существуют? В чем разница между облигациями и акциями? Метод анализа позволяет развеять иногда фатальный для разработки туман двусмысленности, предоставляя специалистам данной конкретной области возможность подготовить необходимые исходные данные. Нет ничего хуже, чем обнаружить на завершающем этапе реализации, что маркетинг и технические отделы заказчика имеют противоречивые взгляды на обслуживание оборудования, по умолчанию учитывалась только одна из этих позиций, и никто не догадался уточнить требования заказчика. Именно к пункту A2 постоянно возвращаются в процессе анализа, если возникают тонкие вопросы или противоречивые интерпретации.

Требования

Практические требования к процессу анализа и поддерживающей нотации следуют из приведенного списка целей:

- возможность участия в анализе и обсуждении результатов неспециалистов в области ПО (A1, A2);
- форма представления результатов анализа должна быть непосредственно пригодной для разработчиков ПО (A7);
- масштабируемость решения (A1);
- нотация не должна допускать неоднозначного толкования (A3);
- возможность для читателя быстро получить общее представление об организации системы или подсистемы (A1, A7).

Масштабируемость необходима для сложных и (или) больших систем. Метод должен обеспечивать описание высокоуровневой структуры проблемы или системы и выделить в этом описании необходимое число уровней абстракции. Это позволит в любой момент сосредоточиться как на большой, так и на маленькой части системы при сохранении полной картины. Свойства структурирования и абстрагирования объектной технологии будут здесь незаменимыми.

Масштабируемость также означает, что критерии расширяемости и повторного использования, занимающие важное место в предшествующих обсуждениях, в той же мере применимы к анализу, как и к проектированию и реализации ПО. При модификации и создании новых систем можно применять **библиотеки элементов спецификаций** аналогично использованию библиотек программных компонент при построении реализаций.

Облака и провалы

Совместить два последних требования непросто. Конфликт, уже обсуждавшийся в контексте абстрактных типов данных, был настоящим бедствием для всех методов анализа и языков спецификаций. Как однозначно задать определенные свойства, не говоря слишком много? Как обеспечить обозримые, достаточно общие структурные описания без риска неопределенности?

Аналитик идет по горной тропе. Слева скрытая за облаками вершина - туманное царство. Справа подстерегают провалы чрезмерной спецификации, в которые так легко угодить, увлеквшись деталями реализации в ущерб общим свойствам системы.

Боязнь риска чрезмерной спецификации характерна для людей, занимающихся анализом. (Говорят, что в этом кругу для уничтожения автора, предлагающего подход X, достаточно сказать "Подход X хороший, но разве он не дитя подхода, ориентированного на реализацию?") По этой причине при проведении анализа часто впадают в другую крайность, полагаясь на описание целостной картины, используя графическую нотацию (часто в виде облаков), неспособную выразить семантические свойства, тогда как для достижения цели A2 требуются точные ответы на конкретные вопросы.

Подобные нотации используются многими традиционными методами анализа. Их успех основан на способности перечисления компонентов системы и графического описания отношений между ними, оставаясь на уровне блок-схем. Для проектов ПО это источник риска, так как при этом слабо отражается семантика. Убежденность, что анализ успешно завершен, в то время как определены лишь основные компоненты и их отношения, не учитывающие глубинные свойства спецификации, может привести к критическим ошибкам.

Далее в данной лекции будут рассмотрены идеи, позволяющие согласовать цели структурного описания и семантической точности.

Изменчивая природа анализа

В литературе почти не упоминается о том, что наиболее значительный вклад объектной технологии в анализ является не техническим, а организационным. Объектная технология не только обеспечивает новые пути проведения анализа, но и затрагивает природу задачи и ее роль в процессе построения ПО.

Эти изменения следуют из того, что акцент переносится на повторное использование. Вместо того чтобы начинать каждый новый проект на пустом месте, рассматривая требования клиента как Евангелие, учитывается наличие постоянно расширяющегося набора программных компонентов, разработанных во внутренних и внешних проектах. Так что задача сводится не к выполнению спущенного сверху заказа, а к ведению переговоров.

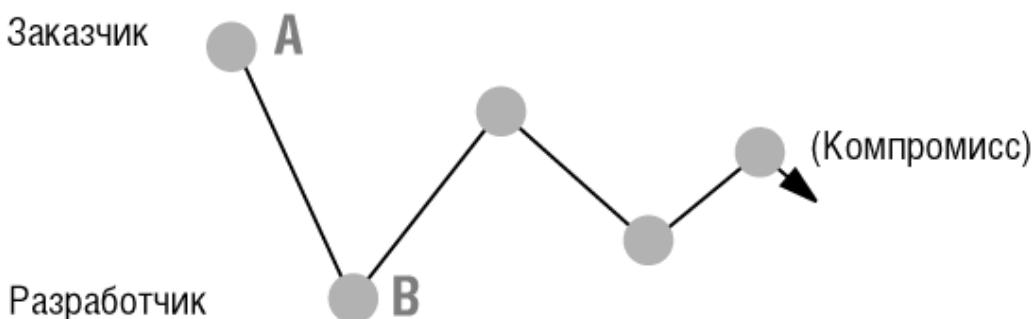


Рис. 9.1. Выработка требований в процессе переговоров

Этот процесс отображен на [рис. 9.1](#). Заказчик начинает с позиции А. Разработчик выступает со своим предложением в В, снимая часть исходных требований или модифицируя их. Его предложения в значительной степени подразумевают повторное использование существующих компонентов, следовательно, снижают затраты средств и времени. Клиенту функциональные потери могут показаться чрезмерными, начинается стадия переговоров, в конечном счете приводящая к приемлемому компромиссу.

Торговля присутствовала всегда. Требования заказчика рассматривались как Евангелие только в некоторых идеализированных описаниях процесса разработки ПО, в учебной литературе и в некоторых правительственные контрактах. В большинстве нормальных ситуаций разработчики обладают возможностью обсуждения требований. Но только с появлением объектной технологии этот неофициальный феномен становится официальной частью процесса разработки ПО, занимая все более важное место по мере развития библиотек повторного использования.

Вклад объектной технологии

Объектная технология оказывает влияние и на методы анализа.

Важно то, что основа, заложенная в предшествующих лекциях, содержит более чем достаточно средств, чтобы приступить к моделированию. "Более чем достаточно" означает, что нотация содержит ненужные для анализа элементы:

- инструкции (присваивания, циклы, вызовы процедур, ...) и все, что с ними связано;
- тела подпрограмм в форме do (отложенные подпрограммы deferred нужны для указания операций, реализация которых отсутствует).

При игнорировании этих императивных элементов совокупность остальных элементов представляет действенный метод моделирования и нотацию. В частности:

- **Классы** дают возможность организовать описания систем на основе типов объектов в широком понимании слова "объект" (не только физические объекты, но также и важнейшие концепции предметной области).
- Подход **АТД** - идея характеризовать объекты с помощью допустимых операций и их свойств - приводит к ясным, абстрактным, эволюционным спецификациям.
- Отношения между компонентами сводятся к двум основным механизмам - отношениям **клиента** и **наследования**. Клиентские отношения, в частности, охватывают такие понятия моделирования, как "быть частью (чего-либо)", агрегирования и соединения.
- При обсуждении объектов было показано, что различия между **ссылочными** и **развернутыми** клиентами соответствуют двум основным видам моделируемых соединений.
- **Наследование** - простое, множественное и дублируемое - поддерживает классификацию. Ценность для моделирования представляют даже такие специфические механизмы наследования, как переименование.
- **Утверждения** необходимы, чтобы охватить семантику систем, позволяя задавать свойства, отличные от структурных. Проектирование по Контракту мощное руководство по анализу.
- **Библиотеки классов** повторного использования, особенно благодаря отложенным классам, обеспечивают готовыми элементами спецификаций.

Это вовсе не подразумевает, что ОО-подход обеспечивает все потребности системного анализа (вопрос, который будет обсужден далее), но он представляет реальную основу. Последующий пример послужит доказательством.

Программирование телевизионного вещания

Рассмотрим конкретное применение ОО-концепций в интересах чистого моделирования.

В качестве примера рассмотрим организацию планирования сетки телевизионного вещания. Поскольку это знакомая всем прикладная задача, можно начать ее решение без привлечения экспертов и будущих пользователей. Для проведения анализа достаточно полагаться на понимание рядового телезрителя.

Хотя данная попытка является лишь прелюдией к созданию автоматизированной системы управления телевизионным вещанием, в данном случае это несущественно, поскольку нас интересуют исключительно вопросы моделирования.

Графики вещания

Сосредоточимся на 24-часовом графике вещания. Его представляет класс (абстракция данных) SCHEDULE. График содержит последовательность отдельных программных сегментов:

```
class SCHEDULE feature
    segments: LIST [SEGMENT]
end
```

При проведении анализа необходимо постоянно помнить об опасности избыточной спецификации. Не является ли избыточным использование LIST? Нет: LIST это отложенный класс, описывающий абстрактное понятие последовательности, что соответствует характеру телевизионного вещания - одновременная передача двух сегментов невозможна. Использование LIST фиксирует свойство проблемы, а не ее решение.

Попутно отметим важность повторного использования: применение классов, подобных LIST, сразу открывает доступ к целому набору операций со списками: команде `ruit` для добавления элементов, запросу `count` для получения номера элемента и другим.

Свести понятие графика к списку его сегментов нельзя. Объектная технология, как следует из обсуждения абстрактных типов данных, является неявной; она описывает абстракции путем перечисления их свойств. График передач - это нечто большее, чем список его сегментов, так что необходим отдельный класс. Другие свойства представляются естественным образом:

```
indexing
    description: "24-часовой график ТВ вещания"
deferred class SCHEDULE feature
    segments: LIST [SEGMENT] is
        -- Последовательность сегментов
        deferred
    end
    air_time: DATE is
        -- Дата вещания
        deferred
    end
    set_air_time (t: DATE) is
        -- Установка даты вещания
    require
```

```

        t.in_future
deferred
ensure
    air_time = t
end
print is
    -- Вывод графика на печать
deferred
end
end

```

Отметим использование отложенной реализации. Это связано с природой анализа, не зависящего от реализации, а часто и от проектирования, так что отложенная форма записи является удобным инструментом. Можно, конечно, вместо отложенной спецификации использовать формализм типа краткой формы. Однако есть два важных довода в пользу полной нотации:

- При записи текста в полном соответствии с синтаксисом можно использовать весь набор средств, предоставляемый средой разработки ПО. В частности, механизм компиляции играет в этом случае ту же роль, что и совершенные CASE-средства, осуществляя контроль спецификации на использование типов и других ограничений, позволяя избежать противоречий и двусмысленностей и существенно снизить затраты времени. Средства просмотра и документирования хорошей ОО среды столь же полезны для анализа, как и для этапов проектирования и реализации.
- Использование стандартной нотации существенно облегчает последующий переход к проектированию и реализации программной системы. В этом случае работа сводится к добавлению новых классов, эффективных версий отложенных реализаций и новых компонентов. Такой подход обеспечивает бесшовный процесс разработки, обсуждаемый в следующей лекции.

Класс содержит булев запрос `in_future` для объекта типа DATE, для указания на будущее время выхода в эфир. Следует отметить первое использование предусловия и постусловия для выражения семантических свойств системы в процессе анализа.

Сегменты

Прежде чем продолжать уточнение и расширение SCHEDULE, необходимо обратиться к понятию SEGMENT. Можно начать со следующего описания:

```

indexing
    description: "Отдельные сегменты графика вещания"
deferred class SEGMENT feature
    schedule: SCHEDULE is deferred end
        -- График, содержащий данный сегмент
    index: INTEGER is deferred end
        -- Положение сегмента в графике
    starting_time, ending_time: INTEGER is deferred end
        -- Время начала и завершения
    next: SEGMENT is deferred end
        -- Следующий сегмент, если он существует
    sponsor: COMPANY is deferred end
        -- Основной спонсор
    rating: INTEGER is deferred end
        -- Рейтинг сегмента (допустимость просмотра детьми и т.д.)
    ... Опущены команды change_next, set_sponsor, set_rating и др. ...
    Minimum_duration: INTEGER is 30
        -- Минимальная длительность сегмента в секундах
    Maximum_interval: INTEGER is 2
        -- Максимальная пауза между соседними сегментами в секундах
invariant
    in_list: (1 "=" index) and (index "=" schedule.segments.count)
    in_schedule: schedule.segments.item(index) = Current
    next_in_list: (next /= Void) implies (schedule.segments.item(index + 1) = next)
    no_next_if_last: (next = Void) = (index = schedule.segments.count)
    non_negative_rating: rating >= 0
    positive_times: (starting_time > 0) and (ending_time " 0)
    sufficient_duration: ending_time - starting_time >= Minimum_duration
    decent_interval: (next.starting_time) - ending_time >= Maximum_interval
end

```

Каждый сегмент "может определить" график, частью которого он является, и свое положение с помощью запросов `schedule` и `index`. Он содержит запросы `starting_time` и `ending_time`, к которым можно добавить и запрос `duration`, с инвариантом, связывающим длительность сегмента с временем начала и завершения. Такая избыточность допустима в системном анализе, добавление избыточных свойств отражает особенности, представляющие интерес для пользователей или разработчиков. Отношения между избыточными элементами фиксируются в соответствующих инвариантах. Инварианты `in_list` и `in_schedule` отражают позицию сегмента в списке сегментов и в графике.

Сегмент также "знает" о следующем сегменте. Инварианты отражают требования последовательности: `next_in_list` указывает, что если позиция текущего сегмента - `i`, то следующего - `i + 1`. Инвариант `no_next_if_last` служит признаком того, является ли данный сегмент последним в графике.

Два последних инварианта выражают ограничения на продолжительности: `sufficient_duration` определяет минимальную продолжительность в 30 секунд для фрагмента программы, являющегося сегментом, а `decent_interval` - максимальную паузу в 2 секунды между двумя последовательными сегментами (темный экран).

Спецификация класса содержит два недостатка, которые почти наверняка придется устраниить при следующей итерации. Во-первых, время и продолжительность выражаются целыми числами (в секундах). Целесообразнее применить более абстрактный вариант - использование библиотечных классов DATE, TIME и DURATION. Во-вторых, понятие SEGMENT охватывает два отдельных понятия: фрагмент телевизионной программы и временное планирование. Разграничение этих понятий достигается добавлением в SEGMENT атрибута

```
content: PROGRAM_FRAGMENT
```

и нового класса PROGRAM_FRAGMENT для описания программного фрагмента вне зависимости от его положения в графике. Компонент duration нужно поместить в PROGRAM_FRAGMENT, а новое инвариантное предложение в SEGMENT примет вид:

```
content.duration = ending_time - starting_time
```

Для краткости в остальной части этого эскиза содержание обрабатывается как часть сегмента. Подобные дискуссии типичны для процесса анализа, поддержанного ОО-методом: мы исследуем различные абстракции, обсуждаем, необходимы ли для них различные классы, перемещаем компоненты, если считаем, что они не на своем месте.

Сегмент имеет основного спонсора и рейтинг. Хотя здесь также более выгоден отдельный класс, рейтинг определен как целое число, большее значение рейтинга означает более строгие ограничения. Значение 0 соответствует сегменту, доступному всем зрителям.

Программы и реклама

Развивая далее понятие SEGMENT, введем два вида сегментов: программные и коммерческие (рекламные сегменты). Это наводит на мысль использовать наследование.

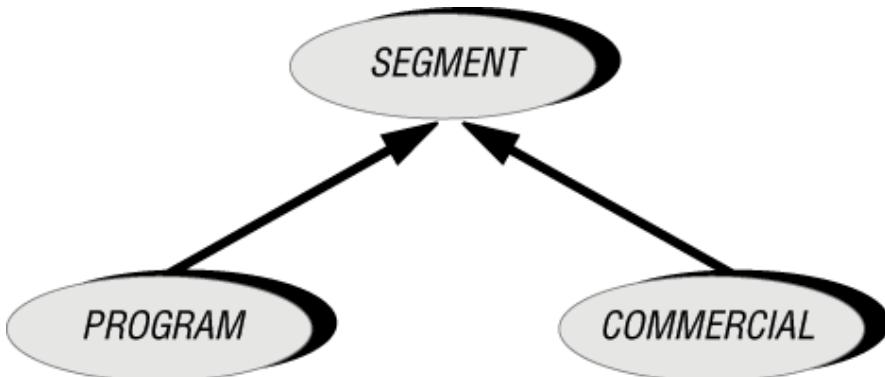


Рис. 9.2. Программные сегменты и рекламные паузы

Использование наследования в процессе анализа всегда вызывает подозрения. Не следует создавать лишних классов там, где достаточно введения отличительного свойства. Руководящий критерий был дан при рассмотрении наследования: действительно ли каждый предложенный класс реально соответствует отдельной абстракции, характеризующейся специфическими особенностями? В данном случае использование нового класса оправдано, поскольку разумно предложить специальные свойства классу COMMERCIAL, как будет показано ниже. Наследование сопровождается преимуществами открытости: можно позже добавить нового наследника INFOMERCIAL (рекламный ролик) для описания сегмента другого вида.

Начнем работу над COMMERCIAL:

```
indexing
  description: "Рекламный сегмент"
deferred class COMMERCIAL inherit
  SEGMENT
    rename sponsor as advertizer end
feature
  primary: PROGRAM is deferred
    -- Программа, с которой связан данный сегмент
  primary_index: INTEGER is deferred
    -- Индекс сегмента primary
  set_primary (p: PROGRAM) is
    -- Связать рекламу с p
  require
    program_exists: p /= Void
```

```

    same_schedule: p.schedule = schedule
    before: p.starting_time <= starting_time
defered
ensure
    index_updated: primary_index = p.index
    primary_updated: primary = p
end
invariant
    meaningful_primary_index: primary_index = primary.index
    primary_before: primary.starting_time <= starting_time
    acceptable_sponsor: advertiser.compatible (primary.sponsor)
    acceptable_rating: rating <= primary.rating
end

```

Использование переименования является еще одним примером полезного средства нотации. Оказывается, оно необходимо не только на этапе реализации, но и для моделирования. Спонсора рекламного фрагмента уместнее называть рекламодателем.

Каждый рекламный сегмент присоединен к некоторому программному (некоммерческому) сегменту, индекс которого в графике задается значением `primary_index`. Два первых инварианта отражают условия последовательности, последние два - совместимости:

- Если программа имеет спонсора, то в течение ее показа приемлема далеко не любая реклама. Никто не будет рекламировать Pepsi-Cola в телешоу, спонсируемом Coca-Cola. Можно выполнить запрос к некоторой базе данных о совместимости.
- Рейтинг рекламы должен соответствовать программе: реклама бульдозера неуместна в передаче для малышей.

Понятие `primary` требует уточнения. На этом этапе анализа становится ясно, что нужно добавить новый уровень: вместо графика, являющегося последовательностью программных и рекламных сегментов, необходимо рассмотреть последовательность телепрограмм (описывается классом `SHOW`), каждая из которых имеет собственные компоненты, спонсора и последовательность сегментов. Такое усовершенствование и уточнение, разработанное на основе лучшего понимания проблемы и опьте первых шагов, является нормальным компонентом процесса анализа.

Деловой регламент

Мы видели, как инварианты и другие утверждения могут охватить семантические ограничения прикладной области. В терминах анализа это называют деловым регламентом: для класса `SCHEDULE` можно планировать размещение сегмента только в будущем; в классе `SEGMENT` определено, что пауза между двумя сегментами не должна превышать установленного значения; в `COMMERCIAL` рейтинг рекламы должен соответствовать рейтингу передачи.

Принципиальным вкладом ОО-метода является возможность для таких правил использования утверждений и принципов Проектирования по Контракту наряду с заданием структуры.

Практическое предупреждение: даже если реализация не предусматривается, остается риск чрезмерной спецификации. Нужно включать в текст анализа только правила, имеющие высокую степень достоверности и долговечности. Если какое-то правило может меняться, то лучше использовать абстракцию, чтобы оставить место для необходимой адаптации. Например, могут измениться правила совместимости спонсора и рекламодателя, поэтому выбранная абстрактная форма инварианта `acceptable_sponsor` является приемлемой. Важнейшим преимуществом анализа является возможность выбора, какие особенности принимать во внимание, а какие игнорировать. Здесь действует то же соображение, которое было высказано при обсуждении абстрактных типов данных: нам нужна правда, только правда и ничего кроме правды.

Оценка

Пример с телевизионной программой находится еще на начальной стадии, но он содержит достаточно для понимания общих принципов подхода. Использованные ОО-концепции и нотация для решения общих задач системного моделирования являются удивительно мощными и интуитивно понятными. А ведь их первоначальное назначение - разработка ПО, и могло казаться, что они непригодны для иных целей. Здесь же метод и нотация в полной мере проявили свои универсальные возможности описания систем различных типов, задавая при этом как общую структуру систем, так и детали семантики.

Рассмотренная выше спецификация не содержит ничего, что связывало бы ее с реализацией или компьютерами. Концепции объектной технологии используются исключительно для описательных целей, компьютеры для этого не нужны.

Естественно, что, если в дальнейшем будет разрабатываться программная система управления работой телестанции, то имеющееся описание обладает неоспоримым преимуществом, поскольку форма его представления в синтаксическом и структурном отношении находится в полном соответствии с описанием ПО. Это основа для бесшовного перехода к проектированию и реализации. В завершенной системе удастся сохранить многие классы, введенные в процессе анализа, снабдив их соответствующей реализацией.

Представление анализа: разные способы

Использование спецификаций, представленных на языке, подобном языку программирования, проиллюстрированное на

примере телепрограмм, ставит очевидный вопрос практичности в обычных условиях.

Источником некоторого скептицизма могут быть неудобства восприятия такой нотации для людей, знакомящихся с результатами анализа. Анализ в большей степени, чем любой другой этап разработки, невозможен без сотрудничества с экспертами в данной области, будущими пользователями, менеджерами, руководителями проектов. Можно ли ожидать, что они будут читать спецификацию, которая на первый взгляд напоминает программный текст (хотя это - чистая модель)?

Неожиданно часто ответ - да. Понимание той части нотации, которая служит для анализа, не требует глубоких знаний в программировании, достаточно понимания элементов логики и способа рассуждений, характерных для любой дисциплины. Автор может засвидетельствовать, что успешно использовал такие спецификации с людьми, имеющими различный уровень опыта и образования.

Но это - не конец истории. Приходится сотрудничать с людьми, нерасположенными к формализму. И даже те, кто ценит мощь формализма, нуждаются в других представлениях, в частности, графических. Сражения между графикой и формализмом, формализмом и естественным языком не имеют смысла. На практике для описания нетривиальной системы могут использоваться дополняющие друг друга способы представления:

- Формальный текст, как в предыдущем примере.
- Графическое представление, отображающее системные структуры в виде диаграмм с помощью "пузырьков и стрелок". Графические образы представляют классы, кластеры, объекты и отношения клиентские и наследования.
- Документ с требованиями на естественном языке.
- Таблица, например, в представлении метода BON далее в этой лекции.

Каждый вариант имеет уникальные преимущества для достижения одних целей анализа и ограничения по отношению к другим целям. В частности:

- Документы на естественном языке незаменимы для передачи основных идей и объяснения тонких нюансов. Их недостатком является склонность к неточности и двусмысленности.
- Таблицы полезны при выделении набора связанных свойств, таких как основные характеристики класса - его родители, компоненты, инварианты.
- Графические представления превосходны для описания структурных свойства проблемы или системы, показывая компоненты и их отношения. Этим объясняется успех "пузырьков-и-стрелок", продвигаемых "структурным анализом". Их ограниченность проявляется, когда наступает время строгого описания семантических свойств. Например, графическое описание является не лучшим местом для ответа на вопрос, какова максимальная длительность рекламной паузы.
- Формальные текстовые представления, являются лучшим инструментом для ответов на такие конкретные вопросы, но не могут конкурировать с графикой, когда нужно быстро понять, как организована система.

Обычный аргумент в пользу графических представлений - шаблонная фраза "изображение стоит тысячи слов". Здесь есть доля правды. Блок-схемы действительно непревзойденно передают впечатление о структуре. Однако это высказывание игнорирует тот факт, что с помощью слов можно передать любые подробности, а неточности рисунка могут приводить к ошибкам. Когда диаграмма предлагается в качестве окончательной спецификации каких-либо тонких особенностей системы, самое время вспомнить загадки типа "найдите все различия" на двух обманчиво похожих рисунках.

Итак, для хорошего метода анализа требуется возможность применения любого из этих представлений и свободный переход от одного к другому.

Возникает проблема синхронизации всех представлений. Для этого одно из представлений выбирается в качестве образца (ссылки), а согласованное внесение добавлений и изменений во все остальные представления выполняется специальным программным инструментарием. Лучшим кандидатом на роль образца (фактически единственным) является формальный текст, поскольку только он строго определен и способен охватить и семантику, и структурные свойства.

При таком подходе формальные описания не являются единственным средством анализа. Это дает возможность использовать все разнообразие инструментальных средств, приспособленных к различным уровням квалификации и личным вкусам участников анализа (программисты, менеджеры, конечные пользователи). Поддержка формальных текстов и дополнительных средств анализа может быть встроена в среду программирования. Графическая нотация может использовать CASE-средства, пригодные для создания структурных диаграмм. Тексты на естественном языке могут поддерживаться системой обработки и управления документами. Можно обеспечить аналогичные средства поддержки таблиц. Различные инструментальные средства могут быть как автономными, так и интегрированными в единую среду разработки или анализа.

Изменения в графическом или табличном способе представления должны немедленно отражаться в формальном представлении и наоборот. Например, если на графике класс С показан как потомок класса А ([рис. 9.3](#)), то при перемещении указателя на класс В соответствующие средства автоматически внесут необходимые изменения в формальный текст и табличное представление. Наоборот, изменения формального описания сопровождаются модификацией графического и табличного представлений.

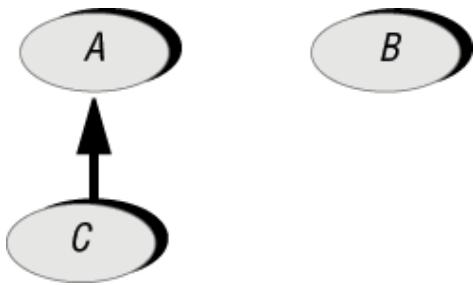


Рис. 9.3. Наследственная Связь

Гораздо труднее с помощью инструментальных средств вносить изменения в описание, сделанное на естественном языке. Но если система предусматривает использование структурированных системных описаний с лекциями, разделами и абзацами, то поддержание связей между формальным текстом и документом на естественном языке возможно. Например, можно указать, что некоторый класс или его компонент связан с определенным абзацем требований. Это особенно полезно, когда инструментарий позволяет при внесении любых изменений в исходные требования вывести список всех зависимых элементов формального описания.

Интересно и другое направление - создание из формальных описаний текстов на естественных языках. Идея заключается в восстановлении из формального системного описания обычного текста, выражающего ту же самую информацию в форме, которая не испугает читателей, нерасположенных к формализму. Не составит труда представить себе инструмент, который на основе нашего эскиза анализа сформировал бы следующий текст:

1. Системные понятия

Основные понятия системы:

SCHEDULE, SEGMENT, COMMERCIAL, PROGRAM ...

SCHEDULE обсуждается в пункте 2; SEGMENT обсуждается в пункте 3; [и т.д.]

2. Понятие SCHEDULE

...

3. ...

4. Понятие COMMERCIAL

1. Общее описание:

Рекламные сегменты

2. Вводные замечания.

Понятие COMMERCIAL - это специализированный вариант понятия SEGMENT, и имеет те же свойства и операции, исключения приведены ниже.

3. Переименованные операции.

Свойство sponsor для SEGMENT названо advertizer для COMMERCIAL.

...

4. Переопределенные операции.

...

5. Новые операции.

Следующие операции характеризуют COMMERCIAL:

primary, запрос, возвращающий связанное понятие PROGRAM

Аргументы: нет [Если нужно, то здесь перечисляются аргументы]

Описание:

Программа, с которой связана реклама

Начальные условия:

...

Конечные условия:

...

...Другие операции...

6. Ограничения.

...Изложение смысла инвариантных свойств...

5. Понятие PROGRAM

...

и т. д.

Все фразы ("Основные понятия системы", "Следующие операции характеризуют..." и т. д.) взяты из стандартного набора предопределенных формулировок, так что это не настоящий "естественный" язык. Тем не менее, может быть создана достаточная иллюзия и достигнут приемлемый для неспециалистов результат. При этом гарантируется совместимость с формальным представлением, так как текст был механически получен именно из него.

Автору неизвестен инструмент, реализующий эту идею, однако цель представляется вполне достижимой. Проект создания такого инструмента гораздо более реалистичен, нежели давно предпринимаемые попытки решения обратной задачи, которые были безуспешными из-за трудностей автоматизированного анализа текстов на естественных языках. Создание текстов более простая задача, точно так же, как реализация синтеза речи гораздо проще ее распознавания.

Такая возможность существует благодаря общности формальной нотации, особенно благодаря наличию поддержки утверждений, что позволяет включить полезные семантические свойства в генерированные тексты на естественном языке. Без утверждений мы остались бы в состоянии неопределенности - в облаках.

Методы анализа

Далее приведен перечень наиболее известных методов ОО анализа приблизительно в хронологическом порядке их опубликования. Несмотря на то, что основное внимание уделяется анализу, большинство методов содержит элементы, относящиеся к разработке и даже реализации. Краткие аннотации не позволяют воздать должное методам и для дальнейшего изучения рекомендуются источники, перечисленные в конце этой лекции.

Метод **Coad-Yourdon** первоначально был направлен на воплощение идей структурного анализа. Он включает в себя пять этапов: поиск классов и объектов, исходя из предметной области и на основе анализа функций системы, идентификация структур путем поиска отношений "обобщение-специализация" и "общее-частное", определение "субъектов" (групп класс-объект), определение атрибутов; определение сервисов.

Метод **OMT** (Object Modeling Technique) объединяет концепции объектной технологии и моделирования, основываясь на понятии "сущность-отношение" (entity-relation). Метод включает статическую и динамическую модели. Статическая модель базируется на концепциях класса, атрибута, операции, отношения и агрегирования, динамическая - на основе диаграмм "событие-состояние" позволяет дать абстрактное описание предполагаемого поведения системы.

Метод **Shlaer-Mellor** изначально ориентирован на создание моделей, допускающих проверку поведения системы, независимо от конкретного проектирования и реализации. Для этого в исходной проблеме выделяются области, задающие различные аспекты: предметная, сервиса (интерфейс пользователя), архитектурная, реализации. Отдельные решения затем связываются воедино для создания завершенной системы.

Наличие в **Shlaer-Mellor** и ряде методов моделирования элементов архитектуры, проектирования и реализации иллюстрирует высказанную ранее мысль о том, что амбиции методов часто выходят за рамки анализа.

Метод **Martin-Odell**, известный также как **OOIE** (Object-Oriented Information Engineering), разделяется на две части. В первой части анализируется объектная структура, идентифицируются типы объектов, их состав, отношения наследования. Вторая часть анализирует поведение объектов, определяемое динамической моделью, учитывающей состояния объектов и события, которые могут изменить эти состояния.

Метод **Booch** использует логическую модель (класс и объектная структура) и физическую модель (модуль и архитектура процесса), включая как статические, так и динамические компоненты, в ней применяются многочисленные графические символы. Планируется его включение в язык анализа UML (Unified Modeling Language) (см. ниже).

Метод **OOSE** (Object-Oriented Software Engineering), также известный как метод Jacobson или как Objectory (название оригинального средства поддержки), основан на использовании сценариев для выявления классов. Рассматривается пять моделей сценариев: доменная модель исходной области приложения и четыре модели этапов разработки - анализа, проектирования, реализации, тестирования.

Метод **OSA** (for Object-oriented Systems Analysis) предназначен скорее для создания общей модели процесса анализа, а не пошаговой процедуры. Он состоит из трех частей: модели объектных отношений, описывающей объекты, классы и их отношения друг с другом и с "реальным миром", модели объектного поведения, обеспечивающей динамическое представление через состояния, переходы, события, действия и исключения и модели объектного взаимодействия, определяющей возможные взаимодействия между объектами. Метод также поддерживает понятия представления, обобщения и специализации, которые используются для описания взаимодействия и моделей поведения.

Метод **Fusion** направлен на объединение некоторых из лучших идей более ранних методов. Для анализа он включает объектную модель для данной прикладной задачи и модель интерфейса для описания поведения системы. Модель интерфейса основана на операционной модели, определяющей события и результирующие действия, и модели жизненного цикла, описывающей сценарии эволюции системы. Аналитики должны поддерживать словарь данных для сбора всей информации от различных моделей.

Метод **Syntropy** определяет три модели. Наиболее важная модель - "модель реальной или воображаемой ситуации", описывающая элементы ситуации, их структуру и поведение. Модель спецификации - абстрактная модель, рассматривающая систему как механизм реакции на воздействия, располагающий неограниченными аппаратными ресурсами. Модель реализации принимает во внимание реальную вычислительную среду. Предусмотрены различные способы представления каждой модели: описание типов объекта и их статических свойств, диаграммы состояний подобные диаграммам переходов в ОМТ для описания динамики поведения, диаграммы механизмов для реализации. Метод поддерживает описание одних и тех же объектов с помощью различных интерфейсов, не ограничиваясь простым разделением интерфейса и реализации.

Метод **MOSES** включает пять моделей: объект-класс, событие для описания сообщений, инициируемых в результате вызова сервисов объекта, "объектные диаграммы" для моделирования динамики изменения состояния, наследование, сервисную структуру для отображения потока данных. Подобно рассматриваемому ниже методу BON, в методе MOSES подчеркивается важность контрактов в определении класса и используются предусловия, постусловия и инварианты в стиле данной книги. Его модель "фонтанирующего процесса" определяет стандартные документы, создаваемые на каждой стадии.

Метод **SOMA** (Semantic Object Modeling Approach) использует "Объектную Модель Задачи", чтобы сформулировать требования и преобразовать их в "Деловую Объектную Модель". Это одна из немногих попыток извлечения выгоды из формальных подходов, использующая понятие контракта для описания деловых правил, применимых к объектам.

Во время написания книги, разрабатывались два самостоятельных проекта объединения существующих методов. Первый (Brian Henderson-Sellers, Don Firesmith, Ian Graham и Jim Odell) направлен на создание объединенного метода OPEN. Целью второго проекта Rational Corporation является разработка UML (унифицированного языка моделирования), используя в качестве отправной точки методы OMT, Booch и Jacobson.

Нотация BON (Business Object Notation)

Каждый из рассмотренных подходов имеет свои сильные стороны. Метод Business Object Notation (BON), предложенный Nerson и Walden, при минимальной сложности обеспечивает максимальные преимущества и может служить примером комплексного подхода к ОО-анализу. Данный краткий обзор основных особенностей метода ограничивается обсуждением его вклада в анализ. Для более подробного знакомства можно рекомендовать указанную в библиографии монографию.

На начальном этапе разрабатывался графический формализм представления системных структур. В дальнейшем BON из способа нотации превратился в законченный метод разработки, но оригинальное название было сохранено. BON используется во многих прикладных областях для анализа и разработки систем, в том числе очень сложных.

Метод BON основан на трех принципах: бесшовность, обратимость и контрактность. Бесшовность - непрерывность процесса на протяжении всего жизненного цикла ПО. Обратимость - поддержка прямого и обратного процессов разработки: от анализа к проектированию и реализации и наоборот. Контрактность (вспомните о Проектировании по Контракту) - точное определение семантических свойств каждого программного элемента. BON - практически единственный популярный метод анализа, использующий развитый механизм утверждений, что позволяет аналитикам определить не только структуру системы, но и ее семантику (ограничения, инварианты, свойства ожидаемых результатов).

Ряд других свойств выделяют BON среди ОО-методов:

- Он обеспечивает "масштабируемость", о которой упоминалось в начале этой лекции. Различные средства и соглашения дают возможность выбрать уровень абстракции системы или описания подсистемы, сосредоточиться на компоненте, скрыть детали. Это выборочное сокрытие предпочтительнее, нежели множественные модели, используемые некоторыми другими методами. Единственность модели обеспечивает бесшовность и обратимость, но в любой момент можно решить, какие аспекты соответствуют текущим потребностям, и скрыть остальное.
- Метод BON был создан в 1990-е годы. В нем изначально предполагается, что в распоряжении его пользователей будут вычислительные ресурсы, а не только бумага и карандаш или доска. Это позволяет использовать мощные инструментальные средства для отображения комплексной информации. Такие средства описаны в последней лекции этой книги. Для небольших задач вполне достаточно карандаша и бумаги.
- При всей амбициозности и способности охватить большие и сложные системы метод замечателен своей простотой. Он содержит небольшое количество основных концепций. Необходимо обратить внимание, что изложение формального подхода занимает всего около двух страниц.

Поддержка больших систем в BON основана в частности на понятии кластера - группы логически связанных классов. Кластеры могут содержать субкластеры, тем самым формируется вложенная структура и аналитики получают возможность работы на различных уровнях. Некоторые кластеры могут быть библиотеками - серьезное внимание уделяется повторному использованию.

Статическая часть модели сосредоточена на классах и кластерах; динамическая часть описывает объекты, взаимодействия объектов и возможные сценарии упорядочения сообщений.

BON поддерживает несколько вариантов формальных описаний: текстовую нотацию, табличную форму и графические диаграммы.

Текстовая нотация аналогична принятой в этой книге. Поскольку не подразумевается непосредственная компиляция, можно использовать ряд расширений в области утверждений. Например, `delta` а означает, что компонент может изменить атрибут `a`, `forall` и `exists` применяются для логических формул исчисления предикатов первого порядка, а `member_of` - для операций с множествами.

Таблица удобна для сжатого описания свойства класса. Общая форма табличного представления класса приведена ниже.

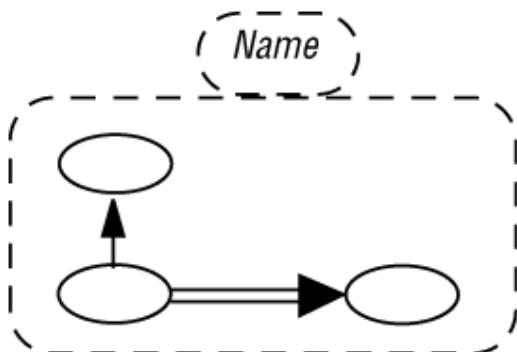
Таблица 9.1. Таблица описания класса в методе BON

CLASS	Class_name	Part:
Short description (Краткое описание)	Indexing information (Индексирующая информация)	
Inherits from (Наследует от)		

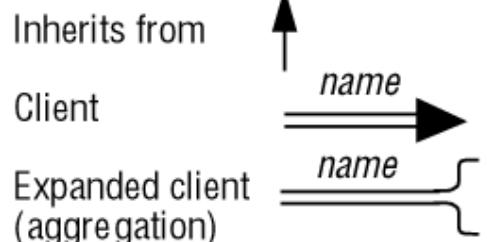
Графические обозначения чрезвычайно просты, их легко изучить и запомнить. Основные соглашения, статические и динамические, приведены на рис. 9.4.

STATIC DIAGRAMS

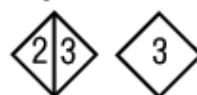
Cluster (with some classes)



Inter-class relations



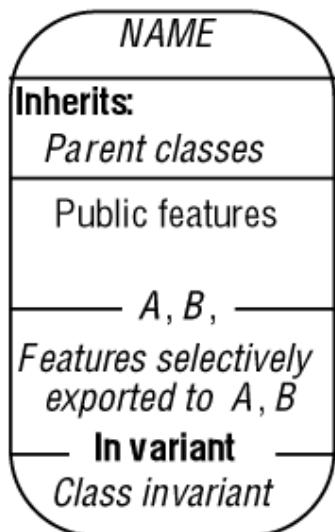
Multiplicity of relations



Class: generic, effective, deferred, reused, persistent, interfaced, root.



Class: detailed interface



Features

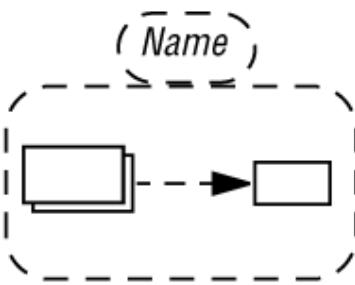
$name^*$, $name^+$, $name^{++}$ deferred, effective, redefined
 $\rightarrow name: TYPE$ input argument
 $?$ $!$ precondition, postcondition

Assertion operators

$\Delta name$ feature may change attribute $name$
 $@, \emptyset$ current object, void reference
 $\exists, \forall, |, \bullet$ symbols for predicate calculus operations
 \in, \notin membership operators

DYNAMIC DIAGRAMS

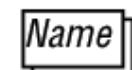
Object group (with some objects)



Object



Objects (one or more)



Inter-object relations

Message passing - - - 7 - - - \rightarrow
 (with message number from scenario)

Рис. 9.4. Основные графические обозначения BON ([Walden 1995], приведено с разрешения автора)

Процесс анализа и разработки состоит из семи задач. Порядок их решения соответствует идеальному процессу. В реальной практике его можно изменить и использовать итерации в соответствии с концепцией обратимости. Стандартными являются задачи:

- **B1 Определение границ системы:** что будет включено и не включено в систему, главные подсистемы, пользовательские метафоры, функциональность, библиотеки повторного использования.
- **B2 Составление списка классов-кандидатов,** в который вначале включают классы, имеющие отношение к данной области.
- **B3 Выбор классов и формирование кластеров:** объединение классов в логические группы, выделение абстрактных, перманентных классов и т. д.
- **B4 Определение классов:** развернутое описание классов в терминах запросов, команд и ограничений.
- **B5 Составление эскиза поведения системы:** определение схем создания объектов, событий и сценариев.
- **B6 Определение общедоступных компонентов:** завершение интерфейсов классов.
- **B7 Совершенствование системы.**

Метод предписывает в течение процесса разработки следовать **терминологии**, принятой в данной области. Опыт показывает, что это существенно при разработке любого большого проекта. Это помогает неспециалистам ориентироваться в профессиональном жаргоне, а также позволяет удостовериться, что все специалисты действительно используют одинаковую терминологию (удивительно видеть, как часто это не так!).

Для каждого шага метод определяет точный список того, что необходимо сделать. Он определяет также и отчетные документы. Эта точность определения организационных обязанностей делает BON не только методом анализа и проектирования, но и стратегическим инструментом для руководства проектом.

Библиографические замечания

Принципиально важным источником по Business Object Notation является [Walden 1995]. Основные концепции содержатся в [Nerson 1992]. Адрес Web-страницы www.tools.com/products/bon/.

Основные ссылки на другие методы. CoadYourdon: [Coad 1990], www.oi.com; OMT: [Rumbaugh 1991]; Shlaer-Mellor: [Shlaer 1992], www.projitech.com; Martin-Odell, [Martin 1992]; Booch: [Booch 1994]; OOSE: [Jacobson 1992]; OSA: [Embley 1992], www.osm7.cs.byu.edu/OSA.html; Syntropy: [Cook 1994], www.objectdesigners.co.uk/syntropy; Fusion, [Coleman 1994]; MOSES: [Henderson-Sellers 1994], www.csse.swin.edu.au/cotar/OPEN/OPEN.html; SOMA, [Graham 1995].

О проекте создания объединенного метода см. [Henderson-Sellers 1996]. [Computer 1996] содержит дискуссию, посвященную Unified Modeling Language (Booch-OMT-Jacobson) Rational Corporation.

Описания ОО-методов и другую полезную информацию можно найти на узле: www.arkhp1.kek.jp/~amako/OOInfo.html.

Основы объектно-ориентированного проектирования

10. Лекция: Процесс разработки ПО

Центральный вопрос объектной технологии - ее воздействие на весь процесс разработки ПО. Пришло время рассмотреть влияние ОО-принципов на общую организацию проектов и их деление на этапы. Это часть более общей проблемы перспектив объектной технологии с позиций менеджмента. Вопросы менеджмента подробно изучаются в книге Object Success. В данной лекции обсуждаются лишь наиболее существенные идеи: кластеры как основная организационная единица, принципы параллельной разработки на основе кластерной модели жизненного цикла ПО, этапы и задачи такой модели, роль обобщения для повторного использования, принципы бесшовности и обратимости.

Кластеры

В основе модульной структуры ОО-метода лежит класс. Классы обычно группируют в коллекции, называемые кластерами.

Кластер - это группа связанных классов или связанных кластеров (рекурсивное определение).

Данные случаи являются взаимоисключающими. Для упрощения считается, что кластер, содержащий подкластеры, не содержит непосредственно классы. Таким образом, рассматриваются **элементарные кластеры**, состоящие из классов и **суперкластеры**, состоящие из других кластеров.

Типичный набор элементарных кластеров может содержать кластер синтаксического анализа для контроля пользовательского ввода, кластер для поддержки графики, коммуникационный кластер. Типичный элементарный кластер содержит от пяти до сорока классов. На уровне примерно двадцати классов следует задуматься о его разбиении на подкластеры. Кластер естественным образом подходит для разработки одним человеком, который полностью в нем разбирается. Напротив, в случае крупного проекта никто не способен осмысливать систему в целом или даже главную подсистему.

Кластеры не являются супермодулями. Ранее мы привели аргументы против введения подобных единиц, например пакетов, вместо этого сохраняется единственный модульный механизм - класс.

В отличие от пакетов кластеры - это не языковая конструкция, а инструмент управления. Они появляются в управляющих файлах Lace, используемых для сборки системы из компонентов. Успешное объединение классов в кластеры определяется главным образом здравым смыслом и опытом руководителя проекта. Этот момент заслуживает особого внимания, поскольку его роль часто недооценивается. Идентификация классов, то есть выбор надлежащих абстракций данных, - действительно трудная задача, удачное решение которой создает условия для благоприятного развития работ, а неудачное может привести к кручу проекта. Группировка же классов в кластеры является организационной проблемой, она может быть решена различными способами в зависимости от доступных ресурсов и квалификации членов группы. Неоптимальное формирование кластеров может причинить неприятности и замедлить разработку, но не может стать причиной неудачи проекта.

Параллельная разработка

Одно из последствий деления на кластеры - уход от недостатков традиционной бескомпромиссной модели жизненного цикла ПО. Известная Модель Водопада, введенная в 1970 г., была реакцией против устаревшего подхода "раньше запрограммируйте, а потом опишите". Заслуга этого подхода выражается в распределении обязанностей, определении основных задач разработки ПО и в подчеркивании важности открытой спецификации.

Модель Водопада помимо других недостатков страдает от жесткости подхода: буквальное следование ей подразумевает, что нельзя перейти к проектированию, пока полностью не закончена спецификация, до завершения проектирования - к реализации. Это может вызвать катастрофу: в механизм попадает единственная песчинка, и останавливается весь проект.

Предлагались различные усовершенствования этой модели, использующие итеративный подход, примером может служить Спиральная модель. Все они сохраняют водопад с одним потоком, что едва ли отражает современное положение, когда разработку ПО ведут большие удаленные друг от друга "виртуальные" команды, поддерживающие связь через Internet.

Успешное внедрение ОО-метода нуждается в схеме **параллельной разработки**, обеспечивающей децентрализацию и гибкость без утраты преимуществ упорядоченности водопада. В то же время ОО-разработка не подразумевает отказа от последовательного компонента, и его также необходимо сохранить. Во всяком случае мощь метода требует от нас еще большей организованности.

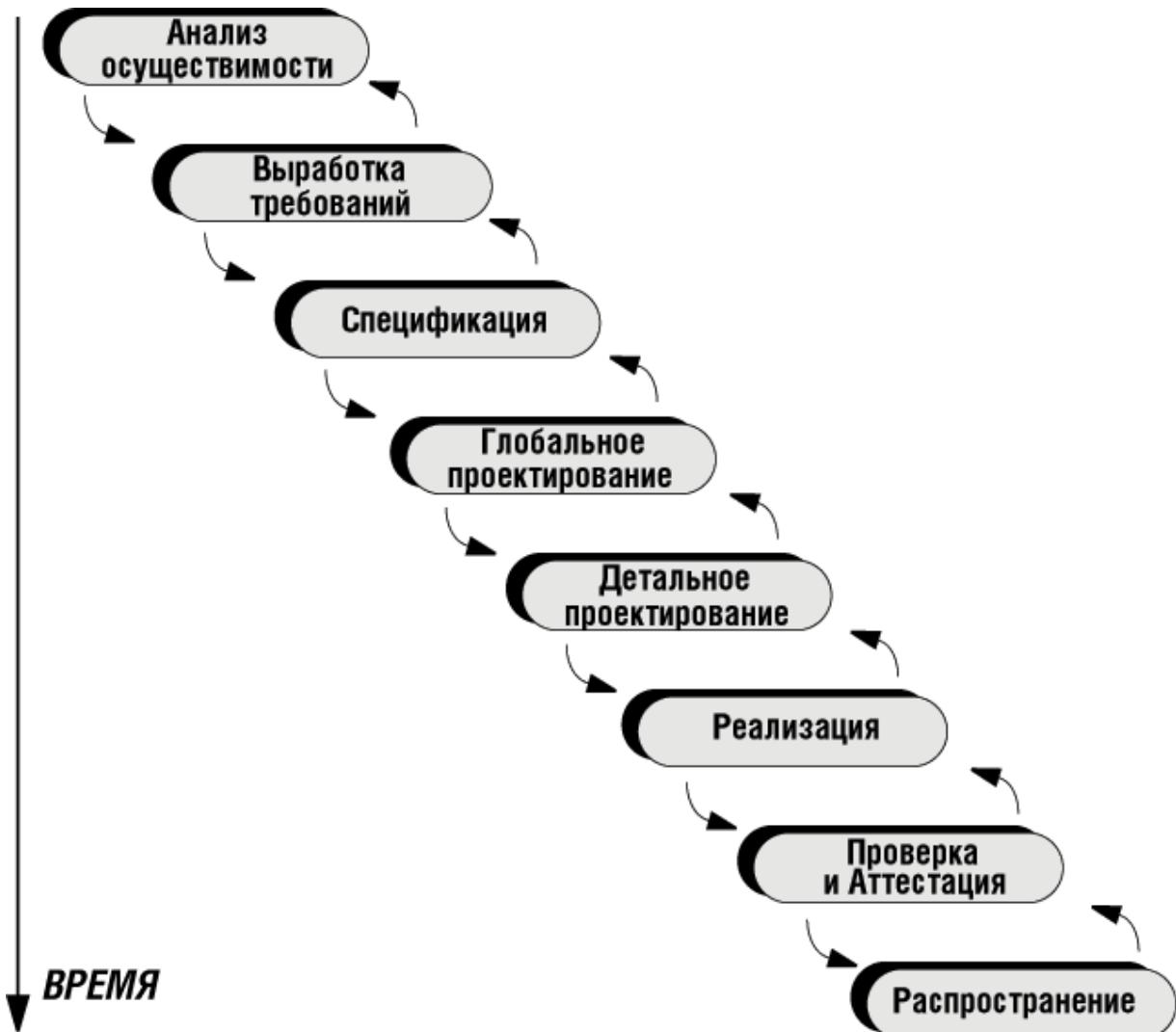


Рис. 10.1. Модель Водопада

Деление на кластеры позволяет обеспечить равновесие между последовательной и параллельной разработкой. Мы получаем последовательный процесс с возможностью обратных корректировок (концепция обратимости обсуждается более подробно в конце этой лекции), но для отдельных кластеров, а не системы в целом.

Вот как выглядит жизненный мини-цикл разработки кластера:



Рис. 10.2. Жизненный цикл отдельного кластера

Форма представления отражает бесшовный характер разработки. Вместо отдельных шагов в модели водопада данный процесс можно уподобить росту сталактита: каждый последующий шаг произрастает из предыдущего и добавляет собственный вклад.

Этапы и задачи

Жизненный цикл каждого кластера включает следующие этапы:

- спецификация: идентификация классов (абстракций данных) кластера и их главных особенностей и ограничений;
- проектирование: определение архитектуры классов и их отношений;
- реализация: завершение классов во всех деталях;
- верификация и Аттестация (Verification & Validation): контроль классов кластера (статическая проверка, тестирование и другие методы);
- обобщение: подготовка к повторному использованию (см. ниже).

Иногда трудно четко разграничить этапы проектирования и реализации. Поэтому возможны варианты модели, объединяющие эти этапы в один, - "проектирования-реализации".

Перед началом работы с кластерами необходимо пройти через две фазы общего характера. Во-первых, данный подход, как и другие, требует анализа осуществимости (feasibility study), на основе которого принимается решение о начале работы над проектом. Во-вторых, следует разбить проект на кластеры. Как уже отмечалось, ответственность за этот шаг возлагается на руководителя проекта, который безусловно может заручиться поддержкой других опытных членов группы.

Кластерная модель жизненного цикла ПО

Общая схема разработки, известная как Кластерная Модель, приведена на [рис. 10.3](#). Вертикальная ось представляет последовательный компонент процесса: чем ниже размещена та или иная работа, тем позже она будет выполнена. Горизонтальное направление отражает параллельную разработку: задачи на одном уровне могут выполняться в одно время.

Различные кластеры и отдельные этапы в пределах каждого кластера могут разрабатываться в индивидуальном темпе в зависимости от трудности задачи. Руководитель проекта отвечает за планирование работы над новым кластером или новой задачей.

Результатом является разумное сочетание порядка и гибкости. Определение задач кластеров обеспечивает порядок, готовую систему управления и контрольные точки, что позволяет отслеживать ход работ (один из самых трудных аспектов руководства проектом). Гибкость достигается за счет возможности нивелировать неожиданные задержки или использовать в своих интересах неожиданно быстрое продвижение путем переноса начала работ на более ранний или поздний срок. Руководитель проекта определяет уровень параллельной разработки. Для небольших групп или на начальных стадиях крупного проекта разрабатывается небольшое количество параллельных кластеров или только один. В случае больших групп после решения принципиальных вопросов можно сразу запустить работу над несколькими кластерами.

АНАЛИЗ ОСУЩЕСТВИМОСТИ

РАЗБИЕНИЕ НА КЛАСТЕРЫ

Кластер 1



Кластер 2



Кластер *n*



ВРЕМЯ

Рис. 10.3. Кластерная модель жизненного цикла ПО

Лучше, чем традиционные подходы, кластерная модель повышает эффективность управления проектом путем гибкого распределения ресурсов.

Во избежание рассогласований необходимо регулярно отслеживать текущие состояния кластеров. Оптимальным является контроль руководителем проекта хода работ через определенное время, например, один раз в неделю. Тем самым гарантируется наличие на каждой стадии текущей демонстрационной версии, не обязательно охватывающей все аспекты системы, но готовой для показа клиентам, менеджерам и другим заинтересованным лицам. Кроме того, это позволяет своевременно информировать участников проекта и устранять любую несогласованность между кластерами.

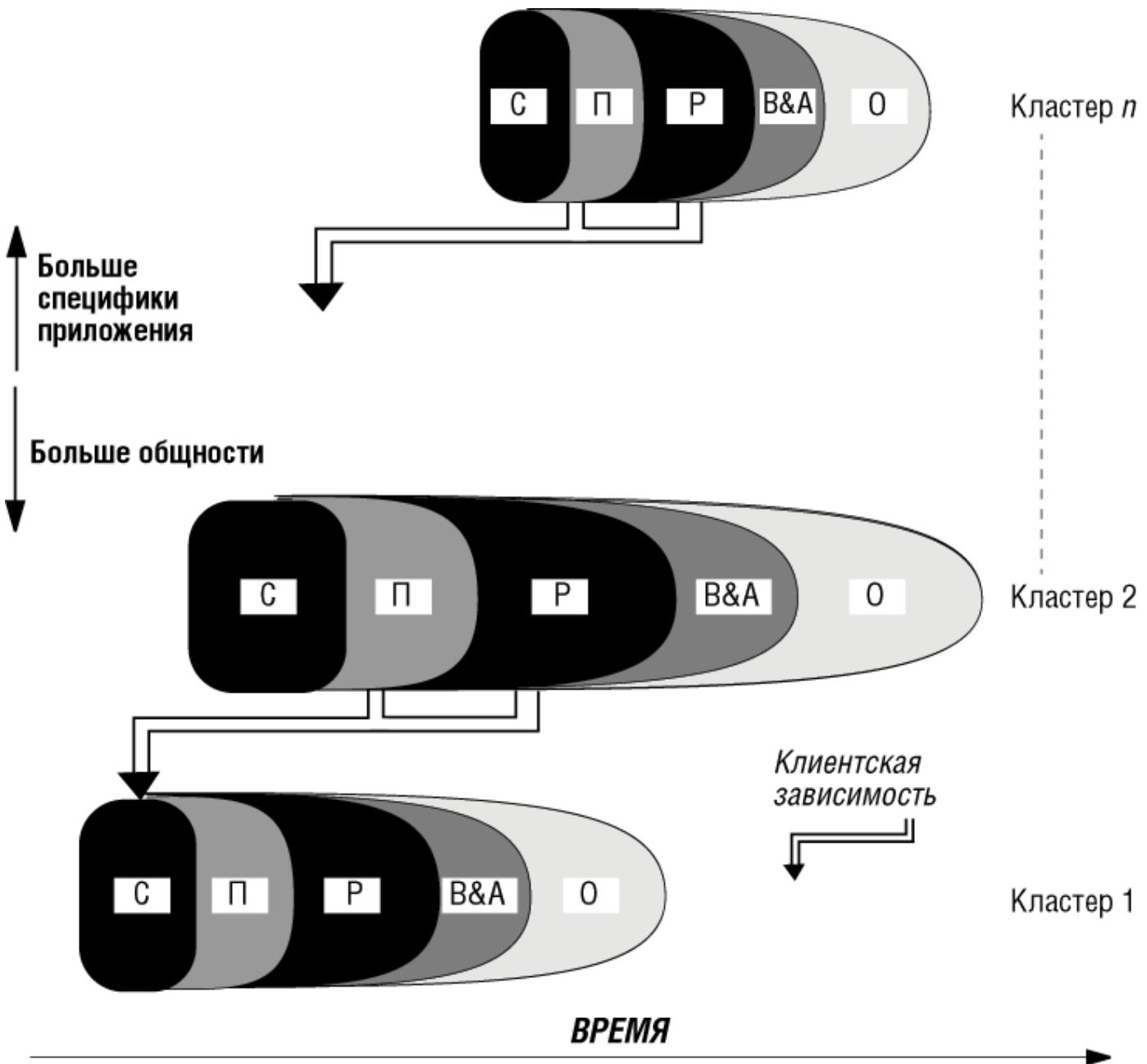


Рис. 10.4. Кластеры проекта как множество уровней абстракции

Возможность параллельной разработки в рамках кластерной модели обеспечивается за счет механизмов скрытия информации ОО-метода. Кластеры могут зависеть друг от друга, например кластер графического интерфейса может нуждаться в классах коммуникационного кластера для реализации удаленного терминала. Благодаря абстрактным данным можно работать над кластером, даже если кластеры, от которых он зависит, еще не завершены. Эта возможность реализуется при наличии законченной спецификации необходимых классов на основе их официального интерфейса, заданного в краткой форме или в виде отложенной версии. Этот аспект модели проще понять, если развернуть [рис. 10.3](#) так, как это показано на [рис. 10.4](#), разместив программные уровни, соответствующие общим кластерам внизу, а отражающие специфику приложения наверху. Проектирование и реализация каждого кластера зависят только от спецификаций расположенных ниже кластеров и не связаны с их реализацией. Кластер может полагаться на любой кластер, расположенный ниже (на [рис. 10.4](#) показаны только зависимости между соседями).

Обобщение

Последний этап жизненного цикла кластера, обобщение, не имеет аналогов в традиционных подходах. Его цель - шлифовка классов для их подготовки к повторному использованию.

Включение шага обобщения немедленно вызывает критику: вместо апостериорных добавлений разве не следовало заботиться, чтобы повторное использование являлось составной частью всего процесса разработки ПО? Можно ли сделать ПО пригодным для повторного использования после его завершения? Такая критика неуместна. Априори следует исходить из того, что возможность повторного использования должна быть заложена с самого начала разработки ПО. Апостериори следует считать, что сразу же по завершению разработки не достигло уровня, пригодного для повторного использования. Эти две точки зрения являются не противоречивыми, а взаимодополняющими. Успех политики повторного использования требует наличия определенной **культуры повторного использования** у каждого участника, выделения достаточных ресурсов для расширения соответствующих возможностей исходных версий классов.

Невзирая на лучшие намерения программные элементы, созданные для конкретной области, не могут быть полностью готовы для повторного использования. Существуют ограничения, влияющие на проект, - давление клиентов, желающих как можно скорее получить следующую версию, конкурентов, выпускающих свои программы, акционеров, жаждущих увидеть результаты. Мы живем в условиях постоянной спешки. Но существует и внутренняя причина недоверия к возможности повторного использования. Нельзя быть уверенными в том, что продукт освобожден от всех явных и неявных связей с конкретной областью применения, корпоративной принадлежностью разработчиков, аппаратно-программной средой, пока **кто-то** не использует его повторно.

Присутствие шага обобщения не означает, что до последнего момента можно не думать о возможности повторного использования. Эта возможность не появится сама собой и наличия политики повторного использования недостаточно. Даже если это образ мыслей каждого, то все равно придется уделить некоторое время классам вашего проекта для того, чтобы их можно было назвать программными компонентами.

Включение обобщающего этапа в официальную модель процесса - вопрос политики. В наши дни очень немногие представители руководства компаний станут явно выступать против повторного использования. **Конечно, мой друг, мы хотим, чтобы наше ПО было готово к повторному использованию!** Как отличить искренние высказывания от запудривания мозгов? Очень легко. Намерения искренни, если руководство готово в каждом проекте резервировать для обобщения дополнительные средства и время. Такое решение требует смелости, потому что не обещает непосредственных выгод, а другие срочные проекты могут немного пострадать. Но только так можно гарантировать, что в результате появятся компоненты повторного использования. Если руководство не готово выделить даже скромные ресурсы (несколько процентов сверх обычной сметы), то Вы можете слушать вежливо напыщенные речи о повторном использовании, но, по правде говоря, компания не готова к повторному использованию и не получит эту возможность.

Даже если дополнительные ресурсы предусмотрены, то только этого недостаточно. Залогом успеха служит комбинация усилий *a priori* и *a posteriori*:

Культура повторного использования

Разрабатывайте все программное обеспечение, подразумевая его повторное использование.

Не верьте в готовность ПО к повторному использованию, пока не увидите, что оно действительно использовано.

Первая часть подразумевает применение концепций повторного использования в течение всей разработки. Вторая учитывает, что для достижения желаемого результата обязательно нужно ввести этап обобщения для удаления всех следов контекстно-зависимых элементов.

Этап обобщения может содержать следующие действия:

- **Абстрагирование:** введение абстрактных классов там, где это необходимо.
- **Факторизацию (Factoring):** распознавание первоначально несвязанных классов, которые являются фактически вариантами того же самого понятия, так что для них можно ввести общего родителя.
- Добавление утверждений, особенно постусловий и инвариантов, отражающих углубленное понимание семантики класса и его особенностей. Вероятно придется также добавить предусловие, но это скорее исправление ошибки, означающее незащищенность подпрограммы на должном уровне.
- Добавление предложений **rescue** для обработки исключений, возможность которых первоначально игнорировалась.
- Добавление документации.

Два первых действия относятся к методологии наследования и отражают нестандартный подход к построению иерархии наследования: не от общего к частному, а как раз наоборот.

Обобщение совершенствует классы, считающиеся вполне подходящими для внутренних целей специфической системы, но требующие шлифовки, когда они становятся частью библиотеки широкого круга применений. Такие простительные в первом случае огрехи, как неполная спецификация или наличие недокументированных предположений, выходят на передний план. В этом причина повышенной трудности разработок, предусматривающих повторное использование, по отношению к обычной практике. Все становится важным, когда ПО доступно всем приложениям различного типа и всем платформам.

Бесшовность и обратимость

Сталактикоподобный характер жизненного цикла кластера отражает одно из самых радикальных различий между ОО-методом и более ранними подходами. Правильно понимаемая объектная технология устраняет барьеры между последовательными шагами жизненного цикла и определяет единую структуру анализа, проектирования, реализации и сопровождения. Это так называемая бесшовная разработка, одним из требований которой является обратимость процесса разработки ПО.

Бесшовная разработка

Отдельные проблемы, конечно, останутся. Существует различие в определении общих свойств системы на начальном и заключительном цикле отладки. Но идея бесшовности сглаживает различия, подчеркивая фундаментальную целостность процесса. На различных этапах разработки возникают одни и те же проблемы, необходимы одинаковые механизмы структурирования, применяется та же логика рассуждений и, как показано в этой книге, можно использовать единую нотацию.

Выгоды от бесшовного подхода многочисленны:

- Устраняются дорогостоящие и подверженные ошибкам резкие переходы между отдельными этапами, использующими различную нотацию, системы взглядов и персонал (аналитики, проектировщики, программисты...). Такие переходы часто называют рассогласованиями импеданса по аналогии с электрическими схемами, собранными из несовместимых элементов. Несоответствия между анализом и проектированием, проектированием и реализацией, реализацией и развитием являются причиной многих неприятностей в традиционной схеме разработки ПО.
- С начала и до конца основной разработки являются классы, что гарантирует полное соответствие между описанием проблемы и ее решением. Прямое отображение упрощает диалог с заказчиками и пользователями и содействует развитию, поскольку все участники пользуются терминами одних и тех же основных концепций. Это та часть поддержки расширяемости, которую обеспечивает ОО-метод.
- Использование единой структуры облегчает корректировки, выполняемые в обратном направлении, неизбежные для поступательного в целом процесса разработки ПО.

Обратимость: мудрость иногда расцветает слишком поздно

Последнее преимущество определяет один из принципиальных вкладов объектной технологии в жизненный цикл ПО - обратимость.

Обратимость - официальное признание влияния более поздних стадий процесса разработки ПО на решения, выработанные на начальных стадиях. Безусловно, эта особенность является неизбежной и универсальной, но это одна из наиболее тщательно охраняемых тайн.

Хотелось бы, конечно, полностью определить проблемы, прежде чем приступить к их решению: анализ завершать до проектирования, проектирование - до начала реализации, реализацию - до поставки. Однако что делать, если в процессе реализации разработчик внезапно понимает, что система может что-то делать лучше или вообще должна иначе работать? Отругать его за то, что занимается не своим делом? А если он действительно прав?

Это явление отражает поговорка - **esprit de l'escalier**, ("лестничное остроумие", остроумие задним числом). Вообразите приятный обед в фешенебельной парижской квартире на третьем этаже. Со всех сторон звучат остроты по поводу телятины Marengo, а Вы будто онемели. Суаре заканчивается, Вы попрощались с хозяевами, спускаетесь по лестнице и вдруг - вот она, разящая остроумная реплика, которая сделала бы Вас героям вечера! Но слишком поздно.

Имеют ли место приступы **esprit de l'escalier** в программном обеспечении? Они существуют с тех пор, как стали замораживать спецификацию прежде, чем приступить к решению. Плохие менеджеры подавляют программистов - пишите код и помалкивайте. Хорошие менеджеры стараются использовать в своих интересах запоздальные идеи спецификации, не обращая внимания на то, кто отвечает за данную проблему, и невзирая на требования стиля водопада.

С развитием ОО-разработки стало ясно, что явление **esprit de l'escalier** - не только результат лени при анализе, но и отражение самой природы процесса разработки ПО. Мудрость иногда приходит слишком поздно. Нигде более, чем в объектной технологии, не проявляется так явно связь между проблемой и решением. Не только потому, что мы иногда понимаем аспекты проблемы только в процессе решения, но и по более глубокой причине. Решение воздействует на проблему и может предложить лучший функциональный подход.

Вспомним пример из [лекции 3](#) с командами отката и повтора и списком истории: фактически в ходе реализации был предложен новый, более удобный интерфейс, облегчающий работу конечных пользователей.

Введение обратимости требует дополнить диаграмму жизненного цикла кластера указаниями на постоянную возможность обратных пересмотров и исправлений:

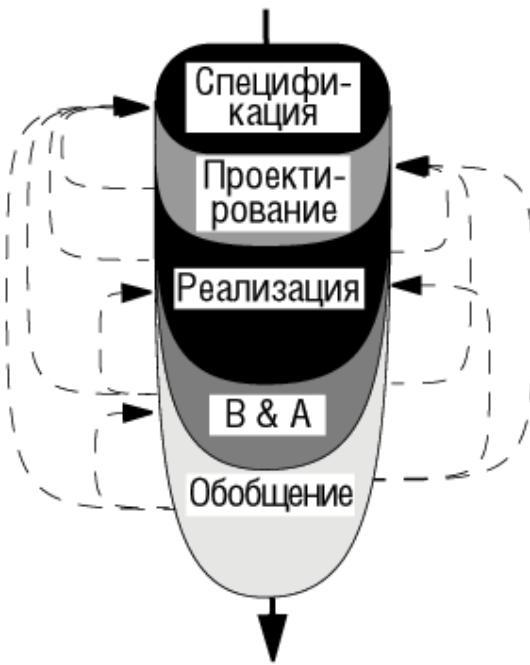


Рис. 10.5. Жизненный цикл отдельного кластера, обратимость

У нас все - лицо

Акцент на бесшовность и обратимость потенциально подрывает устоявшуюся систему взглядов на организацию работы над проектами и сам характер профессии. Стираются барьеры между узкими специальностями - аналитиками, имеющими дело с концепциями, проектировщиками, которых волнует лишь структура, и программистами, пишущими код. Формируется сообщество универсалов, разработчиков в широком смысле этого слова, способных вести свою часть проекта от начала до конца.

Данный подход отличается от подхода, доминирующего в литературе, рассматривающего анализ и реализацию (посредине - проектирование) как принципиально различные действия, использующие различные методы и нотацию, преследующие различные цели. При этом неявно подразумевается, что анализ и проектирование относятся к высоким материям, а реализация - это лишь неизбежная рутина. Такая точка зрения исторически оправдана. Начиная с младенческого периода 1970-х годов, предпринимались попытки внести хоть какой-то порядок в бессистемный процесс разработки ПО. Специалистов призывали подумать, прежде чем стрелять. Поэтому на ранних стадиях разработки ПО особое внимание уделялось выяснению того, что именно планируется реализовать. Сейчас, как и прежде, это полностью справедливо. Однако эти благие намерения завели слишком далеко. Строгое следование последовательной модели приводило к провалам между отдельными этапами в ущерб требованиям бесшовности и обратимости.

Объектная технология позволяет устраниć ненужные различия между анализом, проектированием и реализацией (необходимые проявляются достаточно ясно) и восстановить опороченную репутацию реализации. Пионерам разработки ПО при программировании по ходу дела приходилось решать много машинно-зависимых проблем, изъясняться на низкоуровневом неэлегантном языке, понимаемом компьютером. Эта приземленность мешала изучению абстрактных понятий прикладной области. Но теперь можно сочетать высокий уровень абстракции с пониманием проблем реализации.

Секрет в том, чтобы поднять концепции программирования и соответствующую нотацию на достаточно высокий уровень, позволяющий использовать их в качестве средств моделирования. Именно этого достигает объектная технология.

Следующая история, заимствованная из книги Романа Якобсона "Essays on General Linguistics", поможет поставить точку:

В далекой стране миссионер ругалaborигенов: "Вы не должны ходить обнаженными, показывая ваше тело!". Однажды маленькая девочка возразила, указывая на него: "Но Вы, Отец, также показываете часть вашего тела!". "Ну конечно", - величественно произнес миссионер. - "Это мое лицо". Девочка ответила: "То, что видите Вы, Отец, на самом деле то же самое. Только у нас все - лицо".

Так же обстоит дело с объектной технологией. У нас все - лицо!

Ключевые концепции

- Объектная технология нуждается в новой модели процесса, обеспечивающей бесшовную, обратимую разработку.
- Последовательным элементом жизненного цикла является кластер - набор логически связанных классов. Кластеры допускают произвольную вложенность.
- Модель жизненного цикла основана на параллельной разработке - параллельной работе над несколькими кластерами, использующих спецификации ранее завершенных кластеров.
- Объектная технология восстанавливает в правах реализацию.

Библиографические замечания

Монография [М 1995] посвящена подробному обсуждению тематики данной лекции. В ней содержится подробное изложение кластерной модели, обсуждается влияние ОО-метода на организацию работы групп, роль менеджера, экономику разработки ПО.

[Baudoin 1996] посвящена проблемам жизненного цикла, связанным с объектной технологией, а также охватывает много других важных тем, включая организацию проекта, роль стандартов и некоторые социологические аспекты.

Кластерная модель была впервые представлена в [Gindre 1989]. Другая ОО-модель жизненного цикла, "модель фонтана", была предложена в [Henderson-Sellers 1990] и далее разработана в [Henderson-Sellers 1991], [Henderson-Sellers 1994]. Подчеркивая необходимость итераций, она скорее дополняет кластерную модель, нежели ей противоречит.

Ряд публикаций по ОО-анализу, в частности [Rumbaugh 1991] (оригинальная работа по методу ОМТ) и [Henderson-Sellers 1991], особое внимание уделяют бесшовности разработки. Детальное рассмотрение проблем бесшовности и обратимости см. в [Walden 1995].

Wisdom sometimes blooms late in the season
Or half-way down the stairs.
Is it, my Lords, a crime of high treason
To trust the implementers?¹⁾

¹⁾ У мудрости цветы цветут зимой, И встречи с ней на лестнице не редки. Прости, господь, но разве грех большой, Что программисту доверяются проекты?

Основы объектно-ориентированного проектирования

11. Лекция: Обучение методу

Заканчивая наше изучение методологических проблем, обратимся к одному из принципиальных вопросов, стоящих перед компаниями и университетами: как лучше учить тех, кто должен применять объектные технологии. Эта лекция представляет принципы обучения и отмечает типичные ошибки. В первой части проблема рассматривается с точки зрения лица, организующего профессиональную подготовку в компании, во второй - с позиций профессора университета. Несмотря на то, что речь пойдет о педагогических проблемах профессиональной подготовки, обсуждение касается и тех, кто не занимает указанных позиций и является скорее обучаемым, чем обучающим.

Профессиональная подготовка (тренинг) в индустрии

Давайте начнем с нескольких общих наблюдений по поводу обучения объектным технологиям профессионалов, уже освоивших другие подходы. Обучение может вестись либо на семинарах, либо согласно внутреннему плану переподготовки, разработанному в компании.

Парадоксально, но задача тренера теперь может быть труднее, чем в середине восьмидесятых, когда к этому методу было привлечено широкое внимание. Тогда он был нов для большинства людей и имел ауру еретика, что всегда привлекает слушателей. Сегодня никто не будет шокирован, если кто-то заявит о своем пристрастии к ОО-методу. От постоянного упоминания в компьютерной прессе - ОО это и ОО то - возник своего рода шумовой эффект, называемый **mOOzak**. Слова объект, класс, полиморфизм от частого употребления становятся затертыми, они кажутся знакомыми, но широко ли понимаются стоящие за ними концепции? Зачастую нет! Это накладывает на тренера новую ношу - объяснить обучаемым, что они знают не все. Невозможно научить чему-либо человека, если он думает, что он уже это знает.

Единственная стратегия, гарантирующая преодоление этой проблемы, состоит в следующем:

Начальный тренинг: стратегия "пройди его дважды"

- **T1** Пройди начальный курс.
- **T2** Попытайся выполнить ОО-разработку.
- **T3** Пройди начальный курс.

Выполнение этапа Т3 кажется странным: применив ОО-метод в реальной разработке, снова слушать тот же курс. Компаниям, занимающимся ОО-обучением, не всегда удается применять эту стратегию, так как она выглядит подозрительно - вам дважды предлагаются продать одну и ту же вещь. Но здесь обмана нет.

Понимание концепций по-настоящему приходит во время второй итерации. Хотя первая необходима для обеспечения основы, она не может быть полностью эффективной частично из-за эффекта **mOOzak**, частично из-за сложностей умозрительного восприятия концепций. Только тогда, когда слушатели сталкиваются день изо дня с ежедневными выборами ОО-конструирования - **Нужен ли новый класс для этого понятия?**

Подходит ли здесь наследование? Следует ли ввести из-за этих двух компонентов новый узел в структуре наследования? Соответствует ли образец из курса моей ситуации? - только после этого они получают необходимую подготовку для правильного восприятия курса. Вторая сессия не будет, конечно же, идентична первой (по крайней мере, вопросы аудитории станут интереснее), она скорее находится на границе между тренингом и консультацией, но она действительно должна представлять вторую итерацию того же основного материала, ни в коей мере не являясь углубленным курсом, следующим за начальным.

На практике только наиболее просвещенные компании готовы принять эту стратегию, другие ссылаются на нехватку ресурсов. По моему опыту результат настолько хорош, что заслуживает чрезвычайных усилий. Стратегия является наилучшей из всех мне известных для обучения разработчиков для настоящего понимания ОО-технологии, чтобы они могли эффективно применять ее в интересах компании.

Следующий принцип говорит о тематике обучения:

Принцип Темы Тренинга

В начальном курсе в особенности сфокусируйтесь на реализации и проектировании.

Некоторые полагают, что курс должен начинаться с ОО-анализа. Это грубая ошибка. Понимание ОО-анализа не может прийти к новичку в ОО-технологии (разве только в смысле шумового эффекта **mOOzak**). Для овладения анализом необходимо изучить фундаментальные понятия: класс, контракты, скрытие информации, наследование, полиморфизм, динамическое связывание и тому подобное. Вначале все нужно делать на уровне реализации, где эти понятия непосредственно применяются. Следует практически построить несколько ОО-систем, вначале небольших, а затем наращивать их размер. Все проекты следуют довести до завершения. Только после такой схватки врукопашную можно переходить к задаче ОО-анализа и пониманию

ее роли в бесшовном процессе ОО-конструирования ПО.

Еще два принципа. Во-первых, не ограничивайтесь вводными курсами:

Принцип углубленной учебной программы

По меньшей мере 50% бюджета тренинга должно быть резервировано для невводных курсов.

Наконец, следует учить не только разработчиков:

Принцип Тренинга Менеджеров

Программа тренинга должна включать курсы для менеджеров, также как и для разработчиков.

Для компании, адаптирующей ОО-технологию, нереально надеяться на успех, обучая только разработчиков. Менеджеры, независимо от уровня их технической подготовки, должны быть знакомы с основными ОО-идеями, уметь оценивать их влияние на распределение задач, организацию команды, жизненный цикл проекта, экономику разработки ПО. Жизненный цикл обсуждается в следующей лекции (основательно в книгах, ориентированных на менеджеров, таких как [Goldberg 1995], [Baudoin 1996] и [M 1995]).

Вот пример того, что должно включаться в программу обучения менеджеров во избежание потенциальных трудностей непонимания. В индустрии все еще измеряют производительность работы программиста, основываясь на числе созданных строк в единицу времени. В осознанном процессе создания ПО много времени уделяется **уже хорошо работающим** программным элементам для увеличения их потенциала повторного использования в будущих проектах. Обычно улучшение приводит к обобщениям - важному шагу жизненного цикла модели. В результате удаляется код, например, из-за введения общего родителя двух или более существующих классов и передачи ему общих свойств. Коэффициент производительности при этом будет только падать - числитель уменьшается, знаменатель растет! Менеджеры должны понимать, что старые меры не годятся для новых подходов, что затраченные чрезвычайные усилия, фактически улучшившие ПО, дают ценный вклад в капитал компании. Без такой подготовки может возникнуть серьезное непонимание, сводя на нет эффект технически спланированной стратегии разработки.

Вводные курсы

Обратимся теперь к обучению объектных технологий в академическом окружении (хотя многие замечания относятся и промышленному тренингу).

Когда программистское сообщество осознало значимость ОО-подхода, непосредственно возник вопрос, где, как и когда включать ОО-понятия, языки и инструментарий в учебные планы университетов, колледжей и средней школы.

Филогенез и онтогенез

Когда следует начинать?

Чем раньше, тем лучше. ОО-метод обеспечивает прекрасную интеллектуальную дисциплину. Если вы согласны с ее целями и техникой, то нет причин в задержке обучения им ваших студентов. Фактически это должен быть первый подход, с которого следует начинать обучение. Начинающие положительно воспринимают ОО-подход не потому, что такова тенденция, а по причине его ясности и эффективности.

Эта стратегия предпочтительнее, чем вначале учить старым подходам, а затем переучивать, прививая ОО-мышление. Нет смысла в обходных путях, когда есть прямая дорога ОО-разработки.

Преподаватели неосознанно склонны применять идею, некогда популярную в биологии: онтогенез (развитие индивидуума) повторяет филогенез (развитие вида). Человеческий эмбрион на разных этапах своего развития напоминает лягушку, свинью и т. д. Применительно к нашему предмету рассмотрения это означает, что учитель, возможно, начинавший изучать Algol, затем перешедший к структурному проектированию, затем познавший объекты, захочет, чтобы и его студенты прошли тот же путь. Что было бы с начальным образованием, если бы детей учили считать, вначале используя римские цифры, и лишь потом вводили арабские? Если вы думаете, что знаете правильный подход, учите ему сразу.

Вымощенная дорога к другим подходам

Одна из причин рекомендации (без фанатизма и узости мышления) использования ОО-технологии в начале обучения состоит в общности метода, он готовит студентов к введению других парадигм, таких как логическое и функциональное программирование, которые должны быть частью программистской культуры. Если ваши учебные планы требуют изучения традиционных языков программирования, то и их

предпочтительнее вводить после обучения ОО-методу, поскольку это позволит использовать эти языки безопасным и более разумным способом.

ОО-обучение является также хорошей подготовкой для тематики, имеющей корни в математике и формальной логике и играющей все большую роль в современных учебных планах: формальные подходы к спецификации, конструированию и верификации программ. Использование утверждений и общего подхода Проектирования по Контракту, по моему опыту, является эффективным способом, показывающим необходимость систематического, независимого от реализации и, по меньшей мере, частично формализованного описания программных элементов. Преждевременное знакомство с механизмом языков формальных спецификаций, таких как Z или VDM, может лишь подавить студентов и вызвать реакцию отторжения. Даже если это и не произойдет, студенты вряд ли воспримут достоинства формализмов, не имея важного опыта разработки ПО. ОО-конструирование в соответствии с принципами Проектирования по Контракту дает студентам возможность начать создавать реальные программы и в то же время приводит к плавному прогрессирующему введению формальных методов.

Выбор языка

Использование ОО-метода во вводном курсе имеет смысл только тогда, когда оно основано на окружении и языке, полностью поддерживающем эту парадигму и не отягощенном призраками прошлого. В частности, гибридные подходы, основанные на расширениях старых языков, не подходят для начинающих студентов, поскольку смешивают ОО-концепции с пережитками старых методов, принуждая преподавателя тратить больше времени на извинения, чем на сами концепции.

В языках, основанных на С, например, придется объяснять, почему массив и указатель следует рассматривать как одно и то же понятие - свойство, имеющее корни в технике оптимизации старой архитектуры компьютеров; на эти объяснения потребуется время и энергия в ущерб обучению понятиям проектирования программ. Более того, это ведет к тому, что студенты приучаются мыслить в терминах низкоуровневых механизмов - адресов, указателей, памяти, сигналов. Они будут тратить неоправданно много времени на борьбу с различными жучками в своих программах.

Задачи вводного курса разнообразны. Следует снабдить студентов ясным, логически связанным множеством практических принципов. Нотация должна непосредственно поддерживать эти принципы, устанавливая взаимно однозначное соответствие между методом и языком. Время, затрачиваемое на объяснение языка самого по себе, зря потрачено. Следует объяснять концепции и использовать язык как естественный способ их применения.

Главное качество языка, используемого во вводном курсе, это его структурная простота и поддержка ОО-идей: модульность, основанная на классах, проектирование по контракту, статическая типизация и наследование. Но не следует недооценивать роли синтаксической ясности. Например, тексты на C++ и Java наполнены строками, такими как:

```
public static void main(String[] args {  
    if (this->fd == -1 && !open_fd(this))  
        if ((xfrm = (char*)malloc(xfrm_len + 1)) == NULL) {
```

Как видно, синтаксис этих языков, основанный на многих специальных операциях, затуманен и зашифрован.

Подобные штучки, оправданные историческими причинами, не для новичков. Обучение программированию достаточно трудно и без того, чтобы еще вводить недружелюбную нотацию.

Дэвид Кларк из университета Канберры на основе опыта обучения опубликовал некоторые из своих заключений в Usenet:

В последнем семестре я обучал студентов программированию, используя Java (вторые полгода из годичного курса для первокурсников). Из моего опыта следует, что студенты не находят язык Java простым в обучении. Неоднократно язык мешал мне учить тому, чему бы я хотел. Вот несколько примеров:

- Первое, с чем они сталкиваются, это главная программа с заголовком: `public static void main (String [] args)`, выбрасывающая исключения вида `IOException`. Здесь в одной строке встречается 6 различных понятий, которые студенты не готовы воспринимать.
- Достаточно свободно можно осуществить вывод, но чтобы что-то ввести, потребуется попрыгать (`import, declare, initialize`). Единственный способ ввода числа с клавиатуры - это прочесть строку и разобрать ее. И снова с этим приходится сталкиваться уже на первой лекции.
- Java рассматривает примитивные типы данных (`int, char, boolean, float, long, ...`) не так, как другие объекты. Здесь есть их объектные эквиваленты (`Integer, Boolean, Character` и т. д.). Но нет связи между `int` и `Integer`.
- Класс `String` представляет специальный случай (для эффективности). Он используется только для строк, не меняющих значения. Есть также класс `StringBuffer` для строк, меняющих свое значение.

Все прекрасно, но нет взаимосвязи между этими классами. Есть только несколько общих компонентов.

- *Отсутствие универсальности означает необходимость преобразования типов, например, при использовании коллекций элементов, таких как Stack или Hashtable. Все это создает помехи для начинающих студентов и уводит их в сторону от главных целей обучения.*

Проф. Кларк далее сравнивает этот опыт с его практикой обучения с использованием нотации этой книги, о которой он пишет: **"Я фактически не учил языку, помимо некоторых примеров кода".**

Начальная нотация, используемая при обучении, крайне важна для формирования их будущего видения, она должна быть простой и ясной, позволяя глубоко понимать базисные понятия. Даже Pascal, традиционный выбор вводного курса факультетов, специализирующихся в подготовке по информатике, предпочтительнее во многих отношениях, чем гибридные языки, так как обеспечивает компактный, согласованный базис, от которого позже студенты могут перейти к другому компактному, согласованному подходу. Конечно, было бы лучше, если бы базис был компактным, согласованным и **объектно-ориентированным**.

Некоторые гибридные языки имеют индустриальную значимость, но им следует учить позднее, когда студенты овладеют базисными концепциями. Это не новая идея: когда в семидесятых годах факультеты информатики (computing science departments) приняли Pascal, они также включали специальные курсы по изучению Fortran, Cobol или PL/I, что требовалось тогда индустрии. Аналогично современный учебный план может включать специальные курсы по C++ или Java для удовлетворения требований индустрии, давая возможность студентам включать требуемые шумовые слова в свои резюме. В любом случае студенты лучше поймут C++ и Java после изучения объектной технологии, используя чистый ОО-язык. Начальный курс, формирующий сознание у студентов, должен использовать лучший технический подход.

Некоторые преподаватели пытаются использовать гибриды С из-за ощущаемого давления индустрии. Но этого не стоит делать по ряду причин:

- Требования индустрии изменчивы. В какие-то годы все хотели что-то подобное RPG и Cobol. В конце 1996 все начали требовать Java, но еще в 1995 никто и не слышал о Java. Что же будет стоять в списке 2010 или 2020? Мы не знаем, но мы обязаны наделить наших студентов потенциалом, который будет востребован на рынке и в эти годы. По этой причине особое внимание следует уделять долговременным навыкам проектирования и разумным (intellectual) принципам.
- То, что мы начинаем обучать таким навыкам и принципам, вовсе не исключает обучения специфическим подходам. На самом деле, как отмечалось, скорее помогает. Студент, глубоко освоивший ОО-концепции, используя подходящую нотацию, будет лучшим C++- или Java-программистом, чем тот, для кого первая встреча с программированием включала битву с языком.
- Исторический прецедент с Pascal показывает, что преподаватели информатики могут добиться успеха, благодаря собственному выбору. В середине семидесятых годов в индустрии никто не требовал Pascal; фактически почти никто в индустрии и не слышал о Pascal. В те времена индустрия требовала одного из Трех Теноров - Fortran, Cobol и PL/I. В преподавании и в науке было выбрано другое решение - наилучшее техническое решение, соответствующее тому уровню, на котором находилась программистская методология - структурное программирование. Результат себя оправдал, студентов стали обучать абстрактным концепциям и техническим приемам разработки программ, подготавливая их к изучению новых языков и инструментария.

Другие курсы

Помимо вводных курсов, ОО-метод может играть роль на многих этапах программистского образования, находя соответствующее отражение в учебных планах. Давайте рассмотрим соответствующие возможности.

Терминология

Организация высшего образования в разных странах имеет серьезные отличия. Во избежание недоразумений следует договориться о терминологии, обозначающей разные уровни подготовки. Попытаемся привести все к некоему общему знаменателю:

- Под средним образованием будем понимать средние школы, лицеи, гимназии (High school (US), lycée, Gymnasium).
- Высшее образование - первые несколько лет университета или их эквивалент (то, что называется "undergraduate studies" в США и других англо-саксонских странах, **Gakubu** в Японии). Во Франции и других странах, находящихся под влиянием ее системы образования, это соответствует комбинации классов **préparatoires** и первых двух лет инженерных школ или первому и второму циклу университетов. В системе образования Германии это соответствует **Grundstudium**. Термин высшее образование (undergraduate) будет использоваться ниже для этого уровня.
- Наконец, для последних лет образования (магистратура, аспирантура), заканчивающихся присуждением научных степеней, будем использовать термин аспирантура (**graduate** - в США, **postgraduate** - в Англии, **DEA**, **DESS** - во Франции, **Hauptstudium** - в Германии, **Daigakuin** - в

Японии).

Среднее и высшее образование

На уровне среднего и высшего образования ОО-метод может играть центральную роль, как уже отмечалось, во вводных курсах. Он может также помогать и при изучении многих других курсов. Мы будем различать курсы, полностью использующие ОО-метод, и те, что получают преимущества от частичного использования некоторых ОО-идей.

Вот некоторые из стандартных курсов, изучение которых может быть полностью основано на ОО-подходе:

- **Структуры данных и алгоритмы.** Фундаментом может служить техника Проектирования по Контракту: подпрограммы должны характеризоваться утверждениями, структуры данных специфицироваться инвариантами класса, с алгоритмами должны связываться инварианты и варианты циклов. Помимо этого, новаторским и эффективным способом является организация этого курса как проекта на базе существующей библиотеки программных компонентов в существующем ОО-окружении. Тогда, вместо того чтобы начинать с нуля, студенты могут заниматься повторением и улучшением компонентов. (Смотри ниже подробности на эту тему.)
- **Инженерия ПО.** ОО-метод обеспечивает отличную основу для введения студентов в проблемы индустриальной, командной разработки ПО. В частности, оценка способов управления проектами, метрики, применяемые в процессе разработки, экономика проекта, окружение разработки и другие вопросы, обсуждаемые в литературе по инженерии ПО (в дополнение к объектной ориентации).
- **Анализ и проектирование.** Ясно, что все это может изучаться полностью на объектной основе. Снова в центре будет оставаться Проектирование по Контракту. В курсе должен делаться акцент на бесшовном переходе к реализации и сопровождению.
- **Введение в графику, введение в моделирование** и так далее.

Вот курсы, которые могут получать преимущества от использования "тяжелых" или "легких" объектов.

Операционные системы, где метод помогает освоить понятия процесса, парадигму обмена сообщениями, важность скрытия информации, четкого определения интерфейсов, ограничений коммуникационных каналов при проектировании подходящей архитектуры систем. **Введение в формальные методы**, **Функциональное программирование**, **Логическое программирование**, где упор должен делаться на связь с утверждениями. Введение в искусственный интеллект, где наследование является ключевым элементом представления знаний. **Базы данных**, где центральное место должно отводиться понятию АТД и обсуждению ОО-баз данных.

Даже на курс по архитектуре компьютеров влияют ОО-идеи: концепции модульности, скрытия информации и утверждения могут служить для представления материала ясным и убедительным способом.

Курсы для аспирантов

На аспирантском и магистерском уровне возможны многие продвинутые семинары и курсы: параллельные вычисления, распределенные системы, базы данных, формальные спецификации, углубленные методы анализа и проектирования, управление конфигурацией, управление распределенным проектом, верификация программ.

Полный учебный план (curriculum)

Этот неполный список показывает возможности повсеместного применения метода, так что возникает ощущение, что вокруг него можно построить полный учебный план по специальности (software curriculum). Несколько организаций уже добились некоторого успеха в этом направлении. Нет сомнений, что кто-то может сделать прорыв и довести до сознания руководства университета, что следует идти по этой дороге.

Вперед к новой педагогике для программистов

Эффект объектной технологии не только в том, что можно научить студентов современному программированию. Метод предлагает новые педагогические приемы, исследованием которых мы и займемся.

Важное замечание: стратегии, описываемые в оставшейся части лекции, принадлежат будущему. Я верю, что они должны стать превалирующими в обучении создания ПО, но их полное применение требует инфраструктуры, отсутствующей в настоящее время, в частности, соответствующих учебников и административной политики.

Если вы или ваша организация не готовы принять данные стратегии, то это еще не означает, что не следует использовать объекты при обучении. Вы можете все же, как описано в предыдущих разделах, вводить объектную технологию, достигая совместимости с текущим способом обучения.

Оставшуюся часть этой лекции следует прочитать в любом случае. Если вы и не примете радикальные предложения, возможно, вы используете одну или парочку идей в традиционном контексте.

Стратегия "от потребителя к производителю"

ОО-курс по структурам данным и алгоритмам, как отмечалось выше, может быть построен вокруг библиотеки. Эта идея фактически имеет более широкую область применения.

Разочаровывающий эффект многих курсов зачастую связан с тем, что преподаватели дают только простые примеры и упражнения, так что студенты не работают с реальными интересными приложениями. Можно ли получить удовлетворение от вычисления первых 25 чисел Фибоначчи или от замены в тексте одного слова другим - типичные примеры элементарного программистского курса.

С ОО-методом, хорошим ОО-окружением и, что наиболее важно, хорошими библиотеками возможной становится другая стратегия, при которой студентам дается доступ к библиотекам, как можно раньше. В этой роли студенты выступают просто как потребители, используя библиотечные компоненты, как черные ящики в том их значении, как это было описано в одной из предыдущих лекций. Этот подход предполагает доступность описания компонентов без показа их внутреннего содержания. Тогда студенты могут сразу же начать строить осмысленные приложения: их задача состоит главным образом в том, чтобы собрать систему из существующих компонентов. Проблемы и радости разработки познаются лучше, чем при рассмотрении игрушечных примеров, которыми довольствуются многие вводные курсы.

Повторно используя данное им ПО, студенты за день могут создать впечатляющее приложение. Их первое задание может состоять из написания всего нескольких строчек, сводящихся к вызову предварительно построенного приложения и вывода поразительных результатов (подготовленных кем-либо ранее!). Желательно, кстати, использовать библиотеки, включающие графику, мультимедийные компоненты, чтобы вывод был по-настоящему захватывающим.

Шаг за шагом студенты будут идти дальше: анализируя тексты некоторых компонентов, они могут сделать некоторые модификации и расширения либо в самих классах, либо в их новых потомках. **Наконец**, они перейдут к написанию собственных классов (шаг, который был бы первым в традиционном учебном плане, но который не должен встречаться, пока есть обильное поле деятельности для работы с библиотеками).

Этот обучающий процесс может быть назван "постепенным открытием черных ящиков" или более коротко стратегией "от потребителя к производителю". ("Изнутри наружу" также было бы подходящим названием стратегии.)

Стратегия от потребителя к производителю

- **S1** Научитесь использовать библиотечные классы, зная их абстрактные спецификации.
- **S2** Научитесь понимать внутреннее устройство выбранных классов.
- **S3** Научитесь расширять выбранные классы.
- **S4** Научитесь модифицировать выбранные классы.
- **S5** Научитесь добавлять собственные классы.

Если вам нравятся автомобильные сравнения, то подумайте о том, кто первым научил вас водить машину, затем поднял капот и шаг за шагом рассказал вам о работе мотора, потом научил вас ремонтировать его и много позже, как проектировать собственные автомобили.

Чтобы этот процесс заработал, необходимы хорошие средства абстракции, позволяющие потребителю понять сущность отдельного компонента без полного с ним знакомства. Понятие **краткой формы** класса поддерживает эту идею, перечисляя экспортируемые компоненты с их утверждениями, скрывая свойства реализации. После того как студенты увидят и разберутся в краткой форме, они могут выборочно ознакомиться и с реализацией снова под руководством инструктора.

Абстракция

Наиболее хорошие учебники по программированию проповедуют абстракцию, некоторые из них включают это слово в заголовок. Их авторы, будучи профессионалами в разработке программ и в обучении, понимают, что никто не может справиться с масштабной разработкой программной системы без постоянных усилий в поисках абстракций.

К несчастью, эти проповеди редко доходят до студентов, видящих в них очередное увещевание "быть паинькой". Небольшие программистские упражнения, любимые в традиционных методах обучения, вовсе не требуют поиска абстракций. Так зачем же обращать внимание на заклинания преподавателя о важности абстракции? Она не способна, так им кажется, улучшить их рейтинг. И только тогда, когда они перейдут к большим разработкам, студенты смогут оценить полезность этих советов.

Проповеди не лучший способ обучения. В стратегии "от потребителя к производителю", основанной на библиотеках, абстракция не является чем-то особенным - это практическое и необходимое средство. Без абстракции невозможно использовать библиотеки, альтернативой было бы изучение исходного кода. Только через краткую форму, содержащую утверждения и информацию высокого уровня, студенты могут познакомиться с библиотечным модулем и оценить преимущества библиотечного класса.

Приучаясь с самого начала видеть классы через призму абстрактных интерфейсов, студенты значительно проще начнут применять те же принципы в собственных классах.

Снова замечу, что эти результаты возможны только при условии, что окружение библиотеки поддерживает краткую форму и другой необходимый инструментарий.

Ученичество

Стратегия "от потребителя к производителю" представляет применение к программистскому обучению техники, освященной временем, - ученичество. Ученик, отанный в обучение мастеру, учится, пока мастер не поймет, что техника подмастерья не хуже, чем у мастера. За отсутствием нужного числа доступных мастеров применимость такого способа (мастер - ученик) ограничена. Но, к счастью, нам не нужны сами мастера, достаточно иметь результаты их работы - повторно используемые компоненты.

Этот подход является продолжением тенденции, уже повлиявшей на обучение некоторым темам в программистском образовании, таких как конструирование компиляторов, еще до того, как объектная технология стала популярной. В семидесятых и в начале восьмидесятых годов типичный семестровый курс по компиляторам включал написание компилятора (интерпретатора) с нуля. Однако первые же задачи лексического анализа и разбора требовали столь больших усилий, что на практике компилятор мог быть построен только для игрушечного языка. Обычно дело не доходило до наиболее интересных вещей: семантического анализа, генерации кода и оптимизации. Когда же появился и стал широко применяться инструментарий для лексического анализа и разбора, такой как Lex и Yacc, то это позволило студентам тратить на эти задачи существенно меньше времени. Наша стратегия обобщает этот опыт.

Обращенный учебный план

Стратегия "от потребителя к производителю" имеет интересного двойника в электротехнической инженерии, где Бернар Кохен предложил "обращенный учебный план". Критикуя классическую последовательность изложения (теория поля, теория цепей, энергетика, устройства, теория управления, цифровые системы, VLSI проектирование), он предлагает системно-ориентированную последовательность:

- цифровые системы, использующие VLSI и CAD;
- обратная связь, распараллеливание, верификация;
- линейные системы и управление;
- прием и передача энергии;
- устройства и технологии.

Стратегия программистского образования, предлагаемая выше, аналогична: не повторяя филогенез, начать давать студентам пользовательское видение концепций высокого уровня и методов, фактически применяемых в индустриальном окружении, затем шаг за шагом вводить принципы, лежащие в их основе.

Политика многих семестров

Стратегия "от потребителя к производителю" имеет интересный вариант применения в курсах, ориентированных на приложения, таких как Операционные системы, Графика, Конструирование компиляторов или Искусственный интеллект.

Идея состоит в том, чтобы дать студентам возможность построения системы путем последовательного расширения и обобщения, используя на каждом году обучения труд предыдущего года. Этот метод имеет очевидный недостаток на первом курсе, поскольку его труд служит основой для последующих расширений, но сам он не получает преимуществ повторного использования. Должен признаться, что я не видел систематического применения такого подхода, но на бумаге он выглядит привлекательным. Кажется, что вряд ли есть лучший способ дать студентам почувствовать все преимущества и трудности повторного использования, необходимости построения расширяющегося ПО, проблем улучшения кем-то сделанной работы. Такой опыт подготовил бы студентов к работе в их будущей компании, где шансы заняться сопровождением уже разработанного ПО гораздо выше участия в разработке нового продукта.

Даже если условия не допускают такой многолетней работы, следует избегать стандартных ловушек. Многие учебные планы высшего образования включают курс по "инженерии ПО", часто играющий ключевую роль для проекта, разрабатываемого группой студентов. Такая работа над проектом необходима, но зачастую оставляет разочарование из-за временных ограничений семестрового курса. Если это административно

возможно, то желательно вести эту работу в течение всего года, даже при том же количестве часов. Трехмесячные проекты на границе абсурда, они либо заканчиваются на этапе анализа или проектирования, или результатом будет гонка в работе над кодом в течение последних нескольких недель с применением любых методов, часто противоречащих исходным целям образования. Нужно больше времени, чтобы студенты могли ощутить глубину проблем, стоящих при построении серьезного ПО. Проект, длящийся год, а еще лучше являющийся частью многосеместровой политики, благоприятствует этому процессу. Он плохо укладывается в типичные стартовые планы, но за него стоит сражаться.

Объектно-ориентированный план

Идея стратегии многих семестров, основанная на повторном использовании, а также на организации всего учебного процесса вокруг ОО-концепций, может привести к более амбициозному подходу, захватывающему не только образование, но и исследования и разработки. Хотя эта концепция может быть привлекательной не для всех институтов, она заслуживает рассмотрения.

Рассмотрим факультет (кафедру) университета (информатики, информационных систем или их эквивалент) в поисках многосеместрового объединяющего проекта. Такой проект призван обеспечить лучшее обучение, разработку новых курсов, факультетские исследования, являлся бы источником публикаций, магистерских и кандидатских диссертаций, дипломных работ, допускал бы сотрудничество с индустрией и получение правительственные грантов. Сегодня большинство уважаемых факультетов именно так позиционируют себя, ориентируясь на многолетнюю коллективную работу.

ОО-метод является естественной основой таких попыток. Сосредоточиться необходимо не на разработке компиляторов, интерпретаторов или инструментария (это прерогатива компаний), но на библиотеках. То, что сегодня необходимо ОО-технологии, так это повторно используемые компоненты, ориентированные на приложения, называемые также прикладными библиотеками. Хорошее ОО-окружение уже обеспечивает, как отмечалось, множество библиотек общеподходового назначения, покрывающих такие универсальные потребности, как фундаментальные структуры данных и алгоритмы, графику, проектирование пользовательского интерфейса, грамматического разбора текстов. Открытыми остаются многие области приложений - от Web-документов до мультимедиа, от финансового ПО до анализа сигналов, от компьютерного проектирования документов до обработки документов. В подобных областях необходимость качественных компонентов является прямо кричащей.

Выбор разработки библиотеки в качестве проекта, объединяющего усилия факультета, дает несколько преимуществ:

- Хотя проект рассчитан на длительную перспективу, частные результаты могут быть получены в короткие сроки. Компиляторы и подобные разработки относятся к категории "все или ничего" - пока они полностью не завершены, их распространение скорее повредит вашей репутации, чем поможет ей. С библиотеками дело обстоит по-другому, даже десятка два качественных повторно используемых классов могут оказать потрясающую службу своим пользователям и привлечь к ним значительное внимание.
- Поскольку серьезная библиотека является большим проектом, то в ней найдется место для вклада многих людей - от хорошо подготовленных студентов до доцентов и профессоров. Предполагается, конечно, что правильно выбрана проблемная область, и она соответствует научным и другим ресурсам факультета или кафедры.
- Если говорить о ресурсах, то проект может начинаться достаточно скромно, но быть прямым кандидатом для привлечения внимания организаций, занимающихся распределением фондов. Он также может быть предложен тем компаниям, для которых интересна выбранная проблемная область.
- Построение хорошей библиотеки представляет привлекательную задачу, ставящую новые научные проблемы, так что выходом успешного проекта может быть не только ПО, но и публикации, диссертации. Возникающие при этом научные проблемы могут быть двух видов. Во-первых, конструирование повторно используемых компонентов представляет одну из наиболее интересных и трудных проблем инженерии программ, в решении которых метод оказывает некоторую помощь, но определенно не дает ответа на все вопросы. Во-вторых, любая успешная прикладная библиотека должна опираться на таксономию предметной области, требуя долговременных усилий в классификации известных концепций в этой области. Как хорошо известно, в естественных науках (вспомните обсуждение истории таксономии в [лекции 6](#)) классификация является первым шагом в понимании сути. Такие усилия, принятые для новой проблемной области, известные как анализ предметной области, поднимают новые и интересные проблемы.
- Предполагается возможность междисциплинарной кооперации с исследователями из различных областей приложения.
- Кооперацию следует начинать с людей, работающих в соседних областях. Многие университеты располагают двумя группами специалистов: одних, ориентированных на инженерию программ (часто "**comliuting science**"), других - на проблемы бизнеса (часто "**information systems**"). Независимо от того, разделены эти две группы или являются частями одной структуры, проект может быть привлекателен для обеих групп, обеспечивая возможность сотрудничества.
- Наконец, успешная библиотека, предоставляющая программные компоненты для важной проблемной

области, может стать широко известной, принеся известность ее разработчикам.

Можно надеяться, что в ближайшие годы появятся университеты, вдохновившиеся этими идеями и создавшие "Повторно используемые Финансовые Компоненты X Университета -" или "Библиотека ОО-Обработки Текстов Y Политехнического института". Имена у них будут красивее приведенных, но столь же известны, как в свое время UCSD Pascal, Waterloo Fortran и система X Window, разработанная в MIT.

Ключевые концепции

- В ОО-тренинге основное внимание уделяйте реализации и проектированию.
- В начальном тренинге для профессионалов без колебаний повторяйте сессию, посвященную концепциям, после некоторой фазы практической работы.
- Тренинг в компании должен включать курсы для менеджеров наряду с курсами для разработчиков.
- Начальный курс по программированию и многие другие могут получать преимущества от введения ОО-приемов.
- Для обучения используйте чистый ОО-язык, простой и понятный, поддерживающий полный спектр технологий, в частности утверждения.
- Курсы должны, насколько возможно, основываться на библиотеках повторно используемых компонентов.
- Стратегия "от потребителя к производителю" (подобная идеям обращенного учебного плана) - снабжать студентов существующими компонентами, позволяя им с самого начала создавать полноценные приложения, расширять их и создавать новые компоненты, имитируя процесс ученичества.
- Долговременный проект по созданию библиотеки может объединить усилия кафедры или факультета.

Библиографические замечания

Материалы этой лекции используют статью, опубликованную в **Journal of Object-Oriented Programming**, пересмотренная версия которой была представлена на конференции TOOLS USA 93 и появилась в ее трудах (обе ссылки даны в [M 1993c]). Дальнейшие материалы о проблемах образования и тренинга появились в книге **Object Success** [M1995], из которой и взят термин **mOOzak**, а также некоторые замечания о тренинге в индустрии.

Важные статьи об обучении программированию с использованием ОО-концепций даны в [McKim 1992] и [Helotis 1996].

Понятие обращенного учебного плана в электротехнике ввел Bernard Cohen в [Cohen 1991]. Я благодарен Уоррену Йейтсу, декану факультета электротехники в Технологического университета в Сиднее, привлекшего мое внимание к этой работе. Эта лекция также использует результаты многочисленных дискуссий с преподавателями: Christine Mingins, James McKim, Richard Mitchell, John Potter, Robert Switzer, Jean-Claude Boussard, Roger Rousseau, David Riley, Richard Wiener, Fiorella De Cindio, Brian Henderson-Sellers, Pete Thomas, Ray Weedon, John Kerstholt, Jacob Gore, David Rine, Naftaly Minsky, Peter Lohr, Robert Ogor, Robert Rannou.

Есть несколько хороших учебников вводного курса по программированию на основе ОО-идей. Они были перечислены в библиографии к [лекции 2](#) курса "Основы объектно-ориентированного программирования", но для удобства приводятся здесь еще раз уже без комментариев: [Rist 1995], [Wiener 1996], [Gore 1996], [Wiener 1997], [Jzequel 1996].

Основы объектно-ориентированного проектирования

12. Лекция: Параллельность, распределенность, клиент-сервер и Интернет

Как и люди, компьютеры могут объединяться в команды для достижения результатов, которые ни один из них не может получить в одиночку, но в отличие от людей они могут в выполнять много дел одновременно (или делать вид, что делают их одновременно) и делать их хорошо. Однако до сих пор в обсуждении неявно предполагалось, что вычисление является последовательным, т. е. управляемся одной цепочкой команд. Сейчас мы увидим, что получится, если отказаться от этого предположения и перейти к рассмотрению параллельных вычислений. Параллельность - это не новый предмет, но долгое время интерес к ней ограничивался, в основном, четырьмя областями применения: операционными системами, сетями, реализацией систем управления базами данных и высокопроизводительными научными программами. Хотя эти области стратегически важны и престижны, лишь небольшая часть программистского сообщества вовлечена в их разработку. Но положение дел изменилось. Параллельность быстро становится необходимой во всех видах приложений, включая такие, которые традиционно представлялись как существенно последовательные по своей природе. Наши системы не просто параллельные, они, будучи или нет системами типа клиент-сервер, должны все больше становиться распределенными по сетям, включая сеть сетей - Интернет. Эта эволюция делает особенно настоятельным центральный вопрос этой лекции: можно ли применять ОО-идеи в контексте параллельности и распределенности? Это не только возможно: объектная технология может помочь разрабатывать параллельные и распределенные приложения просто и элегантно.

Предварительный просмотр

Как и обычно, при обсуждении параллелизма мы не предложим заранее подготовленный ответ, но вместо этого тщательно построим решение, исходя из детального анализа проблемы и изучения различных путей ее решения, включая некоторые тупиковые. Хотя такая тщательность необходима для глубокого понимания рассматриваемых методов, она могла бы привести читателя к мысли об их большой сложности, что было бы непростительно, так как тот параллельный механизм, к которому мы в конце концов придем, на самом деле отличается неправдоподобной простотой. Чтобы избежать такого риска, начнем с обзора этого механизма, отложив обоснования на потом.

Если вам не нравится забегать вперед и вы предпочитаете последовательное изучение предмета и продвижение к развязке драмы шаг за шагом и вывод за выводом, то пропустите следующую страницу с резюме и переходите к следующему разделу.

Расширение, полностью охватывающее параллельность и распределенность, будет самым минимальным из всех возможных: к последовательным обозначениям добавляется единственное новое ключевое слово - `separate`. Почему это возможно? Мы используем основную схему ОО-вычислений: вызов компонента `x.f` (`a`), выполняемый от имени некоторого объекта `O1`, и вызывающий компонент `f` объекта `O2`, присоединенного к `x` с аргументом `a`. Но сейчас вместо одного процессора, выполняющего операции всех объектов, мы рассчитываем на возможность использовать разные процессоры для `O1` и `O2`, так что вычисление `O1` может продолжаться, не ожидая завершения указанного вызова, поскольку он обрабатывается другим процессором.

Поскольку результат вызова сейчас зависит от того, обрабатываются ли объекты одним процессором или несколькими, в тексте программы об этом должно быть точно сказано для каждой сущности `x`. Поэтому требуется новое ключевое слово: вместо того, чтобы объявлять просто `x : SOME_TYPE`, будем объявлять `x : separate SOME_TYPE`, чтобы указать, что `x` обрабатывается отдельным процессором, так что вызовы с целью `x` могут выполняться параллельно с остальным вычислением. При таком объявлении всякая команда создания `create x.make (. . .)` будет порождать новый процессор - новую ветвь управления - для обработки будущих вызовов `x`.

Нигде в тексте программы не требуется указывать, какой именно процессор нужно использовать. Все, что утверждается посредством объявления `separate` - это то, что два объекта обрабатываются различными процессорами, и это существенно влияет на семантику системы. Назначение конкретного процессора можно перенести на время исполнения. Мы также не устанавливаем заранее точную природу процессора: он может быть реализован как часть оборудования (компьютера), но может также оказаться заданием (процессом) операционной системы или, в случае многопоточной ОС, стать одной из нитей (потоков) задания. С точки зрения программы "процессор" - это абстрактное понятие; одно и то же параллельное приложение может выполняться на совершенно разных архитектурах (на одном компьютере с разделением времени, в распределенной сети со многими компьютерами, несколькими потоками одного задания под Unix или Windows) без всякого изменения его исходного текста. Все, что потребуется изменить, - это "Файл параллельной конфигурации" - ("Concurrency Configuration File"), задающий отображение абстрактных процессоров на физические ресурсы.

Определим ограничения, связанные с синхронизацией. Эти соглашения достаточно просты:

- Клиенту не требуется никакого специального механизма для повторной синхронизации с сервером после того, как вызов `x.f` (`a`) для объявленной `separate` сущности `x` пойдет на параллельное выполнение. Клиент будет ждать столько, сколько необходимо, когда он запрашивает информацию об объекте с помощью вызова запроса, как в операторе `value := x.some_query`. Этот автоматический механизм называется **ожидание по необходимости (wait by necessity)**.
- Для получения исключительного доступа к отдельному объекту `O2` достаточно использовать присоединенную к нему сущность `a`, объявленную как `separate`, в качестве аргумента соответствующего вызова, например, `r(a)`.
- Если у подпрограммы имеется предусловие, содержащее аргумент, объявленный как `separate` (например, такой как `a`), то клиенту придется ждать, пока это предусловие не выполнится.
- Для контроля за работой ПО и предсказуемости результатов (в частности, поддержания инвариантов класса) нужно разрешать процессору, ответственному за объект, выполнять в каждый момент времени не более одной процедуры.
- Однако иногда может потребоваться **прервать** выполнение некоторой процедуры, уступив ресурсы новому более приоритетному клиенту. Клиент, которого прервали, сможет произвести соответствующие корректирующие мероприятия; наиболее вероятно, что он повторит попытку после некоторого ожидания.

Это описание охватывает основные свойства механизма, позволяющего строить продвинутые параллельные и распределенные приложения, в полной мере используя ОО-методы от множественного наследования до проектирования по контракту. Далее мы рассмотрим этот механизм детально, забыв на время то, что прочли только что в этом кратком обзоре.

Возникновение параллельности

Вернемся к началу. Чтобы понять, как эволюция потребовала от разработчиков сделать параллельность частью их образа мысли, проанализируем различные виды параллельности. В дополнение к традиционным понятиям мультипроцессорной обработки (multiprocessing) и многозадачности (multiprogramming) за несколько последних лет было введено два новых понятия: посредники запроса объекта (object request brokers) и удаленное выполнение в Сети.

Мультипроцессорная обработка

Чем больше хочется использовать огромную вычислительную мощь, тем меньше хотелось бы ждать ответа компьютера (хотя мы вполне миримся с тем, что компьютер ждет нас). Поэтому, если один вычислитель не выдает требуемый результат достаточно быстро, то приходится использовать несколько вычислителей, работающих параллельно. Эта форма параллельности называется мультипроцессорной обработкой.

Впечатляющие приложения мультипроцессорности привлекли исследователей, надеющихся на работу сотен компьютеров, разбросанных по сети Интернет, в то время, когда их (по-видимому, согласные с этим) владельцы в них не нуждаются, к решению задач, требующих интенсивных вычислений, таких, например, как взлом криптографических алгоритмов. Такие усилия прилагаются не только в компьютерных исследованиях. Ненасытное требование Голливудом реалистичной компьютерной графики подбрасывает топливо в топку прогресса этой области: в создании фильма **Toy Story**, одного из первых, в котором играли только искусственные персонажи (люди их лишь озвучивали), участвовала сеть из сотни мощных рабочих станций - это оказалось более экономичным, чем привлечение сотни профессиональных мультипликаторов.

Мультипроцессорность повсеместно используется в высокоскоростных научных вычислениях при решении физических задач большой размерности, в инженерных расчетах, метеорологии, статистике, инвестиционных банковских расчетах.

Во многих вычислительных системах часто применяется некоторый вид балансирования нагрузки (**load balancing**): автоматическое распределение вычислений по разным компьютерам, доступным в данный момент в локальной сети некоторой организации.

Другой формой мультипроцессорности является вычислительная архитектура, называемая **клиент-сервер (client-server computing)**, присваивающая разные роли компьютерам в сети: несколько самых крупных и дорогих машин являются "серверами", выполняющими большие объемы вычислений, работающими с общими базами данных и содержащими другие централизованные ресурсы. Более дешевые машины сети, расположенные у конечных пользователей, выполняют децентрализованные задания, обеспечивая интерфейс и проведение простых вычислений, передавая серверам все задачи, не входящие в их компетенцию, получая от них результаты решений.

Нынешняя популярность подхода клиент-сервер представляется колебанием маятника в направлении, противоположном тенденциям предыдущего десятилетия. В 60-х и 70-х архитектуры были централизованными, заставляя пользователей бороться за ресурсы. Революция, вызванная появлением персональных компьютеров и рабочих станций в 80-х, наделила пользователей ресурсами, ранее приберегаемыми Центром (на промышленном жаргоне "стеклянным домом"). Но вскоре стало очевидным, что персональный компьютер может далеко не все и некоторые ресурсы **должны** быть общими (разделяться). Это объясняет появление архитектуры клиент-сервер в 90-х. Постоянный циничный комментарий - мы возвратились к архитектуре нашей юности: одна центральная машина - много терминалов, только с более дорогими терминалами, называемыми сейчас рабочими станциями, - на самом деле, не вполне оправдан: промышленность просто ищет путем проб и ошибок подходящее соотношение между децентрализацией и разделением ресурсов.

Многозадачность

Другой главной формой параллельности является многозадачность, когда один компьютер выполняет одновременно несколько задач.

Если рассмотреть системы общего назначения (исключая процессоры, встроенные в оборудование от стиральных машин до самолетов и однообразно повторяющие фиксированный набор операций), то компьютеры почти всегда являются многозадачными, выполняя задачи операционной системы параллельно с задачами приложений. Строго говоря, параллелизм при многозадачности скорее мнимый, чем настоящий: в каждый момент времени процессор на самом деле выполняет одно задание, но время переключения с одного задания на другое столь коротко, что внешний наблюдатель может поверить в то, что они выполняются одновременно. Кроме того, сам процессор может делать некоторые вещи параллельно (как, например, в современных схемах выборки команд во многих компьютерах, когда за один такт одновременно с выполнением текущей команды загружается следующая) или может на самом деле быть комбинацией нескольких вычисляющих компонентов, так что многозадачность переплетается с мультипроцессорностью.

Обычным применением многозадачности является **разделение времени компьютера (time-sharing)**, позволяющее одной машине обслуживать одновременно нескольких пользователей. Но, за исключением случая самых мощных компьютеров - "мэйнфреймов", эта идея сегодня предстает гораздо менее привлекательной, чем в те времена, когда компьютеры были большой редкостью. Сегодня наше время является более ценным ресурсом, поэтому хотим, чтобы система выполняла для нас несколько дел одновременно. В частности, многооконный интерфейс пользователя позволяет одновременно выполнять несколько приложений: в одном окне мы осуществляем поиск в Интернете, в другом - редактируем документ, а еще в одном компилируем и отлаживаем некоторую программу. Все это требует мощных механизмов параллельности.

Ответственность за предоставление каждому пользователю многооконного многозадачного интерфейса лежит на операционной системе. Но все больше пользователей разрабатываемых нами программ хотят иметь параллельность **внутри одного приложения**. Причина все та же: они знают, что вычислительные мощности доступны в изобилии, и не хотят сидеть в пассивном ожидании. Так что, если получение пришедших по электронной почте сообщений требует много времени, то хотелось бы иметь возможность в это же время послать исходящие сообщения. В хорошем Интернет-браузере можно получать доступ к новому сайту во время загрузки страниц из другого сайта. В системе биржевой торговли можно одновременно получать информацию с нескольких бирж, покупая в одном месте, продавая в другом и управляя портфелем клиента в третьем.

Именно эта необходимость распараллеливания внутри одного приложения неожиданно выдвинула область параллельных вычислений на передний край разработки ПО и вызвала интерес к ней, в кругах далеко выходящих за рамки первоначальных спонсоров. Между тем традиционные приложения параллельности не потеряли своего значения в новых разработках, относящихся к операционным системам, Интернету, локальным сетям и научным вычислениям - всюду, где непрерывный поиск скорости требует еще высокого уровня многозадачности.

Посредники запросов объектов (брокеры объектных запросов - Object Request Broker)

Другим важным недавним достижением явилось появление предложения CORBA от Группы управления объектами (Object Management Group) и архитектуры OLE 2/ActiveX от фирмы Майкрософт. Хотя их окончательные цели, детали и рынки различны, оба

предложения обещают существенное продвижение в направлении распределенных вычислений¹.

Общая цель состоит в том, чтобы сделать объекты и услуги различных приложений доступными друг для друга наиболее удобным образом локально или через сеть. Усилия CORBA направлены, в частности, на достижение интероперабельности (interoperability).

- Приложения, поддерживающие CORBA, могут взаимодействовать между собой, даже если они основаны на "посредниках запроса объектов" разных производителей.
- Интероперабельность применяется также и на уровне языка: приложение на одном из поддерживаемых языков может получить доступ к объектам приложения, написанного на другом языке. Взаимодействие происходит с помощью внутреннего языка, называемого IDL (язык определения интерфейса - Interface Definition Language); у поддерживаемых языков имеется официальная привязка к IDL, в которой определено, как конструкции языка отображаются на конструкции IDL.

IDL - это общий знаменатель ОО-языка, сконцентрированного на понятии интерфейса. Интерфейс IDL для класса по духу похож на его краткую форму, хотя и более примитивную (в частности, IDL не поддерживает утверждений); в нем описывается набор компонентов, доступных на некотором уровне абстракции. По классу, написанному на ОО-языке, с помощью инструментальных средств будет выводиться IDL-интерфейс класса, представляющий интерес для клиентов. Клиент, написанный на том же или на другом языке, может через этот IDL-интерфейс получать доступ по сети к компонентам, предоставляемым поставщиком класса.

Удаленное выполнение

Другим достижением поздних 90-х является механизм для удаленного выполнения программ через Всемирную Паутину (World-Wide Web).

Первые Веб-браузеры сделали не только возможным, но и весьма удобным использование информации, хранящейся на удаленных компьютерах, расположенных по всему миру, и переходы по логическим связям или гиперссылкам с помощью одного щелчка мыши. Но это был пассивный механизм: некто готовит некоторую информацию, а все остальные получают к ней доступ в режиме чтения.

Следующим шагом был переход к достижению активности, когда щелчок по ссылке вызывает выполнение некоторой операции. Это предполагает наличие внутри браузера некоторой машины выполнения, распознающей загружаемую информацию как выполняемый код и выполняющей его. Эта машина может быть встроенной частью браузера или динамически присоединяется к нему в ответ на запрос соответствующего типа. Последнее известно как подключаемый (plug-in) механизм и предполагает доступность бесплатной загрузки его из Интернета.

Эта идея впервые сделалась популярной благодаря Java, когда машина исполнения Java-программ стала общедоступной. С тех пор появилась возможность подключения и других механизмов. Другим направлением стала трансляция исходных языков в код широко распространенной машины такой, например, как машина исполнения Java; действительно, несколько производителей компиляторов начали создавать генераторы "байт-кода" языка Java (это переносимый код низкого уровня, который может исполняться Java-машиной).

Для нотации, используемой в этой книге, работа шла в двух направлениях: у фирмы ISE имеется свободно распространяемая машина исполнения программ в этой нотации и в то же время разрабатывается проект порождения байт-кода языка Java.

При любом из этих подходов возникают проблемы **безопасности**: насколько можно доверять чужому приложению? Неосторожный щелчок по невинно выглядящей гиперссылке способен запустить нехорошую программу, портящую файлы компьютера или крадущую личную информацию. Следует самому быть осторожным, хотя ответственность лежит на поставщике исполняющей машины.

Решение проблемы предполагает использование тщательно разработанных и сертифицированных исполняющих машин и библиотек, пришедших из авторитетных источников. Часто у них две версии:

- Одна допускает неограниченное использование в Интернет и основана на строгих ограничениях возможностей исполняющей машины.
В средствах, предоставляемых ISE, в библиотеке ввода-вывода этой ограниченной версии допускается только чтение и запись с терминала и на терминал, а не в файлы. Механизм "external" для подключения внешних программ также отключен, так что плохое приложение не сможет причинить вред, например, с помощью перехода к C для выполнения манипуляций с файлами. "Виртуальная машина" языка Java также использует драконовские ограничения того, что разрешается делать апплетам, приходящим из Интернета с файловой системой вашего компьютера.
- В другой версии ограничений существенно меньше либо нет вовсе, она дает возможность использовать всю мощь библиотек, в частности, файлового ввода-вывода. Она предназначена для приложений, работающих в безопасной Инtranете (внутренней сети компании), а не на диком пространстве Интернета.

Несмотря на опасения ненадежности перспектива неограниченного удаленного выполнения - нового шага на пути продолжающейся революции в распространении ПО - породила огромный неослабевающий интерес.

От процессов к объектам

Для поддержки этих захватывающих дух достижений, требующих параллельной обработки, нужна мощная программная поддержка. Как мы собираемся программировать эти вещи? Конечно, для этого предлагается ОО-технология.

Говорят, что Робин Мильнер (Robin Milner) воскликнул в 1991 на одном из семинаров ОО-конференции: "Я не могу понять, почему параллельность объектов [ОО-языков] не стоит на первом месте" (цитируется по [Matsuoka 1993]). Даже, если поставить ее на второе или на третье место, то остается вопрос, как прийти к созданию параллельных объектов?

Если рассмотреть параллельную работу не в ОО-контексте, то она в большой степени основана на понятии процесса. Процесс - программная единица - действует как специализированный компьютер: он выполняет некоторый алгоритм, как правило, многократно, пока некоторое внешнее событие не приведет к его завершению. Типичным примером является процесс управления принтером, который последовательно повторяет:

"Ждать появления задания в очереди на печать"
"Взять задание и удалить его из очереди"
"Напечатать задание"

Разные модели параллельности различаются планированием и синхронизацией, борьбой за ресурсы, обменом информацией. В одних языках параллельного программирования непосредственно описываются процессы, в других, таких как Ada, можно также описывать **типы** процессов, которые во время выполнения реализуются в процессах так же, как классы ОО-ПО реализуются в объектах.

Сходство

Это соответствие кажется очевидным. Когда мы начинаем сравнивать идеи параллельного программирования и ОО-построения программ, то кажется естественным идентифицировать процессы с объектами, а типы процессов с классами. Каждый, кто вначале изучил параллельные вычисления, а затем открыл ОО-разработку (или наоборот) будет удивлен сходством между этими двумя технологиями:

- Обе основаны на автономных, инкапсулированных модулях: процессах или типах процессов и на классах.
- Объекты и процессы сохраняют содержащиеся в них значения от одной активации до следующей.
- Для построения параллельной системы на практике требуется налагать строгие ограничения на межмодульный обмен информацией. ОО-подход, как мы видели, тоже налагает строгие ограничения на межмодульную коммуникацию.
- В обоих случаях механизм коммуникации можно упрощенно описать как "передачу сообщений".

Поэтому неудивительно, что многие люди восклицают "Эврика!", когда впервые начинают размышлять, подобно Мильнеру, о наделении объектов параллельностью. Кажется, что можно легко достичь унификации этих понятий.

К сожалению, это первое впечатление ошибочно: после обнаружения первого сходства сталкиваешься с различиями.

Активные объекты

Основываясь на приведенных выше аналогиях, в многочисленных предложениях параллельных ОО-механизмов было введено понятие "активного объекта". Активный объект - это объект, являющийся также процессом: у него есть собственная исполняемая программа. Вот как он определяется в одной книге по языку Java [Doug Lea 1996]:

Каждый объект является единой идентифицируемой процессоподобной сущностью (не отличающейся (?)) от процесса в Unix) со своим состоянием и поведением.

Однако это понятие приводит к тяжелым проблемам. Легко понять самую важную из них. У процесса имеется собственный план решения задачи: на примере с принтером видно, что он постоянно выполняет некоторую последовательность действий. А у классов и объектов дело обстоит не так. Объект не делает одно и то же, он является хранилищем услуг (компонентов порожденного класса) и просто ожидает, когда очередной клиент запросит одну из этих услуг - она выбирается клиентом, а не объектом. Если сделать объект активным, то он сам станет определять расписание выполнения своих операций. Это приведет к конфликту с клиентами, которые совершенно точно знают, каким должно быть это расписание: им нужно только, чтобы поставщик в любой момент, когда от него потребуется конкретная услуга, был готов немедленно ее предоставить!

Эта проблема возникает и в не ОО-подходах к параллельности и приводит к механизмам синхронизации процессов - иначе говоря, к определению того, когда и как каждый процесс готов общаться, ожидая, если требуется, готовности другого процесса. Например, в очень простой схеме производитель-потребитель (producer-consumer) может быть процесс producer, который последовательно повторяет следующие действия:

"Сообщает, что producer не готов"
"Выполняет вычисление значения x"
"Сообщает, что producer готов"
"Ожидает готовности consumer"
"Передает x consumer"

и процесс consumer, который последовательно повторяет

"Сообщает, что consumer готов"
"Ожидает готовности producer"
"Получает x от producer"
"Сообщает, что consumer не готов"
"Выполняет вычисление, используя значение x"

Графически эту схему можно представить так

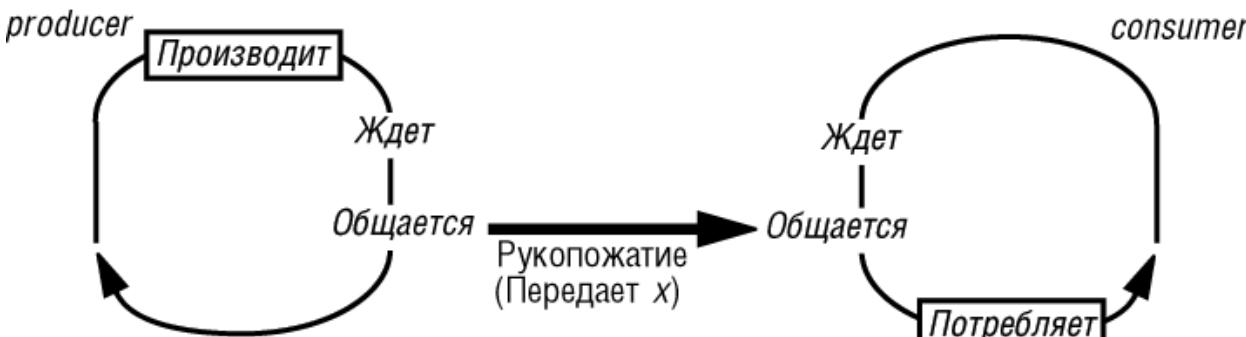


Рис. 12.1. Простая схема производитель-потребитель (producer-consumer)

Общение процессов происходит, когда оба они к этому готовы; это иногда называется **handshake** (рукопожатие) или **rendez-**

vous (рандеву). Проектирование механизмов синхронизации - позволяющих точно выражать смысл команд "Известить о готовности процесса" или "Ждать готовности" - на протяжении нескольких десятилетий является плодотворной областью исследований.

Все это хорошо для процессов, являющихся параллельными эквивалентами традиционных последовательных программ, "делающих одну вещь". Параллельная система, построенная на процессах, похожа на последовательную систему с несколькими главными программами. Но при ОО-подходе мы уже отвергли понятие главной программы и вместо нее определили программные единицы, предоставляющие клиентам несколько компонентов.

Согласование этого подхода с понятием процесса требует тщательно проработанных конструкций синхронизации, когда каждый поставщик будет готов выполнить компонент, затребованный клиентом. Согласование должно быть особенно тщательным, когда клиент и поставщик являются активными объектами, придерживающими своих порядков действий.

Изобретение механизмов, основанных на понятии активного объекта, вполне возможно, о чем свидетельствует обильная литература по этому предмету (в библиографических заметках к этой лекции приведено много соответствующих ссылок). Но это обилие свидетельствует о сложности предлагаемых решений, ни одно из которых не получило широкого признания, из чего можно заключить, что подход с использованием активных объектов сомнителен.

Конфликт активных объектов и наследования

Еще большие сомнения в правильности подхода, использующего активные объекты, появляются при его объединении с другими ОО-механизмами, особенно, с наследованием.

Если класс В является наследником класса А и оба они активны (т. е. описывают экземпляры, которые должны быть активными объектами), то что произойдет в В с описанием процесса А? Во многих случаях потребуется добавлять некоторые новые инструкции, но без специальных встроенных в язык механизмов это повлечет необходимость почти всегда переопределять и переписывать всю часть, относящуюся к процессу, - не очень привлекательное предложение.

Приведем пример одного специального языкового механизма. Хотя язык Simula 67 не поддерживает параллельность, в нем есть понятие активного объекта: класс Simula помимо компонентов содержит инструкции, называемые телом класса (см. [лекцию 17](#)). В теле класса А может содержаться специальная инструкция inner, не влияющая на сам класс, означающая подстановку собственного тела в потомке В. Так что, если тело А имеет вид:

```
some_initialization; inner; some_termination_actions
```

а тело В имеет вид:

```
specific_B_actions
```

то выполнение тела в В на самом деле означает:

```
some_initialization; specific_B_actions; some_termination_actions
```

Хотя необходимость механизмов такого рода для языков, поддерживающих понятие активного объекта, не вызывает сомнений, на ум сразу приходят возражения. Во-первых, эта нотация вводит в заблуждение, поскольку, зная только тело В, можно получить неверное представление его выполнения. Во-вторых, это заставляет родителя предугадывать действия наследников, что в корне противоречит основополагающим принципам ОО-проектирования (в частности принципу Открыт-Закрыт) и годится только для языка с единственным наследованием.

Основная проблема останется и при другой нотации: как соединить спецификацию процесса в классе со спецификациями процессов в его потомках, как примирить спецификации процессов нескольких родителей в случае множественного наследования?

Позже в этой лекции мы увидим и другие проблемы, известные как "аномалия наследования", возникающие при использовании наследования с ограничениями синхронизации.

Встретившись с этими трудностями, некоторые из ранних предложений по ОО-параллельности предпочли вообще отказаться от наследования. Хотя это оправдано, как временная мера, призванная помочь пониманию предмета путем разделения интересов, такое исключение наследования не может оставаться при окончательном выборе подхода к построению параллельного ОО-ПО; это было бы похоже на желание отрезать руку из-за того, что чешутся пальцы. (Для оправдания в некоторых источниках добавляют, что все равно наследование - это сложное и темное понятие, это - как сказать пациенту после операции, что иметь руку с самого начала было плохой идеей.)

Вывод, к которому мы можем придти, проще. Проблема не в ОО-технологии как таковой, и в частности, не в наследовании; она не в параллельности и даже не в комбинации этих идей. Источником неприятностей является понятие активного объекта.

Программируемые процессы

Поскольку мы готовы избавиться от активных объектов, полезно заметить, что на самом деле мы не хотим ни от чего отказываться. Объект способен выполнять много операций: все компоненты породившего его класса. Превращая объект в процесс, приходится выбирать одну из этих операций в качестве единственной реально вычисляемой. Это не дает абсолютно никаких преимуществ! Зачем ограничивать себя одним алгоритмом, когда можно иметь их столько, сколько нужно?

Заметим, что понятие процесса не обязательно должно быть встроено внутрь механизма параллельности; процессы можно **программировать**, рассматривая их как обычные программы. Процесс для принтера, приведенный в начале лекции, с ОО-точки зрения может трактоваться как одна из подпрограмм, скажем, live, соответствующего класса:

```
indexing
description: "Принтер, выполняющий в каждый момент одно задание"
note: "Улучшенная версия, основанная на общем классе PROCESS, %
      %появится далее под именем PRINTER"
class
```

```

PRINTER_1
feature -- Status report
  stop_requested: BOOLEAN is do ... end
  oldest: JOB is do ... end
feature -- Basic operations
  setup is do ... end
  wait_for_job is do ... end
  remove_oldest is do ... end
  print (j: JOB) is do ... end
feature -- Process behavior
  live is
    -- Выполнение работы принтера
    do
      from setup until stop_requested loop
        wait_for_job; print (oldest); remove_oldest
      end
    end
  ... Другие компоненты ...
end

```

Отметим заготовку для других компонентов: хотя до сих пор все наше внимание было уделено `live` и окружающим его компонентам, мы можем снабдить процесс и многими другими желательными компонентами, чему способствует ОО-подход, развитый в других частях этого курса. Превращение объектов класса `PRINTER_1` в процессы означало бы ограничение этой свободы, это была бы существенная потеря в выразительной силе без всякой видимой компенсации.

Абстрагируясь от этого примера, который описывает конкретный тип процесса просто как некоторый класс, можем попытаться предложить более общее описание всех типов процессов с помощью специального отложенного класса - **класса поведения**, как это уже не раз делалось в предыдущих лекциях. Процедура `live` будет применима ко всем процессам. Мы можем оставить ее отложенной, но нетрудно заметить, что большинство процессов будут нуждаться в некоторой инициализации, некотором завершении, а между ними - в некотором основном шаге, повторяемом некоторое число раз. Поэтому мы можем учесть это на самом абстрактном уровне:

```

indexing
  description: "Самое общее понятие процесса"
deferred class
  PROCESS
feature -- Status report
  over: BOOLEAN is
    -- Нужно ли сейчас прекратить выполнение?
    deferred
    end
feature -- Basic operations
  setup is
    -- Подготовка к выполнению операций процесса
    -- (по умолчанию: ничего)
    do
    end
  step is
    -- Выполнение основных операций
    deferred
    end
  wrapup is
    -- Выполнение операций завершения процесса
    -- (по умолчанию: ничего)
    do
    end
feature -- Process behavior
  live is
    -- Выполнение жизненного цикла процесса
    do
      from setup until over loop
        step
      end
      wrapup
    end
  end
end

```

Методологическое замечание: компонент `step` является отложенным, но `setup` и `wrapup` являются эффективными процедурами, которые по определению ничего не делают. Так можно заставить каждого эффективного потомка обеспечить собственную реализацию основного действия процесса `step`, не беспокоясь об инициализации и завершении, если на этих этапах не требуется специальных действий. При проектировании отложенных классов выбор между отложенной версией и пустой эффективной версией приходится делать регулярно. Ошибки не страшны, поскольку в худшем случае потребуется выполнить больше работы по эффективизации или переопределению у потомков.

Используя данный образец, можно определить специальный класс, охватывающий принтеры:

```

indexing
  description: "Принтеры, выполняющие в каждый момент одно задание"
  note: "Пересмотренная версия, основанная на классе PROCESS"
class PRINTER inherit

```

```

PROCESS
    rename over as stop_requested end
feature -- Status report
    stop_requested: BOOLEAN
        -- Является ли следующее задание в очереди запросом на
        -- завершение работы?
    oldest: JOB is
        -- Первое задание в очереди
        do ... end
feature -- Basic operations
    step is
        -- Обработка одного задания
        do
            wait_for_job; print (oldest); remove_oldest
        end
    wait_for_job is
        -- Ждать появления заданий в очереди
        do
            ...
        ensure
            oldest /= Void
        end
    remove_oldest is
        -- Удалить первое задание из очереди
        require
            oldest /= Void
        do
            if oldest.is_stop_request then stop_requested := True end
            "Удалить первое задание из очереди"
        end
    print (j: JOB) is
        -- Печатать j, если это не запрос на остановку
        require
            j /= Void
        do
            if not j.is_stop_request then "Печатать текст, связанный с j"
        end
    end
end

```

Этот класс предполагает, что запрос на остановку принтера посыпается как специальное задание на печать *j*, для которого выполнено условие *j.is_stop_request*. (Было бы лучше устраниТЬ проверку условия в *print* и *remove_oldest*, введя специальный вид задания - "запрос на остановку"; это нетрудно сделать [см. У12.1]).

Уже сейчас видны преимущества ОО-подхода. Точно так же, как переход от главной программы к классам расширил наши возможности, предоставив абстрактные объекты, не ограничивающиеся "только одним делом", рассмотрение процесса принтера как объекта, описанного некоторым классом, открывает возможность новых полезных свойств. В случае принтера можно сделать больше, чем просто выполнять обычную операцию печати, обеспечивающую *live* (которую нам, возможно, придется переименовать в *operate*, при наследовании ее из PROCESS).

Можно добавить компоненты: *perform_internal_test* (**выполнить внутренний тест**), *switch_to_Postscript_level_1* (**переключиться на уровень Postscript1**) или *set_resolution* (**установить разрешение**). Стабилизирующее влияние ОО-метода здесь так же важно, как и для последовательного ПО.

Классы, очерченные в этом разделе, показывают, как можно применять нормальные ОО-механизмы - классы, наследование, отложенные элементы, частично реализованные образцы - к реализации процессов. Нет ничего плохого в понятии процесса в контексте ОО-программирования, и на самом деле оно требуется во многих параллельных приложениях. Но, вместо включения некоторого примитивного механизма, оно просто охватывается **библиотечным классом** PROCESS, основанным на версии, приведенной выше в этом разделе или, может быть, несколькими такими классами, охватывающими различные варианты этого понятия.

Что касается новой конструкции для параллельной ОО-технологии, то она будет рассмотрена далее.

Введение параллельного выполнения

Что же, если не понятие процесса, фундаментально отличает параллельное вычисление от последовательного?

Процессоры

Для понимания специфики параллельности, полезно снова взглянуть на рисунок (он впервые появился в [лекции 5](#) курса "Основы объектно-ориентированного программирования"), который помог нам установить основы объектной технологии путем анализа трех основных ингредиентов вычисления:

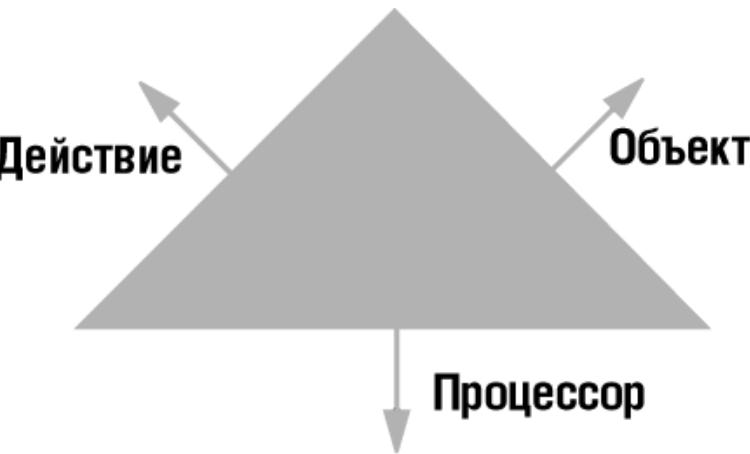


Рис. 12.2. Три силы вычисления

Выполнить программную систему - значит использовать некоторые **процессоры**, чтобы применить некоторые **действия** к некоторым объектам. В объектной технологии действия присоединяются к объектам (точнее, к типам объектов), а не наоборот.

А что же процессоры? Разумеется, нам нужен механизм для выполнения действий над объектами. Но последовательное вычисление образует лишь одну ветвь управления, для которой нужен лишь один процессор, большую часть времени присутствовавший в предыдущих лекциях неявно.

Однако в параллельном случае у нас будет несколько процессоров. Это, конечно, является самым существенным в идее параллельности и может быть даже принято за определение этого понятия. В этом и состоит основной ответ на поставленный выше вопрос: процессоры (а не процессы) будут главным новым понятием, позволяющим включить параллельность в рамки последовательных ОО-вычислений. У параллельной системы может быть любое число процессоров в отличие от последовательной системы, имеющей лишь один.

Природа процессоров

Определение: процессор

Процессор - это автономная ветвь управления, способная поддерживать последовательное выполнение инструкций одного или нескольких объектов.

Это абстрактное понятие, его не надо путать с физическими устройствами, называемыми процессорами, для которых мы далее будем использовать термин ЦПУ (**CPU**), обычно используемый в компьютерной инженерии для обозначения процессорных единиц компьютеров. "ЦПУ" - это сокращение для названия "Центральное процессорное устройство", хотя почти ничего центрального в ЦПУ нет. ЦПУ можно использовать для реализации процессора, но понятие процессора существенно более общее и абстрактное. Например, процессор может быть:

- компьютером (со своим ЦПУ) в сети;
- заданием, также называемым процессом, - поддерживается такими операционными системами, как Unix, Windows и многими другими;
- сопрограммой (сопрограммы будут более детально рассмотрены далее, они моделируют реальную параллельность, выполняясь по очереди на одном ЦПУ, после каждого прерывания каждая сопрограмма продолжает выполнение с того места, где оно остановилось);
- "потоком", который поддерживается в таких многопоточных операционных системах как Solaris, OS/2 и Windows NT.

Потоки являются минипроцессами. Настоящий процесс может включать много потоков, которыми сам управляет; операционная система (ОС) видит только процесс, а не его потоки. Обычно, потоки процесса разделяют одно и то же адресное пространство (в ОО-терминологии они имеют потенциальный доступ к одному и тому же множеству объектов), а у каждого процесса имеется свое собственное адресное пространство. Потоки можно рассматривать, как сопрограммы внутри процесса. Главное достоинство потоков в их эффективности. Создание процесса и его синхронизация с другими процессами являются дорогими операциями, требующими прямого взаимодействия с ОС (для размещения адресного пространства и кода процесса). Операции над потоками производятся более просто, не затрагивая дорогостоящих операций ОС, поэтому они выполняются в сотни и даже в тысячи раз быстрее.

Различие между процессорами и ЦПУ было ясно описано Генри Либерманом ([Lieberman 1987])(для другой модели параллельности):

Не нужно ограничивать заранее число [процессоров] и, если их оказывается больше, чем имеется реальных физических [ЦПУ] у вашего компьютера, то они автоматически будут разделять время. Таким образом, пользователь может считать, что ресурс процессоров у него практически бесконечен.

Чтобы не было неверного толкования, пожалуйста, запомните, что в этой лекции "процессоры" означают виртуальные потоки управления: при ссылках на физические устройства для вычислений будет использоваться термин ЦПУ.

Раньше или позже потребуется назначать вычислительные ресурсы процессорам. Это отображение будет представлено с помощью "файла управления параллелизмом" ("Concurrency Control File"), описываемого ниже, или соответствующих библиотечных средств.

Операции с объектом

Каждый компонент должен быть обработан (выполнен) некоторым процессором. Вообщe, каждый объект O2 **обрабатывается** некоторым процессором - его **обрабочиком**, обработчик ответственен за выполнение всех вызовов компонентов O2 (т. е. всех

вызовов вида `x.f(a)`, где `x` присоединен к `O2`).

Можно пойти дальше и решить, что обработчик связывается с объектом во время его создания и остается неизменным во время всей жизни объекта. Это предположение поможет получить простой механизм. На первый взгляд оно может показаться слишком жестким, так как некоторые распределенные системы должны поддерживать **миграцию объектов** по сети. Но с этой трудностью можно справиться двумя способами:

- позволив переназначать процессору выполняющее его ЦПУ (при таком подходе все объекты, обрабатываемые некоторым процессором, будут мигрировать вместе);
- трактуя миграцию объекта как создание нового объекта.

Дуальная семантика вызовов

При наличии нескольких процессоров мы сталкиваемся с необходимостью пересмотра обычной семантики основной операции ОО-вычисления - вызова компонента, имеющего один из видов:

```
x.f(a)      -- если f - команда  
y := x.f(a)  -- если f - запрос
```

Пусть, как и раньше, `O2` - объект, присоединенный в момент вызова к `x`, а `O1` - объект, от имени которого выполняется вызов. (Иными словами, команда любого указанного вида является частью подпрограммы, имеющей цель `O1`).

Мы привыкли понимать действие вызова как выполнение тела `f`, примененного к `O2` с использованием `a` в качестве аргумента и возвратом некоторого результата в случае запроса. Если такой вызов является частью последовательности инструкций:

```
... previous_instruction; x.f(a); next_instruction; ...
```

(или ее эквивалента в случае запроса), то выполнение `next_instruction` не начнется до того, как завершится вызов `f`.

В случае нескольких процессоров дело обстоит иначе. Главная цель параллельной архитектуры состоит в том, чтобы позволить вычислению клиента продолжаться, не ожидая, когда поставщик завершит свою работу, если эта работа выполняется другим процессором. В приведенном в начале лекции примере с принтером приложение клиента захочет послать запрос на печать ("задание") и далее продолжить работу в соответствии со своим планом.

Поэтому вместо одной семантики вызова у нас появляются две:

- Если у `O1` и `O2` один и тот же обработчик, то всякая следующая операция `O1` (`next_instruction`) должна ждать завершения вызова. Такие вызовы называются **синхронными**.
- Если `O1` и `O2` обрабатываются разными процессорами, то операции `O1` могут продолжаться сразу после того, как он инициирует вызов `O2`. Такие вызовы называются **асинхронными**.

Асинхронный случай особенно интересен для выполнения команды, так как результаты вызова `O2` могут вовсе не понадобиться или понадобиться оставшейся ее части гораздо позже. `O1` может просто отвечать за запуск одного или нескольких параллельных вычислений и за их завершение. В случае запроса результат, конечно, нужен, например, выше его значение присваивается `y`, но ниже будет объяснено, как можно продолжать параллельную работу и в этом случае.

Сепаратные сущности

Общее правило разработки ПО заключается в том, что семантическое различие всегда должно отражаться на различии текстов программ.

Сейчас, когда у нас появилось два варианта семантики вызова, нужно сделать так, чтобы в тексте программы можно было однозначно указать, какой из них имеется в виду. Ответ определяется тем, совпадает ли обработчик (процессор) цели вызова `O2` с обработчиком инициатора вызова `O1`. Поэтому нужно маркировать не вызов, а сущность `x`, обозначающую целевой объект. В соответствии с выработанной в предыдущих лекциях политикой статической проверки типов соответствующая метка должна появиться в объявлении `x`.

Это рассуждение приводит к тому, что для поддержки параллельности достаточно одного расширения нотации. Наряду с обычным объявлением:

```
x: SOME_TYPE
```

мы будем использовать объявление вида:

```
x: separate SOME_TYPE
```

для указания того, что `x` может присоединяться только к объектам, обрабатываемым специальным процессором. Если класс предназначен только для объявления сепаратных сущностей, то его можно объявить как:

```
separate class X ... Остальное как обычно ...  
вместо обычных объявлений class X ... или deferred class X ...
```

Это соглашение аналогично тому, что можно объявить `у` как сущность типа `expanded T` или, что эквивалентно, как сущность типа `T`, если `T` - это класс, объявленный как `expanded class T`. Три возможности - развернутый (`expanded`), отложенный (`deferred`), сепаратный (`separate`) - являются взаимно исключающими, только одно из этих квалифицирующих слов может стоять перед словом `class`.

Это даже поразительно, что достаточно добавить одно ключевое слово для превращения последовательной ОО-нотации в систему обозначений, поддерживающую параллельные вычисления.

Уточним терминологию. Слово "сепаратный" ("separate") можно применять к различным элементам, как статическим

(появляющимся в тексте программы), так и динамическим (существующим во время выполнения). Статически: сепаратный класс - это класс, объявленный как `separate class`; сепаратный тип основывается на сепаратном классе; сепаратная сущность это сущность сепаратного типа или сущность, объявленная как `separate T` для некоторого `T`; `x.f (...)` - это сепаратный вызов, если его цель `x` является сепаратной сущностью. Динамики: значение сепаратной сущности является сепаратной ссылкой; если она не пуста, то присоединяется к объекту, обрабатываемому отдельным процессором - сепаратному объекту.

Типичными примерами сепаратных классов являются:

- `BOUNDED_BUFFER` (**ОГРАНИЧЕННЫЙ_БУФЕР**) задает буфер, позволяющий параллельным компонентам обмениваться данными (некоторые компоненты - производители - помещают объекты в буфер, а другие - потребители - получают объекты из него).
- `PRINTER` (**ПРИНТЕР**), который, по-видимому, правильней называть `PRINT_CONTROLLER` (**КОНТРОЛЕР_ПЕЧАТИ**), управляет одним или несколькими принтерами. Считая контроллеры печати сепаратными объектами, приложения не должны ждать завершения заданий на печать (в отличие от ранних Макинтошей, в которых вы застревали до тех пор, пока последняя страница не выползла из принтера).
- `DATABASE` (**БАЗА ДАННЫХ**), клиентская часть которой в архитектуре клиент-сервер может служить для описания базы данных, расположенной на удаленном сервере, которому клиент может посыпать запросы по сети.
- `BROWSER_WINDOW` (**ОКНО_БРАУЗЕРА**) позволяет порождать новое окно для просмотра запрошенной страницы.

Получение сепаратных объектов

Как показывают предыдущие примеры, на практике встречаются сепаратные объекты двух видов:

- В первом случае приложение при вызове захочет порождать **новый** сепаратный объект, заняв следующий свободный процессор. (Напомним, что такой процессор всегда можно получить, так как процессоры - это не материальные ресурсы, а абстрактные устройства, и их число не ограничено). Эта ситуация типична для `BROWSER_WINDOW`: новое окно создается тогда, когда это нужно. Объекты классов `BOUNDED_BUFFER` или `PRINT_CONTROLLER` также могут создаваться при необходимости.
- Приложению может потребоваться доступ к уже существующему сепаратному объекту, обычно разделяемому многими клиентами. Это имеет место для класса `DATABASE`: приложение-клиент использует сепаратную сущность `db_server`: `separate DATABASE` для доступа к базе данных через сепаратные вызовы вида `db_server.ask_query(sql_query)`. У сервера должно быть полученное на некотором шаге извне значение указателя на базу данных `server`. Аналогичные схемы используются для доступа к существующим объектам классов `BOUNDED_BUFFER` и `PRINT_CONTROLLER`.

Скажем, что в первом случае сепаратные объекты создаются, а во втором являются внешними.

Для создания сепаратного объекта применяется обычная инструкция создания:

```
create x.make (...)
```

В дополнение к своему обычному действию по созданию и инициализации нового объекта ему назначается новый процессор. Такая инструкция называется **сепаратным созданием**:

Для получения существующего внешнего объекта, как правило, используется внешняя процедура, например:

```
server (name: STRING; ... Другие аргументы ...): separate DATABASE
```

Ее аргументы служат для идентификации запрашиваемого объекта. Такая процедура посыпает сообщение по сети и получает в ответ ссылку на объект.

Для визуализации понятия сепаратного объекта, полезно сказать о возможных реализациях. Предположим, что каждый из процессоров связан с некоторой **задачей** (процессом) операционной системы (например, Windows или Unix), имеющей свое адресное пространство; конечно, это только одна из возможных архитектур. Тогда одним из способов представления сепаратного объекта внутри задачи является использование небольшого локального объекта, называемого **заместителем** или **прокси** (*proxy*):

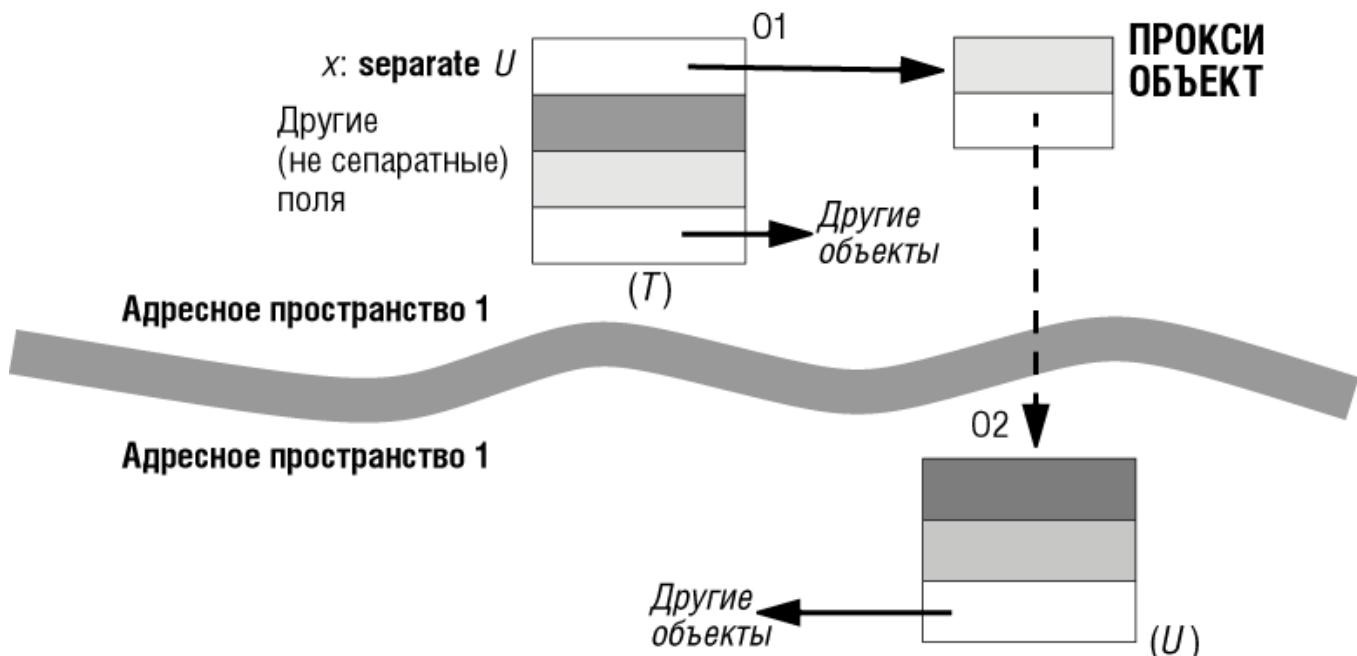


Рис. 12.3. Прокси для сепаратного объекта

На этом рисунке показан объект O1, экземпляр класса T с атрибутом x: separate U. Соответствующее поле - ссылка в O1 - концептуально привязано к объекту O2, обрабатываемому другим процессором. Фактически ссылка ведет к прокси-объекту, обрабатываемому процессором для O1. Прокси - это внутренний объект, не видимый автору параллельного приложения. Он содержит достаточно информации для идентификации O2: задачу, которая служит обработчиком O2, и адрес O2 внутри этой задачи. Все операции над x, проводимые от имени O1 или других клиентов той же задачи, будут проходить через прокси. У всякого другого процессора, также обрабатывающего объекты, содержащие ссылки на O2, будет свой собственный прокси для O2.

Подчеркнем, что это лишь один из возможных методов, а не обязательное свойство рассматриваемой модели. Задачи операционной системы с раздельными адресными пространствами являются лишь одним из способов реализации процессоров. Для потоков методы могут быть другими.

Объекты здесь и там

Некоторые люди, впервые познакомившись с понятием сепаратной сущности, выражают недовольство тем, что оно чрезсур детализировано: "Я не хочу знать, где расположен объект! Я хотел бы лишь запрашивать операцию x.f (...) , а остальное пусть делает машина - выполняет f на x , где бы x не находился".

Будучи вполне законным, такое желание не устраивает необходимость в сепаратных декларациях. Действительно, точное положение объекта часто остается деталью реализации, не влияющей на ПО. Но одно "да-нет" свойство местоположения объекта остается существенным: обрабатывается ли один объект тем же процессором, что и другой. Оно задает важное семантическое различие, поскольку определяет, будут ли вызовы объекта синхронными или асинхронными - будет ли клиент ждать их завершения или нет. Пренебрежение этим свойством в ПО было бы не удобством, а ошибкой.

Если известно, что объект сепаратный, то в большинстве случаев на функционирование использующей его программы (но не на ее эффективности) не должно сказываться, с каким процессором он связан. Он может быть связан с другим потоком того же процесса, другим процессом на том же компьютере или на другом компьютере в той же комнате, в другой комнате того же здания, другим сайтом в частной сети фирмы или узлом Интернета на другом конце мира. Но то, что он сепаратный, существенно.

Параллельная архитектура

Использование объявлений separate для ответа на важный вопрос "этот объект находится здесь или в другом месте?", оставляя возможности различных физических реализаций параллельности, предполагает двухуровневую архитектуру ([рис. 12.4](#)), аналогичную той, которая подходит и для механизмов графики (с библиотекой Vision, находящейся выше библиотек, специфических для разных платформ, см. [лекцию 14](#)).

На верхнем уровне этот механизм не зависит от платформы. Большая часть приложений, рассматриваемых в этой лекции, использует этот уровень. Для выполнения параллельного вычисления приложения просто используют механизм объявлений separate.

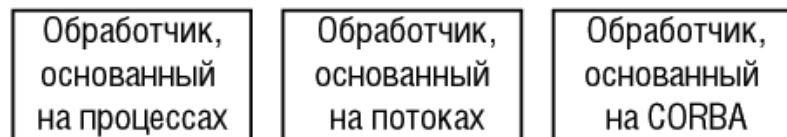


Рис. 12.4. Двухуровневая архитектура механизма параллельности

Внутренняя реализация будет опираться на некоторую конкретную параллельную архитектуру (на [рис. 12.4](#) это нижний уровень). На [рис. 12.4](#) показаны следующие возможности:

- Реализация может использовать процессы, предоставляемые операционной системой. Каждый процессор связывается с некоторым процессом. Такое решение поддерживает распределенные вычисления: процесс сепаратного объекта может находиться как на удаленной машине, так и на локальной. Для нераспределенной обработки его преимущество в том, что процессы стабильны и хорошо известны, а недостаток - в том, что оно приводит к интенсивной загрузке ЦПУ, так как и создание нового процесса, и обмен информацией между процессами являются дорогими операциями.
- Реализация может использовать потоки. Как уже отмечалось, потоки - это облегченная версия процессов, минимизирующая стоимость создания и переключения контекстов. Однако потоки должны располагаться на одной машине.
- Возможна также реализация, использующая механизм распределения CORBA в качестве физического уровня для обмена объектами в сети.
- Другими возможными механизмами являются PVM (Parallel Virtual Machine) (параллельная виртуальная машина - Parallel Virtual Machine), язык параллельного программирования Linda, потоки Java...

Как всегда, в случае двухуровневых архитектур соответствие между конструкциями верхнего уровня и реальным распределением на уровне платформы (**описателем (handle)** в терминах предыдущей лекции) в большинстве случаев устанавливается автоматически, так что разработчики приложений будут видеть только верхний уровень. Но при необходимости и готовности отказаться от платформенной независимости им также должны быть доступны механизмы нижнего уровня.

Распределение процессоров: файл управления параллелизмом (Concurrency Control File)

Если в программе не задаются физические ЦПУ, то их спецификация должна быть помещена в каком-то другом месте. Вот один из способов позаботиться об этом. Подчеркнем, что это лишь одно из возможных решений, не претендующее на фундаментальность; точный формат здесь несущественен, но любой способ конфигурирования будет так или иначе предоставлять одну и ту же информацию.

В качестве примера мы выбрали "Файл управления параллелизмом" (ФУП) ("Concurrency Control File" (CCF)), описывающий доступные программам ресурсы параллельных вычислений. ФУПы по целям и по виду похожи на файлы Ace, используемые для управления сборкой системы ([лекция 7](#) курса "Основы объектно-ориентированного программирования"). Типичный ФУП выглядит так:

```
creation
  local_nodes:
    system
      "pushkin" (2): "c:\system1\appl.exe"
      "akhmatova" (4): "/home/users/syst1"
      Current: "c:\system1\appl2.exe"
    end
  remote_nodes:
    system
      "lermontov": "c:\system1\appl.exe"
      "tiutchev" (2): "/usr/bin/syst2"
    end
  end
external
  Ingres_handler: "mandelstam" port 9000
  ATM_handler: "pasternak" port 8001
end
default
  port: 8001; instance: 10
end
```

Для всех рассматриваемых свойств имеются значения по умолчанию, поэтому ни одна из трех частей (creation, external, default) не является обязательной, как и сам ФУП.

Часть creation определяет, какие ЦПУ используются для сепаратного создания (инструкций вида `create x.make (...)` для сепаратной `x`). В примере используются две группы ЦПУ: `local_nodes`, предположительно включающие локальные машины, и `remote_nodes`. Программа может выбрать группу ЦПУ с помощью вызова вида:

```
set_cpu_group ("local_nodes")
```

указывающего, что последующие операции сепаратного создания будут использовать ЦПУ группы `local_nodes` до появления следующего вызова `set_cpu_group`. Эта процедура описана в классе CONCURRENCY, предоставляющем средства для механизма управления, который мы подробней рассмотрим ниже.

Соответствующие элементы ФУП указывают, какие ЦПУ следует использовать для группы `local_nodes`: первые два объекта будут созданы на машине `pushkin`, следующие четыре - на машине `akhmatova`, а следующие десять - на текущей машине (т. е. на той, на которой выполняются инструкции создания). После этого схема распределения будет повторяться - два объекта на машине `pushkin` и т. д. Если число процессоров отсутствует, как у `Current` в примере, то оно извлекается из пункта `instance` в части `default` (здесь оно равно 10), а если такого пункта нет, то берется равным 1. Система, используемая для создания каждого экземпляра, указывается для каждого элемента, например, для `pushkin` это будет `c:\system1\appl.exe` (очевидно, машина работает под Windows или OS/2).

В этом примере все процессоры привязаны к процессам. ФУП также поддерживает назначение процессоров потокам (для описателей, основанных на потоках) и другие механизмы параллельности, хотя мы здесь не будем входить в их детали.

Часть `external` указывает, где располагаются существующие внешние сепаратные объекты. ФУП ссылается на эти объекты через их абстрактные имена, в примере `Ingres_handler` и `ATM_handler`, используемые в качестве аргументов функций при установлении связи с ними. Например, для функции `server` с аргументами:

```
server (name: STRING; ... Другие аргументы ...): separate DATABASE
```

вызов вида `server ("Ingres_handler", ...)` даст сепаратный объект, обозначающий сервер базы данных `Ingres`. ФУП указывает, что соответствующий объект расположен на машине `mandelstam` и доступен через порт 9000. Если порт явно не задан, то его значение извлекается из части `defaults`, а если и там его нет, то используется некоторое универсальное предопределенное значение.

ФУП существует отдельно от программ. Можно откомпилировать параллельное или распределенное приложение без всякой ссылки на специфическое оборудование и архитектуру сети, а затем во время выполнения каждый отдельный компонент этого приложения будет использовать свой ФУП для связи с другими существующими компонентами (часть `external`) и для создания новых компонентов (часть `creation`).

Этот набросок соглашений, принятых в ФУП, показал, как можно отобразить абстрактные понятия параллельных ОО-вычислений - процессоры и сепаратные объекты (внешние и создаваемые) - на физические ресурсы. Как уже отмечалось, эти соглашения являются только примером того, как это можно сделать, но не являются частью базового механизма параллельности. Они показывают возможность отделения архитектуры параллельной системы от архитектуры параллельного оборудования.

Библиотечные механизмы

При подходах типа ФУП программа приложения может быть никак не связана с конкретной физической параллельной архитектурой. Однако некоторым разработчикам ПО требуется более тонкий контроль, осуществляемый приложением (ценой возможного

удорожания при динамической реконфигурации). Некоторые функции ФУП должны быть доступны непосредственно самому приложению, позволяя ему, например, выбрать для некоторого процессора определенный процесс или поток. Они будут доступны через библиотеки, как часть двухуровневой параллельной архитектуры; это не приведет ни к каким трудным проблемам. Позже в этой лекции мы еще столкнемся с необходимостью дополнительных библиотечных механизмов.

С другой стороны, некоторым приложениям может потребоваться неограниченное реконфигурирование во время исполнения. Тогда недостаточно иметь возможность прочесть ФУП или аналогичную информацию о конфигурации во время старта, а затем ее придерживаться. Но также плохо перечитывать информацию о конфигурации перед выполнением каждой операции, поскольку это убило бы эффективность. Снова разумное решение - использование библиотечного механизма: должна быть доступна процедура для динамического чтения информации о конфигурации, позволяющая приложению адаптироваться к новой конфигурации тогда (и только тогда), когда оно готово это сделать.

Правила обоснования корректности: разоблачение предателей

Так как для сепаратных и несепаратных объектов семантика вызовов различна, то важно гарантировать, что несепаратная сущность (объявленная как x : T для несепаратного T) никогда не будет присоединена к сепаратному объекту. В противном случае, вызов $x.f(a)$ был бы неверно понят - в том числе и компилятором - как синхронный, в то время как присоединенный объект, на самом деле, является сепаратным и требует асинхронной обработки. Такая ссылка, ошибочно объявленная несепаратной, но хранящая верность другой стороне, будет называться **ссылкой-предателем (traitor)**. Нам нужно простое правило обоснования корректности, чтобы гарантировать отсутствие предателей в ПО, а именно, что каждый представитель или лоббист сепаратной стороны надлежащим образом зарегистрирован как таковой соответствующими властями.

У этого правила будут четыре части. Первая часть устраниет риск создания предателей посредством присоединения, т. е. путем присваивания или передачи аргументов:

Правило (1) корректности сепаратности

Если источник присоединения (в инструкции присваивания или при передаче аргументов) является сепаратным, то его целевая сущность также должна быть сепаратной.

Присоединение цели x к источнику у является либо присваиванием $x := u$, либо вызовом $f(\dots, u, \dots)$, в котором u - это фактический аргумент, соответствующий x . Такое присоединение, в котором u сепаратная, а x нет, делает x предателем, поскольку сущность x может быть использована для доступа к сепаратному объекту (объекту, присоединенному к u) под несепаратным именем, как если бы он был локальным объектом с синхронным вызовом. Приведенное правило это запрещает.

Отметим, что синтаксически x является сущностью, а u может быть произвольным выражением. Поэтому нам следует определить понятие "сепаратного выражения". Простое выражение - это сущность; более сложные выражения являются вызовами функций (напомним, в частности, что инфиксное выражение вида $a + b$ формально рассматривается как вызов: нечто вроде $a.\text{plus}(b)$). Отсюда сразу получаем определение: выражение является сепаратным, если оно является сепаратной сущностью или сепаратным вызовом.

Как станет ясно из последующего обсуждения, присоединение несепаратного источника к сепаратной цели безвредно, хотя, как правило, не очень полезно.

Нам нужно еще дополнительное правило, охватывающее случай, когда клиент передает сепаратному поставщику ссылку на локальный объект. Пусть имеется сепаратный вызов:

$x.f(a)$,

в котором a типа T не является сепаратной, а x является. Объявление процедуры f в классе, порождающем x , будет иметь вид:

$f(u:\text{SOME_TYPE})$

а тип T сущности a должен быть совместен с SOME_TYPE . Но этого недостаточно! Глядя с позиций поставщика (т. е. обработчика x) объект O_1 , присоединенный к a , расположен на другой стороне - имеет другого обработчика, поэтому, если не объявить соответствующий формальный аргумент и как сепаратный, он станет предателем, так как даст доступ к сепаратному объекту так, как будто он несепаратный:

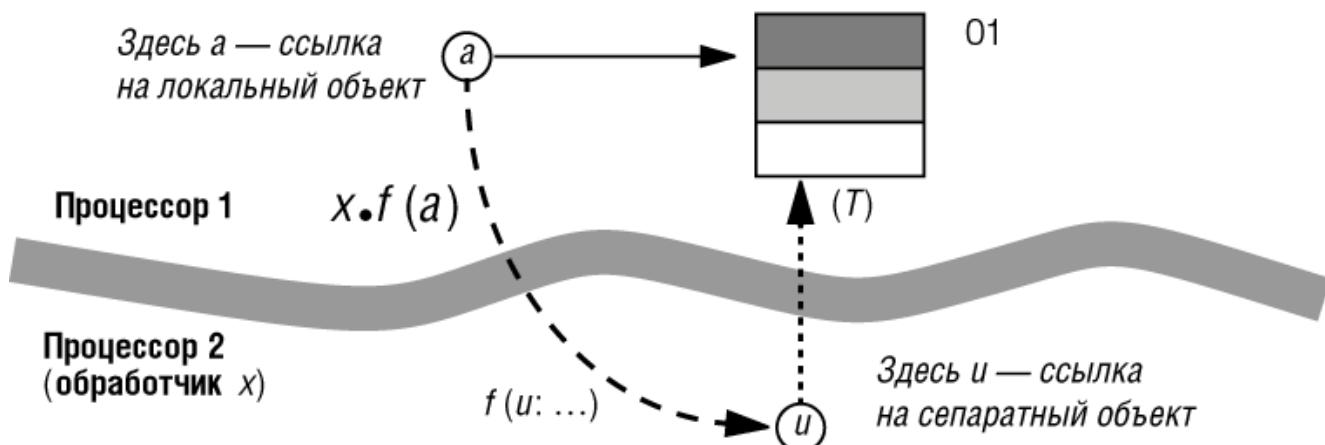


Рис. 12.5. Передача ссылки в качестве аргумента сепаратному вызову

Таким образом, SOME_TYPE должен быть сепаратным, например, это может быть `separate T`. Отсюда получаем второе правило корректности:

Правило (2) корректности сепаратности

Если фактический аргумент сепаратного вызова имеет тип ссылки, то соответствующий формальный аргумент должен быть объявлен как сепаратный.

Здесь рассматриваются только ссылочные аргументы. Случай развернутых типов, включая базовые типы, такие как INTEGER, рассматривается далее.

Иллюстрируя эту технику, рассмотрим объект, порождающий большое число сепаратных объектов, наделяя их возможностью обращаться в дальнейшем к его ресурсам, т. е. говоря им: "Вот вам моя визитная карточка, звоните мне, если потребуется". Типичным примером будет ядро операционной системы, которое создает сепаратные объекты, оставаясь готовым выполнять операции по их запросам. Создание объектов будет иметь вид:

```
create subsystem.make (Current, ... Другие аргументы ...),
```

где Current - это визитная карточка, позволяющая subsystem запомнить своего создателя (progenitor) и в случае необходимости попросить у него помочь. Поскольку Current - это ссылка, то соответствующий формальный аргумент в make должен быть объявлен как сепаратный. Чаще всего make будет иметь вид:

```
make (p: separate PROGENITOR_TYPE; ... Другие аргументы ...) is
do
    progenitor := p
    ... Остальные операции инициализации ...
end
```

при котором значение аргумента создателя запоминается в атрибуте progenitor объемлющего класса. Второе правило корректности сепаратности требует, чтобы p была объявлена как сепаратная, а первое правило требует того же от атрибута progenitor. Тогда вызовы ресурсов создателя вида progenitor.some_resource (...) будут корректно трактоваться как сепаратные.

Аналогичное правило нужно и для результатов функций.

Правило (3) корректности сепаратности

Если источник присоединения является результатом вызова функции, возвращающей тип ссылки, то цель должна быть объявлена как сепаратная.

Поскольку последние два правила относятся к фактическим аргументам и результатам только ссылочных типов, то нужно еще одно правило для развернутых типов.

Правило (4) корректности сепаратности

Если фактический аргумент или результат сепаратного вызова имеет развернутый тип, то его базовый класс не может содержать непосредственно или опосредовано никакой несепаратный атрибут ссылочного типа.

Иными словами, единственные развернутые значения, которые можно передавать в сепаратный вызов - это "полностью развернутые" объекты, не содержащие никаких ссылок на другие объекты. Иначе можно снова попасть в неприятную ситуацию с предателем, так как присоединение развернутого значения приводит к копированию объекта.

[Рис. 12.6](#) иллюстрирует случай, когда формальный аргумент и является развернутым. Тогда при присоединении поля объекта O1 просто копируются в соответствующие поля объекта O'1, присоединенного к и (см. [лекцию 8](#) курса "Основы объектно-ориентированного программирования"). Если разрешить O1 содержать ссылку, то это приведет к полю-предателю в O'1. Та же проблема возникнет, если в O1 будет подобъект со ссылкой; это отмечено в правиле фразой "непосредственно или опосредовано".

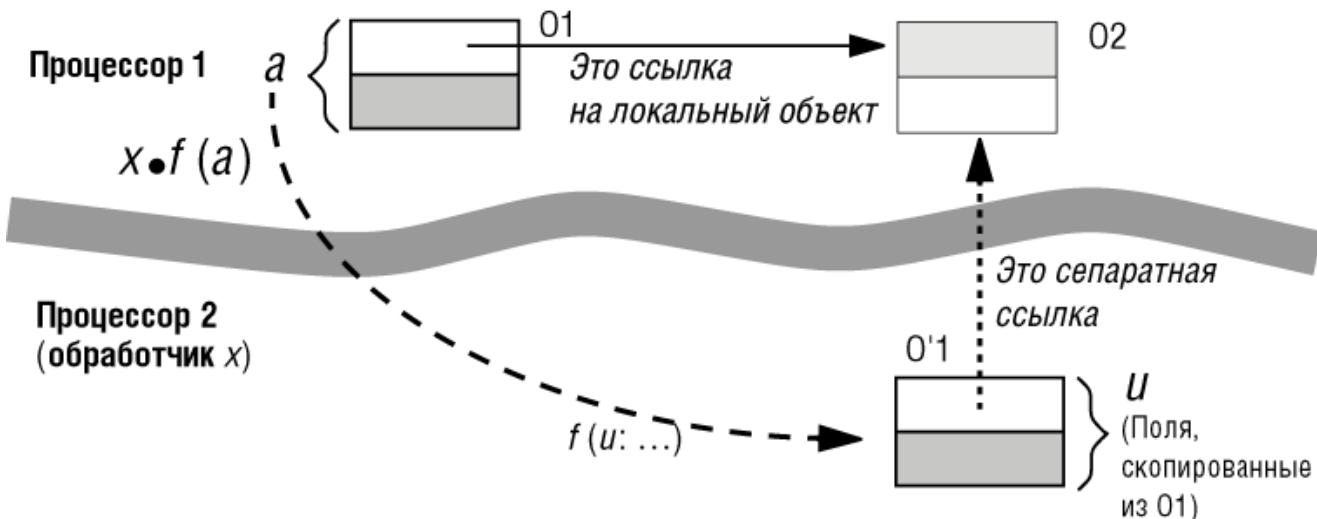


Рис. 12.6. Передача объекта со ссылками сепаратному вызову

Если формальный аргумент и является ссылкой, то присоединение является клоном; вызов будет создавать новый объект O'1, как показано на последнем рисунке, и присоединять к нему ссылку и. В этом случае можно предложить перед вызовом явно создавать клон на стороне клиента:

```
a: expanded SOME_TYPE; a1: SOME_TYPE
```

```
... a1 := a; -- Это клонирует объект и присоединяет a1 к клону
x.f (a1)
```

Согласно второму правилу корректности формальный аргумент и должен иметь тип, согласованный с типом сепаратной ссылки `separate SOME_TYPE`. Вызов в последней строке делает и сепаратной ссылкой, присоединенной ко вновь созданному клону на стороне клиента.

Импорт структур объекта

Одно из следствий правил корректности сепарации состоит в том, что для получения объекта, обрабатываемого другим процессором, нельзя использовать функцию `clone` (из универсального класса `ANY`). Эта функция была объявлена как:

```
clone (other: GENERAL): like other is
    -- Новый объект с полями, идентичными полям other
...
```

Поэтому попытка использовать `у := clone (x)` для сепарированного `x` нарушила бы часть 1-го правила: `x`, которая является сепарированной, не соответствует несепарированной `other`. Это то, чего мы и добивались. Сепарированный объект, обрабатываемый на машине во Владивостоке, может содержать (несепарированные) ссылки на объекты, находящиеся во Владивостоке. Если же его клонировать в Канзас Сити, то в результирующем объекте окажутся предатели - ссылки на сепарированные объекты, хотя в породившем их классе соответствующие атрибуты сепарированными не объявлялись.

Следующая функция из класса `GENERAL` позволяет клонировать структуру сепарированного объекта без создания предателей:

```
deep_import (other: separate GENERAL): GENERAL is
    -- Новый объект с полями, идентичными полям other
...
```

Результатом будет структура несепарированного объекта, рекурсивно скопированная с сепарированной структуры, начиная с объекта `other`. По только что объясненной причине операция поверхностного импорта может приводить к предателям, поэтому нам нужен эквивалент функции `deep_clone` (см. [лекцию 8](#) курса "Основы объектно-ориентированного программирования"), применяемый к сепарированному объекту. Таковым является функция `deep_import`. Она будет создавать копию внутренней структуры, делая все встречающиеся при этом копии объектов несепарированными. (Конечно, она может содержать сепарированные ссылки, если в исходной структуре были ссылки на объекты, обрабатываемые другими процессорами).

Для разработчиков распределенных систем функция `deep_import` является удобным и мощным механизмом, с помощью которого можно передавать по сети сколь угодно большие структуры объектов без необходимости писать какие-либо специальные программы, гарантирующие точное дублирование.

Вопросы синхронизации

У нас имеется базовый механизм для начала параллельных вычислений (сепарированное создание) и для запроса операций в этих вычислениях (обычный механизм вызова компонентов). Всякое параллельное вычисление, ОО или не ОО, должно также предоставлять возможности для **синхронизации** параллельных вычислений, т. е. для определения временных зависимостей между ними.

Если вы знакомы с параллелизмом, то, возможно, будете удивлены заявлением, что одного языкового механизма - сепарированных объявлений - достаточно, чтобы включить полную поддержку параллелизма в ОО-подход. Нужен ли на самом деле специальный механизм синхронизации? Оказывается, нет. Основные ОО-конструкции достаточны, чтобы покрыть большую часть потребностей в синхронизации при условии, что определения их семантики будут приспособлены для применения к сепарированным элементам. Это является еще одним свидетельством силы ОО-метода, легко и элегантно адаптирующегося к параллельным вычислениям.

Синхронизация versus взаимодействия

Для понимания того, как следует поддерживать синхронизацию в ОО-параллелизме, полезно начать с обзора не ОО-решений. Процессы (единицы параллельности в большинстве этих решений) нуждаются в механизмах двух видов:

- Синхронизации, обеспечивающей временные ограничения. Типичное ограничение утверждает, что некоторая операция одного процесса (например доступ к элементу базы данных), может выполняться только после некоторой операции другого процесса (инициализации этого элемента).
- Взаимодействия, позволяющего процессам обмениваться информацией, представляющей в ОО-случае объекты (как частный случай, простые значения) или объектные структуры.

Одни подходы к параллельности основываются на механизме синхронизации, используя для коммуникации обычные непараллельные методы такие, как передача аргументов. Другие рассматривают в качестве основы взаимодействие, добиваясь затем синхронизации. Можно говорить о **механизмах, основанных на синхронизации**, и о **механизмах, основанных на взаимодействии**.

Механизмы, основанные на синхронизации

Наиболее известный и элементарный механизм, основанный на синхронизации, - это семафор, средство блокировки для управления распределенными ресурсами. Семафор - это объект с двумя операциями: `reserve` (занять) и `free` (освободить), традиционно обозначаемыми `P` и `V`, но мы предпочитаем использовать содержательные имена. В каждый момент времени семафор либо занят некоторым клиентом, либо свободен. Если он свободен и клиент выполняет `reserve`, то семафор занимается этим клиентом. Если клиент, занявший семафор, выполняет `free`, то семафор становится свободным. Если семафор занят некоторым клиентом, а новый клиент выполняет `reserve`, то он будет ждать, пока семафор освободится. Эта спецификация отражена в следующей таблице:

Таблица 30.1. Операции семафора

Операция	Состояние		
	Свободен	Занят мной	Занят кем-то другим
reserve	Занимается мной		Я жду
free		Становится свободным	

Предполагается, что события, соответствующие пустым клеткам таблицы, не происходят, они могут рассматриваться как ошибки или как не вызывающие изменений.

Правила захвата семафора после его освобождения, когда этого события ожидают несколько клиентов, могут быть частью спецификации семафора или вовсе не задаваться. (Обычно для клиентов выполняется свойство **равнодоступности (fairness)**, гарантирующее, что никто не будет ожидать бесконечно, если получающий доступ к семафору обязательно его освобождает).

Это описание относится к бинарным семафорам. Целочисленный вариант допускает наличие одновременного обслуживания до $n > 0$ клиентов для некоторого целого $n > 0$.

Хотя семафоры используются во многих практических приложениях, они сейчас считаются средствами низкого уровня для построения больших надежных систем. Но они дают хорошую отправную точку для обсуждения усовершенствованных методов.

Критические интервалы (critical regions) представляют более абстрактный подход. Критический интервал - это последовательность инструкций, которая может выполняться в каждый момент не более чем одним клиентом. Для обеспечения исключительного доступа к объекту можно написать нечто вроде:

```
hold a then ... Операции с полями а ...end.
```

Здесь критический интервал выделен ключевыми словами `then...` `end`. Только один клиент может выполнять критический интервал в каждый момент, другие клиенты, выполняющие `hold`, будут ждать.

Большей части приложений требуется общий вариант - **условный критический интервал (conditional critical region)**, в котором выполнение критического интервала подчинено некоторому логическому условию. Рассмотрим некоторый буфер, разделяемый производителем, который может только писать в буфер, если тот не полон, и потребителем, который может только читать из буфера, если тот не пуст. Они могут использовать две соответствующие схемы:

```
hold buffer when not buffer.full then "Записать в буфер, сделав его непустым" end
hold buffer when not buffer.empty then "Прочесть из буфера, сделав его неполным" end
```

Такое взаимодействие между входным и выходным условиями требует введения утверждений и придания им важной роли в синхронизации. Эта идея будет развита далее в этой лекции.

Другим хорошо известным механизмом синхронизации, объединяющим понятие критического интервала с модульной структурой некоторых современных языков программирования, являются **мониторы (monitor)**. Монитор - это программный модуль, похожий на модули Modula или Ada. Основной механизм синхронизации прост: взаимное исключение достигается на уровне процедур. В каждый момент времени только один клиент может выполнять процедуру монитора.

Интересно также понятие "**путевого выражения**" (**path expression**). Путевое выражение задает возможный порядок выполнения процессов. Например, выражение:

```
init ; (reader* | writer)+ ; finish
```

описывает следующее поведение: вначале активизируется процесс `init`, затем активным может стать один процесс `writer` или произвольное число процессов `reader`; это состояние может повторяться конечное число раз, затем приходит черед заключительного процесса `finish`. В записи выражения звездочка (*) означает произвольное число параллельных экземпляров, точка с запятой (;) - последовательное применение, символ черты (|) - "или-или", (+) - любое число последовательных повторений. Часто цитируемый аргумент в пользу путевого выражения заключается в том, что они задают процессы и синхронизацию раздельно, устранивая помехи, возникающие между описанием отдельных алгоритмических задач и составлением расписания их выполнения.

Еще одна категория методов для описания синхронизации основана на анализе множества **состояний**, через которое может проходить система или ее компонент, и переходов между этими состояниями. В частности, **сети Петри** основаны на графическом описании таких переходов. Хотя такие методы достаточно понятны для простых устройств, они быстро приводят к комбинаторному взрыву числа состояний и переходов, и трудны для иерархического проектирования (независимой спецификации подсистем, а затем рекурсивного вложения их спецификаций в большие системы). Поэтому они вряд ли применимы к большим эволюционирующими программным системам.

Механизмы, основанные на взаимодействии

Начиная с книги Хоара "Взаимодействующие последовательные процессы" (ВПП), появившейся в конце 70-х, большинство работ по не ОО-параллельности сосредоточено на подходах, основанных на взаимодействии.

Причину этого легко понять. Если решена проблема синхронизации, то все равно нужно искать способы взаимодействия параллельных компонентов. Но если разработан хороший механизм взаимодействия, то, скорее всего, решены и вопросы синхронизации: так как два компонента не могут взаимодействовать, если отправитель не готов послать, а получатель не готов принять информацию. Взаимодействие влечет синхронизацию. Чистая синхронизация может выглядеть как крайний случай взаимодействия с пустым сообщением. Общий механизм взаимодействия обеспечит необходимую синхронизацию.

Подход ВПП основан на взгляде: "Я взаимодействую, следовательно, я синхронизирую". Исходным пунктом является обобщение фундаментального понятия "вход-выход": процесс получает информацию `v` по некоторому "каналу" с помощью конструкции `c ? v`; он посылает информацию в канал с помощью конструкции `c ! v`. Передача информации по каналу и получение информации из него являются двумя примерами возможных событий.

Для большей гибкости ВПП использует понятие недетерминированного ожидания, представляемого символом |, позволяющее процессу ожидать нескольких возможных событий и выполнять действие, связанное с первым из случившихся. Рассмотрим систему, позволяющую клиентам банка получать сведения о состоянии их счетов и производить переводы средств на них, а менеджеру банка проверять, что происходит:

```
(balance_enquiry ? customer R
  (ask_password.customer ? password R
    (password_valid R (balance_out.customer ! balance)
      | (password_invalid R (denial.customer ! denial_message)))
  | transfer_request ? customer R ...
  | control_operation ? manager R ...)
```

Система находится в ожидании одного из трех возможных входных событий: запроса о балансе (balance_enquiry), требования о переводе (transfer_request), контроля операции (control_operation). Событие, произошедшее первым, запустит на выполнение поведение, описываемое с использованием тех же механизмов (справа от соответствующей стрелки).

В примере часть справа от стрелки заполнена только для первого события: после получения запроса о балансе от некоторого клиента, ему посыпается сообщение запрос пароля (ask_password), в результате которого ожидаем получить пароль (password). Затем проверяется правильность пароля и клиенту посыпается одно из двух сообщений: balance_out с балансом счета (balance) в качестве аргумента или отказ (denial).

После завершения обработки события система возвращается в состояние ожидания следующего входного события.

Первоначальная версия ВПП в значительной степени повлияла на механизм параллельности в языке Ada, чьи "задачи" являются процессами, способными ожидать несколько возможных "входов" посредством команды "принять" (см. [лекцию 15](#)). Язык Occam, непосредственно реализующий ВПП, является основополагающим программным средством для транспьютеров (transputer) семейства микропроцессоров, разработанных фирмой Inmos (сейчас SGS-Thomson) для создания высокопараллельных архитектур.

Синхронизация параллельных ОО-вычислений

Многие из только что рассмотренных идей помогут выбрать правильный подход к параллельности в ОО-контексте. В полученном решении будут видны понятия, пришедшие из ВПП, из мониторов и условных критических интервалов.

Упор ВПП на взаимодействии представляется нам правильным, поскольку главный метод нашей модели вычислений - вызов компонента с аргументами для некоторого объекта - является механизмом взаимодействия. Но есть и другая причина предпочтеть решение, основанное на взаимодействии: механизм, основанный на синхронизации, может конфликтовать с наследственностью.

Этот конфликт наиболее очевиден при рассмотрении путевых выражений. Идея использования выражений на путях привлекла многих исследователей ОО-параллельности. Она дает возможность разделить реальную обработку, заданную компонентами класса, и ограничения синхронизации, задаваемые путевыми выражениями. При этом параллельность не затрагивала бы чисто вычислительные аспекты ПО. Например, если у класса BUFFER имеются компоненты `remove` (удаление самого старого элемента буфера) и `put` (добавить элемент), то можно выразить синхронизацию с помощью ограничений, используя обозначения в духе путевых выражений:

```
empty: {put}
partial: {put, remove}
full: {remove}
```

Эти обозначения и пример взяты из [Matusoka 1993], где введен термин "аномалия наследования". Более подробный пример смотри в У12.3.

Здесь перечислены три возможных состояния и для каждого из них указаны допустимые операции. Но предположим далее, что потомок класса NEW_BUFFER задал дополнительный компонент `remove_two`, удаляющий из буфера два элемента одновременно (если размер буфера не менее трех). В этом случае придется почти полностью изменить множество состояний:

```
empty: {put}
partial_one: {put, remove}      -- Состояние, в котором в буфере ровно один
                                -- элемент
partial_two_or_more: {put, remove, remove_two}
full: {remove, remove_two}
```

и если в процедурах определяются вычисляемые ими состояния, то их необходимо переопределять при переходе от BUFFER к NEW_BUFFER, что противоречит самой сути наследования.

Эта и другие проблемы, выявленные исследователями, получили название **аномалии наследования (inheritance anomaly)** и привели разработчиков параллельных ОО-языков к подозрительному отношению к наследованию. Например, из первых версий параллельного ОО-языка POOL наследование было исключено.

Заботы о проблеме "аномалии наследования" породили обильную литературу с предложениями ее решения, которые по большей части сводились к уменьшению объема переопределений.

Однако при более внимательном рассмотрении оказывается, что эта проблема связана не с наследованием или с конфликтом между наследованием и параллелизмом, а с идеей отделения программ от определения ограничений синхронизации.

Для читателя этой книги, знакомого с принципами проектирования по контракту, методы, использующие явные состояния и список компонентов, применимых в каждом из них, выглядят чересчур низкоуровневыми. Спецификации классов BUFFER и NEW_BUFFER следует задавать с помощью предусловий: `put` требует выполнения условия `require not full, remove_two - require count >= 2` и т. д. Такие более компактные и более абстрактные спецификации проще объяснять, адаптировать и связывать с пожеланиями клиентов (изменение предусловия одной процедуры не влияет на остальные процедуры). Методы, основанные на состояниях, налагают больше ограничений и подвержены ошибкам. Они также увеличивают риск комбинаторного взрыва,

отмеченный выше для сетей Петри и других моделей, использующих состояния: в приведенных выше элементарных примерах число состояний уже равно три в одном случае и четыре - в другом, а в более сложных системах оно может быстро стать совершенно неконтролируемым.

"Аномалия наследования" происходит лишь потому, что такие спецификации стремятся быть жесткими и хрупкими: измените хоть что-нибудь, и вся спецификация рассыплется.

В начале этой лекции мы наблюдали другой мнимый конфликт между параллельностью и наследованием, но оказалось, что в этом виновно понятие активного объекта. В обоих случаях наследование находится в противоречии не с параллельностью, но с конкретными подходами к ней (активные объекты, спецификации, основанные на состояниях). Поэтому прежде, чем отказываться от наследования или ограничивать его - отрубить руку, на которой чешется палец, - следует поискать более подходящие механизмы параллельности.

Одно из практических следствий этого обсуждения состоит в том, что следует попытаться использовать для синхронизации в параллельном вычислении то, что уже имеется в ОО-модели, в частности утверждения. И действительно, предусловия сыграют центральную роль в синхронизации, хотя придется изменить их семантику по сравнению с последовательным случаем.

Доступ к сепаратным объектам

Сейчас у нас уже достаточно сведений, чтобы предложить подходящие механизмы синхронизации параллельных ОО-систем.

Параллельный доступ к объекту

Первый вопрос, на который требуется ответить, - это сколько вычислений может одновременно работать с объектом. Ответ на него неявно присутствует в определениях процессора и обработчика: если все вызовы компонентов объекта выполняются его обработчиком (ответственным за него процессором) и процессор реализует один поток вычисления, то отсюда следует, что лишь один компонент может выполняться в каждый момент времени.

Следует ли разрешить одновременное выполнение нескольких подпрограмм на данном объекте? Основная причина ответить "нет" заключена в желании сохранить способность корректно рассуждать о нашем ПО.

Изучение корректности класса в предыдущей лекции позволяет найти верный подход. Жизненный цикл объекта можно изобразить следующим образом:

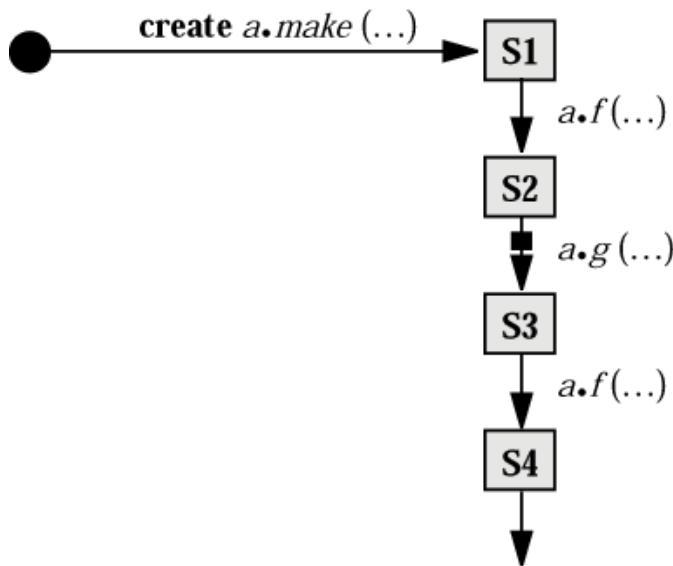


Рис. 12.7. Жизненный цикл объекта

На этом рисунке объект извне наблюдается только в состояниях, заключенных в квадратики: сразу после создания (S1), после каждого применения некоторого компонента клиентом (S2 и последующие состояния). Они называются "стабильными моментами" жизни объекта. Как следствие, мы получили формальное правило: чтобы доказать корректность класса, достаточно проверить одно свойство для каждой из процедур создания и одно свойство для каждого экспортируемого компонента (здесь оно несколько упрощено, полная формулировка была дана в [лекции 11](#) курса "Основы объектно-ориентированного программирования"). Если р - это процедура создания, то проверяемое свойство имеет вид:

```
{Default and prep} Bodyp {postp and INV}
```

Для экспортируемой подпрограммы г проверяемое свойство имеет вид:

```
{prer and INV} Bodyr {postr and INV}
```

Число проверяемых свойств невелико и не требует анализа сложных сценариев во время исполнения. Указанные свойства дают возможность понимания класса, рассматривая его подпрограммы независимо друг от друга, убеждаясь, пусть и неформально, в том, что каждая подпрограмма, начав работу в "правильном" состоянии, завершит ее в нужном заключительном состоянии.

Выведите параллельность в этот простой корректный мир и все пойдет прахом. Даже при простом чередовании, когда, начав выполнение некоторой подпрограммы, мы прерываем ее для выполнения другой, затем возвращаемся к первой и т. д., мы лишаемся возможности делать выводы о поведении ПО на основании текстов программ. У нас не будет никакой зацепки, помогающей понять, что может произойти во время выполнения, попытки угадать это заставят проверить все возможные варианты чередований, что сразу же приведет к уже указанному комбинаторному взрыву.

Поэтому для обеспечения простоты и корректности разрешим в каждый момент времени выполнять не более одной подпрограммы каждого объекта. Заметим, что существует возможность прервать клиента в случае крайней необходимости или при слишком долгой задержке объекта. Пока это делается насищенным способом - запуском исключения. При этом гарантируется, что неудачливый клиент получит извещение, позволяющее ему предпринять, если потребуется, корректирующие действия. Рассматриваемый далее механизм дуэлей (*duels*) дает такую возможность.

В конце обсуждения выясним, могут ли какие-либо обстоятельства позволить нам ослабить запрет на одновременный доступ к подпрограммам одного объекта.

Резервирование объекта

Нам нужен способ, обеспечивающий некоторому клиенту исключительные права доступа к некоторому ресурсу, предоставляемому некоторым объектом.

Идея, привлекательная на первый взгляд (но недостаточная), основана на использовании понятия *сепаратного вызова*. Рассмотрим вызов `x.f(...)` для сепаратной сущности `x`, присоединенной во время выполнения к объекту `O2`, где вызов выполняется некоторым объектом-клиентом `O1`. Ясно, что после начала выполнения этого вызова `O1` может безопасно перейти к своему следующему делу, не дожидаясь его завершения, но само выполнение этого вызова не может начаться до тех пор, пока `O2` не освободится для `O1`. Отсюда можно заключить, что клиент дождается, когда целевой объект станет свободным, и клиент сможет выполнять над ним свою операцию.

К сожалению, эта простая схема недостаточна, так как она не позволяет клиенту удерживать объект, пока в этом есть необходимость. Предположим, что `O2` - это некоторая разделяемая структура данных (например, буфер) и что соответствующий класс предоставляет процедуру `remove` для удаления одного элемента. Клиенту `O1` может потребоваться удалить два соседних элемента, но если просто написать:

```
Buffer.remove; buffer.remove;
```

то это не сработает, так как между выполнением этих двух инструкций может вклиниваться другой клиент, и удаленные элементы могут оказаться не соседними.

Одно решение состоит в добавлении к порождающему классу `buffer` (или его потомку) процедуры `remove_two`, удаляющей одновременно два элемента. Но в общем случае это решение нереалистично: нельзя изменять поставщика из-за каждой потребности в синхронизации кода их клиентов. У клиента должна быть возможность удерживать поставленный ему объект так долго, как это требуется.

Другими словами, нужно нечто в духе механизма критических интервалов. Ранее был введен их синтаксис:

```
hold a then действия_требующие_исключительного_доступа end
```

Или в условном варианте:

```
hold a when a.некоторое_свойство then действия_требующие_исключительного_доступа end
```

Тем не менее, мы перейдем к более простому способу обозначений, возможно, вначале несколько странному. Наше соглашение состоит в том, что, если `a` - это непустое сепаратное выражение, то вызов вида:

```
действия_требующие_исключительного_доступа (a)
```

автоматически заставляет ожидать до тех пор, пока объект, присоединенный к `a`, не станет доступным. В инструкции `hold` нет никакой необходимости - для резервирования сепаратного объекта достаточно указать его в качестве фактического аргумента вызова.

Заметим, что ожидание имеет смысл, только если подпрограмма содержит хоть один вызов `x.some_routine` с формальным аргументом `x`, соответствующим `a`. В противном случае, например, если она выполняет только присваивание вида `some_attribute := x`, ждать нет никакой необходимости. Это будет уточнено в полной форме правила, которое будет сформулировано далее в этой лекции.

Возможны и другие подходы, в которых авторы предлагают сохранить инструкцию `hold`. Но передача аргумента в качестве механизма резервирования объекта сохраняет простоту и легкость освоения модели параллелизма. Схема с `hold` привлекательна для разработчиков, поскольку соответствует девизу ОО-разработки "Инкапсулировать повторения", а главное, объединяет в одной подпрограмме действия, требующие исключающего доступа к объекту. Поскольку этой подпрограмме неизбежно потребуется аргумент, представляющий сам объект, то мы пошли дальше и считаем наличие такого аргумента достаточным для обеспечения резервирования объекта без введения ключевого слова `hold`.

Это соглашение также означает, что (достаточно парадоксально) **большинство сепаратных вызовов не должно ждать**. Выполняя тело подпрограммы с сепаратным формальным аргументом `a`, мы уже зарезервировали соответствующий присоединенный объект, поэтому всякий вызов с целью `a` может исполняться немедленно. Как уже отмечено, не требуется ждать, когда он завершится. В общем случае для подпрограммы вида:

```
r (a: separate SOME_TYPE) is
  do
    ...; a.r1 (...); ...
    ...; a.r2 (...); ...
  end
```

реализация может продолжать выполнение остальных инструкций, не дожидаясь завершения любого из двух вызовов при условии, что она протоколирует вызовы для `a` так, что они будут выполняться в требуемом порядке. (Нам еще нужно понять, как, если это потребуется, ждать завершения сепаратного вызова; пока же, мы просто запускаем вызовы и никогда не ждем!)

Если у подпрограммы имеется несколько сепаратных аргументов, то вызов клиента будет ждать, пока не сможет зарезервировать все соответствующие объекты. Это требование тяжело реализовать при компиляции, которая должна порождать код, используя протоколы для множественного одновременного резервирования. По этой причине компилятор мог бы требовать, чтобы подпрограмма использовала не более одного сепаратного формального аргумента. Но при полной реализации предложенного механизма разработчики приложений получают значительные удобства; в частности, рассматриваемая в качестве типичного примера далее в этой лекции знаменитая проблема "обедающих философов" допускает почти тривиальное решение.

Доступ к сепаратным объектам

Последний пример показывает, как формальный сепаратный аргумент подпрограммы `r` используется в качестве цели сепаратных вызовов в ее теле. Преимущество этого подхода в том, что для внутренних вызовов не нужно беспокоиться о получении доступа к целевому объекту: об этом будет заботиться вызов `r`, резервирующий объект, ожидая, если требуется, его освобождения.

Можно пойти дальше и объявить эту схему **единственной** для сепаратных вызовов:

Правило сепаратного вызова

Цель сепаратного вызова должна быть формальным аргументом подпрограммы, осуществляющей этот вызов.

Напомним, что вызов `a.r (...)` является сепаратным, если его цель `a` - это сепаратная сущность или выражение. Правило запрещает вызов компонентов `a`, если `a` не является формальным аргументом вызывающей подпрограммы. Поэтому приходится вводить дополнительные подпрограммы. Например, если `attrib` - это атрибут, объявленный как сепаратный, то вместо непосредственного вызова `attrib.r (...)` придется вызывать подпрограмму `rf(attrib, ...)`, где:

```
rf (x: separate SOME_TYPE; ... Другие аргументы ...) is
    -- Вызов r.
    do
        x.r (...)
    end
```

Может показаться, что это правило ложится чрезмерным бременем на плечи разработчиков параллельных приложений, поскольку оно вынуждает их встраивать все использование сепаратных объектов в соответствующие подпрограммы. Можно было бы разработать вариант модели из этой лекции без правила сепаратного вызова, но, когда вы начнете использовать эту модель, то, я уверен, поймете, что это правило на самом деле очень полезно. Оно поощряет разработчиков выделять места доступа к сепаратным объектам и отделять их от остальной части вычисления. Самое главное, что оно устраниет тяжелые ошибки, которые почти наверняка случились бы без него.

Следующий пример весьма типичен. Предположим, что имеется разделяемая структура данных - снова буфер - с компонентами `remove` для удаления элемента и `count` для запроса числа элементов. Тогда вполне "естественно" написать:

```
if buffer.count >= 2 then
    buffer.remove; buffer.remove
end
```

Замысел состоит в удалении двух элементов. Но, как мы уже отмечали, это не всегда работает - по крайней мере, до тех пор, пока не обеспечен безопасный исключающий доступ к `buffer`. Иначе между моментом, когда проверяется условие для `count` и моментом, когда выполняется первое удаление `remove`, любой другой клиент может прийти и удалить элемент, так что эта программа аварийно завершится, пытаясь применить `remove` к пустой структуре.

В следующем примере предполагается, что компонент `item`, не имеющий побочного эффекта, возвращает элемент, удаляемый компонентом `remove`:

```
if not buffer.empty then
    value := buffer.item; buffer.remove
end
```

Без защиты буфера `buffer` другой клиент может добавить или удалить элемент в промежутке между вызовами `item` и `remove`. В один прекрасный день автор этого фрагмента получит доступ к одному элементу, а удалит другой, так что можно, например, (при повторении указанной схемы) получить доступ к одному и тому же элементу дважды! Все это очень плохо.

Сделав `buffer` аргументом вызывающей подпрограммы, мы устраним эти проблемы: гарантируется, что `buffer` будет зарезервирован на все время выполнения вызова подпрограммы.

Конечно, вина за ошибку в рассмотренных примерах лежит на невнимательных разработчиках. Но без правила сепаратного вызова такие ошибки совершаются легко. По-настоящему плохо то, что поведение во время выполнения становится недетерминированным, поскольку оно зависит от относительной скорости клиентов. Из-за этого ошибки будут блуждающей, сейчас в одном месте программы, при следующем запуске - в другом. Еще хуже то, что она, вероятно, будет проявляться редко: во всяком случае (в первом примере) конкурирующий клиент должен оказаться очень удачливым, чтобы прописнуться между проверкой `count` и первым вызовом `remove`. Поэтому такую ошибку очень трудно повторить и изолировать.

Такие коварные ошибки ответственны за кошмарную репутацию отладки параллельных систем. Всякое правило, существенно уменьшающее вероятность их появления, оказывает большую помощь разработчикам.

Учитывая правило сепаратного вызова, наши примеры следует записать в виде следующих процедур, использующих сепаратный тип `BOUNDED_BUFFER`:

```
remove_two (buffer: BOUNDED_BUFFER) is
    -- Удаляет два самых старых элемента
    do
        if buffer.count >= 2 then
```

```

        buffer.remove; buffer.remove
    end
end
get_and_remove (buffer: BOUNDED_BUFFER) is
    -- Присваивает самый старый элемент value и удаляет его
do
    if not buffer.empty then
        value := buffer.item; buffer.remove
    end
end

```

Эти процедуры могут быть частью некоторого класса приложения; в частности, они могут быть описаны в классе BUFFER_ACCESS (**ДОСТУП_К_БУФЕРУ**), инкапсулирующем операции работы с буфером и служащем родительским классом для различных видов буферов.

Обе эти процедуры вызывают о предусловии. Вскоре мы позаботимся о нем.

Ожидание по необходимости

Предположим, что после ожидания необходимых сепараторных аргументов началось выполнение некоторого сепараторного вызова (например buffer.remove). Мы уже видели, что это не блокирует клиента, который может спокойно продолжать свои вычисления. Но, конечно, клиенту может потребоваться синхронизация с поставщиком, и для продолжения работы ему нужно дождаться завершения вызова.

Может показаться, что для этого нужен специальный механизм (он, действительно, был предложен в некоторых языках параллельного ОО-программирования, например в Hybrid) для воссоединения вычисления родителя с его расточительным вызовом. Но вместо этого можно использовать предложенную Денисом Каромелем (см. раздел "Библиографические заметки" этой лекции) идею ожидания по необходимости. Она состоит в том, чтобы ждать столько, сколько действительно необходимо.

Когда клиенту нужно точно знать, что вызов a.r (...) для сепараторной сущности a, присоединенной к сепараторному объекту O1, завершился? В тот момент, когда нужен доступ к некоторому свойству O1, требуется, чтобы объект был доступен, а все предыдущие его вызовы были завершены. До этого можно делать что-либо с другими объектами, даже запускать новый вызов процедуры a.r (...) на том же сепараторном объекте, поскольку, как мы видели, при разумной реализации можно просто ставить такие вызовы в очередь так, что они будут выполняться в порядке поступления.

Напомним, что компоненты подразделяются на команды (процедуры), выполняющие преобразование целевого объекта, и на запросы (функции и атрибуты), возвращающие информацию о нем. Завершения вызовов команд ждать не нужно, а завершения запросов - вполне возможно.

Рассмотрим, например, сепараторный стек s и последовательные вызовы:

```
s.put (x1); ... Другие инструкции...; s.put (x2); ... Другие инструкции ...; value := s.item
```

(которые в соответствии с правилом сепараторного вызова должны входить в некоторую подпрограмму с формальным аргументом s). Если предположить, что ни одна из "Других инструкций" не использует s, то единственной инструкцией, требующей ожидания, является последняя; ей необходима информация о стеке - его верхнее значение (которое в данном случае должно равняться x2).

Эти наблюдения приводят к главному в понимании ожидания по необходимости: после запуска сепараторного вызова клиент должен ожидать его завершения, только если это вызов запроса. Более точная формулировка правила будет приведена ниже после рассмотрения практического примера.

Ожидание по необходимости (также называемое "ленивым ожиданием" и похожее на механизмы "вызыва по необходимости" и "ленивого вычисления", знакомые лиспovцам и студентам, изучающим теоретическую информатику) позволяет запускать произвольные параллельные вычисления, устранивая ненужные ожидания и гарантируя ожидание, когда это действительно требуется.

Мультипускатель

Приведем типичный пример, показывающий преимущества ожидания по необходимости. Предположим, что некоторый объект должен создать множество других объектов, каждый из которых далее живет сам по себе:

```

launch (a: ARRAY [separate X]) is
    -- Запустить для каждого элемента a
    require
        -- Все элементы a непусты
local
    i: INTEGER
do
    from i := a.lower until i > a.upper loop
        launch_one (a @ i); i := i + 1
    end
end
launch_one (p: separate X) is
    -- Запустить для p
    require
        p /= Void
do
    p.live
end

```

Если процедура `live` класса `X` описывает бесконечный процесс, то корректность этой схемы основана на том, что каждая итерация цикла будет выполняться сразу же после запуска `Launch_one`, не ожидая, когда завершится этот вызов: иначе бы цикл никогда бы не ушел дальше его первой итерации. Эта схема будет далее использована в одном из примеров.

Читатели, знакомые с моделированием дискретных событий, основанным на сопрограммах, изучаемым в одной из следующих лекций, легко распознают схему, близкую к используемой, когда оператор языка Simula `detach` возвращает управление после запуска процесса моделирования.

Оптимизация

(В этом разделе рассматриваются достаточно тонкие темы, и при первом чтении его можно опустить).

Для завершения обсуждения ожидания по необходимости аккуратно определим, когда клиенту следует ожидать завершения сепараторного вызова.

Мы уже видели, что причиной ожидания могут быть только вызовы запросов. Но можно пойти дальше и выяснить, чем является результат запроса - развернутым типом или ссылкой. Если это развернутый тип, например `INTEGER` или какой-нибудь другой базисный тип, то выбора нет - нам нужно его значение, поэтому вычисление клиента должно ждать, пока запрос не вернет результат. Но в случае ссылочного типа разумная реализация сможет продолжить работу в то время, пока будет вычисляться сепараторный объект-результат запроса; в частности, если эта реализация использует "заместителей" сепараторных объектов, объект-заместитель может быть создан немедленно, так что ссылка на него доступна даже, если этот заместитель пока не ссылается на требуемый сепараторный объект.

Такая оптимизация, естественно, усложняет механизм параллельности, так как в этом случае заместители должны иметь логический атрибут "готов или нет", и все операции над сепараторными ссылками должны ожидать до тех пор, пока заместитель не станет "готов". Представляется также, что она ориентирована на конкретную реализацию - с помощью заместителей. Поэтому мы не включили ее в общее правило:

Ожидание по необходимости

Если клиент запустил на некотором сепараторном объекте один или несколько вызовов и выполняет на этом объекте вызов запроса, то этот вызов может начать выполнение лишь после завершения всех предыдущих вызовов, а все последующие операции клиента на этом объекте будут ожидать завершения этого вызова запроса.

Чтобы учесть только что рассмотренную возможную оптимизацию, измените "вызов запроса" на "вызов запроса, возвращающего развернутый тип".

Устранение блокировок (тупиков)

Наряду с несколькими важными примерами передачи сепараторных ссылок в сепараторные вызовы, мы видели, что возможна также передача несепараторных ссылок при условии, что соответствующие формальные аргументы объявлены как сепараторные (поскольку на стороне поставщика они представляют чужие объекты и нам не нужны предатели). Несепараторные ссылки увеличивают риск блокировок, и с ними нужно обходиться аккуратно.

Обычный способ передачи несепараторных ссылок состоит в использовании схемы **визитной карточки**: используется сепараторный вызов вида `x.f(a)`, где `x` - сепараторная сущность, а `a` - нет; иначе говоря, `a` - это ссылка на локальный объект клиента, возможно, на сам `Current`. На стороне поставщика `f` имеет вид:

```
f (u: separate SOME_TYPE) is
  do
    local_reference := u
  end
```

где `local_reference` типа `separate SOME_TYPE` является атрибутом объемлющего класса поставщика. Далее поставщик может использовать `local_reference` в подпрограммах, отличных от `f`, для выполнения операций над объектами на стороне клиента с помощью вызовов вида `local_reference.some_routine(...)`.

Эта схема корректна. Предположим, однако, что `f` делает еще что-то, например, включает для некоторого `g` вызов вида `u.g(...)`. Это с большой вероятностью приведет к тупику: клиент (обработчик объекта, присоединенного к `u` и `a`) занят выполнением `f` или, быть может, ожиданием по необходимости выполнения другого вызова, резервирующего тот же объект.

Следующее правило позволяет избежать таких ситуаций:

Принцип визитной карточки

Если сепараторный вызов использует несепараторный фактический аргумент типа ссылки, то соответствующий формальный аргумент должен использоваться в подпрограмме только в качестве источника присваиваний.

Пока это только методологическое руководящее указание, хотя было бы желательно ввести соответствующее формальное правило (в упражнениях У12.4 и У12.13 эта идея рассматривается глубже). Дополнительные комментарии о блокировках появятся еще при общем обсуждении.

Условия ожидания

Осталось рассмотреть еще одно правило синхронизации. Оно имеет отношение к двум вопросам:

- как можно заставить клиента ожидать выполнения некоторого условия (как это сделано в условных критических интервалах);
- что означают утверждения, в частности, предусловия, в контексте параллелизма?

Буфер - это сепаратная очередь

Нам нужен рабочий пример. Чтобы понять, что происходит с утверждениями, рассмотрим (понятие уже несколько раз неформально появляющееся в этой лекции) **ограниченный буфер**, позволяющий различным компонентам параллельной системы обмениваться данными. Производитель, порождающий объект, не должен ждать, пока потребитель будет готов его использовать, и наоборот. Взаимодействие происходит через разделяемую структуру - буфер. Ограниченный буфер может содержать не более maxcount элементов и поэтому может переполняться. При этом ожидание происходит только тогда, когда потребитель хочет получить элемент из пустого буфера или когда производителю нужно поместить элемент, а буфер полон. В хорошо отрегулированной системе с буфером такие события будут происходить гораздо реже, чем при взаимодействии без буфера, а их частота будет уменьшаться с ростом его размера. Правда, возникает еще один источник задержек из-за того, что доступ к буферу должен быть исключающим: в каждый момент лишь один клиент может выполнять операцию помещения в буфер (`put`) или извлечения из него (`item`, `remove`). Но это простые и быстрые операции, поэтому обычно общее время ожидания мало.

Как правило, порядок, в котором производятся объекты, важен для потребителей, поэтому буфер должен поддерживать дисциплину очереди "первым-в, первым-из (FIFO)".

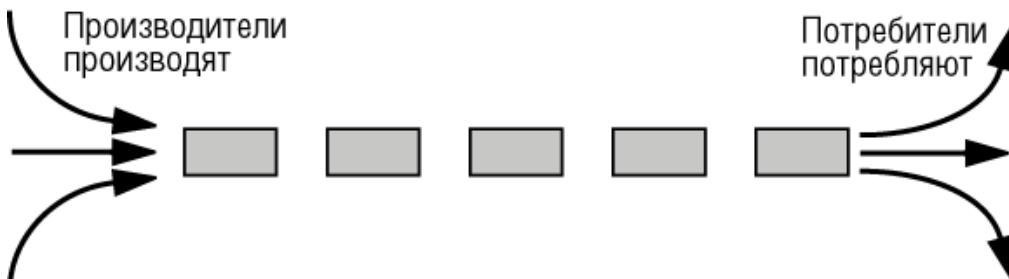


Рис. 12.8. Ограниченный буфер

Типичная реализация - несущественная для нашего рассмотрения, но дающая более конкретное представление о буфере - может использовать кольцевой массив representation размера capacity = maxcount + 1; число `oldest` будет номером самого старого элемента, а `next` - это индекс позиции, в которую нужно вставлять следующий элемент. Можно изобразить этот массив в виде кольца, в котором позиции 1 и capacity являются соседними (см. [рис. 12.9](#)).

Процедура `put`, используемая производителем для добавления элемента `x`, будет реализована как:

```
Representation.put (x, next); next := (next \ maxcount) + 1
```

где `\` - это операция получения остатка при целочисленном делении; запрос `item`, используемый потребителями для получения самого старого элемента, просто возвращает `representation @ oldest` (элемент массива в позиции `oldest`), а процедура `remove` просто выполняет `oldest := (oldest \ maxcount) + 1`. Ячейка массива с индексом `capacity` (на рисунке она серая) остается свободной; это позволяет отличить проверку условия пустоты `empty`, выражаемую как `next = oldest`, от проверки на полное заполнение `full`, выражаемой как `(next \ maxcount) + 1 = oldest`.

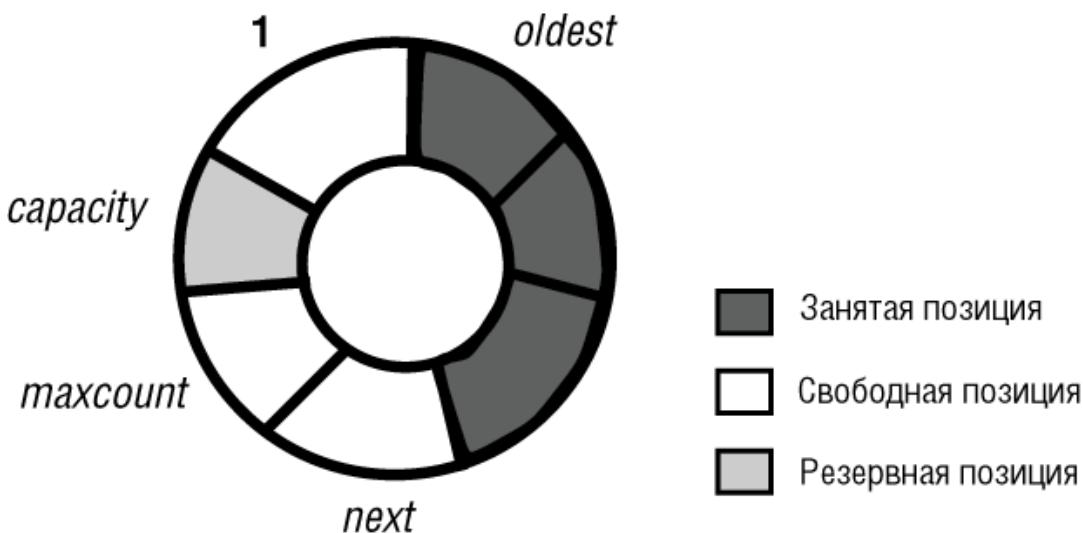


Рис. 12.9. Ограниченный буфер, реализованный массивом

Такая структура с политикой FIFO и представлением буфера в виде кольцевого массива, конечно, не является специфически параллельной: это просто **ограниченная очередь**, похожая на многие структуры, изученные в предыдущих лекциях. Нетрудно написать соответствующий класс, используя в качестве образца схему, использованную в [лекции 3](#) для команды возврата Undo. Ниже представлена краткая форма этого класса в упрощенном виде (только основные компоненты и главные утверждения без комментариев заголовков):

```
class interface BOUNDED_QUEUE [G] feature
    empty, full: BOOLEAN
    put (x: G)
        require
            not full
        ensure
            not empty
    remove
```

```

require
    not empty
ensure
    not full
item: G
require
    not empty
end

```

Получить из этого описания класс, задающий ограниченные буфера, проще, чем об этом можно было бы мечтать:

```

separate class BOUNDED_BUFFER [G] inherit
    BOUNDED_QUEUE [G]
end

```

Спецификатор `separate` относится только к тому классу, в котором он появляется, но не к его наследникам. Поэтому сепаратный класс может быть, как в данном случае, наследником несепаратного класса и наоборот. Соглашение такое же, как и для двух других спецификаторов, применимых к классам: `expanded` и `deferred`. Как уже отмечалось, эти три спецификатора являются взаимно исключающими, так что не более одного из них может появиться перед ключевым словом `class`.

Мы снова видим, как просто разрабатывать параллельное ОО-ПО, а гладкий переход от последовательных понятий к параллельным стал возможен, в частности, благодаря использованию инкапсуляции. Оказалось, что ограниченный буфер (понятие, для которого в литературе по параллелизму можно найти много усложненных описаний) - это не что иное как ограниченная очередь, сделанная сепаратной.

Предусловия при параллельном выполнении

Давайте рассмотрим типичное использование ограниченного буфера `buffer` клиентом, посылающим в него объект `u` с помощью процедуры `put`. Предположим, что `buffer` - это атрибут объемлющего класса, объявленный как `buffer : BOUNDED_BUFFER [T]` с элементами типа `T` (пусть `u` имеет этого же тип).

Клиент может, например, инициализировать `buffer` с помощью ссылки на реальный буфер, переданной из процедуры его создания, используя предложенную выше схему визитной карточки:

```
make (b: BOUNDED_BUFFER [T], ...) is do ...; buffer := b; ... end
```

Так как `buffer`, имеющий сепаратный тип, является сепаратной сущностью, то всякий вызов вида `buffer.put (u)` является сепаратным и должен появляться лишь в подпрограмме, одним из аргументов которой является `buffer`. Поэтому мы должны вместо него использовать `put(buffer, u)`, где `put` - подпрограмма из класса клиента (ее не следует путать с `put` из класса `BOUNDED_BUFFER`), объявленная как:

```

put (b: BOUNDED_BUFFER [T]; x: T) is
    -- Вставить x в b. (Первая попытка)
do
    b.put (x)
end

```

Но это не совсем верное определение. У процедуры `put` из `BOUNDED_BUFFER` имеется предусловие `not full`. Поскольку `u` не имеет смысла пытаться вставлять `x` в полный `b`, то нам нужно скопировать это условие в новой процедуре из класса клиента:

```

put (b: BOUNDED_BUFFER [T]; x: T) is
    -- Вставить x в b
    require
        not b.full
    do
        b.put (x)
    end

```

Уже лучше. Как же можно вызвать эту процедуру для конкретных `buffer` и `u`? Конечно, при входе требуется уверенность в выполнении предусловия. Один способ состоит в проверке:

```
if not full (buffer) then put (buffer, u)           -- [PUT1]
```

но можно также учитывать контекст вызова, например, в:

```
remove (buffer); put (buffer, u)           -- [PUT2]
```

где постусловие `remove` включает `not full`. (В примере PUT2 предполагается, что начальное состояние удовлетворяет соответствующему предусловию `not empty` для самой операции `remove`.)

Будет ли это верно работать? В свете предыдущих замечаний о непредсказуемости ошибок в параллельных системах ответ неутешителен - **может быть**. Между проверкой на полноту `full` и вызовом `put` в варианте PUT1 или между `remove` и `put` в PUT2 может вклиниваться какой-то другой клиент и снова сделать буфер полным. Это тот же дефект, который ранее потребовал от нас обеспечить резервирование объекта через инкапсуляцию.

Мы снова можем попробовать инкапсуляцию, написав PUT1 или PUT2 как процедуры, в которые `buffer` передается в качестве аргумента, например, для PUT1:

```
put_if_possible (b: BOUNDED_BUFFER [T]; x: T) is
```

```
-- Вставить x в b, если это возможно; иначе вернуть в was_full - значение true
do
    if b.full then was_full:= True else
        put (b, x); was_full := False
    end
end
```

Но на самом деле это не очень поможет клиенту. Во-первых, причиняет неудобство проверка условия `was_full` при возврате, а затем что делать, если оно истинно? Попытаться снова - возможно, но нет никакой гарантии успеха. На самом деле хотелось бы иметь способ выполнить `put` в тот момент, когда буфер будет наверняка неполон, даже если придется ждать, пока это случится.

Парадокс предусловий

Только что обнаруженная ситуация может обесокоить, поскольку, на первый взгляд, она показывает несостоятельность в параллельном контексте ключевой методологии проектирования по контракту. Для очереди при последовательных вычислениях у нас были полностью определены спецификации взаимных обязательств и преимуществ (аналогично стекам в [лекции 11](#) курса "Основы объектно-ориентированного программирования"):

Таблица 30.2. Контракт программы `put` для ограниченных очередей

<code>put</code>	Обязательства	Преимущества
Клиент	(Выполнить предусловие:) Вызывать <code>put(x)</code> только для непустой очереди	(Из постусловия:) Получить обновленную, непустую очередь с добавленным x
Поставщик	(Выполнить постусловие:) Обновить очередь, добавив x и обеспечив выполнение <code>not empty</code>	(Из постусловия:) Обработка защищена предположением о том, что очередь неполна

Неявно за такими контрактами стоит принцип "**отсутствия скрытых условий**": предусловие является единственным требованием, которое должен выполнить клиент, чтобы получить результат. Если вы вызываете `put` с неполной очередью на входе, то вам предоставляется результат этой подпрограммы, удовлетворяющий ее постусловию.

Но в параллельном контексте при наличии сепаратных поставщиков, таких как `BOUNDED_BUFFER`, дела клиента складываются весьма плачевно: как бы мы не старались ублажить поставщика, обеспечивая выполнение требуемого им предусловия, мы никогда не можем быть уверены в том, что его пожелания удовлетворены! Однако выполнение предусловия необходимо для корректной работы поставщика. Например, вполне вероятно, что тело подпрограммы `put` из класса `BOUNDED_QUEUE` (то же, что и в классе `BOUNDED_BUFFER`) не будет работать, если не гарантирована ложность условия `full`.

Подведем итоги: поставщики не могут выполнять свою работу без гарантии выполнения предусловий, а клиенты не способны обеспечить выполнение этих предусловий для сепаратных аргументов. Это можно назвать **парадоксом параллельных предусловий**.

Имеется аналогичный парадокс **постусловий**: при возврате из сепаратного вызова `put` нельзя быть уверенными, что для клиента выполнено условие `not empty` и другие постусловия. Эти свойства имеют место сразу после завершения подпрограммы, но другой клиент может их нарушить прежде, чем вызывающий клиент продолжит работу. Поскольку проблема является более серьезной для предусловий, определяющих корректную работу поставщиков, то они и будут рассматриваться.

Эти парадоксы возникают только для сепаратных формальных аргументов. Если аргумент не сепаратный, например, является значением развернутого типа, то можно продолжать рассчитывать на обычные свойства утверждений. Но это слабое утешение.

Хотя это еще не до конца осознано в литературе, парадокс параллельных предусловий является одним из центральных пунктов в конструировании параллельного ОО-ПО, а безрезультативность попыток сохранения обычной семантики утверждений является одним из главных факторов, отличающих параллельные вычисления от их последовательных вариантов.

Парадокс предусловий может также возникнуть и в ситуациях, когда обычно не думают о параллельности, например при доступе к файлу. Это изучается в упражнении Y12.6.

Параллельная семантика предусловий

Для разрешения парадокса параллельных предусловий выделим три аспекта возникшей ситуации:

- **A1** Поставщикам нужны предусловия для защиты тел их подпрограмм. Например, `put` из классов `BOUNDED_BUFFER` и `BOUNDED_QUEUE` требует гарантии неполноты входной очереди.
- **A2** Сепаратные клиенты не могут рассчитывать на обычную (последовательную) семантику предусловий. Проверка полноты `full` перед вызовом буфера еще не дает гарантий.
- **A3** Так как каждый клиент может соперничать с другими за доступ к ресурсам, то клиент должен быть готов ждать получения требуемых ресурсов. Наградой за ожидание является гарантия корректной обработки.

Отсюда неизбежен вывод: нам все еще нужны предусловия, но у них должна быть другая семантика. Они перестают быть условиями корректности, как в последовательном случае. Примененные к сепаратным аргументам они становятся **условиями ожидания**. Их можно назвать "предложениями сепаратного предусловия" и они применяются ко всякому предложению предусловия, содержащему вызов, целью которого является сепаратный аргумент. Типичным предложением сепаратного предусловия является `not b.full` для `put`.

Вот соответствующее правило:

Семантика сепаратного вызова

Прежде чем начать выполнение тела подпрограммы, сепаратный вызов должен дождаться момента, когда будут свободны все блокирующие объекты и будут выполнены все предложения сепаратного предусловия.

В этом определении объект называется блокирующим, если он присоединен к некоторому фактическому аргументу, а соответствующий формальный аргумент используется в подпрограмме в качестве цели хотя бы одного вызова.

Сепараторный объект является свободным, если он не используется в качестве фактического аргумента никакого сепараторного вызова (откуда следует, что на нем не исполняется никакая подпрограмма).

Это правило требует ожидания только для сепараторных аргументов, появляющихся в теле подпрограммы в качестве цели вызова (в нем для соответствующих объектов использовано слово "блокирующий", поскольку они могут заблокировать ее вызов в процессе выполнения). Для программы в виде "визитной карточки":

```
r (x: separate SOME_TYPE) is do some_attribute := x end
```

или в каком-либо другом виде, не содержащем вызова вида `x.some_routine`, не требуется ждать фактического аргумента, соответствующего `x`.

Если же такой вызов имеется, то для удобства авторов клиентов он должен быть отражен в краткой форме класса. Это будет указываться в заголовке подпрограммы как `r (x: blocking SOME_TYPE)...`

С помощью нашего правила приведенная выше версия `put` в классе клиента достигнет желаемого результата:

```
put (b: BOUNDED_BUFFER [T]; x: T) is
  require
    not b.full
  do
    b.put (x)
  ensure
    not b.empty
end
```

Вызов вида `put (buffer, y)` из клиента-производителя будет ждать, пока `buffer` не станет свободным (доступным) и не полным. Если `buffer` свободен, но полон, то данный вызов не может выполняться, но какой-нибудь другой клиент-потребитель может получить доступ к буферу (поскольку предусловие `not b.empty`, интересующее потребителей, будет в данном случае выполнено); после того, как такой клиент удалит некоторый элемент, сделав буфер неполным, клиент-производитель сможет начать выполнение своего вызова.

Как реализации решить, какой из двух или более клиентов, условия запуска вызовов которых выполнены (свободны блокирующие объекты и выполнены предусловия), должен получить доступ? Некоторые люди предпочитают передавать такие решения компилятору. Предпочтительнее определить по умолчанию политику FIFO, улучшающую переносимость и равнодоступность. Разработчикам приложений и в этом случае будут доступны библиотечные механизмы для изменения принятой по умолчанию политики.

Подчеркнем еще раз, что специальная семантика предусловий как условий ожидания применяется только к тому, что мы назвали предложениями сепараторных вызовов, т. е. к предложениям, включающим условия вида `b.some_property`, где `b` - это сепараторный аргумент. Несепараторное предложение, такое как `i >= 0`, будет иметь обычную семантику корректности, так как к нему неприменим парадокс параллельных предусловий: если клиент обеспечивает выполнение указанного условия перед вызовом, то оно будет выполнено и в момент запуска подпрограммы, а если это условие не выполнено, то никакое ожидание не приведет к изменению ситуации.

Последовательные и параллельные утверждения

Мысль о том, что утверждения, в частности предусловия, могут иметь две разные семантики - условий корректности и условий ожидания, может показаться странной. Но без этого не обойтись: последовательная семантика не применима в случае сепараторных предусловий.

Нужно ответить на одно возможное возражение. Мы видели, что проверку утверждения во время выполнения можно включать или отключать. Не опасно ли тогда придавать так много семантической важности предусловиям в параллельных ОО-системах? Нет. Утверждения являются неотъемлемой частью ПО независимо от того, проверяются ли они во время исполнения. В последовательной системе корректность означает, что утверждения всегда выполняются, поэтому из соображений эффективности можно отключить их проверку, если есть уверенность в том, что ошибок не осталось, хотя концептуально утверждения все еще присутствуют. В параллельном случае отличие состоит в том, что некоторые утверждения - предложения сепараторных предусловий - могут нарушаться во время выполнения даже в корректной системе и должны служить условиями ожидания.

Ограничение проверки правильности

Для устранения тупиков требуется наложить ограничение на предложения предусловий и постусловий. Предположим, что разрешены подпрограммы вида:

```
f (x: SOME_TYPE) is
  require
    some_property (separate_attribute)
  do
    ...
end
```

где `separate_attribute` - сепараторный атрибут объемлющего класса. В этом примере, кроме `separate_attribute`, ничто не должно быть сепараторным. Вычисление предусловия `f` (как части мониторинга утверждений корректности либо как условия синхронизации, если фактический аргумент, соответствующий `x`, является сепараторным) может вызвать блокировку, когда присоединенный объект недоступен.

Эта ситуация запрещается следующим правилом:

Правило аргументов утверждения

Если утверждение содержит вызов функции, то любой фактический аргумент этого вызова, если он сепаратный, должен быть формальным аргументом объемлющей подпрограммы.

Отметим, что из этого правила следует, что такое утверждение не может появиться в инварианте класса, который не является частью подпрограммы.

Состояния и переходы

[Рис. 12.10](#) подводит итог предшествующего обсуждения, показывая различные возможные состояния, в которых могут находиться объекты и процессоры, и их изменения в результате вызовов.

Вызов является **успешным**, если обработчик его цели не занят или приостановлен, все его непустые сепаратные аргументы присоединены к свободным объектам, а все имеющиеся предложения соответствующего сепаратного предусловия выполнены. Заметим, что это делает определения состояний объекта и процессора взаимно зависимыми.

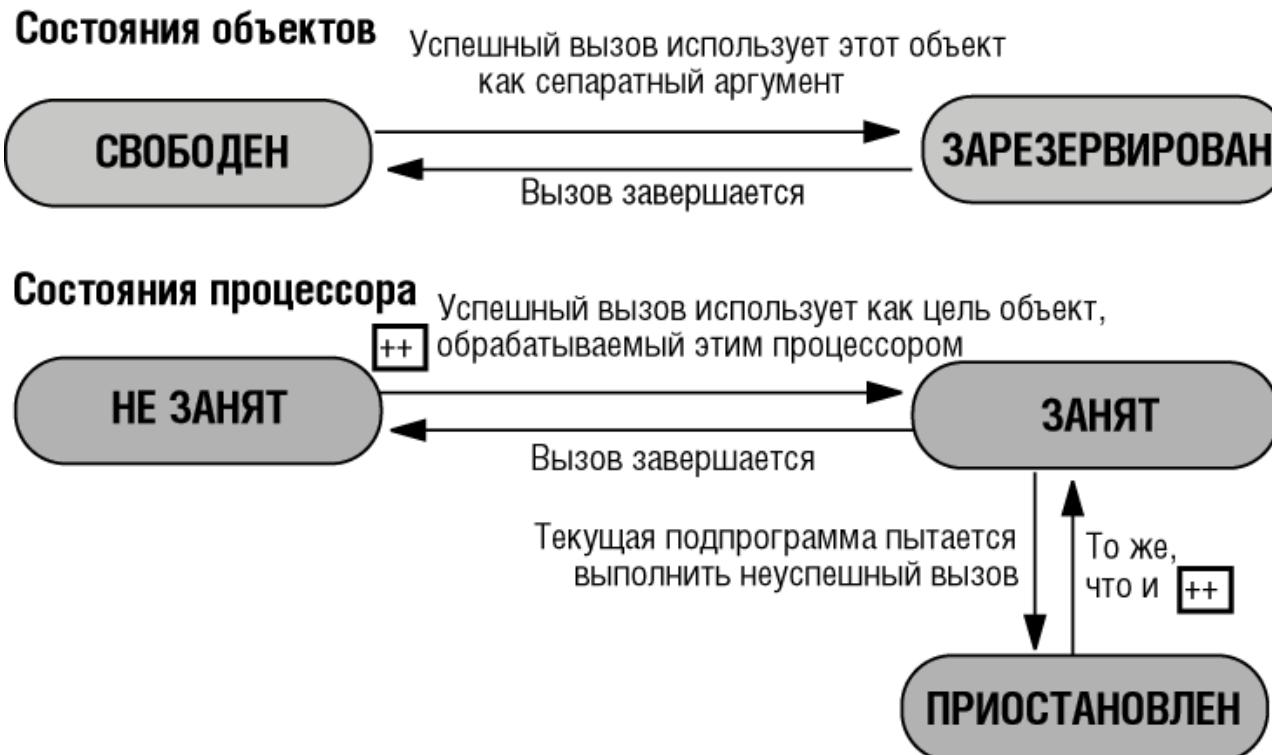


Рис. 12.10. Состояния объектов и процессоров и переходы между ними

Запросы специальных услуг

Мы завершили описание основ политики взаимодействия и синхронизации. Для большей гибкости полезно было бы определить еще несколько способов прерывания нормального процесса вычисления, доступных в некоторых случаях.

Так как эти возможности не являются частью основной модели параллелизма, а добавляются для удобства, то они вводятся не как конструкции языка, а как библиотечные компоненты. Мы будем считать, что они помещены в класс CONCURRENCY, из которого могут наследоваться классами, нуждающимися в таких механизмах. Аналогичный подход был уже дважды использован в этой книге:

- для дополнения базисной обработки исключений более тонкими средствами управления с помощью библиотечного класса EXCEPTIONS ([лекция 12](#)) курса "Основы объектно-ориентированного программирования";
- для дополнения стандартного механизма управления памятью и сбором мусора более тонкими средствами управления с помощью библиотечного класса MEMORY (см. [лекцию 9](#) курса "Основы объектно-ориентированного программирования").

Экспресс сообщения

В параллельном языке ABCL/1 ([Yonezawa 1987a]) введено понятие "экспресс-сообщение" для тех случаев, когда объекту-поставщику нужно позволить обслужить "вне очереди" некоторого VIP-клиента, даже если он в данный момент занят обслуживанием другого клиента.

При некоторых подходах экспресс-сообщение прерывает нормальное сообщение, получает услугу, возобновляя затем нормальное сообщение. Для нас это неприемлемо. Ранее мы уже поняли, что в каждый момент на любом объекте может быть активизировано лишь одно вычисление. Экспресс-сообщение, как и всякий экспортруемый компонент, нуждается в выполнении инвариант в начальном состоянии, но кто знает, в каком состоянии окажется прерываемая программа, когда ее заставят уступить место экспресс-сообщению? И кто знает, какое состояние в этом случае будет создано в результате? Все это открывает дорогу к созданию противоречивого объекта. При обсуждении статического связывания в [лекции 14](#) курса "Основы объектно-ориентированного программирования" это было названо "одним из худших событий, возникающих во время выполнения программной системы". Как мы тогда отметили, "если возникает такая ситуация, то нельзя надеяться на предсказание

результата вычисления".

Тем не менее, это не означает полного отказа от экспресс-сообщения. Нам на самом деле может потребоваться прервать клиента либо потому, что появилось нечто более важное, что нужно сделать с зарезервированным им объектом, либо потому, что он слишком затянул владение объектом. Но такое прерывание - это не вежливая просьба отступить ненадолго. Это убийство или по меньшей мере попытка убийства. Устранивая конкурента, в него стреляют, так что он погибнет, если не сможет излечиться в больнице. В программных терминах прерывание, заданное клиентом, должно вызывать исключение, приводящее в итоге либо к смерти (fail), либо к излечению и повторной попытке (retry) выполнить свою работу.

При таком поведении подразумевается превосходство претендента над конкурентом. В противном случае, у него самого возникнут неприятности - исключение.

Дуэли и их семантика

Почти неизбежная метафора предполагает, что вместо терминологии "экспресс-сообщение" можно говорить о дуэли (в предшествующей эре к дуэли приводили попытки увести чью-либо законную супругу).

Пусть объект выполнил инструкцию:

r (b)

для сепаратного b. После возможного ожидания освобождения b и выполнения сепаратного предусловия объект захватывает b, становясь его текущим **владельцем**. От имени владельца начинается выполнение r на b, но в некий момент времени, когда действие еще не завершилось, другой сепаратный объект, **претендент**, выполняет вызов:

s (c)

Пусть сущность с сепаратной и присоединена к тому же объекту, что и b. В обычном случае претендент будет ждать завершения вызова r. Но что случится, если претендент нетерпелив?

С помощью процедур класса CONCURRENCY можно обеспечить необходимую гибкость. Владелец мог уступить, вызвав процедуру yield, означающую: "Я готов отдать свое владение более достойному". Конечно, большинство владельцев не будут столь любезны: если вызов yield явно не выполнен, то владелец будет удерживать то, чем владеет. Сделав уступку, владелец позже может от нее отказаться, вернувшись к установленному по умолчанию поведению, для чего использует вызов процедуры retain.

У претендента, желающего захватить занятый ресурс, есть два разных способа сделать это. Он может выполнить:

- Demand означает "сейчас или никогда!". Непреклонный владелец (не вызвавший yield), ресурс не отдаст, и у претендента, не сумевшего захватить предмет своей мечты, возникнет исключение (так что demand - это своего рода попытка самоубийства). Уступчивый владелец ресурса отдаст его претенденту, исключение возникнет у него.
- Insist более мягкая процедура: вы пытаетесь прервать программу владельца, но, если это невозможно, то вас ждет общий жребий - ждать, пока не освободится объект.

Для возврата к обычному поведению с ожиданием владельца претендент может использовать вызов процедуры wait_turn.

Вызов одной из этих процедур класса CONCURRENCY будет сохранять свое действие до тех пор, пока другая процедура его не отменит. Отметим, что эти два набора не исключают друг друга. Например, претендент может одновременно использовать insist, требуя специальной обработки, и yield, допуская прерывание своей работы другими. Можно также добавить схему с приоритетами, в которой претенденты ранжированы в соответствии с приоритетами, но здесь мы не будем ее уточнять.

В следующей таблице показаны все возможные результаты дуэлей - конфликтов между владельцем и претендентом. По умолчанию считается, что процедуры из класса CONCURRENCY не вызываются (эти случаи в таблице подчеркнуты).

Таблица 30.3. Семантика дуэлей

Владелец	Претендент		
	Wait_turn	demand	insist
retain	Претендент ждет	Исключение у претендента	Владелец ждет
yield	Претендент ждет	Исключение в программе владельца	Исключение в программе владельца

"Программа владельца", в которой возбуждается исключение в двух нижних правых клетках, - это программа поставщика, исполняемая от лица владельца. При отсутствии retry она будет передавать исключение владельцу, а претендент будет получать объект.

Как вы помните, каждый вид исключений имеет свой код, доступный через класс EXCEPTIONS. Для выделения исключений, вызванных ситуациями из приведенной таблицы, класс предоставляет запрос is_concurrency_interrupt.

Обработка исключений: алгоритм "Секретарь-регистратор"

Приведем пример, использующий дуэли. Предположим, что некоторый управляющий объект-контроллер запустил несколько объектов-партнеров, а затем занялся своей собственной работой, для которой требуется некоторый ресурс shared. Но другим объектам также может потребоваться доступ к этому разделяемому ресурсу, поэтому контроллер готов в таком случае прервать выполнение своего текущего задания и дать возможность поработать с ресурсом каждому из них, а когда партнер отработает, контроллер возобновит выполнение прерванного задания.

Приведенное общее описание среди прочего охватывает и ядро операционной системы (контроллер), запускающее процессоры ввода-вывода (партнеры), но не ждущее завершения их операций, поскольку операции ввода-вывода выполняются на нескольких порядков медленнее основного вычисления. По завершении операции ввода-вывода ее процессор требует внимания к себе и посыпает запрос на прерывание ядра. Это традиционная схема управления вводом-выводом с помощью прерываний - проблема, давшая много лет назад первоначальный импульс изучению параллелизма.

Эту общую схему можно назвать алгоритмом "Секретарь-регистратор" по аналогии с тем, что наблюдается во многих организациях: регистратор сидит в приемной, приветствует, регистрирует и направляет посетителей, но кроме этого, он выполняет и обычную секретарскую работу. Когда появляется посетитель, регистратор прерывает свою работу, занимается с посетителем, а затем возвращается к прерванному заданию.

Возврат к выполнению некоторого задания после того, как оно было начато и прервано, может потребовать некоторых действий, поэтому приведенная ниже процедура работы секретаря передает в вызываемую ей процедуру operate значение interrupted, позволяющее проверить, запускалось ли уже текущее задание. Первый аргумент operate, здесь это next, идентифицирует выполняемое задание. Предполагается, что эта процедура является частью класса, наследника CONCURRENCY (yield и retain), и EXCEPTIONS (is_concurrency_interrupt). Выполнение процедуры operate может занять много времени, поэтому она является прерываемой частью.

```
execute_interruptibly is
    -- Выполнение собственного набора действий с прерываниями
    -- (алгоритм Секретарь-регистратор)
local
    done, next: INTEGER; interrupted: BOOLEAN
do
    from done := 0 until termination_criterion loop
        if interrupted then
            process_interruption (shared); interrupted := False
        else
            next := done + 1; yield
            operate (next, shared, interrupted) -- Это прерываемая часть
            retain; done := next
        end
    end
rescue
    if is_concurrency_interrupt then
        interrupted := True; retry
    end
end
```

Некоторые из выполняемых контроллером шагов могут быть на самом деле затребованы одним из прерывающих партнеров. Например, при прерывании ввода-вывода его процессор будет сигнализировать об окончании операции и (в случае ввода) о доступности прочитанных данных. Прерывающий партнер может использовать объект shared для размещения этой информации, а для прерывания контроллера он будет выполнять:

```
insist; interrupt (shared); wait_turn
    - Требует внимания контроллера, если нужно, прерывает его.
    -- Размещает всю необходимую информацию в объекте shared.
```

По этой причине process_interruption, как и operate, использует в качестве аргумента shared: объект shared можно проанализировать, выявив информацию, переданную прерывающим партнером. Это позволит ему при необходимости подготовить одно из последующих заданий для выполнения от имени этого партнера. Подчеркнем, что в отличие от operate сама процедура process_interruption не является прерываемой; любому партнеру придется ждать (в противном случае некоторые заявки партнеров могли бы потеряться). Поэтому process_interruption должна выполнять простые операции - регистрировать информацию, требуемую для последующей обработки. Если это невозможно, то можно использовать несколько иную схему, в которой process_interruption надеется на сепаратный объект, отличный от shared.

Следует принять меры предосторожности. Хотя заявки партнеров могут обрабатываться позднее (с помощью вызовов operate на последующих шагах), важно то, что ни одна из них не будет потеряна. В приведенной схеме после выполнения некоторым партнером interrupt другой может сделать то же самое, стерев информацию до того, как у контроллера появится время для ее регистрации. Такое нельзя допустить. Для устранения этой опасности можно добавить в класс, порождающий shared, логический атрибут deposited с соответствующими процедурами его установки и сброса. Тогда у interrupt появится предусловие not shared.deposited, и придется ждать, пока предыдущий партнер не зарегистрируется и не выполнит перед выходом вызов shared.set_deposited, а process_interruption перед входом будет выполнять shared.set_not_deposited.

Партнеры инициализируются вызовами по схеме "визитной карточки" вида create partner.make (shared, ...), в которых им передается ссылка на объект shared, сохраняемая для дальнейших нужд.

Процедура execute_interruptibly должна быть расписана полностью с включением специфических для приложения элементов, представляемых вызовами подпрограмм operate, process_interruption, termination_criterion, которые в стиле класса поведения предполагаются отложенными. Это подготавливает возможное включение этих процедур в библиотеку параллелизма.

О том, что будет дальше в этой лекции

Представив механизм дуэлей, мы завершили определение набора инструментов, необходимых для реализации параллельности. В оставшейся части этой лекции приведен большой набор примеров для различных приложений, использующий эти инструменты. После примеров вы найдете:

- набросок правил доказательства для читателей, склонных к математике;
- сводку средств предложенного механизма параллельности с его синтаксисом, правилами корректности и семантикой;
- обсуждение целей этого механизма и дальнейшей необходимой работы;
- подробную библиографию других работ в этой области.

Примеры

Для иллюстрации предложенного механизма мы сейчас приведем несколько примеров, выбранных из различных источников - от традиционных примеров параллельных программ до приложений реального времени.

Обедающие философы

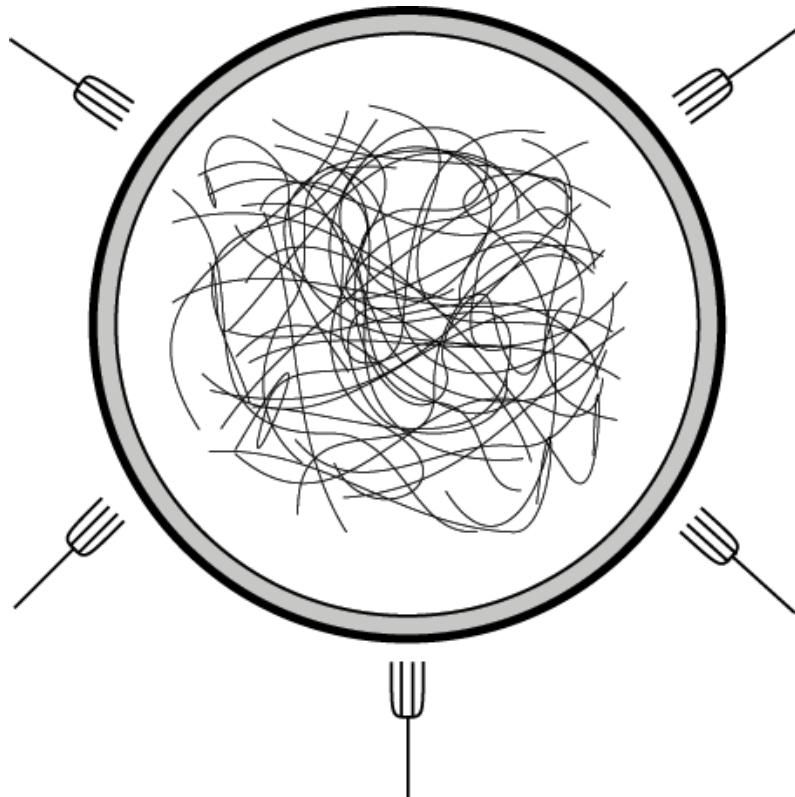


Рис. 12.11. Блюдо спагетти обедающих философов

Знаменитые "обедающие философы" Дейкстры - искусственный пример, призванный проиллюстрировать поведение процессов операционной системы, конкурирующих за разделяемые ресурсы, является обязательной частью всякого обсуждения параллелизма. Пять философов, сидящих за столом, проводят время в размышлениях, затем едят, затем снова размышляют и т. д. Чтобы есть спагетти, каждому из них нужны две вилки, лежащие непосредственно слева и справа от философов, что создает предпосылку для возникновения блокировок.

Следующий класс описывает поведение философов. Благодаря механизму резервирования объектов с помощью сепаратных аргументов в нем, по существу, отсутствует явный код синхронизации (в отличие от обычно предлагаемых в литературе решений):

```
separate class PHILOSOPHER creation
  make
  inherit
    GENERAL_PHILOSOPHER
    PROCESS
      rename setup as getup undefine getup end
  feature {BUTLER}
    step is
      -- Выполнение задач философа
      do
        think
        eat (left, right)
      end
  feature {NONE}
    eat (l, r: separate FORK) is
      -- Обедать, захватив вилки l и r
      do ... end
  end
```

Все, что связано с синхронизацией, вложено в вызов `eat`, который использует аргументы `left` и `right`, представляющие обе необходимые вилки, резервируя тем самым эти объекты.

Простота этого решения объясняется способностью резервировать несколько сепаратных аргументов в одном вызове, здесь это `left` и `right`. Если бы мы ограничили число сепаратных аргументов в вызове одним, то в решении пришлось бы использовать один из многих опубликованных алгоритмов для захвата двух вилок без блокировки.

Главная процедура класса `PHILOSOPHER` не приведена выше, поскольку она приходит из класса поведения `PROCESS`: это процедура `live`, которая по определению в `PROCESS` просто выполняет `from setup until over loop step end`, поэтому все, что требуется переопределить, это `step`. Надеюсь, вам понравилось переименование обозначения начальной операции философа `setup` в `getup`.

Благодаря использованию резервирования нескольких объектов с помощью аргументов описанное выше решение не создает блокировок, но нет гарантии, что оно обеспечивает равнодоступность. Некоторые из философов могут организовать заговор, уморив

голодом коллег. Во избежание такого исхода предложены разные решения, которые можно интегрировать в описанную выше схему.

Чтобы избежать смешения жанров, независящие от параллельности компоненты собраны в класс GENERAL_PHILOSOPHER:

```
class GENERAL_PHILOSOPHER creation
    make
feature -- Initialization
    make (l, r: separate FORK) is
        -- Задать l как левую, а r как правую вилки
        do
            left := l; right := r
        end
feature {NONE} -- Implementation
    left, right: separate FORK
        -- Две требуемые вилки
    setup is
        -- Выполнить необходимую инициализацию
        do ... end
    think is
        -- Любое подходящие действие или его отсутствие
        do ... end
end
```

Остальная часть системы относится к инициализации и включает описания вспомогательных абстракций. У вилок никаких специальных свойств нет:

```
class FORK end
Класс BUTLER ("дворецкий") используется для настройки и начала сессии:
class BUTLER creation
    make
feature
    count: INTEGER
        -- Число философов и вилок
    launch is
        -- Начало полной сессии
        local
            i: INTEGER
        do
            from i := 1 until i > count loop
                launch_one (participants @ i); i := i + 1
            end
        end
    end
feature {NONE}
    launch_one (p: PHILOSOPHER) is
        -- Позволяет начать актуальную жизнь одному философу
        do
            p.live
        end
    participants: ARRAY [PHILOSOPHER]
    cutlery: ARRAY [FORK]
feature {NONE} -- Initialization
    make (n: INTEGER) is
        -- Инициализация сессии с n философами
        require
            n >= 0
        do
            count := n
            create participants.make (1, count); create cutlery.make (1, count)
            make_philosophers
        ensure
            count = n
        end
        make_philosophers is
            -- Настройка философов
        local
            i: INTEGER; p: PHILOSOPHER; left, right: FORK
        do
            from i := 1 until i > count loop
                p := philosophers @ i
                left := cutlery @ i
                right := cutlery @ ((i \\\ count) + 1
                create p.make (left, right)
                i := i + 1
            end
        end
    invariant
        count >= 0; participants.count = count; cutlery.count = count
end
```

Обратите внимание, `launch` и `launch_one`, используя образец, обсужденный при введении ожидания по необходимости, основаны на том, что вызов `r.live` не приведет к ожиданию, допуская обработку следующего философа в цикле.

Полное использование параллелизма оборудования

Следующий пример иллюстрирует, как использовать ожидание по необходимости для извлечения максимальной пользы от параллелизма в оборудовании. Он показывает изощренную форму **балансировки загрузки** компьютеров в сети. Благодаря понятию процессора, можно опереться на механизм параллельности для автоматического выбора компьютеров.

Сам этот пример - вычисление числа вершин бинарного дерева - имеет небольшое практическое значение, но иллюстрирует общую схему, которая может быть чрезвычайно полезна для больших, сложных вычислений, встречающихся в криптографии или в машинной графике, для которых разработчикам нужны все доступные ресурсы, но не хочется вручную заниматься назначением абстрактных вычислительных единиц реальным компьютерам.

Рассмотрим сначала набросок класса, без параллелизма:

```
class BINARY_TREE [G] feature
    left, right: BINARY_TREE [G]
    ... Другие компоненты ...
    nodes: INTEGER is
        -- Число вершин в данном дереве
    do
        Result := node_count (left) + node_count (right) + 1
    end
feature {NONE}
    node_count (b: BINARY_TREE [G]): INTEGER is
        -- Число вершин в b
    do
        if b /= Void then Result := b.nodes end
    end
end
```

Функция `nodes` использует рекурсию для вычисления числа вершин в дереве. Эта косвенная рекурсия проходит через вызовы `node_count`.

В параллельном окружении, предлагающем много процессоров, можно было бы загрузить вычисления для отдельных вершин в разные процессоры. Сделаем это, объявив класс **сепаратным** (`separate`), заменив `nodes` атрибутом и введя соответствующие процедуры:

```
separate class BINARY_TREE1 [G] feature
    left, right: BINARY_TREE1 [G]
    ... Другие компоненты ...
    nodes: INTEGER
    update_nodes is
        -- Модифицировать nodes, подсчитав число вершин дерева
    do
        nodes := 1
        compute_nodes (left); compute_nodes (right)
        adjust_nodes (left); adjust_nodes (right)
    end
feature {NONE}
    compute_nodes (b: BINARY_TREE1 [G]) is
        -- Модифицировать информацию о числе вершин в b
    do
        if b /= Void then
            b.update_nodes
        end
    end
    adjust_nodes (b: BINARY_TREE1 [G]) is
        -- Добавить число вершин в b
    do
        if b /= Void then nodes := nodes + b.nodes end
    end
end
```

В этом случае рекурсивные вызовы `compute_nodes` будут запускаться параллельно. Операция сложения будет ждать, пока не завершатся два параллельных вычисления.

Если доступно неограниченное число ЦПУ (физических процессоров), то это решение, по-видимому, обеспечивает максимальное использование параллелизма оборудования. Если же число имеющихся процессоров меньше числа вершин дерева, то ускорение вычисления по сравнению с последовательным вариантом будет зависеть от того, насколько удачно реализовано распределение (виртуальных) процессоров по ЦПУ.

Наличие двух проверок пустоты `b` может показаться неприятным. Однако это требуется для отделения распараллеливаемой части - вызовы процедур, запускаемых параллельно на `left` и `right`, - от сложений, которые по своему смыслу должны ждать готовности своих операндов.

В этом решении привлекает то, что все проблемы, связанные с назначением конкретных компьютеров, полностью игнорируются. Программа занимает процессоры по мере необходимости. Это происходит в не приведенных здесь командах создания, появляющихся, в частности, в процедуре вставки. Для вставки нового элемента в бинарное дерево создается вершина вызовом

create new_node.make (new_element). Поскольку new_node имеет сепаратный тип BINARY_TREE1[G], для нее выделяется процессор. Связывание этих виртуальных процессоров с доступными физическими ресурсами происходит автоматически.

Замки

Предположим, что мы хотим разрешить многим клиентам, которых будем называть ключниками (lockers), получать исключительный доступ к сейфам - закрываемым ресурсам (lockable) - без явного выделения разделов, где происходит этот доступ, исключающий других ключников. Это даст нам механизм типа семафоров. Вот решение:

```
class LOCKER feature
  grab (resource: separate LOCKABLE) is
    -- Запрос исключительного доступа к ресурсу
    require
      not resource.locked
    do
      resource.set_holder (Current)
    end
  release (resource: separate LOCKABLE) is
    require
      resource.is_held (Current)
    do
      resource.release
    end
  end
class LOCKABLE feature {LOCKER}
  set_holder (l: separate LOCKER) is
    -- Назначает l владельцем
    require
      l /= Void
    do
      holder := l
    ensure
      locked
    end
  locked: BOOLEAN is
    -- Занят ли ресурс каким-либо ключником?
    do
      Result := (holder /= Void)
    end
  is_held (l: separate LOCKER): BOOLEAN is
    -- Занят ли ресурс l?
    do
      Result := (holder = l)
    end
  release is
    -- Освобождение от текущего владельца
    do
      holder := Void
    ensure
      not locked
    end
  feature {NONE}
    holder: separate LOCKER
  invariant
    locked_iff_holder: locked = (holder /= Void)
  end
```

Всякий класс, описывающий ресурсы, будет наследником LOCKABLE. Правильное функционирование этого механизма предполагает, что каждый ключник выполняет последовательность операций grab и release в этом порядке. Другое поведение приводит, как правило, к блокировке работы, эта проблема уже была отмечена при обсуждении семафоров как один из существенных недостатков этого метода. Но можно и в этом случае получить требуемое поведение системы, основываясь на силе ОО-вычислений. Не доверяя поведению каждого ключника, можно требовать от них вызова процедуры use, определенной в следующем классе поведения:

```
deferred class LOCKING_PROCESS feature
  resource: separate LOCKABLE
  use is
    -- Обеспечивает дисциплинированное использование resource
    require
      resource /= Void
    do
      from create lock; setup until over loop
        lock.grab (resource)
        exclusive_actions
        lock.release (resource)
      end
      finalize
    end
  set_resource (r: separate LOCKABLE) is
    -- Выбирает r в качестве используемого ресурса
```

```

require
    r /= Void
do
    resource := r
ensure
    resource /= Void
end
feature {NONE}
lock: LOCKER
exclusive_actions
    -- Операции во время исключительного доступа к resource
deferred
end
setup
    -- Начальное действие; по умолчанию: ничего не делать
do
end
over: BOOLEAN is
    -- Закончилось ли закрывающее поведение?
deferred
end
finalize
    -- Заключительное действие; по умолчанию: ничего не делать
do
end
end

```

В эффективных наследниках класса LOCKING_PROCESS процедуры exclusive_actions и over будут эффективизированы, а setup и finalize могут быть доопределены. Отметим, что желательно писать класс LOCKING_PROCESS как наследник класса PROCESS.

Независимо от того, используется ли LOCKING_PROCESS, подпрограмма grab не отбирает сейфу всех возможных клиентов: она исключает только ключников, не соблюдающих протокол. Для закрытия доступа к ресурсу любому клиенту нужно включить операции доступа в подпрограмму, которой ресурс передается в качестве аргумента.

Подпрограмма grab из класса LOCKER является примером того, что называется схемой визитной карточки: ресурсу resource передается ссылка на текущего ключника Current, трактуемая как сепаратная ссылка.

Основываясь на представляемых этими классами образцах, нетрудно написать и другие реализации разных видов семафоров (см. У12.7). ОО-механизмы помогают пользователям таких классов избежать классической опасности семафоров: выполнить для некоторого ресурса операцию резервирования reserve и забыть выполнить соответствующую операцию освобождения free. Разработчик, использующий класс поведения типа LOCKING_PROCESS, допишет отложенные операции в соответствии с нуждами своего приложения и сможет рассчитывать на то, что предопределенная общая схема обеспечит выполнение после каждой reserve соответствующей операции free.

Сопрограммы (Coroutines)

Хотя наш следующий пример и не является полностью параллельным (по крайней мере в его первоначальном виде), но он важен как способ проверки применимости нашего параллельного механизма.

Первым (и, возможно, единственным) из главных языков программирования, включившим конструкцию сопрограмм, был также и первый ОО-язык Simula 67; мы будем рассматривать его механизм сопрограмм при его описании в [лекции 17](#). Там же будут приведены примеры практического использования сопрограмм.

Сопрограммы моделируют параллельность на последовательном компьютере. Они представляют собой программные единицы, отражающие симметричную форму взаимодействия:

- При вызове обычной подпрограммы имеется хозяин и раб. Хозяин запускает подпрограмму, ожидает ее завершения и продолжает с того места, где закончился вызов; однако подпрограмма при вызове всегда начинает работу с самого начала. Хозяин вызывает, а подпрограмма-раб возвращает.
- Отношения между сопрограммами - это отношения равных. Когда сопрограмма a застревает в процессе своей работы, то она призывает сопрограмму b на помощь; b запускается с того места, где последний раз остановилась, и продолжает выполнение до тех пор, пока сама не застрянет или не выполнит все, что от нее в данный момент требуется; затем a возобновляет свое вычисление. Вместо разных механизмов вызова и возврата здесь имеется одна операция возобновления вычисления resume c, означающая: запусти сопрограмму c с того места, в котором она последний раз была прервана, а я буду ждать, пока кто-нибудь не возобновит (resumes) мою работу.

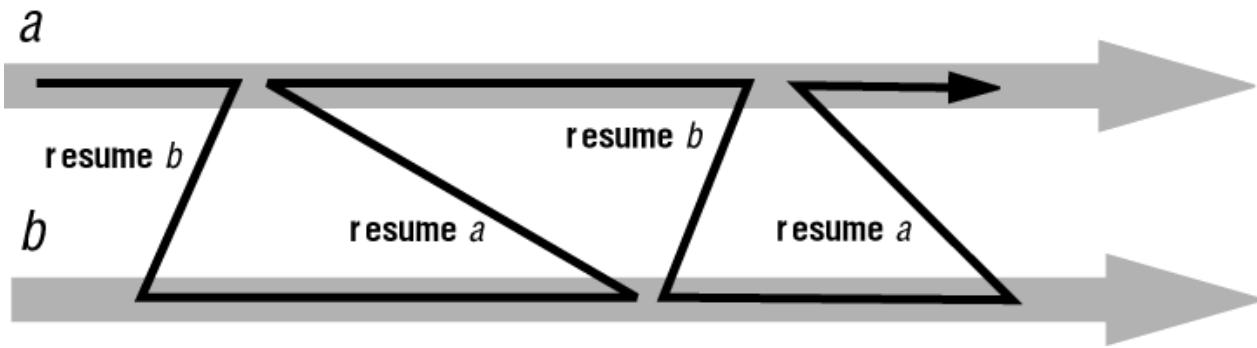


Рис. 12.12. Последовательность выполнения сопрограмм

Все это происходит строго последовательно и предназначено для выполнения в одном процессе (задании) одного компьютера. Но сама идея получена из параллельных вычислений; например, операционная система, выполняемая на одном ЦПУ, будет внутри себя использовать механизм сопрограмм для реализации разделения времени, многозадачности и многопоточности.

Можно рассматривать сопрограммы как некоторый ограниченный вид параллельности: бедный суррогат параллельного вычисления, которому доступна лишь одна ветвь управления. Обычно полезно проверять общие механизмы на их способность элегантно упрощаться для работы в ограниченных ситуациях, поэтому давайте посмотрим, как можно представить сопрограммы. Эта цель достигается с помощью следующих двух классов:

```

separate class COROUTINE creation
  make
feature {COROUTINE}
  resume (i: INTEGER) is
    -- Разбудить сопрограмму с идентификатором i и пойти спать
    do
      actual_resume (i, controller)
    end
feature {NONE} -- Implementation
  controller: COROUTINE_CONTROLLER
  identifier: INTEGER
  actual_resume (i: INTEGER; c: COROUTINE_CONTROLLER) is
    -- Разбудить сопрограмму с идентификатором i и пойти спать.
    -- (Реальная работа resume).
    do
      c.set_next (i); request (c)
    end
  request (c: COROUTINE_CONTROLLER) is
    -- Запрос возможного повторного пробуждения от c
    require
      c.is_next (identifier)
    do
      -- Действия не нужны
    end
feature {NONE} -- Создание
  make (i: INTEGER; c: COROUTINE_CONTROLLER) is
    -- Присвоение i идентификатору и с контроллеру
    do
      identifier := i
      controller := c
    end
end
separate class COROUTINE_CONTROLLER feature {NONE}
  next: INTEGER
feature {COROUTINE}
  set_next (i: INTEGER) is
    -- Выбор i в качестве следующей пробуждаемой сопрограммы
    do
      next := i
    end
  is_next (i: INTEGER): BOOLEAN is
    -- Является ли i индексом следующей пробуждаемой сопрограммы?
    do
      Result := (next = i)
    end
end

```

Одна или несколько сопрограмм будут разделять один контроллер сопрограмм, создаваемый не приведенной здесь однократной функцией (см. У12.10). У каждой сопрограммы имеется целочисленный идентификатор. Чтобы возобновить сопрограмму с идентификатором i, процедура resume с помощью actual_resume установит атрибут next контроллера в i, а затем приостановится, ожидая выполнения предусловия next = j, в котором j - это идентификатор самой сопрограммы. Это и обеспечит требуемое поведение.

Хотя это выглядит как обычная параллельная программа, данное решение гарантирует (в случае, когда у всех сопрограмм разные идентификаторы), что в каждый момент сможет выполняться лишь одна сопрограмма, что делает ненужным назначение более одного физического ЦПУ. (Контроллер мог бы использовать собственное ЦПУ, но его действия настолько просты, что этого не

следует делать.)

Обращение к целочисленным идентификаторам необходимо, поскольку передача `resume` аргумента типа `COROUTINE`, т. е. сепаратного типа, вызвала бы блокировку. На практике можно воспользоваться объявлениями `unique`, чтобы не задавать эти идентификаторы вручную. Такое использование целых чисел имеет еще одно интересное следствие: если мы допустим, чтобы две или более сопрограмм имели одинаковые идентификаторы, то при наличии одного ЦПУ получим механизм недетерминированности: вызов `resume (i)` позволит перезапустить любую сопрограмму с идентификатором `i`. Если же ЦПУ будет много, то вызов `resume (i)` позволит параллельно выполнятся всем сопрограммам с идентификатором `i`.

Таким образом, у приведенной схемы двойной эффект: в случае одного ЦПУ она обеспечивает работу механизма сопрограмм, а в случае нескольких ЦПУ - механизма управления максимальным числом одновременно активных процессов определенного вида.

Система управления лифтом

Следующий пример демонстрирует случай использования ОО-технологии и определенного в этой лекции механизма для получения привлекательной децентрализованной управляемой событиями архитектуры для некоторого приложения реального времени.

В этом примере описывается ПО управления системой нескольких лифтов, обслуживающих много этажей. Предлагаемый ниже проект объектно-ориентирован до фанатичности. Каждый сколь нибудь существенный компонент физической системы - например, кнопка с номером этажа в кабине лифта - отображается в свой сепаратный класс. Каждый соответствующий объект, такой как кнопка, имеет свой собственный поток управления (процессор). Тем самым мы приближаемся к процитированному в начале лекции пожеланию Мильнера сделать все объекты параллельными. Преимущество такой системы в том, что она полностью управляет событиями, не требуя никаких циклов для постоянной проверки состояний объектов (например, нажата ли кнопка).

Тексты классов, приведенные ниже, являются лишь набросками, но они дают хорошее представление о том, каким будет полное решение. В большинстве случаев мы не приводим процедуры создания.

Данная реализация примера с лифтом, приспособленная к показу управления на экранах нескольких компьютеров через Интернет (а не для реальных лифтов), была использована на нескольких конференциях для демонстрации ОО-механизмов параллельности и распределенности.

Класс `MOTOR` описывает мотор, связанный с одной кабиной лифта и интерфейс с механическим оборудованием:

```
separate class MOTOR feature {ELEVATOR}
  move (floor: INTEGER) is
    -- Переместиться на этаж floor и сообщить об этом
    do
      "Приказать физическому устройству переместится на floor"
      signal_stopped (cabin)
    end
  signal_stopped (e: ELEVATOR) is
    -- Сообщить, что лифт остановился на этаже e
    do
      e.record_stop (position)
    end
  feature {NONE}
    cabin: ELEVATOR
    position: INTEGER is
      -- Текущий этаж
      do
        Result := "Текущий этаж, считанный с физических датчиков"
      end
  end
```

Процедура создания этого класса должна связать кабину лифта `cabin` с мотором. В классе `ELEVATOR` имеется обратная информация: с помощью атрибута `puller` указывается мотор, перемещающий данный лифт.

Причиной для выделения лифта и его мотора как сепаратных объектов является желание уменьшить "зернистость" запираний: сразу после того, как лифт пошлет запрос своему мотору, он станет готов, благодаря политике ожидания по необходимости, к приему запросов от кнопок внутри и вне кабины. Он будет рассинхронизирован со своим мотором до получения вызова процедуры `record_stop` через процедуру `signal_stopped`. Экземпляр класса `ELEVATOR` будет только на очень короткое время зарезервирован вызовами от объектов классов `MOTOR` или `BUTTON`.

```
separate class ELEVATOR creation
  make
feature {BUTTON}
  accept (floor: INTEGER) is
    -- Записать и обработать запрос на переход на floor
    do
      record (floor)
      if not moving then process_request end
    end
  feature {MOTOR}
    record_stop (floor: INTEGER) is
      -- Записать информацию об остановке лифта на этаже floor
      do
        moving := false; position := floor; process_request
      end
  feature {DISPATCHER}
```

```

position: INTEGER
moving: BOOLEAN
feature {NONE}
puller: MOTOR
pending: QUEUE [INTEGER]
    -- Очередь ожидающих запросов
    -- (каждый идентифицируется номером нужного этажа)
record (floor: INTEGER) is
    -- Записать запрос на переход на этаж floor
do
    "Алгоритм вставки запроса на floor в очередь pending"
end
process_request is
    -- Обработка очередного запроса из pending, если такой есть
local
    floor: INTEGER
do
    if not pending.empty then
        floor := pending.item
        actual_process (puller, floor)
        pending.remove
    end
end
actual_process (m: separate MOTOR; floor: INTEGER) is
    -- Приказать m переместится на этаж floor
do
    moving := True; m.move (floor)
end
end

```

Имеются кнопки двух видов: кнопки на этажах, нажимаемые для вызова лифта на данный этаж, и кнопки внутри кабины, нажимаемые для перемещения лифта на соответствующий этаж. Эти два вида кнопок посылают разные запросы: запросы кнопок, расположенных внутри кабины, направляются этой кабине, а запросы кнопок на этажах могут обрабатываться любым лифтом, и поэтому они посылаются объекту-диспетчеру, опрашивающему разные лифты для выбора того, кто будет выполнять этот запрос. (Мы не приводим реализацию этого алгоритма выбора, поскольку это не существенно для данного рассмотрения, то же относится и к алгоритмам, используемым лифтами для управления их очередями запросов pending в классе ELEVATOR).

В классе FLOOR_BUTTON предполагается, что на каждом этаже имеется только одна кнопка. Нетрудно изменить этот проект так, чтобы поддерживались две кнопки: одна для запросов на движение вверх, а другая - вниз.

Удобно, хотя и не очень существенно, иметь общего родителя BUTTON для классов, представляющих оба вида кнопок. Напомним, что компоненты, экспортируемые ELEVATOR в BUTTON, также экспортируются в соответствии со стандартными правилами скрытия информации в оба потомка этого класса:

```

separate class BUTTON feature
    target: INTEGER
end
separate class CABIN_BUTTON inherit BUTTON feature
    cabin: ELEVATOR
    request is
        -- Послать своему лифту запрос на остановку на этаже target
    do
        actual_request (cabin)
    end
    actual_request (e: ELEVATOR) is
        -- Захватить e и послать запрос на остановку на этаже target
    do
        e.accept (target)
    end
end
separate class FLOOR_BUTTON inherit
    BUTTON
feature
    controller: DISPATCHER
    request is
        -- Послать диспетчеру запрос на остановку на этаже target
    do
        actual_request (controller)
    end
    actual_request (d: DISPATCHER) is
        -- Послать d запрос на остановку на этаже target
    do
        d.accept (target)
    end
end

```

Вопрос о включении и выключении света в кнопках здесь не рассматривается. Нетрудно добавить вызовы подпрограмм, которые будут этим заниматься.

Наконец, вот класс DISPATCHER. Чтобы разработать алгоритм выбора лифта в процедуре accept, потребуется разрешить ей

доступ к атрибутам position и moving класса ELEVATOR, которые в полной системе должны быть дополнены булевским атрибутом going_up (движется_вверх). Такой доступ не вызовет никаких проблем, поскольку наш проект гарантирует, что объекты класса ELEVATOR никогда не резервируются на долгое время.

```

separate class DISPATCHER creation
  make
feature {FLOOR_BUTTON}
  accept (floor: INTEGER) is
    -- Обработка запроса о посылке лифта на этаж floor
    local
      index: INTEGER; chosen: ELEVATOR
    do
      "Алгоритм определения лифта, выполняющего запрос для этажа floor"
      index := "Индекс выбранного лифта"
      chosen := elevators @ index
      send_request (chosen, floor)
    end
feature {NONE}
  send_request (e: ELEVATOR; floor: INTEGER) is
    -- Послать лифту e запрос на перемещение на этаж floor
    do
      e.accept (floor)
    end
  elevators: ARRAY [ELEVATOR]
feature {NONE} -- Создание
  make is
    -- Настройка массива лифтов
    do
      "Инициализировать массив лифтов"
    end
end

```

Сторожевой механизм

Как и предыдущий, следующий пример показывает применимость нашего механизма к задачам реального времени. Он также хорошо иллюстрирует понятие дуэли.

Мы хотим дать возможность некоторому объекту вызывать некоторую процедуру action при условии, что этот вызов будет прерван и булевскому атрибуту failed будет присвоено значение истина (true), если процедура не завершит свое выполнение через t секунд. Единственным доступным средством измерения времени является процедура wait (t), которая будет выполняться в течение t секунд.

Приведем решение, использующее дуэль. Класс, которому нужен указанный механизм, будет наследником класса поведения TIMED и предоставит эффективную версию процедуры action, отложенной в классе TIMED. Чтобы разрешить action выполнятся не более t секунд, достаточно вызвать timed_action (t). Эта процедура запускает сторожа (экземпляр класса WATCHDOG), который выполняет wait (t), а затем прерывает клиента. Если же сама процедура action завершится в предписанное время, то сам клиент прервет сторожа. Отметим, что в приведенном классе у всех процедур с аргументом $t: REAL$ имеется предусловие $t \geq 0$, опущенное для краткости.

```

deferred class TIMED inherit
  CONCURRENCY
feature {NONE}
  failed: BOOLEAN; alarm: WATCHDOG
  timed_action (t: REAL) is
    -- Выполняет действие, но прерывается после t секунд, если не завершится
    -- Если прерывается до завершения, то устанавливает failed в true
    do
      set_alarm (t); unset_alarm (t); failed := False
    rescue
      if is_concurrency_interrupt then failed := True end
    end
  set_alarm (t: REAL) is
    -- Выдает сигнал тревоги для прерывания текущего объекта через t секунд
    do
      -- При необходимости создать сигнал тревоги:
      if alarm = Void then create alarm end
      yield; actual_set (alarm, t); retain
    end
  unset_alarm (t: REAL) is
    -- Удалить последний сигнал тревоги
    do
      demand; actual_unset (alarm); wait_turn
    end
  action is
    -- Действие, выполняемое под управлением сторожа
  deferred
end
feature {NONE} -- Реальный доступ к сторожу
  actual_set (a: WATCHDOG; t: REAL) is

```

```

-- Запуск а для прерывания текущего объекта после t секунд
do
    a.set (t)
end
...Аналогичная процедура actual_unset предоставается читателю...
feature {WATCHDOG} -- Операция прерывания
    stop is
-- Пустое действие, чтобы позволить сторожу прервать вызов timed_action
    do -- Nothing end
end
separate class
    WATCHDOG
feature {TIMED}
set (caller: separate TIMED; t: REAL) is
    -- После t секунд прерывает caller;
    -- если до этого прерывается, то спокойно завершается
    require
        caller_exists: caller /= Void
    local
        interrupted: BOOLEAN
    do
        if not interrupted then wait (t); demand; callerl stop; wait_turn end
    rescue
        if is_concurrency_interrupt then interrupted:= True; retry end
    end
unset is
    -- Удаляет сигнал тревоги (пустое действие, чтобы дать
    -- клиенту прервать set)
    do -- Nothing end
feature {NONE}
    early_termination: BOOLEAN
end

```

За каждым использованием yield должно, как это здесь сделано, следовать retain в виде: yield; "Некоторый вызов"; retain. Аналогично каждое использование demand (или insist) должно иметь вид: demand; "Некоторый вызов "; wait_turn. Для принудительного выполнения этого правила можно использовать классы поведения.

Организация доступа к буферам

Завершим рассмотрение примером ограниченного буфера, уже встречавшегося при описании механизма параллельности. Этот класс можно объявить как separate class BOUNDED_BUFFER [G] inherit BOUNDED_QUEUE [G] end в предположении, что имеется соответствующий последовательный класс BOUNDED_QUEUE .

Чтобы для сущности q типа BOUNDED_BUFFER [T] выполнить вызов вида q.remove, его нужно включить в подпрограмму, использующую q как формальный аргумент. Для этой цели полезно было бы разработать класс BUFFER_ACCESS (**ДОСТУП_К_БУФЕРУ**), инкапсулирующий понятие ограниченного буфера, а классы приложений могли бы стать его наследниками. В написании такого класса поведения нет ничего трудного. Он дает хороший пример того, как можно инкапсулировать сепаратные классы (непосредственно выведенные из последовательных, таких как BOUNDED_QUEUE) для облегчения их непосредственного использования в параллельных приложениях.

```

indexing
    description: "Инкапсуляция доступа к ограниченным буферам"
class BUFFER_ACCESS [G] is
    put (q: BOUNDED_BUFFER [G]; x: G) is
        -- Вставляет x в q, ожидая, при необходимости свободного места
        require
            not q.full
        do
            q.put (x)
        ensure
            not q.empty
        end
    remove (q: BOUNDED_BUFFER [G]) is
        -- Удаляет элемент из q, ожидая, при необходимости его появления
        require
            not q.empty
        do
            q.remove
        ensure
            not q.full
        end
    item (q: BOUNDED_BUFFER [G]): G is
        -- Старейший неиспользованный элемент
        require
            not q.empty
        do
            Result := q.item
        ensure
            not q.full
        end

```

end

О правилах доказательств

Этот параграф предназначен для читателей, склонных к математике, остальные могут сразу перейти к обсуждению в следующем разделе. Хотя основные идеи можно понять и без формального освоения теории языков программирования, полное понимание требует знакомства хотя бы с основами этой теории, изложенными, например, в книге [М 1990], обозначения которой здесь будут использоваться.

Основное математическое свойство последовательного ОО-вычисления было полуформально приведено при обсуждении Проектирования по Контракту:

{INV and pre} body {INV and post}

где pre, post и body - это предусловие, постусловие и тело программы, а INV - это инвариант класса. При подходящей аксиоматизации исходных инструкций оно может послужить основой полной формальной аксиоматической семантики ОО-ПО.

Выразим это свойство более строго в виде правила доказательства вызовов. Такое правило послужит фундаментом математического изучения ОО-ПО, поскольку стержнем всякого ОО-вычисления - последовательного, как раньше, или параллельного, рассматриваемого сейчас, - являются операции вида:

t.f (... , a, ...)

вызывающие компонент f, возможно, с аргументами такими, как a, для цели t, присоединенной к объекту. Правило доказательства для последовательного случая можно содержательно сформулировать так:

Основной последовательный метод доказательства

Если можно доказать, что тело f, запущенное в состоянии, удовлетворяющем предусловию f, завершит работу в состоянии, удовлетворяющем постусловию, то для указанного выше вызова можно вывести то же свойство, в котором формальные аргументы заменены фактическими. Каждый неквалифицированный вызов в утверждениях вида some_boolean_property заменяется соответствующим свойством t вида t.some_boolean_property.

Например, если мы можем доказать, что конкретная реализация put в классе BOUNDED_QUEUE, запускаемая при выполнении условия not full, выдает состояние, удовлетворяющее not empty, то для любой очереди q и любого элемента a приведенное правило позволяет вывести:

{not q.full} q.put (a) {not q.empty}

Более формально основное правило доказательства можно выразить, приспособив к ОО-вычислениям известное правило Хоара (Hoare) для доказательства вызовов процедур:

$$\{ \text{INV} \wedge \bigwedge_{p \in \text{Pre}(r)} p \} \text{ Body } (r) \{ \text{INV} \wedge \bigwedge_{q \in \text{Post}(r)} q \}$$

$$\{ \bigwedge_{p \in \text{Pre}(r)} p' \} \text{ Call } (r) \{ \bigwedge_{q \in \text{Post}(r)} q' \}$$

Здесь INV - инвариант класса, Pre (f) - множество предложений предусловия f, а Post (f) - множество предложений постусловия. Напомним, что утверждение является конъюнкцией набора предложений вида:

clause1; ...; clauseN

Знаки больших "и" означают конъюнкцию всех предложений. Фактические аргументы f явно не указаны, но выражения со штрихами, такие как t.q', означают подстановку фактических аргументов вызова вместо формальных аргументов f.

Из соображений краткости правило приведено в форме, не поддерживающей доказательства вызовов рекурсивных процедур. Однако добавление такой поддержки никак не повлияет на наше обсуждение. Детали, связанные с рекурсией, можно найти в [М 1990].

Причина, по которой предложения утверждения рассматриваются по отдельности, а затем соединяются посредством "и", заключается в том, что в таком виде правило подготовлено для перехода к описанному ниже случаю сепаратных вызовов при параллелизме. Для подготовки к параллельному случаю также интересно, что инвариант INV учитывается при доказательстве для тела подпрограммы (в числителе правила), но невидим при доказательстве вызова (в знаменателе).

Что изменится в параллельном случае? В предложении предусловия может появиться ожидание только для предусловий в виде t.cond, где t - это сепаратная сущность, являющаяся формальным аргументом подпрограммы, содержащей рассматриваемый вызов. В подпрограмме вида:

```
f (... , a: T, ...) is
    require
        clause1; clause2; ...
    do
        ...
    done
```

end

любое из предложений предусловия, не содержащее сепаратный вызов на сепаратном формальном аргументе, является условием корректности: любой клиент должен обеспечить выполнение этого условия перед каждым вызовом, иначе вызов будет ошибочным. Всякое предложение предусловия, включающее вызов вида `a.some_condition`, в котором `a` является сепаратным формальным аргументом, является условием ожидания, которое будет блокировать вызов до своего выполнения.

Эти наблюдения можно выразить в виде правила доказательства, которое заменяет для сепараторного вычисления предыдущее последовательное правило:

$$\{ \text{INV} \wedge \bigwedge_{p \in \text{Pre}(r)} p \} \text{ Body}(r) \{ \text{INV} \wedge \bigwedge_{q \in \text{Post}(r)} q \}$$

$$\{ \bigwedge_{p \in \text{Nonsep_Pre}(r)} p' \} \text{ Call}(r) \{ \bigwedge_{q \in \text{Nonsep_Post}(r)} q' \}$$

где `Nonsep_Pre(f)` - множество предложений предусловия `f`, не содержащих сепаратных вызовов, а `Nonsep_Post(f)` - аналогичное множество для постусловия.

Это правило частично отражает суть параллельного вычисления. Для доказательства правильности программы все еще требуется доказать те же условия (в числите), что и в последовательном правиле. Но следствия этого для свойств вызова различны: клиенту нужно обеспечивать меньше свойств перед вызовом, поскольку, как это подробно обсуждено выше, по меньшей мере бесполезно пытаться обеспечить выполнение сепаратной части предусловия; но и на выходе мы получаем меньше гарантированных утверждений. Первое отличие является хорошей новостью для клиента, а последнее - плохой.

Таким образом, сепаратные предложения предусловий и постусловий объединяются с инвариантами как свойства, которые должны быть включены во внутреннее доказательство правильности тела подпрограммы, но они не используются явно как свойства ее вызова.

Это правило также восстанавливает симметрию между предусловиями и постусловиями, дополняя обсуждение, в котором выделялась роль предусловий.

Резюме параллельного механизма

Приведем теперь точное описание параллельных конструкций, введенных в предыдущих разделах. В этом разделе не будет нового материала, он служит только для ссылок и может быть пропущен при первом чтении. Описание состоит из 4-х частей: синтаксиса, ограничений правильности, семантики и библиотечных механизмов. Все это является расширением последовательных механизмов, развитых в предыдущих лекциях.

Синтаксис

Синтаксис расширяется за счет введения только одного нового ключевого слова `separate`.

Объявление сущности или функции, которое в обычном случае выглядит как:

`x : TYPE`

сейчас может также иметь вид:

`x : separate TYPE`

Кроме этого, объявление класса, которое обычно начиналось с `class C`, `deferred class C` или `expanded class C`, сейчас может также иметь вид `separate class C`. В этом случае `C` называется сепаратным классом. Из этого синтаксического соглашения вытекает, что у класса может быть не более одного определяющего ключевого слова, например, он не может быть одновременно отложенным и сепаратным. Как и в случае развернутости и отложенности, свойство сепараторности класса не наследуется: класс является или не является сепаратным в соответствии с его собственным объявлением независимо от статуса сепараторности его родителя.

Тип `T` называется сепаратным, если он основан на сепаратном классе или определен как `separate T` (если `T` сам сепаратный, то это не ошибка, хотя и избыточно, действует то же соглашение, что и для развернутых типов). Сущность или функция являются сепаратными, если имеют сепаратный тип. Выражение является сепаратным, если это сепаратная сущность или вызов сепаратной сущности. Вызов или инструкция создания являются сепаратными, если их цель (выражение) является сепаратной. Предложение предусловия сепаратно, если оно содержит сепаратный вызов (чья цель, в соответствии с приведенным далее правилом, является формальным аргументом).

Ограничения

Правило корректной сепараторности состоит из четырех частей и управляет правильностью сепаратных вызовов:

1. если источник присоединения (в инструкции присваивания или при передаче аргументов) является сепаратным, то его целевая сущность также должна быть сепаратной;
2. если фактический аргумент сепаратного вызова имеет тип ссылки, то соответствующий формальный аргумент должен быть объявлен как сепаратный;

3. если источник присоединения является результатом сепаратного вызова функции, возвращающей тип ссылки, то цель должна быть объявлена как сепаратная;
4. если фактический аргумент или результат сепаратного вызова имеет развернутый тип, то его базовый класс не может содержать непосредственно или опосредованно никакой несепаратный атрибут ссылочного типа.

Ранее не приведенное, простое правило корректности для типов утверждает: в описании `separate` TYPE базовый класс для TYPE не должен быть ни отложенным, ни развернутым.

Для правильности сепаратного вызова его целью должен быть формальный аргумент подпрограммы, включающей этот вызов.

Если утверждение содержит вызов функции, то любой фактический аргумент этого вызова, если он сепаратный, должен быть формальным аргументом объемлющего класса (правило аргументов утверждения).

Семантика

Каждый объект обрабатывается некоторым процессором - его обработчиком. Если цель `t` инструкции создания не является сепаратной, то новый объект будет обрабатываться тем же процессором, что и создавший его объект. Если `t` сепаратна, то для обработки нового объекта будет назначен новый процессор.

Будучи создан, объект в дальнейшем находится в одном из двух состояний: свободен или зарезервирован (занят). Он свободен, если в данный момент не выполняется никакой его компонент и никакой сепаратный клиент не выполняет подпрограмму, использующую в качестве фактического аргумента присоединенную к нему сепаратную ссылку.

Процессор может находиться в трех состояниях: "не занят", "занят" и "приостановлен". Он "занят", если выполняет программу, чьей целью является обрабатываемый им объект. Он переходит в состояние "приостановлен" при попытке выполнить неуспешный вызов (это определено ниже), чья цель - обрабатываемый этим процессором объект.

Семантика вызова меняется только, если один или более вовлеченный в него элемент - цель или фактический аргумент - является сепаратным. Мы будем предполагать, что вызов имеет общий вид `t.f(..., s, ...)`, в котором `f` - это подпрограмма. (Если `f` является атрибутом, то для простоты будем считать, что имеется вызов неявной функции, возвращающей значение этого атрибута.)

Пусть вызов выполняется как часть выполнения подпрограммы некоторого объекта `C_OBJ`, обработчик которого на этом шаге может быть только в состоянии "занят". Основным является следующее понятие:

Определение: выполнимый вызов (satisfiable call)

При отсутствии компонентов `CONCURRENCY` (описываемых далее) вызов подпрограммы `f`, выполняемый от имени объекта `C_OBJ`, является выполнимым тогда и только тогда, когда каждый сепаратный фактический аргумент, присоединенный к некоторому сепаратному объекту `A_OBJ`, чей соответствующий формальный аргумент используется подпрограммой как цель хотя бы одного вызова, удовлетворяет двум следующим условиям:

- **S1**`A_OBJ` свободен или зарезервирован (обработчиком) `C_OBJ`.
- **S2**Каждое сепаратное предложение предусловия `f` истинно для `A_OBJ` и заданных фактических аргументов.

Если процессор исполняет выполнимый вызов, то этот вызов называется успешным и осуществляется немедленно; `C_OBJ` остается зарезервированным, а его процессор остается в состоянии "занят", каждый объект `A_OBJ` становится зарезервированным, цель остается зарезервированной, обработчик цели становится занятым и начинает выполнение подпрограммы вызова. Когда этот вызов завершается, обработчик цели возвращается в свое предыдущее состояние ("не занят" или "приостановлен"), и каждый из объектов `A_OBJ` также возвращается в свое предыдущее состояние (свободен или зарезервирован (обработчиком) `C_OBJ`).

Если вызов не является выполнимым, то он называется неуспешным; (обработчик) `C_OBJ` переходит в состояние "приостановлен". Эта попытка вызова никак не влияет на его цель и фактические аргументы. Если в некоторый момент один или больше ранее неуспешных вызовов становятся выполнимыми, то процессор выбирает один из них для выполнения (делает успешным). По умолчанию из нескольких выполнимых вызовов выбирается тот, который ожидал дольше других.

Последнее изменение в семантике связано с ожиданием по необходимости: если клиент запускает на некотором сепаратном объекте один или более вызовов и выполняет на этом объекте вызов запроса, то этот вызов может выполняться лишь после завершения всех ранее начавшихся, а все остальные операции клиента будут ждать завершения этого вызова запроса. (Мы уже отмечали, что оптимизирующая реализация может применять это правило только к запросам, возвращающим развернутый результат.) Во время ожидания завершения этих вызовов клиент остается в состоянии "зарезервирован".

Библиотечные механизмы

Компоненты класса `CONCURRENCY` позволяют в некоторых случаях считать, что условие выполнимости вызова `S1` имеет место, даже если `A_OBJ` зарезервирован другим объектом ("обладателем") в случае, когда `C_OBJ` ("претендент") вызвал `demand` или `insist`; если в результате этого вызов становится выполнимым, то обладатель получает некоторое исключение. Это может произойти только, если обладатель находится в состоянии "готов уступить", в которое можно попасть, вызвав `yield`.

Для возврата в предопределенное состояние "неуступчивости" обладатель может выполнить `retain`; булевский запрос `yielding` позволяет узнать текущее состояние. Состояние претендента задается числовым запросом `Challenging`, который может иметь значения `Normal`, `Demanding` или `Insisting`.

Для возврата в предопределенное состояние `Normal` претендент может выполнить `wait_turn`. Различие между `demand` и `insist` заключается в том, что, если обладатель не находится в состоянии `yielding`, то при `demand` претендент получит исключение, а при `insist` он далее просто ожидает, как и при `wait_turn`.

Когда эти механизмы возбуждают исключение в обладателе или в претенденте, то булевский запрос `is_concurrency_exception` из класса `EXCEPTIONS` имеет значение "истина" (`true`).

Обсуждение

В заключение перечислим критерии, которыми мы руководствовались при создании параллельного ОО-механизма. Как будет видно, в ряде случаев для полного успеха нужно проделать еще определенную работу. Вот основные цели:

- минимальность механизма;
- полное использование наследования и других ОО-методов;
- совместимость с Проектированием по Контракту;
- доказуемость;
- поддержка различия между командами и запросами;
- применимость ко многим видам параллельности;
- поддержка программирования сопрограмм;
- адаптируемость с помощью библиотек;
- поддержка использования непараллельного ПО;
- поддержка устранения блокировок.

Бросим также последний взгляд на вопрос поочередного доступа к объекту.

Минимальность механизма

ОО-конструирование ПО - это богатая и мощная парадигма, которая, как отмечалось в начале этой лекции, интуитивно выглядит готовой для поддержания параллельности.

Это существенно для достижения цели наименьшего возможного расширения. Минимализм здесь - это не просто вопрос проектирования красивого языка. Если параллельное расширение не минимально, то некоторые параллельные конструкции оказываются излишними для ОО-конструкций или будут с ними конфликтовать, делая задачу проектировщика более трудной или невыполнимой. Чтобы избежать такой ситуации, нам нужно найти наименьший синтаксический и семантический эпсилон, который предоставит нашим ОО-программам возможность параллельного исполнения.

Представленное в предыдущих параграфах расширение действительно является синтаксически минимальным, поскольку нельзя добавить меньше чем одно ключевое слово.

Полное использование наследования и других ОО-методов

Было бы недопустимо построить параллельный ОО-механизм, не использующий всех преимуществ ОО-метода, в частности наследование. Мы заметили, что "аномалия наследования" и другие потенциальные конфликты внутренне не присущи параллельному ОО-проектированию. Они являются следствиями специфического выбора параллельных механизмов - активных объектов, синхронизации с помощью путевых выражений. По этой причине мы отказались от этих конструкций и сохранили наследование.

Мы неоднократно видели, как можно использовать наследование для создания классов поведения высокого уровня (таких как PROCESS), описывая общие образцы, наследуемые их потомками. Большинство из приведенных примеров невозможно было бы реализовать без множественного наследования.

Отметим также, что скрытие информации играет центральную роль среди всех ОО-методов.

Совместимость с Проектированием по Контракту

Крайне важно сохранение систематического подхода, выраженного в принципах Проектирования по Контракту. Результаты этой лекции основывались на изучении утверждений и их существования в параллельном контексте.

По ходу изучения обнаружилось поразительное свойство. Парадокс параллельных предусловий заставил нас определить новую семантику утверждений для параллельного случая, что подтвердило важную роль утверждений в параллельном механизме.

Поддержка различия между командами и запросами

В предыдущих лекциях был обоснован важный принцип различия команд и запросов. Он предписывает не смешивать команды (процедуры), изменяющие объекты, и запросы (функции и атрибуты), возвращающие информацию, что устраняет функции с побочными эффектами.

Существовала точка зрения, что в параллельном контексте этот принцип не выполняется, например нельзя написать:

```
next_element := buffer.item  
buffer.remove
```

и быть уверенным в том, что удаленный во втором вызове элемент будет тем же, что и присвоенный в первой строке переменной next_item. Другой клиент мог испортить дело, получив доступ к буферу между этими двумя инструкциями. Такого рода примеры часто использовались для доказательства необходимости функций с побочным эффектом, в данном случае - функции get, возвращающей элемент и удаляющей его из контейнера в качестве побочного эффекта.

Этот аргумент заведомо ложен. В нем смешаны два понятия: исключительный доступ и спецификация программы. Понятия этой лекции позволяют получить исключительный доступ, не жертвуя принципом Команд и Запросов. Достаточно включить эти две инструкции, заменив buffer на b, в некоторую процедуру с формальным аргументом b, а затем вызывать эту процедуру с атрибутом buffer в качестве аргумента. Или, если вам не требуется, чтобы обе операции применялись к одному элементу, а нужно минимизировать время захвата общего ресурса, то напишите две отдельные подпрограммы. Такая гибкость важна для разработчиков. Она обеспечивается благодаря простому механизму исключительного доступа, не связанному с наличием или отсутствием побочного эффекта у функций.

Применимость ко многим видам параллельности

Общим критерием при разработке параллельного механизма была необходимость поддержки многих различных видов параллельности: разделяемой памяти, многозадачности, сетевого программирования, программирования для архитектуры клиент-сервер, распределенной обработки, реального времени.

Нельзя ожидать, что один языковый механизм обеспечит ответы на все вопросы из таких разных прикладных областей. Но должна быть возможность приспособления его к потребностям всех перечисленных видов параллельности. Это достигается за счет применения абстрактного понятия процессора и использования специальных средств (файлов управления параллельностью, библиотек) для адаптации к любой доступной архитектуре оборудования.

Адаптируемость с помощью библиотек

В течение ряда лет было предложено много механизмов параллельности, некоторые наиболее известные из них были рассмотрены в начале этой лекции. У каждого из них есть свои сторонники, и каждый может предложить наилучший подход к какой-либо проблемной области.

Поэтому важно, чтобы предлагаемый механизм смог бы поддержать хотя бы некоторые из известных механизмов. Более точно, предлагаемое решение должно позволять запрограммировать в его терминах другие параллельные конструкции.

И здесь ОО-метод проявляется с лучшей стороны. Один из наиболее важных аспектов этого метода состоит в том, что он поддерживает создание специальных библиотек для широко используемых схем. Имеющиеся для построения библиотек средства (классы, утверждения, ограниченная и неограниченная универсальность, множественное наследование, отложенные классы и др.) позволяют выразить многие параллельные механизмы в виде библиотечных компонентов. Некоторые примеры таких инкапсулированных механизмов уже были приведены в этой лекции (например, класс PROCESS и класс поведения для замков), в упражнениях будут предложены дополнительные примеры.

Вполне вероятно, что, используя и дополняя базисные средства, проектировщики построят множество библиотек, поддерживающих параллельные модели, удовлетворяющие специфическим требованиям и вкусам.

Мы также видели, как, используя такие библиотечные классы, как CONCURRENCY, можно уточнять базовую схему, задаваемую предложенным параллельным механизмом языка.

Поддержка программирования сопрограмм

Как специальный случай, сопрограммы представляют собой вид квазипараллельности, интересный как сам по себе (в частности, в сфере моделирования), так и как простой тест на применимость нашего механизма, поскольку общее решение должно легко адаптироваться к крайним случаям. Мы видели, как можно выразить сопрограммы, основываясь на общем параллельном механизме и используя ОО-средства построения библиотек.

Поддержка использования непараллельного ПО

Необходимо поддерживать возможности повторного использования существующего непараллельного ПО, особенно библиотек переиспользуемых программных компонентов.

Мы уже видели, насколько плавно можно переходить от последовательных классов (таких как BOUNDED_QUEUE) к их параллельным двойникам (таким как BOUNDED_BUFFER; надо просто написать `separate class BOUNDED_BUFFER [G] inherit BOUNDED_QUEUE [G] end.`) Этот результат несколько ослабляется тем, что часто желательно иметь инкапсулированные классы, такие как наш BUFFER_ACCESS. Однако такая инкапсуляция представляется полезной и, по-видимому, является неизбежным следствием семантического различия между последовательными и параллельными вычислениями. Отметим также, что такие классы-оболочки пишутся достаточно легко.

Поддержка устранения блокировок

Один из вопросов, требующий дальнейшей работы, - это гарантирование отсутствия блокировок.

Потенциальная возможность блокировки - это факт параллельной жизни. Например, любой механизм, который можно использовать для программирования семафоров (а механизм, недостаточный для их реализации, выглядел бы весьма подозрительно), может вызвать блокировку, поскольку семафоры trivialно допускают такую возможность.

Частичное решение состоит в использовании механизмов инкапсуляции высокого уровня. Например, набор классов, инкапсулирующих семафоры, вроде представленного выше для замков, должен поступать вместе с классами поведения, которые автоматически обеспечивают операцию `free` после каждой `reserve`, гарантируя тем самым отсутствие блокировок для приложений, которые следуют рекомендованной практике наследования классов поведения. Мой опыт показывает, что это наилучший рецепт для устранения блокировок.

Этот подход, конечно, может оказаться недостаточным. Можно изобрести простые правила недопущения блокировок, автоматически проверяемые статическими средствами. Одно такое правило - принцип визитной карточки - приведено выше (из-за страха, что оно налагает чересчур сильные ограничения, оно представлено в виде методологического принципа, а не в виде правила языка). Но одного этого правила мало.

Допускается ли одновременный доступ?

Заключительное замечание относится к одному из важных свойств предложенного подхода - требованию, чтобы в каждый момент времени не более чем один клиент мог иметь доступ к каждому объекту-поставщику. Для VIP-клиентов предоставляется механизм дуэлей.

Причина запрета понятна: если бы некоторый клиент мог бы прервать в любой момент выполнение программы, запущенной его претендентом, то была бы потеряна возможность рассуждать о классах (используя свойства вида `{INV and pre} body {INV and post}`), поскольку прерванный претендент мог бы оставить объект в произвольном состоянии.

Такой недостаток исчез бы, если бы мы разрешили претендентам выполнять только программы весьма специального вида: applicативные программы (в определенном в предыдущих лекциях для функций смысле), которые либо вовсе не изменяют объект, либо, изменения его, устраниют все свои изменения перед тем, как его покинуть. Для этого нужен языковой механизм, позволяющий утверждать, что некоторая программа applicативна, и компиляторы, обеспечивающие это свойство.

Ключевые концепции

- Параллельность и распределенность играют все возрастающую роль в большинстве областей приложения компьютеров.
- Существует много вариантов параллельности, включая многопроцессорность и мультипрограммность. Еще больше возможностей принесли Интернет, Web и брокеры запросов объектов.
- Для достижения наибольшей выгоды разработчики параллельных и распределенных приложений могут использовать фундаментальные схемы ОО-технологии: классы, инкапсуляцию, множественное наследование, отложенные классы, утверждения и т. п.
- Не требуется различать активные и пассивные объекты. По своей природе объекты способны выполнять много операций, если делать их активными, то придется ограничиться только одной операцией.
- Все главные области приложений параллельности покрываются расширением последовательных ОО-обозначений одним новым ключевым словом (*separate*).
- Каждый объект обрабатывается некоторым процессором. Процессоры - это абстрактное понятие, описывающее потоки управления; система может использовать столько процессоров, сколько ей необходимо, независимо от числа доступных физических вычислительных устройств (ЦПУ). Должна быть предоставлена возможность определять отображение процессоров на ЦПУ вне текстов программ.
- Объект, обрабатываемый отдельным процессором, называют сепаратным.
- Вызовы сепаратных целей имеют разную семантику, скорее, асинхронную, чем синхронную. Поэтому сущности, представляющие сепаратные объекты, должны объявляться таковыми с помощью ключевого слова *separate*.
- Правила совместности, из которых, в частности, следует, что сепаратная сущность не может быть присвоена несепаратной, гарантируют отсутствие "предателей" - никакая несепаратная сущность не присоединится к сепаратному объекту.
- Для получения исключительного доступа к сепаратному объекту достаточно использовать соответствующую ссылку в качестве аргумента сепаратного вызова (вызова с сепаратной целью).
- Цель сепаратного вызова в свою очередь должна быть сепаратным формальным аргументом объемлющей подпрограммы.
- Предусловия сепаратных целей не могут сохранить свою обычную семантику условий корректности (это называется "парадоксом параллельных предусловий"). Они служат условиями ожидания.
- Разработанный в этой лекции механизм охватывает многозадачность, разделение времени, многопоточность, вычисления в архитектуре клиент-сервер, распределенную обработку в сетях (в частности, в Интернет), сопрограммы и приложения реального времени.

Библиографические замечания

Описанный в этой лекции подход к параллельности вырос из доклада на конференции TOOLS EUROPE [M 1990a], а затем был пересмотрен в работе [M 1993b], из которой перенесена некоторая часть материала данной лекции (в частности, примеры). Сейчас он известен под аббревиатурой SCOOP ("Simple Concurrent Object-Oriented Programming" - Простое Параллельное ОО-Программирование). Джон Поттер (John Potter) и Гинва Жалул (Ghinwa Jalloul) разработали вариант, который включает явную инструкцию *hold* [Jalloul 1991], [Jalloul, 1994]. Ожидание по необходимости было введено Дэнисом Каромелем (Denis Caromel) [Caromel 1989], [Caromel1993].

Первая реализация описанной здесь модели была выполнена Терри Тангом (Terry Tang) и Ксавьером ле Вуршем (Xavier Le Vourch). Оба внесли свой вклад.

Хорошим учебником по традиционным подходам к параллельности является [Ben Ari 1990]. Ссылки на оригинальные работы: по семафорам [Dijkstra 1968a] (там же появилась задача об "обедающих философах"), по мониторам [Hoare 1974], по путевым выражениям [Campbell 1974]. Первоначально модель CSP была описана в [Hoare 1978]; в книге [Hoare 1985] эта модель уточнена с упором на ее математические свойства. Occam2 описан в [Inmos 1988]. Архив по CSP и Occam доступен в Оксфордском университете: <http://www.comlab.ox.ac.uk/archive/csp.html> (Я благодарен Биллу Роскоу (Bill Roscoe) из Оксфорда за помощь в понимании деталей CSP.) Другой важной математически обоснованной моделью являются CCS (Communicating Concurrent Systems - Взаимодействующие Параллельные Системы) [Milner 1989]. Мимоходом упомянутые в этой лекции метод и инструментальное средство Linda Карьеро (Carriero) и Гелернтера (Gelernter) [Carriero 1990] должны знать все, кто интересуется параллелизмом.

В специальном выпуске журнала Communications of the ACM [M 1993a] представлены многие подходы к параллельному ОО-программированию, первоначально описанные в статьях по параллелизму на различных конференциях TOOLS.

Примерно в то же время появился и другой сборник статей [Agha 1993]. Более ранняя коллективная книга [Yonezawa 1987] под редакцией Йонезава (Yonezawa) и Токоро (Tokoro) послужила катализатором для многих работ в этой области и до сих пор является хорошим источником. Среди других обзоров отметим диссертацию [Papathomas 1992] и статью [Wyatt 1992]. Еще один сборник ряда авторов [Wilson 1996] охватывает параллельные расширения C++.

Модель акторов Хьюита (Hewitt) и Агха (Agha) повлияла на многие подходы к ОО-параллельности; она описана в статье [Agha 1990] и в книге [Agha1986]. Акторы - это вычислительные агенты, похожие на активные объекты, каждый со своим собственным почтовым адресом и поведением. Они взаимодействуют друг с другом с помощью сообщений, посыпаемых по их адресам; для достижения асинхронного поведения эти сообщения буферизируются. Актор обрабатывает сообщения с помощью функций, после того как некоторое сообщение обработано, прежнее поведение актора изменяется на "замещающее поведение".

Одним из самых ранних и наиболее исследованных параллельных ОО-языков является POOL [Amerika 1989]; в POOL используется понятие активного объекта, которое в сочетании с наследованием приводит к известным проблемам. Поэтому наследование было введено в этот язык только после тщательного изучения, приведшего к разделению механизмов наследования и выделения подтипов. Проект POOL примечателен также тем, что с самого начала продемонстрировал заинтересованность в формальной спецификации языка.

Много важной работы по параллельным ОО-языкам было проведено в Японии. Уже цитированная книга [Yonezawa 1987] содержит описание нескольких важных японских достижений, таких как ABCL/1 [Yonezawa 1987a]. ОО-операционная система MUSE, разработанная в лаборатории информатики корпорации Сони (Sony), была представлена Токоро (Tokoro) и его коллегами на

конференции TOOLS EUROPE 1989 [Yokote 1989]. Термин "аномалия наследования" был введен Мацуокой (Matsuoka) и Йонезавой (Yonezawa) [Matsuoka 1993], а в последующих статьях Мацуоки и его сотрудников были предложены разные средства ее разрешения.

Работы по распределенным системам особенно активно велись во Франции, где были созданы операционная система CHORUS с описанным в [Lea 1993] ОО-расширением, язык GUIDE и система Krakowiak (Krakowiak) и др. [Balter 1991], система Шапиро (Shapiro) [Shapiro 1989]. В области программирования массивных параллельных архитектур, ориентированных на научные приложения, Жан-Марк Жезекель (Jean-Marc Jezequel) разработал систему EPEE [Jezequel 1992], [Jezequel 1996], [Guidec 1996].

Важной также оказалась работа Nierstrasz и его коллег из университета Женевы по языку Hybrid [Nierstrasz 1992], [Papathomas 1992], в котором нет двух категорий объектов (активных и пассивных), но вместо этого используется понятие потока управления, называемого активностью (activity). Основным механизмом взаимодействия является удаленный вызов процедур, синхронный или асинхронный.

Среди других важных проектов отметим DRAGOON [Atkinson 1991], в котором, как и в механизме данной лекции, для выражения синхронизации используются пред- и постусловия, и pSather [Feldman 1993], базирующийся на понятии потока и предопределенном классе MONITOR.

К этому списку следовало бы добавить еще много других работ. Более подробные обзоры были процитированы в начале этого раздела. В трудах семинаров, регулярно проходящих при конференциях ECOOP и OOPSLA, таких как [Agha 1988], [Agha 1991], [Tokoro 1992], описано многообразие исследовательских проектов, и они весьма ценные для тех, кто хочет узнать, какие проблемы исследователи считают наиболее безотлагательными.

На разных стадиях для предложенной в этой лекции работы были полезны комментарии и критические замечания многих людей. В дополнение к уже процитированным в первых двух параграфах этого раздела коллегам перечислим Мордехая Бен-Ари (Mordechai Ben-Ari), Ричарда Бельяка (Richard Bielak), Джона Бруно (John Bruno), Поля Дюбуа (Paul Dubois), Карло Гецци (Carlo Ghezzi), Петера Лёра (Peter Lohr), Дино Мандриоли (Dino Mandrioli), Жан-Марка Нерсона (Jean-Marc Nerson), Роберта Светцера (Robert Switzer) и Кима Вальдена (Kim Walden).

Упражнения

У12.1 Принтеры

Завершите определение класса PRINTER из третьего раздела этой лекции, реализующего очередь с помощью ограниченного буфера. Обратите внимание на то, что подпрограммы для работы с очередью, такие как print, не нуждаются в обработке специального "запроса на остановку" задания печати (у print в качестве предусловия может быть not j.is_stop_request).

У12.2 Почему импорт должен быть глубоким

Предположим, что доступен только механизм поверхностного (а не deep_import). Постройте пример, в котором порождалась бы некорректная структура, в которой сепаратный объект был бы присоединен к несепаратной сущности.

У12.3 "Аномалия наследования"

Предположим, что в примере BUFFER, использованном для иллюстрации "аномалии наследования" каждая подпрограмма специфицирует свое выходное состояние с помощью инструкции yield, например, как в:

```
put (x: G) is
  do
    "Добавляет x к структуре данных, представляющей буфер"
    if "Все места сейчас заняты" then
      yield full
    else
      yield partial
    end
  end
```

Напишите соответствующую схему для remove. Затем определите класс NEW_BUFFER с дополнительной процедурой remove_two (удалить_два) и покажите, что в этом классе должны быть переопределены оба наследуемых компонента (одновременно определите, какие компоненты применимы в каких состояниях).

У12.4 Устранение тупиков (проблема для исследования)

Начав с принципа визитной карточки, выясните, реально ли устранение некоторых из возможных тупиков с помощью введения правила корректности, налагающего ограничения на использование несепаратных фактических аргументов в сепаратных вызовах. Это правило должно быть разумным (т. е. оно не должно мешать широко используемым схемам), его выполнение должно поддерживаться компилятором (в частности, инкрементным компилятором) и быть легко объяснимым разработчикам.

У12.5 Приоритеты

Исследуйте, как добавить схему приоритетов к механизму дуэлей класса CONCURRENCY, сохранив совместимость вверх с семантикой, определенной для процедур yield, insist и других процедур, связанных с ними.

У12.6 Файлы и парадокс предусловия

Рассмотрите следующий простой фрагмент некоторой подпрограммы для работы с файлом:

```
f: FILE
...

```

```
if f /= Void and then f.readable then
    f.some_input_routine
        -- some_input_routine - программа, которая читает
        -- данные из файла; ее предусловием является readable
end
```

Обсудите, как, несмотря на отсутствие явной параллельности в этом примере, к нему может примениться парадокс предусловий. (**Указание:** файл - это сепаратная постоянная структура, поэтому другой интерактивный пользователь или другая программная система могут получить к нему доступ между выполнением двух операций из указанного фрагмента.) К чему в данном случае может привести эта проблема, каковы возможные пути ее решения?

у12.7 Замки (Locking)

Перепишите класс LOCKING_PROCESS как наследника класса PROCESS.

у12.8 Бинарные семафоры

Напишите один или несколько классов, реализующих понятие бинарного семафора. (**Указание:** начните с классов, реализующих замки.) Как было предложено в конце обсуждения замков, разработайте классы поведения высокого уровня, используемые с помощью наследования и предоставляемые образцы для операций reserve и free.

у12.9 Целочисленные семафоры

Напишите один или несколько классов, реализующих понятие целочисленного семафора.

у12.10 Контроллер сопрограмм

Завершите реализацию сопрограмм, объяснив, как создать для них контроллер.

у12.11 Примеры сопрограмм

В представлении языка Simula имеется несколько примеров сопрограмм. Примените классы сопрограмм из данной лекции для реализации этих примеров.

у12.12 Лифты

Завершите пример с лифтами, добавив все процедуры создания и все опущенные алгоритмы, в частности алгоритм для выбора запросов кнопок на этажах.

у12.13 Сторожа и принцип визитной карточки

Покажите, что процедура set класса WATCHDOG нарушает принцип визитной карточки. Объясните, почему в этом случае все в порядке.

у12.14 Однократные подпрограммы и параллельность

Какова подходящая семантика для однократных подпрограмм в параллельном контексте (как программ, исполняемых один раз при каждом запуске системы, или как программ, исполняемых не более одного раза каждым процессором)?

¹⁾ В момент написания этой книги понятия Web-сервиса еще не существовало.

Основы объектно-ориентированного проектирования

13. Лекция: Сохранение объектов и базы данных (БД)

Выполнение ОО-приложения означает создание и манипулирование некоторым числом объектов. Что происходит с этими объектами, когда текущее вычисление завершается? Часть объектов исчезает с завершением сессии. Многим приложениям нужны также и сохраняемые объекты, остающиеся между сессиями. Сохраняемые объекты могут использоваться разными приложениями, что приводит к необходимости баз данных. В данном обзоре вопросов и решений проблемы сохраняемости объектов мы рассмотрим три подхода, имеющихся в распоряжении ОО-разработчиков. Во-первых, они могут рассчитывать на механизмы поддержки сохраняемости, включенные в язык программирования и окружение разработки, позволяющие помещать и извлекать объекты из некоторого постоянного хранилища. Во-вторых, они могут объединить объектную технологию с наиболее доступным видом (не ОО) БД - реляционными базами данных. И, в-третьих, можно использовать новые объектно-ориентированные системы баз данных, которые переносят на БД идеи ОО-технологии. Эти методы по очереди описываются в данной лекции, в ней также имеется обзор технологии ОО-баз данных с упором на два наиболее известных продукта. Завершается лекция обсуждением гипотетической будущей судьбы баз данных в ОО-контексте.

Сохраняемость средствами языка

Для удовлетворения многих потребностей в сохраняемости достаточно иметь связанный с окружением разработки набор механизмов для записи объектов в файлах и их чтения. Для простых объектов, таких как числа или символы, можно использовать средства ввода-вывода, аналогичные средствам традиционного программирования.

Сохранение и извлечение структур объектов

Как только появляются составные объекты, простое запоминание и извлечение индивидуальных объектов становится недостаточным, поскольку в них могут находиться ссылки на другие объекты, а объект, лишенный своих связников (см. [лекцию 8](#) курса "Основы объектно-ориентированного программирования"), некорректен. Это наблюдение привело нас в [лекцию 8](#) курса "Основы объектно-ориентированного программирования" к принципу Замыкания Сохраняемости, утверждающему, что всякий механизм сохранения и возвращения должен обрабатывать вместе с некоторым объектом всех его прямых и непрямых связников, что показано на [рис. 13.1](#).

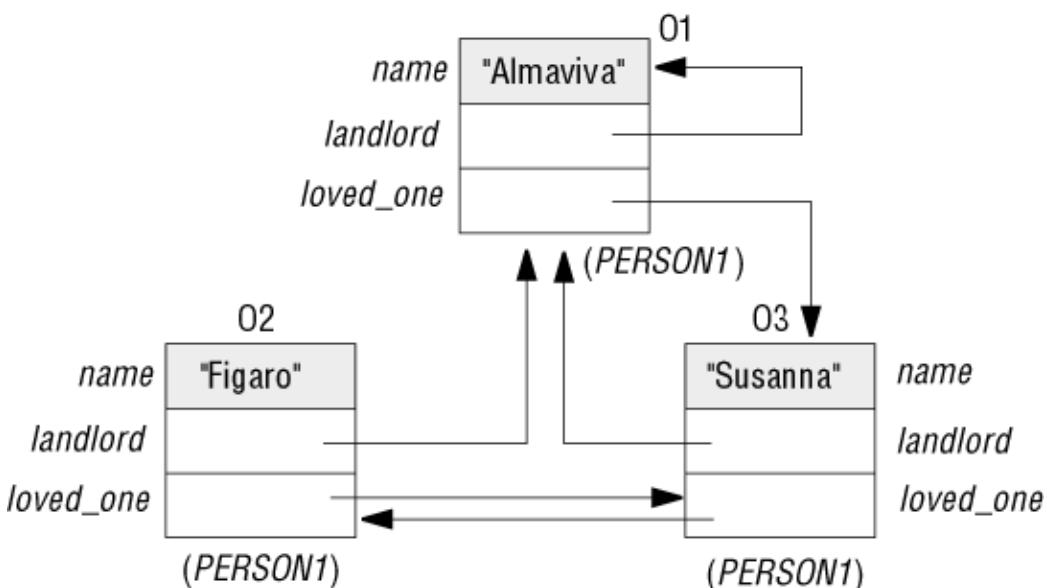


Рис. 13.1. Необходимость в замыкании при сохранении

Принцип Замыкания Сохраняемости утверждает, что всякий механизм, сохраняющий О1, должен также сохранить все объекты, на которые он непосредственно или опосредованно ссылается, иначе при извлечении этой структуры можно получить бессмысленное значение ("висящую ссылку") в поле loved_one объекта О1.

Мы рассмотрели механизмы класса STORABLE, предоставляющие соответствующие средства: store для сохранения структуры объекта и retrieved для ее извлечения. Это ценный механизм, чье присутствие в ОО-окружении само по себе является большим преимуществом перед традиционными окружениями. В [лекции 8](#) курса "Основы объектно-ориентированного программирования" приведен типичный пример его использования: реализация команды редактора SAVE. Вот еще один пример из практики нашей фирмы ISE. Наш компилятор выполняет несколько проходов по представлениям текста программы. Первый проход создает внутреннее представление, называемое Деревом Абстрактного Синтаксиса (Abstract Syntax Tree (AST)). Задача последующих проходов заключается в постепенном добавлении семантической информации в AST ("украшении дерева") до тех пор, пока ее не станет достаточно для генерации целевого кода компилятором. Каждый проход завершается операцией store; а следующий проход начинается восстановлением AST с помощью операции retrieved.

Механизм STORABLE работает не только на файлах, но и на сетевых соединениях таких, как сокеты; он на самом деле лежит в основе библиотеки клиент-сервер Net.

Форматы сохранения

У процедуры `store` имеется несколько вариантов. Один, `basic_store` (базовое_сохранение), сохраняет объекты для их последующего возвращения в ту же систему, работающую на машине той же архитектуры, в процессе того же или последующего ее исполнения. Эти предположения позволяют использовать наиболее компактную форму представления объектов.

Другой вариант, `independent_store` (независимое_сохранение), обходится без этих предположений; представление объекта в нем не зависит от платформы и от системы. Поэтому оно занимает несколько больше места, так как использует переносимое представление для чисел с плавающей точкой и для других числовых значений, а также должно включать некоторую простую информацию о классах системы. Но он важен для систем типа клиент-сервер, которые должны обмениваться потенциально большими и сложными наборами объектов, находящимися на машинах весьма разных архитектур, работающих в различных системах. Например, сервер на рабочей станции и клиент на PC могут выполнять два разных приложения и взаимодействовать с помощью библиотеки Net. Сервер приложения выполняет основные вычисления, а приложение клиента реализует интерфейс пользователя, используя графическую библиотеку, например Vision.

Заметим, что только сохранение требует нескольких процедур - `basic_store`, `independent_store`. Хотя реализация операции возврата для каждого формата своя, всегда можно использовать один компонент `retrieved`, реализация которого определит формат возвращаемых данных, используемый в файле или сети, и автоматически применит соответствующий алгоритм извлечения.

Вне рамок замыкания сохраняемости

Принцип Замыкания Сохраняемости теоретически применим ко всем видам сохранения. Как мы видели, это позволяет сохранить совместность сохраненных и восстановленных объектов.

Но в некоторых практических ситуациях требуется немного изменить структуру данных перед тем, как к ней будут применены такие механизмы, как STORABLE или средства ОО-баз данных, рассматриваемые далее в этой лекции. Иначе можно получить больше, чем хотелось бы.

Такая проблема возникает, в частности, из-за разделяемых структур, как в следующем примере.

Требуется заархивировать сравнительно небольшую структуру данных. Так как она содержит одну или более ссылок на большую разделяемую структуру, то принцип замыкания сохраняемости требует архивирования и этой структуры. В ряде случаев этого делать не хочется. Например, как показано на [рис. 13.1](#), объект личность может через поле `address` ссылаться на гораздо большее множество объектов, представляющих географическую информацию.

Аналогичная ситуация возникает в продукте ArchiText фирмы ISE, позволяющем пользователям манипулировать структурами таких документов, как программы или спецификации. Каждый документ, подобно структуре FAMILY на [рис. 13.2](#), содержит ссылку на структуру, представляющую основную грамматику, играющую ту же роль, что и структура CITY для FAMILY. Мы хотели бы сохранять документ, а не грамматику, которая уже где-то имеется и которую разделяют многие документы.

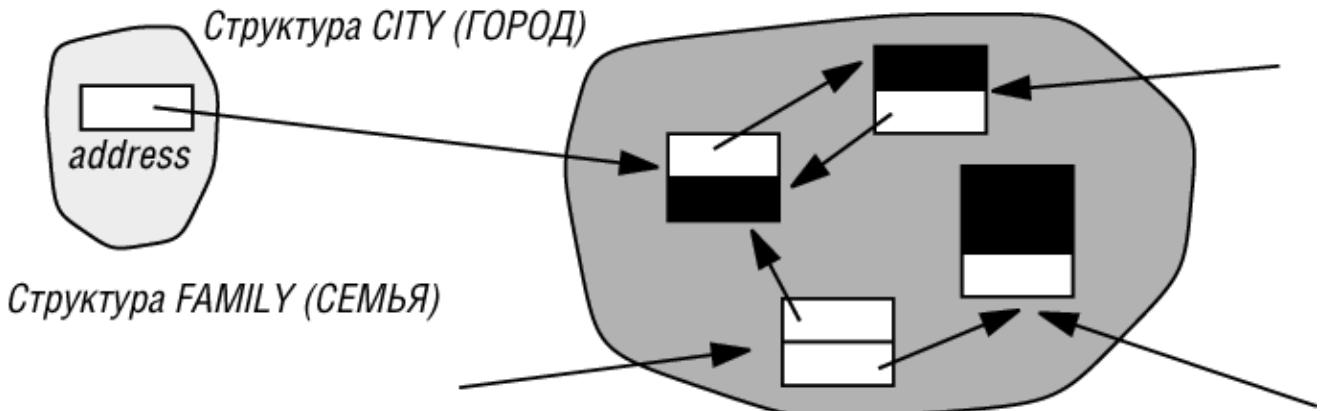


Рис. 13.2. Малая структура, ссылающаяся на большую разделяемую структуру

В таких случаях хочется " оборвать" ссылки на разделяемые структуры перед сохранением ссылающейся структуры. Однако это тонкая процедура. Во-первых, нужно, как всегда, быть уверенным, что в момент новой загрузки объекты останутся совместными - будут удовлетворять своим инвариантам. Но здесь есть и практическая проблема: как обойтись без усложнений и ошибок? На самом деле, не хотелось бы менять исходную структуру, ссылки нужно оборвать только в сохраняемой версии.

И вновь методы построения ОО-ПО дают элегантное решение проблемы, основанное на идеях классов поведения, рассмотренных при обсуждении наследования. Одна из версий процедуры сохранения `custom_independent_store` работала так же, как и предопределенная процедура `independent_store`. Но она позволяла также каждому потомку библиотечного класса ACTIONABLE переопределять ряд процедур, которые по умолчанию ничего не делали, например, процедуры `pre_store` и `post_store`, выполняющиеся непосредственно перед и после сохранения объекта. Таким образом, можно, чтобы `pre_store` выполняла:

```
preserve; address := Void,
```

где `preserve` - это тоже компонент `ACTIONABLE`, который куда-нибудь безопасно копирует объект. Тогда `post_action` будет выполнять вызов:

```
restore,
```

восстанавливающий объект из сохраненной копии.

В общем случае того же эффекта можно добиться с помощью вызова вида:

```
store_ignore ("address"),
```

где `store_ignore` получает в качестве аргумента имя поля. Так как реализация `store_ignore` может просто пропускать поле, устранив необходимость двустороннего копирования посредством `preserve` и `restore`, то в данном случае это будет более эффективно, но механизм `pre_store-post_store` является общим, позволяя выполнять необходимые действия до и после сохранения. Разумеется, нужно убедиться в том, что эти действия не будут неблагоприятно влиять на объекты.

Аналогично можно справиться и с проблемой несовместности во время возвращения, для этого достаточно переопределить процедуру `post_retrieve`, выполняемую непосредственно перед тем, как восстанавливаемый объект присоединится к сообществу уже одобренных объектов. Например, приложение может переопределить `post_retrieve` в нужном классе-наследнике `ACTIONABLE`, чтобы она работала так:

```
address := my_city_structure.address_value (...)
```

тем самым снова делая объект представительным, еще до того, как он сможет нарушить инвариант своего класса или какое-нибудь неформальное ограничение.

Конечно, нужно соблюдать некоторые правила, связанные с механизмом класса `ACTIONABLE`; в частности, `pre_store` не должна вносить в структуры данных никаких изменений, которые не были бы сразу же исправлены процедурой `post_store`. Нужно также обеспечить, чтобы `post_retrieve` выполняла необходимые действия (часто те же, что и `post_store`) для корректировки всех несовместностей, внесенных в сохраненные данные процедурой `pre_store`. Предложенный механизм, используемый с соблюдением указанных правил, позволит вам остаться верным духу принципа Замыкания Сохраняемости, делая его применение более гибким.

Эволюция схемы

Этот общий вопрос возникает при всех подходах к ОО-сохраняемости. Классы могут меняться. Что произойдет, если изменится класс, экземпляры которого находятся где-то на постоянном хранении? Этот вопрос называют проблемой эволюции схемы.

Слово **схема (schema)** пришло из мира реляционных баз данных, где она задает архитектуру БД: множество ее отношений с указанием того, что называется их типами, - число полей и тип каждого поля. В ОО-контексте схема тоже будет множеством типов, определяемых в этом случае классами.

Хотя некоторые средства разработки и системы баз данных обладают интересными средствами для эволюции ОО-схем, ни одно из них не дает полностью удовлетворительного решения. Давайте, определим компоненты полного подхода.

Будет полезно ввести некоторую точную терминологию. **Эволюция схемы** имеет место, если хотя бы один класс системы, возвращающей объекты (**возвращающая система**), отличается от своего прототипа в системе, сохранившей эти объекты (**сохраняющая система**). Рассогласование при возврате объекта или просто **рассогласование объекта** имеет место, когда возвращающая система реально возвращает некоторый объект, у которого изменился породивший его класс. Рассогласование объекта - это следствие эволюции схемы одного или нескольких классов, отражающееся на конкретном объекте.

Напомним, несмотря на термины "сохраняющая система" и "возвращающая система" наше обсуждение применимо не только к сохранению и возврату, использующим файлы и БД, но также и к передаче объектов по сети, как в библиотеке Net. В этом случае более аккуратными терминами были бы "посылающая система" и "получающая система".

Для упрощения обсуждения примем обычное предположение, что программная система не изменяется в процессе ее выполнения. Это означает, что все сохраненные экземпляры класса относятся к одной и той же версии, поэтому во время возвращения либо все они приведут к рассогласованию, либо ни один из них не будет рассогласован. Это предположение не слишком ограничительное, оно не исключает случая баз данных, которые содержат экземпляры многих разных версий одного класса, созданные различными выполнениями системы.

Наивные подходы

Мы можем исключить два крайних подхода к эволюции схем:

- Отказ от ранее сохраненных объектов (революция схемы!). Разработчиков нового приложения эта идея может привлечь, так как облегчит им жизнь. Но пользователи этого приложения вряд ли будут в восторге.

- Переход к новому формату, требующий единовременного преобразования всех старых объектов. Хотя это решение может в ряде случаев подойти, оно не годится для большого хранилища объектов или для хранилища, которое должно быть постоянно доступно.

На самом деле нам нужен способ трансформации объектов "**на лету**" в то время, когда они возвращаются или изменяются. Такое решение является наиболее общим, и далее мы будем рассматривать только его.

Если потребуется механизм одновременной трансформации многих объектов, то механизм "на лету" легко позволит это сделать: достаточно написать маленькую систему, которая возвращает все существующие объекты, используя новые классы, при необходимости применяет трансформацию на лету и все сохраняет.

Преобразование объектов на лету

Механика преобразования на лету может оказаться весьма мудреной: мы должны быть очень внимательны, чтобы не получить в результате испорченные объекты или БД.

Во-первых, у приложения может не оказаться права изменять запомненный объект из-за существования разных версий породившего его класса. Это вполне разумно, поскольку другие приложения могут все еще использовать старую версию этого объекта. Проблема эта не нова для баз данных. Можно сделать так, чтобы используемый приложением объект был совместим с описанием собственного класса; механизм преобразования на лету обеспечит выполнение этого свойства. Заносить ли преобразованный объект обратно - это отдельный вопрос, классический вопрос привилегированного доступа, возникающий всякий раз, когда несколько приложений или несколько сессий одного и того же приложения получают доступ к сохранению данных. Различные его решения предлагаются базами данных, обычными и ОО.

Независимо от ответа на вопрос о сохранении после изменения более новая и трудная проблема состоит в том, что каждое приложение должно делать с устаревшими объектами. Эволюция схемы включает три отдельных аспекта - выявление, извещение и исправление:

- выявление (Detection)** - обнаружение рассогласований объекта (восстановливаемый объект устарел);
- извещение (Notification)** - уведомление системы о рассогласовании объекта, чтобы она смогла соответствующим образом на это прореагировать, а не продолжала работать с неправильным объектом (вероятная причина главной неприятности в будущем!);
- исправление (Correction)** - приведение рассогласованного объекта в согласованное состояние, т. е. по превращению его в корректный экземпляр новой версии своего класса - гражданина или по крайней мере постоянного резидента системы.

Все три задачи являются весьма тонкими. К счастью, их можно решать по отдельности.

Выявление

Мы определим две общих категории политики выявления: **номинальную (nominal)** и **структурную (structural)**.

В обоих случаях задача состоит в выявлении рассогласования между двумя версиями породившего объект класса, существующих в сохраняющей и возвращающей системах.

При номинальном подходе каждая версия класса идентифицируется именем. Это предполагает наличие специального механизма регистрации, который может иметь два варианта:

- При использовании системы управления конфигурацией можно регистрировать каждую новую версию класса и получать в ответ имя этой версии (или самому задавать это имя).
- Возможна и автоматическая схема, аналогичная возможности автоматической идентификации в OLE 2 фирмы Microsoft или методам, используемым для присвоения "динамических IP-адресов" компьютерам в Интернете. Эти методы основаны на присвоении случайных номеров, достаточно больших для того, чтобы сделать вероятность совпадения бесконечно малой.

Для каждого из этих решений требуется некоторый централизованный регистр. Если вы хотите избежать связанных с этим трудностей, то используйте структурный подход. Его идея в том, что каждая версия класса имеет свой дескриптор, строящийся по текущей структуре, заданной в объявлении класса. Механизм сохранения объектов должен сохранять дескрипторы их классов. (Конечно, при сохранении многих экземпляров одного класса достаточно запомнить только одну копию его дескриптора.) После этого механизм выявления рассогласований прост: достаточно сравнить дескриптор класса каждого сохраненного объекта с новым дескриптором этого класса. Если они различны, то имеется рассогласованный объект.

Что входит в дескриптор класса? Ответ связан с соотношением между эффективностью и надежностью. Из соображений эффективности не хотелось бы тратить много места на информацию о классе или чрезмерно много времени на сравнение дескрипторов во время возвращения, но надежность требует минимизации риска пропуска рассогласования. Вот несколько возможных стратегий:

- C1** Одна крайность состоит в том, чтобы в качестве дескриптора класса взять его имя. В общем случае этого недостаточно: если имя класса, породившего объект, в сохранившей его системе совпадет с именем класса в системе, возвратившей этот объект, то объект будет принят, даже если эти два класса совершенно несовместимы. Неизбежно последуют неприятности.

- **C2** Другая крайность - использовать в качестве дескриптора класса весь его текст, не обязательно в виде строки, но в некоторой подходящей внутренней форме (дерева абстрактного синтаксиса). Понятно, что с точки зрения эффективности это самое плохое решение: и занимаемая память, и время сравнения дескрипторов максимальны. Но оно может оказаться неудачным и с точки зрения надежности, так как некоторые изменения класса являются безвредными. Предположим, например, что к тексту класса добавилась новая процедура, но атрибуты класса и его инвариант не изменились. Тогда нет ничего плохого в том, чтобы рассматривать возвращаемый объект как соответствующий современным требованиям, а определение его как рассогласованного может привести к неоправданным затруднениям (таким как исключение) в возвращающей системе.
- **C3** Более реалистичный подход состоит в том, чтобы включить в дескриптор класса его имя и список имен атрибутов и их типов. По сравнению с номинальным подходом остается риск того, что два совершенно разных класса могут иметь одинаковые имена и атрибуты, но (в отличие от C1) такие случайные совпадения на практике чрезвычайно маловероятны.
- **C4** Еще один вариант C3 включает не только список атрибутов, но и инвариант класса. Это приведет к тому, что добавление или удаление подпрограммы, не приводящей к рассогласованию объекта, окажется безвредным, так как, если бы изменилась семантика класса, то изменился бы и его инвариант.

C3 - это минимальная разумная политика, и в обычных случаях она представляется хорошим выбором, по крайней мере для начала.

Извещение

Что должно произойти после того, как номинальный или структурный механизм выявления выловит рассогласование объекта?

Хотелось бы, чтобы возвращающая система узнала об этом и сумела предпринять необходимые корректирующие действия. Этой проблемой будет заниматься некоторый библиотечный механизм. Класс GENERAL (предок всех классов) должен содержать процедуру:

```
correct_mismatch is
  do
    ...См. полную версию ниже...
  end
```

и правило, что любое выявленное рассогласование объекта приводит к вызову `correct_mismatch` (**корректировать_рассогласование**) на временно возвратившейся версии объекта. Каждый класс может переопределить стандартную версию `correct_mismatch` аналогично всякому переопределению процедур создания и стандартной обработки исключений `default_rescue`. Любое переопределение `correct_mismatch` должно сохранять инвариант класса.

Что должна делать стандартная (определенная по умолчанию) версия `correct_mismatch?` Дать ей пустое тело, демонстрируя ненавязчивость, не годится. Это означало бы по умолчанию игнорирование рассогласования, что привело бы к всевозможным ненормальностям в поведении системы. Для глобальной стандартной процедуры лучше возбудить соответствующее исключение:

```
correct_mismatch is
  -- Обработка рассогласования объекта при возврате
  do
    raise_mismatch_exception
  end
```

где процедура, вызываемая в теле, делает то, что подразумевается ее именем. Это может привести к некоторым неожиданным исключениям, но лучше это, чем разрешить рассогласованиям остаться незамеченными. Если в проекте требуется переделать это предопределенное поведение, например, выполнять пустую инструкцию, а не возбуждать исключение, то всегда можно переопределить `correct_mismatch`, на свой страх и риск, в классе ANY. (Как вы помните, определенные разработчиками классы наследуют GENERAL не прямо, а через класс ANY, который может быть переделан при проектировании или инсталляции.)

Для большей гибкости имеется также компонент `mismatch_information` (**информация_о_рассогласовании**) типа ANY, определенный как однократная функция. Процедура `set_mismatch_information` (`info: ANY`) позволяет передать в `correct_mismatch` больше информации, например, о различных предыдущих версиях класса.

Если вы предполагаете, что у объектов некоторого класса возникнут рассогласования, то лучше не рассчитывать на обработку исключений по умолчанию, а переопределить `correct_mismatch` так, чтобы сразу изменять возвращаемый объект. Это приводит нас к последней задаче - исправлению.

Исправление

Как следует исправлять объект, для которого при возвращении обнаружено рассогласование? Ответ требует аккуратного анализа и более сложного подхода, чем обычно реализуется в существующих системах или предлагается в литературе.

Ситуация такова: механизм возвращения (с помощью компонента retrieved класса STORABLE, соответствующей операции БД или другого доступного примитива) создал в возвращающей системе новый объект, исходя из некоторого сохраненного объекта того же класса, но обнаружил при этом рассогласование. Новый объект в его временном состоянии может быть неправильным, например, он может потерять некоторое поле, присутствовавшее у сохраненного объекта, или приобрести поле, которого не было у оригинала. Рассматривайте его как иностранца без визы.

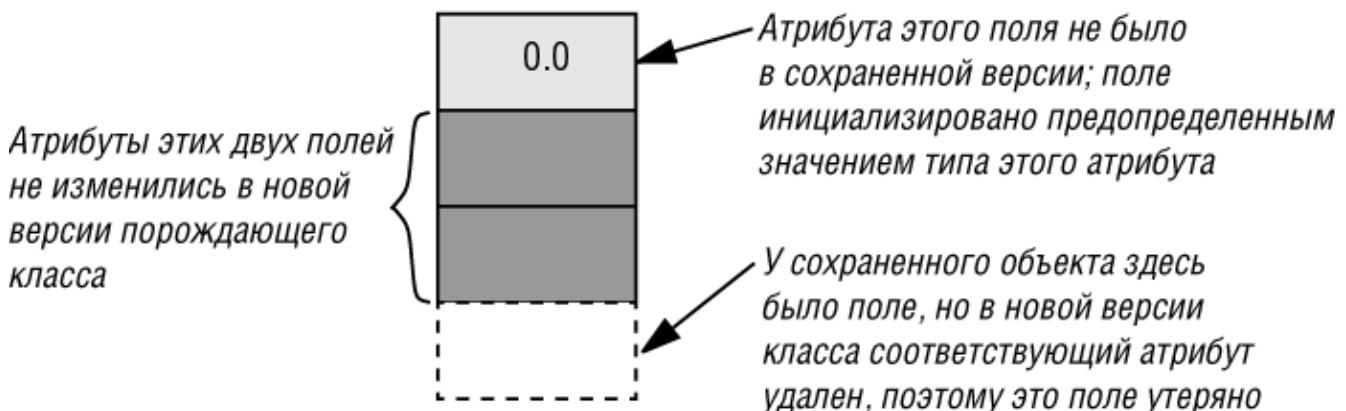


Рис. 13.3. Рассогласование объекта

Такое состояние объекта аналогично промежуточному состоянию объекта, создаваемого - вне всяких рассуждений о сохранении - с помощью инструкции создания `x.make (. . .)` сразу после распределения ячеек памяти объекта и инициализации их предопределенными значениями, но перед вызовом `make` (см. [лекцию 8](#) курса "Основы объектно-ориентированного программирования"). На этой стадии у объекта имеются все требуемые компоненты, но он еще не готов быть принятым в обществе, поскольку может иметь неверные значения некоторых полей; как мы видели, официальная цель процедуры `make` состоит в замене при необходимости предопределенных значений инициализации на значения, обеспечивающие инвариант.

Предположим для простоты, что метод выявления является структурным и основан на атрибутах (т. е. на определенной выше политике С3), хотя приведенное далее обсуждение распространяется и на другие решения, как номинальные, так и структурные. Рассогласование является следствием изменения свойств атрибутов класса. Можно свести все такие изменения к комбинациям некоторого числа **добавлений и удалений атрибутов**. На приведенном выше рисунке показано одно добавление и одно удаление.

Удаление атрибута не вызывает никаких трудностей: если в новом классе отсутствует некоторый атрибут старого класса, то соответствующие поля в объекте больше не нужны и их можно просто убрать. Фактически, процедура `correct_mismatch` ничего не должна делать с такими полями, поскольку механизм возвращения при создании временного экземпляра нового класса будет их отбрасывать. На рисунке это показано для нижнего поля - скорее, уже не поля - изображенного объекта.

Можно было бы, конечно, проявить больше заботы об отбрасываемых полях. А что, если они были действительно необходимы, а без них объект потеряет свой смысл? В таком случае нужно иметь более продуманную политику выявления, например, такую, как структурная политика С4, которая учитывает инварианты.

Более тонкая вещь - **добавление атрибута** в новый класс, приводит к появлению нового поля в возвращаемых объектах. Что делать с таким полем? Нужно его как-то инициализировать. В известных мне системах, поддерживающих эволюцию схемы и преобразование объектов, решение состоит в использовании предопределенных значений, заданных по умолчанию (обычно для чисел выбирается ноль, для строк - пустая строка). Но, как следует из обсуждения похожих проблем, возникающих, например, в контексте наследования, это решение может оказаться очень плохим!

Вспомним стандартный пример - класс ACCOUNT с атрибутами `deposits_list` и `withdrawals_list`. Предположим, в новой версии добавлен атрибут `balance`. Система, используя новую версию, пытается возвратить некоторый экземпляр, созданный в предыдущей версии.

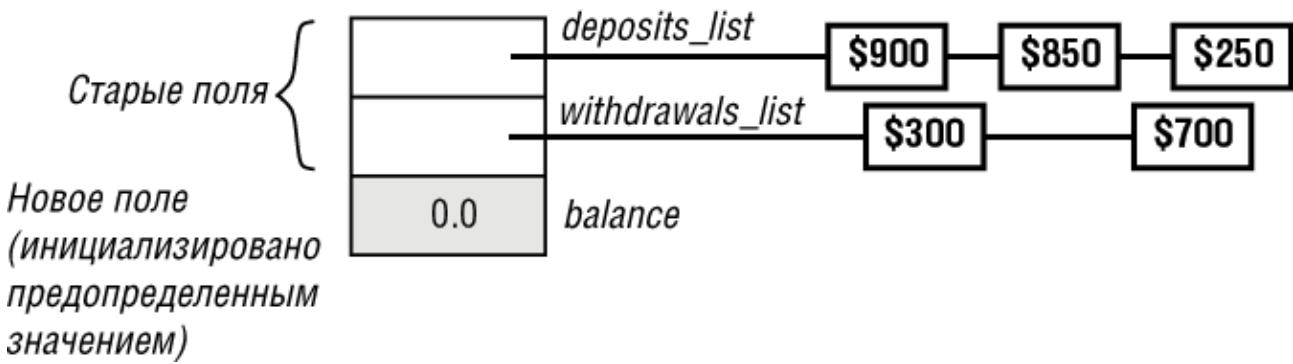


Рис. 13.4. Возвращение объекта account (счет). (Подумайте, что не в порядке на этом рисунке?)

Цель добавления атрибута `balance` понятна: вместо того, чтобы перевычислять баланс счета по каждому требованию,

мы держим его в объекте и обновляем при необходимости. Инвариант нового класса отражает это с помощью предложения вида:

```
balance = deposits_list.total - withdrawals_list.total
```

Но, если применить к полю balance возвращаемого объекта инициализацию по умолчанию, то получится совершенно неправильный результат, в котором поле с балансом счета не согласуется с записями вкладов и расходов. На приведенном рисунке balance из-за инициализации по умолчанию нулевой, а в соответствии со списком вкладов и расходов он должен равняться \$1000.

Это показывает важность механизма корректировки correct_mismatch. В данном случае можно просто переопределить эту процедуру:

```
correct_mismatch is
-- Обработать рассогласование объекта, правильно установив balance
do
    balance := deposits_list.total -withdrawals_list.total
end
```

Если автор нового класса ничего не запланирует на этот случай, то предопределенная версия correct_mismatch возбудит исключение, которое аварийно остановит приложение, если не будет обработано retry (реализующим другую возможность восстановления). Это правильный выход, поскольку продолжение вычисления может нарушить целостность структуры выполняемого объекта и, что еще хуже, структуры сохраненного объекта, например БД. Используя предыдущую метафору, можно сказать, что мы будем отвергать объект до тех пор, пока не сможем присвоить ему надлежащий иммигрантский статус.

От сохраняемости к базам данных

Использование класса STORABLE становится недостаточным для приложений, полностью основанных на БД. Его ограниченность отмечалась уже выше: имеется лишь один входной объект, нет поддержки для запросов, основанных на содержимом, каждый вызов retrieved заново создает всю структуру, без всякого разделения объектов в промежутках между последовательными вызовами. Кроме того, в STORABLE не поддерживается одновременный доступ разных приложений клиента к одним и тем же сохраненным данным.

Хотя различные расширения этого механизма могут облегчить или устранить некоторые из этих проблем, полностью отработанное решение требует отдать предпочтение технологии баз данных.

Набор механизмов, ОО или нет, предназначенных для сохранения и извлечения элементов данных (в общем случае "объектов") заслуживает названия системы управления базой данных (СУБД), если он поддерживает следующие свойства:

- Живучесть (Persistence): объекты могут пережить завершение отдельных сессий использующих их программ, а также сбои компьютера.
- Программируемая структура (Programmable structure): система рассматривает объекты как структурированные данные, связанные некоторыми точно определенными отношениями. Пользователи системы могут сгруппировать множество объектов в некоторую совокупность, называемую базой данных, и определить структуру конкретной БД.
- Произвольный размер (Arbitrary size): нет никаких заранее заданных ограничений (вытекающих, например, из размера основной памяти компьютера или ограниченности его адресного пространства) на число объектов в базе данных.
- Контроль доступа (Access control): пользователь может "владеть" объектами и определять права доступа к ним.
- Запросы, основанные на свойствах (Property-based querying): имеются механизмы, позволяющие пользователям и программам находить объекты в базе данных, задавая их абстрактные свойства, а не местоположение.
- Ограничения целостности (Integrity constraints): пользователи могут налагать некоторые семантические ограничения на объекты и заставлять базу данных поддерживать их выполнение.
- Администрирование (Administration): доступны средства для осуществления текущего контроля, аудита, архивации и реорганизации БД, добавления и удаления ее пользователей, распечатки отчетов.
- Разделение (Sharing): несколько пользователей или программ могут одновременно получать доступ к базе данных.
- Закрытие (Locking): пользователи или программы могут получать исключающий доступ (только для чтения, для чтения и записи) к одному или нескольким объектам.
- Транзакции (Transactions): можно так определять последовательности операций БД, называемые транзакциями, что либо вся транзакция будет выполнена нормально, либо при неудачном завершении не оставит никаких видимых изменений в состоянии БД.

Стандартный пример транзакции - это перевод денег в банке с одного счета на другой, требующий двух операций - занесения в дебет первого счета и в кредит второго, которые должны либо успешно завершиться, либо вместе не выполниться. Если они завершаются неудачей, то всякое частичное изменение, такое как занесение в дебет первого счета, нужно отменить; это называется **откатом (rolling back)** транзакции.

Приведенный список свойств не является исчерпывающим; он отражает то, что предлагается большинством коммерческих систем и ожидается пользователями.

Объектно-реляционное взаимодействие

Безусловно, сегодня наиболее общей формой СУБД является **реляционная (relational) модель**, базирующаяся на идеях, предложенных Коддом (E. F. Codd) в статье 1970 года.

Определения

Реляционная БД - это набор **отношений (relations)**, каждого из которых состоит из множества кортежей (**tuples**) (или записей [**records**]). Отношения также называются **таблицами**, а кортежи **строками**, так как отношения удобно представлять в виде таблиц. Как пример, рассмотрим таблицу BOOKS (**КНИГИ**):

Таблица 13.1. Отношение КНИГИ (BOOKS)

title (название)	date (дата)	pages (страницы)	author (автор)
"The Red and the Black"	1830	341	"STENDHAL"
"The Charterhouse of Parma"	1839	307	"STENDHAL"
"Madame Bovary"	1856	425	"FLAUBERT"
"Euge_nie Grandet"	1833	346	"BALZAC"

Каждый кортеж состоит из нескольких **полей (fields)**. У всех кортежей одного отношения одинаковое число и типы полей; в примере первое и последнее поля являются строками, а два других - целыми числами. Каждое поле идентифицируется **именем**: в примере с книгами это title, date и т. д. Имена полей (столбцов) называются **атрибутами (attributes)**.

Реляционные базы обычно являются **нормализованными**, среди прочего это означает, что каждое поле имеет простое значение (целое число, вещественное число, строка, дата) и не может быть ссылкой на другой кортеж.

Операции

Реляционной модели баз данных сопутствует **реляционная алгебра**, в которой определено много операций над отношениями. Три типичные операции - это выбор (selection), проекция (projection) и соединение (join).

Выбор выдает отношение, содержащее подмножество строк данного отношения, удовлетворяющее некоторому условию на значения полей. Применяя условие выбора "pages меньше, чем 400" к BOOKS, получим отношение, состоящее из первой, второй и последней строки BOOKS.

Проекция отношения на один или несколько атрибутов получается пропуском всех других полей и устранением повторяющихся строк в получившемся результате. Если спроектировать наше отношение на последний атрибут, то получим отношение с одним полем и с тремя кортежами: "STENDHAL", "FLAUBERT" и "BALZAC". Если же спроектировать его на три первых атрибута, то получится отношение с тремя полями, полученное из исходного вычеркиванием последнего столбца.

Соединение двух отношений это комбинированное отношение, полученное путем выбора атрибутов с согласованными типами в каждом из них и объединением строк с одинаковыми (в общем случае, согласованными) значениями этих атрибутов. Предположим, что у нас имеется еще отношение AUTHORS (**АВТОРЫ**):

Таблица 13.2. Отношение AUTHORS (АВТОРЫ)

Name (имя)	real_name (настоящее_имя)	Birth (год_рождения)	death (год_смерти)
"BALZAC"	"Honore_de Balzac"	1799	1850
"FLAUBERT"	"Gustave Flaubert"	1821	1880
"PROUST"	"Marcel Proust"	1871	1922
"STENDHAL"	"Henry Beyle"	1783	1842

Тогда соединение отношений BOOKS и AUTHORS по согласованным атрибутам author и name будет следующим отношением:

Таблица 13.3. Соединение отношений BOOKS и AUTHORS по полям author и name

title	date	pages	author/name	real_name	birth	death
"The Red and the Black"	1830	341	"STENDHAL"	"Henry Beyle"	1783	1842
"The Charterhouse of Parma"	1839	307	"STENDHAL"	"Henry Beyle"	1783	1842
"Madame Bovary"	1856	425	"FLAUBERT"	"Gustave Flaubert"	1821	1880
"Euge_nie Grandet"	1833	346	"BALZAC"	"Honore_de Balzac"	1799	1850

Запросы

Реляционная модель допускает запросы - одно из главных требований к базе данных в нашем списке - в

стандартизированном языке, называемом SQL. Они используются в двух формах: одна применяется непосредственно людьми, а другая ("встроенный SQL") используется в программах. В первой форме типичный SQL-запрос выглядит так:

```
select title, date, pages from BOOKS
```

Он выдает названия, даты и число страниц для всех книг из таблицы BOOKS. Как мы видели, этот запрос в реляционной алгебре является операцией проекции. Другой пример:

```
select title, date, pages, author where pages < 400
```

соответствует в реляционной алгебре выбору. Запрос:

```
select
    title, date, pages, author, real_name, birth, date
from AUTHORS, BOOKS where
    author = name
```

это внутреннее соединение, дающее тот же результат, что и приведенный пример соединения.

Использование реляционных баз данных с ОО-ПО

Основные понятия реляционных СУБД, кратко описанные выше, демонстрируют явное сходство с основной моделью ОО-вычисления. Можно сопоставить отношению класс, а кортежу этого отношения - объект, экземпляр класса. Нам потребуется библиотечный класс, предоставляющий операции реляционной алгебры (соответствующий встроенному SQL).

Многие ОО-окружения предоставляют такую библиотеку для C++, Smalltalk или для языка этой книги (библиотека Store). Этот подход, который можно назвать объектно-реляционным взаимодействием, был успешно испробован во многих разработках. Он подходит в одном из следующих случаев:

- ОО-система должна использовать и, возможно, обновлять существующие общие данные, находящиеся в реляционных базах данных. Тогда нет иного выбора, как использовать объектно-реляционный интерфейс.
- ОО-система должна сохранять объекты, структура которых хорошо соответствует реляционному взгляду на вещи. (Далее будут объяснены причины, по которым это бывает не всегда.)

Если ваши требования к сохраняемости не подпадают под эти случаи, то вы будете испытывать то, что в литературе называют **сопротивлением несогласованности (impedance mismatch)** между ОО-моделью данных вашей разработки ПО и реляционной моделью данных БД. Тогда полезно будет взглянуть на новейшие разработки в области БД: объектно-ориентированные системы баз данных.

Основания ОО-баз данных

Становление ОО-БД подкреплялось тремя стимулами.

- **D1** Желанием предоставления разработчикам ОО-ПО механизма сохранения объектов, сопоставимого с их методом разработки и устранившего сопротивление несогласованности.
- **D2** Необходимостью преодоления концептуальных ограничений реляционных баз данных.
- **D3** Возможностью предложения более развитых средств работы с базами данных, отсутствующих в ранних системах (реляционных и других), но сделавшихся возможными и необходимыми благодаря прогрессу технологий.

Первый стимул наиболее очевиден для освоивших объектную разработку ПО, когда они сталкиваются с необходимостью сохранения объектов. Но он не обязательно является самым важным. Два других полностью относятся к области баз данных и не зависят от метода разработки.

Изучение понятия ОО-БД начнем с выявления ограничений реляционных систем D2 и того, чем они могут не устроить разработчиков ОО ПО (D1), а затем перейдем к новаторским достижениям движения за ОО-БД.

На чем застопорились реляционные БД

Было бы абсурдом отрицать вклад систем реляционных БД. (На самом деле в то время как первые публикации по ОО-БД в восьмидесятых склонялись к критике реляционной технологии, современная тенденция состоит в том, чтобы рассматривать эти два подхода как взаимодополняющие.) Реляционные системы являются одним из важнейших компонентов роста информационных технологий, начиная с семидесятых, и будут оставаться им еще долгое время. Они хорошо приспособились к ситуациям, связанным с данными (возможно, больших размеров), в которых:

- **R1** структура данных регулярна: все объекты данного типа имеют одинаковое число и типы компонентов;
- **R2** эта структура простая: для типов компонентов имеется небольшое множество заранее определенных возможностей;
- **R3** эти типы выбираются из небольшой группы заранее определенных возможных типов (целые числа, строки, даты, ...), для каждого из которых фиксированы размеры.

Типичным примером является БД с данными о налогоплательщиках с большим количеством объектов,

представляющих людей, описываемых фиксированными компонентами: ФИО (строка), дата рождения (дата), адрес (строка), зарплата (число) и еще несколько свойств.

Свойство (R3) исключает многие приложения, связанные с мультимедиа, CAD-CAM и обработкой изображений, в которых некоторые элементы данных, такие как битовые образы изображений, имеют сильно различающиеся и иногда очень большие размеры. Этому также мешает требование, чтобы отношения находились в "нормальной форме", налагаемое существующими коммерческими системами, из-за которого один объект не может ссылаться на другой. Это, конечно, очень сильное ограничение, если сравнить его с тем, что мы доказали раньше в дискуссиях этой книги.

Как только у нас есть некоторый объект, ссылающийся на другой объект, то ОО-модель обеспечивает простой доступ к непрямым свойствам этого объекта. Например, `redblack.author.birth_year` возвращает значение 1783, если переменная `redblack` присоединена к объекту слева на [рис. 13.5](#). Реляционное описание неспособно представить поле со ссылкой `author` (**автор**), чьим значением является обозначение другого объекта.

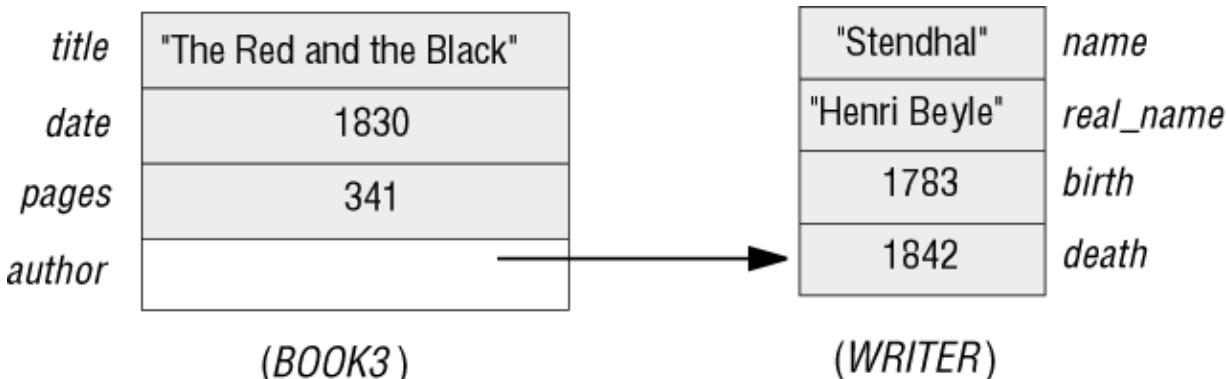


Рис. 13.5. Объект со ссылкой на другой объект

В реляционной модели имеется обходной путь для этой проблемы, но он тяжелый и непрактичный. Для представления описанной ситуации будут необходимы два отношения BOOKS и AUTHORS, введенные выше. Для их связывания можно выполнить уже показанную операцию соединения, использующую соответствие между полями `author` первого отношения и `name` - второго.

Для ответа на вопросы вида: "В каком году родился автор "Красного и черного"?" реляционная реализация должна будет вычислять соединения, проекции и т. п. В данном случае можно использовать указанное выше соединение, а затем взять его проекцию на атрибут `birth`.

Этот метод работает и широко используется, но он применим только для простых схем. Число операций соединения быстро возрастает в сложных случаях для систем, постоянно обрабатывающих запросы со многими связями, например: "Сколько комнат имеется в предыдущем доме менеджера отдела, из которого дама, закончившая на первом месте среднюю школу вместе с младшим дядей моей жены, была переведена, когда компания-учредитель провела второй тур реорганизации?" Для ОО-системы, поддерживающей во время выполнения сеть соответствующих объектов, ответ на этот запрос не представляет никакой сложности.

Идентичность объектов

Простота реляционной модели частично объясняется тем, что объекты однозначно идентифицируются значениями своих атрибутов. Отношение (таблица) является подмножеством декартового произведения $A \times B \times \dots$ некоторых множеств A , B , \dots ; иными словами, каждый элемент отношения, каждый объект, это кортеж $\langle a_1, b_1, \dots \rangle$, в котором a_1 принадлежит A и т. д. Поэтому он не существует вне своего значения, в частности, вставка объекта в отношение не будет иметь никакого эффекта, если в отношении уже имелся идентичный кортеж. Например, вставка $\langle "The Red and the Black", 1830, 341, "STENDHAL" \rangle$ в приведенное выше отношение BOOKS не приведет к изменению этого отношения. Это сильно отличается от динамичной модели ОО-вычислений, в которой могут существовать два идентичных объекта.

Напомним, что отношение `equal (obj1, obj2)` истинно, если `obj1` и `obj2` - это ссылки, присоединенные к этим объектам, но равенство `obj1 = obj2` будет ложным.

Быть идентичными - не значит быть одними и теми же (спросите об этом близнецам). Такая способность различать два этих понятия частично определяет силу моделирования в ОО-технологии. Она основана на понятии идентичности объекта: всякий объект существует независимо от его содержания.

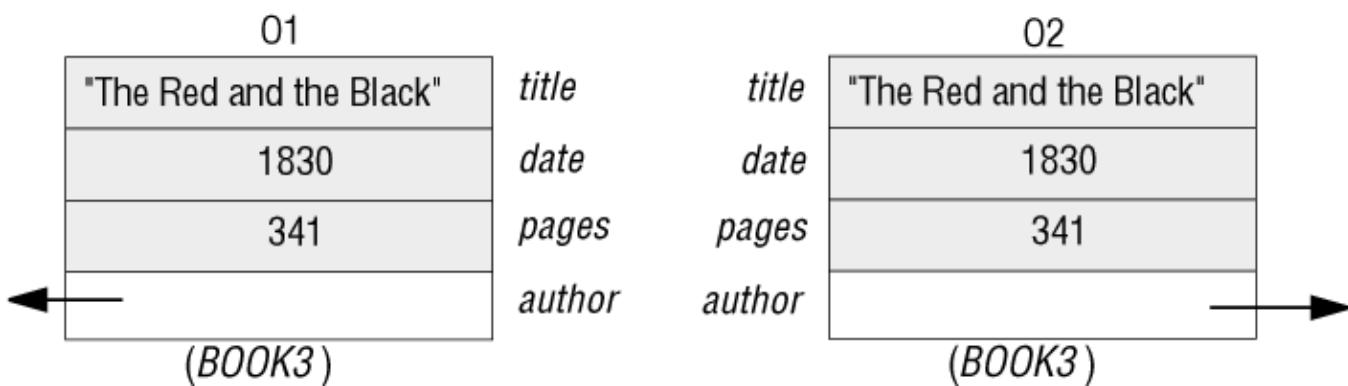


Рис. 13.6. Отдельные, но равные (обе нижние ссылки присоединены к одному объекту)

Посетителям Императорского дворца в Киото говорят, что эти здания очень древние и каждое перестраивается приблизительно раз в сто лет. С учетом понятия идентичности объекта в этом нет никакого противоречия: объект остается тем же, даже если его содержание меняется.

Вы та же личность, что и десять лет назад, хотя ни одной из молекул, составляющих ваше тело в то время, сейчас не осталось.

Разумеется, в реляционной модели тоже можно выразить идентичность объектов: достаточно добавить к каждому объекту специальное ключевое поле с уникальным для объектов данного типа значением. Но придется об этом заботиться явно. А в ОО-модели идентичность объектов имеется по умолчанию.

При создании ОО-ПО, не требующего хранить объекты, поддержка идентичности объекта получается почти случайно: в простейшей реализации каждый объект постоянно хранится по некоторому адресу, а ссылки на объект используют этот адрес, который служит неизменяемым идентификатором данного объекта. (Это неверно для более сложных реализаций, например, фирмы ISE, которые могут перемещать объекты в процессе эффективного сбора мусора; в этих реализациях идентичность объекта является более абстрактным понятием.) Если требуется хранить объекты, то идентичность объекта становится важным фактором ОО-модели.

Поддержка идентичности объектов в разделяемых БД приводит к новым проблемам: каждый клиент, которому нужно создавать объекты, должен получать для них уникальные идентификаторы; это значит, что специальный модуль, ответственный за присвоение идентификаторов, должен быть разделяемым ресурсом, что в условиях сильной параллельности создает потенциальное узкое горлышко.

Пороговая модель

Из предыдущих обсуждений можно вывести то, что может быть названо пороговой моделью ОО-БД: минимальное множество свойств, которым должна удовлетворять система БД, чтобы заслужить название ОО-БД (по работе [Zdonik 1990]). (Другие, также весьма желательные, свойства будут обсуждены ниже.) Имеются четыре требования, которым должна удовлетворять пороговая модель: база данных, инкапсуляция, идентифицируемость объектов и ссылки. Такая система должна:

- **T1** предоставлять все стандартные функции баз данных, перечисленные выше в этой лекции;
- **T2** поддерживать инкапсуляцию, т. е. позволять скрытие внутренних свойств объектов и делать их доступными через официальный интерфейс;
- **T3** связывать с каждым объектом его уникальный для данной базы идентификатор;
- **T4** разрешать одним объектам содержать ссылки на другие объекты.

Примечательно, что в этом списке отсутствуют некоторые ОО-механизмы, необходимые для этого метода, в частности наследование. Но это не так странно, как может показаться на первый взгляд. Все зависит от того, что мы ожидаем от БД. Система на пороговом уровне должна быть хорошей **машиной ОО-БД**, предоставляющей набор механизмов для сохранения, возвращения и обхода структур объектов, но оставляющей знания более высокого уровня о семантике этих объектов (например, отношения наследования) для уровня языка программирования или окружения разработки.

Опыт ранних систем ОО-БД подтверждает, что подход машины базы данных разумен. Некоторые из первых систем ударились в другую крайность и обзавелись полной "моделью данных" с соответствующим ОО-языком, поддерживающим наследование, родовыми классами, полиморфизм и т.п. Их производители обнаружили, что эти языки в конкуренции с языками ОО-разработки проигрывают (поскольку язык базы данных, как правило, менее общий и практичный, чем язык, который с самого начала проектировался как универсальный); тогда они стремглав побежали заменять свои собственные предложения интерфейсами с основными ОО-языками.

Дополнительные возможности

Имеется много желательных свойств БД, не входящих в пороговую модель. Большинство коммерческих систем предлагают, по крайней мере, некоторые из них.

Первая категория включает непосредственную поддержку более глубоких свойств ОО-метода: наследования (одиночного или множественного), типизации, динамического связывания. Не нужно подробней разъяснять эти

свойства читателям данной книги. Другие возможности, которые мы кратко рассмотрим ниже, включают: версии объектов, эволюцию схемы, длинные транзакции, блокировка, ОО-запросы.

Версии объекта

Так называется способность запоминать предыдущие состояния объекта после, того, как вызовы процедур его изменили. Это особенно важно в случае параллельного доступа. Предположим, что объект O1 содержит ссылку на объект O2. Клиент изменяет некоторые поля O1, отличные от этой ссылки. Другой клиент изменяет O2. Тогда, если первый клиент попытается проследовать по ссылке, он может обнаружить версию O2, несовместную с O1.

Некоторые ОО-СУБД справляются с этой проблемой, трактуя каждую модификацию объекта как создание нового объекта, тем самым, поддерживая доступ к старым версиям объектов.

Версии классов и эволюция схемы

Объекты - это не единственные элементы, для которых требуется поддержка версий: со временем могут изменяться и порождающие их классы. Это проблема эволюции схемы, которая обсуждалась в начале этой лекции. Только очень немногие ОО-СУБД полностью поддерживают эволюцию схем.

Длинные транзакции

Понятие транзакции уже давно является очень важным для СУБД, но классические механизмы транзакций ориентированы на **короткие** транзакции, которые начинаются и завершаются одной операцией, выполняемой одним пользователем во время одной сессии работы компьютерной системы. Исходным примером, процитированным в начале этой лекции, служит банковский перевод денег с одного счета на другой; он является транзакцией, поскольку требует либо полного выполнения обеих операций (снятия денег с одного счета и зачисления их на другой), либо (при неудаче) - сохранения исходного состояния. Время, занимаемое этой транзакцией, составляет несколько секунд (даже меньше, если не учитывать взаимодействие с пользователем).

У приложений, связанных с проектированием сложных систем, таких как CAD-CAM (системы автоматизированного проектирования и производства инженерной продукции) или системы автоматизированного проектирования ПО, возникает потребность в длинных транзакциях, которые могут выполняться в течение дней или даже месяцев. Например, в процессе проектирования автомобиля одна из групп инженеров может прекратить работу над частью карбюратора, чтобы внести какие-то изменения, и вернуться к ней через неделю или две. У такой операции имеются все свойства транзакции, но методы, разработанные для коротких транзакций, здесь напрямую не применимы.

Область разработки ПО имеет очевидную потребность в длинных транзакциях, возникающую всякий раз, когда несколько человек или команд работают над общим набором модулей. Интересно, что технология БД не получила широкого распространения (несмотря на многие предложения в литературе) в сфере разработки ПО. Вместо этого, разрабатывались собственные средства управления конфигурациями (configuration management), которые ориентировались на специфические запросы разработки компонентов ПО, а также дублировали некоторые стандартные функции СУБД, как правило, не используя достижений технологии БД. Эта, на первый взгляд, странная ситуация имеет вполне вероятное простое объяснение: отсутствие длинных транзакций в традиционных СУБД.

Хотя длинные транзакции концептуально могут и не требовать использования объектной технологии, усилия последнего времени по их поддержке пришли со стороны ОО-СУБД, некоторые из которых предлагают способ проверки любого объекта как в базе данных, так и вне ее.

Блокировка

Каждая СУБД должна предоставлять некоторую форму блокировки объектов для того, чтобы обеспечить безопасный параллельный доступ и обновление. Ранние ОО-СУБД поддерживали блокировку **на уровне страниц**, при которой границы блокируемой области определялись операционной системой. Это было неудобно как для больших объектов (которые могут занимать несколько страниц), так и для маленьких (которых может быть много на одной странице, так что блокировка одного из них повлечет блокировку и остальных). Новые системы предлагают блокировку **на уровне объекта**, позволяя приложению клиента блокировать объекты индивидуально.

Последние усилия направлены на минимизацию размера блокировки в процессе реального выполнения, поскольку блокировка может вызвать конфликты и замедление выполнения операций БД. **Оптимистическая блокировка** - это общее название целого класса методов, которые пытаются устранить априорное навешивание замка на объект, выполняя вместо этого спорные операции на копии объекта, откладывая насколько возможно обновление главной копии, затем блокируют ее, согласовывая конфликтующие обновления. Мы увидим далее пример оптимистической блокировки в системе Matisse.

Запросы

Как было подчеркнуто выше, СУБД поддерживают запросы. Здесь ОО-системы в случае эволюции схем могут оказаться более гибкими, чем реляционные. Изменение схемы реляционной БД часто означает необходимость изменения текстов запросов и их перекомпиляцию. В ОО-БД запросы формулируются относительно объектов; вы спрашиваете о некоторых компонентах экземпляров некоторого класса. Здесь в качестве экземпляров могут выступать как прямые экземпляры данного класса, так и экземпляры его собственных потомков. Поэтому, если у класса, для которого сформулирован запрос, появляется новый потомок, то данный запрос применим и к экземплярам этого нового

класса и позволяет извлекать из них требуемую информацию.

ОО-СУБД: примеры

Начиная с середины восьмидесятых появилось большое число продуктов с ОО-СУБД. Некоторыми из наиболее известных являются: Gemstone, Itasca, Matisse, Objectivity, ObjectStore, Ontos, O2, Poet, Versant. Недавно несколько компаний, таких как UniSQL, разработали объектно-реляционные системы, пытаясь объединить наилучшие черты обоих подходов. Главные производители реляционных СУБД также предлагают или анонсируют комбинированные решения, такие как Illustra фирмы Informix (частично базируется на проекте POSTGRES Калифорнийского университета в Беркли) и объявленная фирмой Oracle система Oracle 8.

Чтобы облегчить возможность взаимодействия, многие производители ОО-СУБД объединили свои силы в Object Database Management Group, которая предложила стандарт ODMG для унификации общего интерфейса ОО-БД и их языков запросов.

Давайте взглянем на две особенно интересные системы: Matisse и Versant.

Matisse

MATISSE от фирмы ADB Inc., - это ОО-СУБД, поддерживающая C, C++, Smalltalk и нотацию данной книги.

Matisse - это смелая разработка со многими необычными идеями. Она ориентирована на большие базы данных с богатой семантической структурой и может манипулировать с очень большими объектами, такими как изображения, фильмы и звуки. Хотя она поддерживает основные ОО-понятия, в частности, множественное наследование, но не налагает сильных ограничений на модель данных, а скорее служит мощной машиной ОО-БД. Перечислим некоторые из ее сильных сторон:

- оригинальный метод представления, позволяющий разбивать объект - особенно большой объект - на части, помещаемые на нескольких дисках, и таким образом оптимизировать время доступа;
- оптимизированное размещение объекта на дисках;
- механизм автоматического дублирования, обеспечивающий программное решение проблемы устойчивости к машинным сбоям: объекты (а не сами диски) могут быть дублированы и автоматически восстановлены в случае сбоя на диске;
- встроенный механизм поддержки версий объектов (см. ниже);
- поддержка транзакций;
- Поддержка архитектуры клиент-сервер, в которой центральный сервер управляет данными возможно большего числа клиентов и ведет "кэш" недавно использованных объектов.

Matisse использует оригинальный подход к проблеме минимизации блокировок. Многие системы применяют следующее правило взаимного исключения: несколько клиентов могут читать объект одновременно, но как только один из клиентов начинает писать, ни один из других не может читать или писать. Причина, объясненная в лекции о параллельности, состоит в сохранении целостности объекта, выраженной инвариантами класса. Если разрешить одновременную запись двум клиентам, то объект может стать несовместным, а если некоторый клиент находится в середине процесса записи, то объект может оказаться в нестабильном состоянии (не удовлетворяющем инвариант), так что другой клиент, который его в этот момент читает, получит неверный результат.

Очевидно, что блокировка вида писатель-писатель необходима. Что касается исключений вида писатель-читатель, то некоторые системы их не придерживаются, разрешая операции чтения даже при наличии блокировки записи. Такие операции уместно назвать грязным чтением (dirty reads).

Matisse, чьи создатели явно преследовали цель минимизации блокировок, предложил радикальное решение этого вопроса, основанное на организации управления объектами: никаких операций записи. Вместо изменения существующего объекта операция записи (из ПО клиента) создает новый объект. В результате можно читать объекты без всяких блокировок: у вас всегда будет доступ к некоторой версии БД, на которую не повлияли операции записи, которые могли произойти после начала вашего чтения. Вы также можете получить доступ к большому числу объектов с гарантией, что все они принадлежат одной и той же версии БД, в то время как при традиционном подходе для достижения того же результата потребовалось бы использовать глобальные блокировки или транзакции, что привело бы к большой потере эффективности.

Следствием такой политики является возможность возврата к предыдущим версиям объекта или самой БД. По умолчанию, старые версии сохраняются, но система предоставляет "сборщик версий", позволяющий избавляться от нежелательных версий.

Система Matisse предоставляет интересные возможности для работы с отношениями. Например, если у класса EMPLOYEE (**СЛУЖАЩИЙ**) имеется атрибут supervisor (**руководитель**): MANAGER, то Matisse (по требованию разработчика) автоматически отслеживает обратные связи, так что можно получить доступ не только к руководителю служащего, но также и ко всем служащим, подчиненным данному руководителю. Кроме того, возможны запросы, ищащие объекты по ключевым словам.

Versant

Versant от фирмы Versant Object Technology - это ОО-СУБД, работающая с C++, Smalltalk и нотацией данной книги. Ее модель данных и язык интерфейса поддерживают многие из основных концепций ОО-разработки, в частности классы,

множественное наследование, переопределение компонентов, переименование компонентов, полиморфизм и универсальность.

Versant - это одна из СУБД, отвечающих стандарту ODMG. Она предназначена для архитектуры клиент-сервер и, как и Matisse, допускает кэширование недавно использованной информации на уровне страниц на стороне сервера и на уровне объектов на стороне клиента.

При разработке Versant особое внимание было уделено блокировке и транзакциям. В ней можно блокировать отдельные объекты. Приложение может запросить чтение, обновление или запись заблокированного объекта. Обновления служат для устранения взаимных блокировок: если вам нужно прочесть заблокированный объект и записать в него, то требуется вначале запросить право на его обновление, предоставляемое при условии, что никакой другой клиент им в данный момент не пользуется. При этом остальные клиенты могут читать объект, пока не начнется (гарантированное) выполнение вашего запроса на запись. Непосредственный переход от чтения заблокированного объекта к записи мог бы привести к взаимной блокировке: каждый из двух клиентов мог бы ждать до бесконечности, пока другой не снимет свою блокировку.

Механизм транзакций обеспечивает как короткие, так и длинные транзакции; приложение может оставить объект на любое время. Поддерживаются версии объектов и оптимистическая блокировка.

Механизм запросов позволяет запросить все экземпляры класса, включая и экземпляры его собственных потомков. Как уже отмечалось, это позволяет добавлять новый класс без переопределения запросов, применимых к его ранее определенным предкам.

Другой особенностью Versant является ее механизм, позволяющий сообщать приложению о различных событиях в БД, например, об удалении или обновлении объекта. Получив такое извещение, приложение может выполнить предусмотренные на этот случай действия.

СУБД Versant предоставляет пользователям богатый набор типов данных, включая и множество заранее определенных коллекций классов. Это позволяет проводить эволюцию схемы при условии, что новые поля инициализируются предопределенными значениями. В ней также имеются возможности индексации, используемые в механизме запросов.

Обсуждение: за пределами ОО-баз данных

Завершим этот обзор мечтами о возможных направлениях развития сохраняемости в будущем. Следующие далее наблюдения носят предварительный характер, их цель - побудить дальнейшие размышления, а не предложить конкретные ответы.

Является ли "ОО-база данных" оксюмороном?

Понятие базы данных произошло от взгляда на мир, в центре которого сидят Данные, а расположенным вокруг программам разрешены доступ и модификация этих Данных:

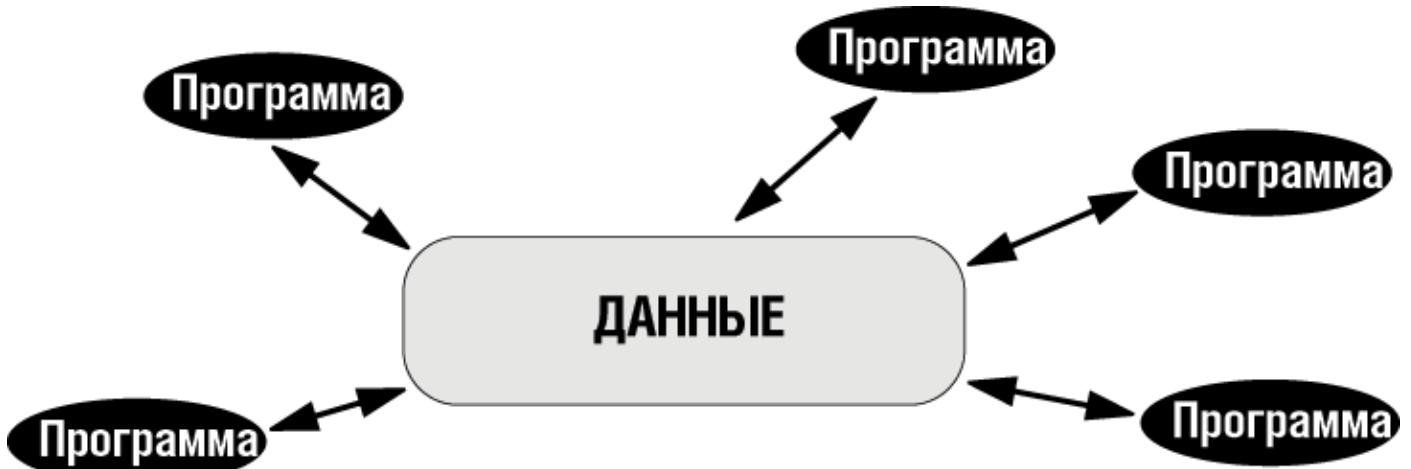


Рис. 13.7. Взгляд со стороны баз данных

Однако в объектной технологии мы научились понимать данные как сущности, полностью определяемые применяемыми к ним операциями:

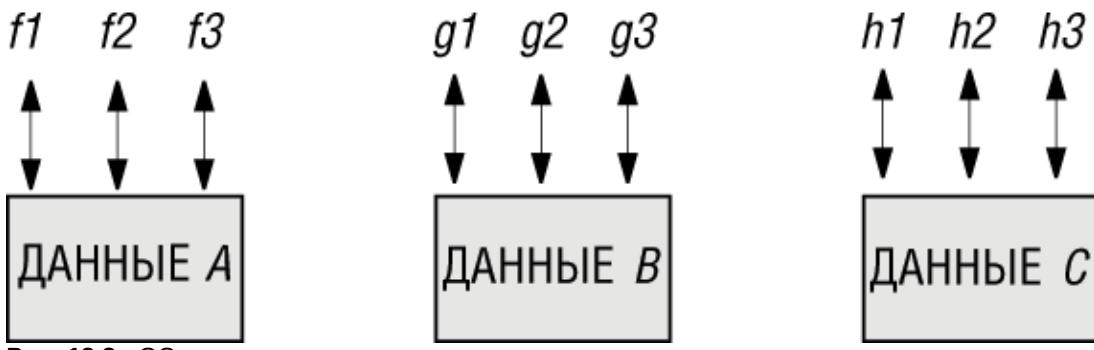


Рис. 13.8. ОО-взгляд

Эти два взгляда кажутся несовместимыми! Понятие данных, существующих независимо от обрабатывающих их программ ("независимость данных", догмат, повторяемый на первых страницах любой книги по БД) является проклятием для ОО-разработчика. Должны ли мы считать выражение "ОО-база данных" оксюмороном?¹⁾

Возможно, нет, но, быть может, стоит понять, как в догматическом ОО-контексте можно получить эффект баз данных, реально не имея их. Если мы дадим определение (упрощая до самого существенного данное ранее в этой лекции определение БД)

$$\text{БАЗА ДАННЫХ} = \text{СОХРАНЯЕМОСТЬ} + \text{РАЗДЕЛЕНИЕ ДАННЫХ},$$

то догматический взгляд будет рассматривать второй компонент, разделение данных, как несовместимый с ОО-идеями, и сосредоточится только на сохраняемости. Но тогда можно подойти к разделению данных, используя другой метод - параллелизм! Это показано на следующем рисунке.

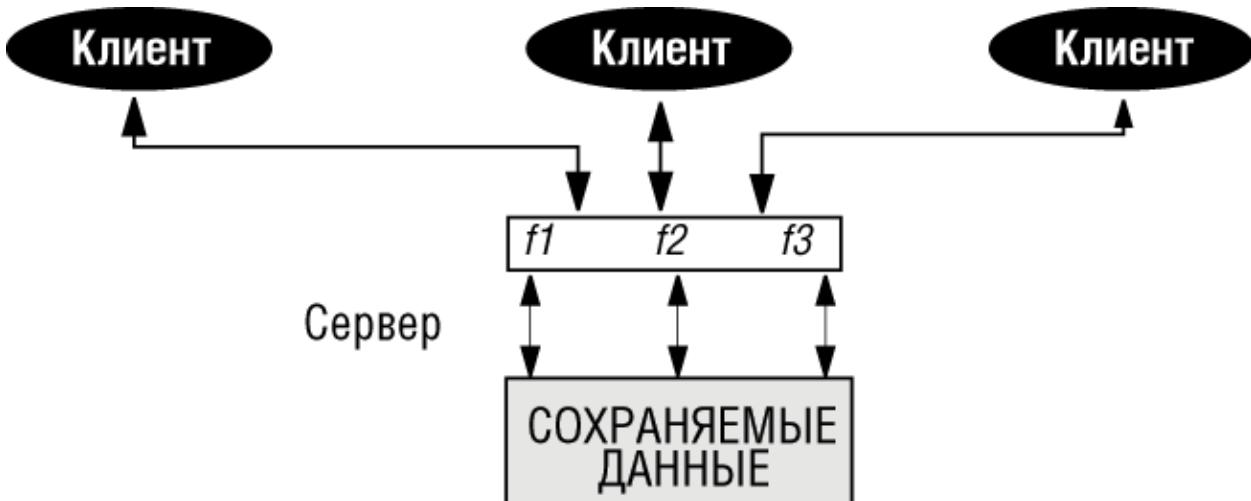


Рис. 13.9. Отделение сохраняемости от разделения данных

Следуя ОО-принципам, сохраняемые данные реализуются как множество объектов - экземпляров некоторых абстрактных типов данных - и управляются некоторой серверной системой. Системы клиентов, которым требуется работать с данными, будут делать это через сервер. Так как эта схема требует разделения и параллельного доступа, то клиенты будут рассматривать сервер как сепаратный в смысле, определенном при обсуждении параллельности в [лекции 12](#). Например:

```

flights: separate FLIGHT_DATABASE; ...
flight_details (f: separate FLIGHT_DATABASE;
               rf: REQUESTED_FLIGHTS): FLIGHT is
do
  Result := f.flight_details (rf)
end
reserve (f: separate FLIGHT_DATABASE; r: RESERVATION) is
do
  f.reserve (r); status := f.status
end
  
```

Тогда на стороне сервера не требуется никакого механизма разделения, а только общий механизм сохранения. Нам могут также понадобиться средства и методы для поддержки версий объектов, но они, по существу, относятся к вопросам сохраняемости объектов, а не к базам данных.

В этом случае механизм сохранения может стать чрезвычайно простым, отбросив многое из багажа БД. Можно даже считать, что все объекты по умолчанию являются постоянно хранимыми, а временные объекты становятся исключением, обрабатываемым механизмом, обобщающим сбор мусора. Такой подход, который невозможно было представить при изобретении БД, становится менее абсурдным при постоянном уменьшении стоимости памяти и росте доступности 64-битовых виртуальных адресных пространств, в которых, как было уже замечено в [Sombrero-Web],

"можно создавать каждую секунду новый 4-гигабайтный объект (вся память обычного 32-битного процессора) в течение 136 лет и все еще не исчерпать доступные адреса. Этого достаточно, чтобы сохранить все данные, связанные с почти любым приложением на протяжении всего его существования".

Все это весьма умозрительно и нет никаких оснований для того, чтобы отказываться от традиционного понятия БД. Не следует также немедленно мчаться и продавать акции компаний, создающих ОО-БД. Рассматривайте это обсуждение как интеллектуальное упражнение: приглашение прозондировать будущее общепринятого понятия ОО-БД, проверив, может ли существующий подход по-настоящему успешно справиться со страшным сопротивлением несогласованности между методом разработки ПО и механизмами, поддерживающими накопление и хранение данных.

Неструктурированная информация

Последнее замечание о БД. Взрывной рост Интернета и появление средств поиска, основанного на контексте (в момент написания книги наиболее известными примерами таких средств были AltaVista, Web Crawler и Yahoo), показал, что можно получать доступ к данным и при отсутствии БД.

СУБД требуют, чтобы перед сохранением любых данных вы сначала конвертировали их в строго определенный формат схемы БД. Недавние исследования, тем не менее, показали, что 80% электронных данных в компаниях являются неструктурированными (т. е. располагаются вне БД, как правило, в текстовых файлах), несмотря на многолетнее использование баз данных. Сюда и внедряются средства поиска по контексту: по заданным пользователем критериям, включающим ключевые слова и фразы, они могут извлечь данные из неструктурированных или минимально структурированных документов. Почти каждый, кто испробовал эти средства, был ослеплен блеском скорости, с которой они извлекают информацию: секунды или двух достаточно, чтобы найти иголку в стоге байтов размером в тысячи гигабайт. Это неизбежно приводит к вопросу: нужны ли нам на самом деле структурированные БД?

Пока еще ответ - да. Неструктурированные и структурированные данные будут сосуществовать. Но БД больше не являются единственной выбором; все более и более изощренные средства для запросов смогут извлекать информацию, даже если она не имеет формата, требуемого БД. Разумеется, для создания таких средств лучше всего подходит ОО-технология.

Ключевые концепции

- ОО-окружение должно позволять, чтобы объекты сохранялись - существовали и после завершения создавшей их сессии.
- Механизм сохраняемости должен предложить эволюцию схемы, чтобы преобразовывать на лету возвращаемые объекты к формату изменившегося породившего их класса ("рассогласование объекта"). Он решает три задачи: выявление, извещение и исправление. По умолчанию рассогласование должно возбуждать некоторое исключение.
- Кроме сохраняемости, многим приложениям требуется поддержка БД, обеспечивающая параллельный доступ разных клиентов.
- Другими свойствами БД являются запросы, блокировка и транзакции.
- ОО-разработку можно применять совместно с реляционными БД, имея в виду простое соответствие: классы - отношения, объекты - кортежи.
- Чтобы получить все выгоды от использования ОО-технологии и избежать сопротивления несогласованности между разработкой и моделью данных, можно использовать ОО-БД.
- Были рассмотрены две интересных ОО-СУБД: Matisse, в которой оригинально решены проблемы версий объектов и избыточности, и Versant, обладающей развитыми механизмами блокировок и транзакций.
- Некоторые вопросы были рассмотрены на уровне предварительного обсуждения: насколько совместимы принципы БД с ОО-взглядом на мир, каковы потребности в доступе как к структурированным, так и к неструктурированным данным.

Библиографические замечания

Первой работой о реляционной модели была статья [Codd 1970]; имеется множество книг по этой тематике. По-видимому, наиболее известный учебник по БД с упором на реляционную модель - книга [Date 1995] (это шестое издание книги, впервые опубликованной в середине семидесятых). Другой полезный текст общего назначения - [Elmasri 1989].

[Walden 1995] содержит детальное практическое обсуждение того, как выполнять работу по объектно-реляционному взаимодействию. [Khoshafian 1986] выдвигает вопрос об идентификации объектов на передний план обсуждения ОО-БД.

Хорошей отправной точкой для понимания целей ОО-БД и чтения некоторых оригинальных статей является сборник [Zdonik 1990], включающий работы некоторых пионеров этой области, а его вводная лекция явилась источником понятия "пороговой модели", использованного в этой лекции. Широко циркулирующий "Манифест ОО-систем баз данных" [Atkinson 1989], являющийся результатом сотрудничества большого числа экспертов, существенно повлиял на цели движения ОО-БД. Сейчас имеется много учебников по этой тематике, перечислим некоторые из наиболее известных (в порядке публикации): [Kim 1990], [Bertino 1993], [Khoshafian 1993], [Kemper 1994], [Loomis 1995]. Дальнейшие постоянно обновляемые ссылки можно найти в он-лайн библиографии Майкла Лея (Michael Ley) по системам БД [Ley-Web]. Группа Клауса Дитриха (Klaus Dittrich) в Цюрихском университете ведет список часто задаваемых вопросов о ОО-БД http://www_ifi.unizh.ch/groups/dbtg/ObjectDB_FAQ.html. В [Cattel 1993] описан стандарт ODMG. Оценки (несколько вымученные) достижений и неудач ОО-БД одного из пионеров этой области можно

найти в [Stein 1995].

Эта лекция очень выиграла от важных комментариев Ричарда Биляка (Richard Bielak), особенно об эволюции схем, Замыкании Сохраняемости, запросам к ОО-БД, Versant и Sombrero. Описание Versant базируется на [Versant 1994], а Matisse - на [ADB 1995] (см. также <http://www.adb.com/techow/features.html>). Я благодарен Ш. Финкельштейну (Shel Finkelstein) за помощь в знакомстве со свойствами Matisse. Система O2 описана в [Banchilhon 1992]. Проект Sombrero [Sombrero -Web] исследует влияние большого адресного пространства на традиционные подходы к сохраняемости и БД.

Предварительный вариант материала этой лекции, посвященного эволюции схем, появился в [M 1996c]. Обсуждение вопроса о том, как соответствуют друг другу ОО-понятия и понятия БД, пришло из неопубликованных заметок лекций, прочитанных в 1995 г. на конференциях TOOLS USA и European Software Engineering Conference [M 1996d].

Упражнения

У13.1 Динамическая эволюция схем

Предложите, как расширить методы эволюции схем, развитые в этой лекции, чтобы учесть случай, при котором классы программной системы могут изменяться во время выполнения системы.

У13.2 Объектно-ориентированные запросы

Обсудите, в каком виде могут формулироваться запросы в ОО-БД.

-
- 1) Оксюморон (oxymoron) - соединение несовместимых понятий (горячий лед, оглушительная тишина).

Основы объектно-ориентированного проектирования

14. Лекция: ОО-метод для графических интерактивных приложений

Элегантный интерфейс пользователя стал неотъемлемой частью всякого успешного программного продукта. Достижения в создании новых мониторов, эргономике (изучении человеческого фактора) и в разработке ПО привели к всеобщему распространению интерактивных методов и средств, прокладывающих себе дорогу с семидесятых. Многооконные системы позволяют выполнять одновременно несколько работ, мышь позволяет быстро указывать на требуемый элемент, меню ускоряет возможность выбора, значки представляют визуальные понятия, рисунки демонстрируют информацию визуально, кнопки выполняют стандартные операции. Аббревиатура GUI, обозначающая Graphical User Interface (Графический Интерфейс Пользователя), служит общим лозунгом для такого стиля взаимодействия. Связанные с ним модные термины - WYSIWYG (What You See Is What You Get (Что Вы видите, то и имеете)), WIMP (Windows, Icons, Menus, Pointing device (Окна, Значки, Меню, Указатели)) и фраза "прямое манипулирование" - характеризуют приложения, у пользователей которых создается впечатление, что они работают непосредственно с объектами, изображенными на экране. Эти впечатляющие средства, ранее доступные только пользователям самых передовых систем, работающих на дорогостоящем оборудовании, сейчас стали стандартными для самых обычных персональных машин. Настолько стандартными и популярными, что разработчики ПО заведомо постигнут неудачу, если в его продукте будет использован не графический интерфейс, а строковый или полноэкранный. Еще до недавнего времени построение интерактивных приложений с развитыми графическими возможностями представлялось весьма трудным делом, подтверждающим то, что можно назвать гипотезой об интерфейсе: чем более удобным и простым приложение кажется пользователю, тем тяжелее его создать разработчику. Вспоминая достижение последних лет в области ПО явились опровергнувшие гипотезы об интерфейсе. Произошло это благодаря появлению хороших средств разработки, таких как конструкторы интерфейсов. Объектная технология оказалась здесь чрезвычайно полезной. Области, обозначаемые двумя модными терминами GUI и ОО, имеют тесно связанную историю. Цель этой лекции - показать, как опровергается гипотеза об интерфейсе: приложение, дружественное для пользователя, не должно быть вражеским для разработчика. ОО-методы помогут нам выделить надлежащие абстракции данных и повторно использовать все, что только может быть переиспользовано. Полное изложение ОО-методов для построения графических и интерактивных приложений потребовало бы отдельной книги. В этой лекции у нас более скромная цель. В ней выбрано несколько не самых простых аспектов построения GUI и описано несколько важных методов, широко применимых при создании графических систем.

Недавно Знаменитый Конструктор сконструировал новый автомобиль. У него не было ни прибора с показателем горючего, ни спидометра, ни множества других глупых кнопок и табло, заполонивших современные машины. Вместо этого в нужный момент посреди приборной доски загорался большой "?". "Опытный водитель", - сказала Знаменитость, - "всегда знает, что вышло из строя".

Фольклор Unix'a. (Вместо "Знаменитый Конструктор" использовалось настоящее имя одного из главных создателей ОС Unix.)

Необходимые средства

Какие средства нужны для создания полезных и приятных интерактивных приложений?

Конечные пользователи, разработчики приложений и разработчики инструментальных средств

Для устранения непонимания начнем с терминологии. Слово "пользователь" (одно из самых оскорбительных для компьютерщиков) здесь может ввести в заблуждение. Некоторые люди, называемые **разработчиками приложений**, создают интерактивные приложения, используемые другими людьми, называемыми **конечными пользователями**. Разработчики приложений, в свою очередь, рассчитывают на графические средства, созданные третьей группой - **разработчиками инструментальных средств**. Существование этих трех категорий объясняет, почему слово "пользователь" без дальнейшего уточнения неоднозначно: конечные пользователи являются пользователями разработчиков инструментальных средств.

Приложение - это интерактивная система, созданная разработчиком. Конечный пользователь начинает сессию и исследует возможности системы, задавая ей различные входы. Сессии для приложений, что объекты для классов: индивидуальные экземпляры общего образца.

Проанализируем потребности разработчиков, желающих предоставить своим конечным пользователям полезные приложения с графическим интерфейсом.

Графические системы, оконные системы, инструментальные средства

Многие вычислительные платформы предлагают средства для построения графических интерактивных приложений. Для реализации графики имеются соответствующие библиотеки такие, как GKS и PHIGS. Что касается интерфейса пользователя, то базовые оконные системы (такие, как Windows API, Xlib API под Unix'ом и Presentation Manager API под OS/2) имеют чересчур низкий уровень, чтобы ими было удобно пользоваться разработчикам приложений, но они дополняются "инструментариями", например, основанными на протоколе интерфейса пользователя Motif.

Все эти системы, удовлетворяя определенным потребностям, недостаточны для выполнения всех требований разработчиков. Перечислим некоторые ограничения.

- Их трудно использовать. Чтобы освоить инструментальные средства, основанные на протоколе Motif, разработчики должны изучить многотомную документацию, описывающую сотни встроенных функций на Си и структур, носящих такие впечатляющие благородный ужас имена как XmPushButtonCallbackStruct, где в Button буква В большая, а в back - b малая. К трудностям и небезопасности С добавляется сложность инструментария. Использование базового интерфейса программирования приложений API в Windows также утомительно.
- Хотя предлагаемый инструментарий включает объекты пользовательского интерфейса - кнопки, меню и т. п., - у некоторых из них хромает графика (геометрические фигуры и их преобразования). Для добавления в интерфейс

настоящей графики требуются значительные усилия.

- Различные инструментальные средства несовместимы друг с другом. Графика Motif, Windows и Presentation Manager, основанная на похожих понятиях, имеет множество различий. Некоторые из них существенны. Так, в Windows и PM создаваемый объект интерфейса сразу же выводится на экран, а в Motif сначала строится соответствующая структура, а затем вызов операции "реализовать" ее показывает. Некоторые различия связаны с разными соглашениями (координаты экрана откладываются от верхнего левого угла в PM и от нижнего левого угла у других). Многие соглашения об интерфейсах пользователя также различны. Большинство этих различий доставляет неприятности конечным пользователям, желающим иметь нечто работающее и "приятно выглядящее" и которым неважно, какие углы у окна - острые или слегка закругленные. Эти различия еще больше неприятны разработчикам, которые должны выбирать между потерей части их потенциального рынка итратой драгоценного времени на усилия по переносу.

Библиотека и конструктор приложений

Чтобы удовлетворить нужды разработчиков и дать им возможность создавать приложения, устраивающие их конечных пользователей, требуется пойти дальше инструментария. Разработчикам необходимо предоставить переносимые средства высокого уровня, освобождающие их от утомительной и регулярно повторяющейся работы, позволив им посвятить свое творчество действительно новаторским аспектам.

Наборы инструментов содержат много необходимых механизмов и дают хорошую основу. Остается скрыть лишние детали и пополнить их полезными средствами.

В основе предлагаемого решения лежит библиотека повторно используемых классов. Эти классы поддерживают основные абстракции данных, отвечающие понятиям: окно, меню, контекст, событие, команда, состояние, приложение.

Для решения некоторых задач, встречающихся при создании приложения, разработчикам было бы удобней не писать, как обычно, тексты программ, но использовать интерактивную систему, называемую конструктором приложений. Такая система позволяет им выражать свои потребности в графическом, WYSIWIG виде; иными словами, использовать в их собственной работе стиль интерфейса, который они предлагают своим пользователям. Конструктор приложений - это инструмент, чьими конечными пользователями являются сами разработчики, они используют конструктор приложений для создания тех частей своих систем, которые могут быть заданы визуально и интерактивно. Термин "конструктор приложений" показывает, что это средство гораздо более амбициозное, чем простой "конструктор интерфейса", позволяющий создавать только интерфейс приложения. Конструктор приложения должен идти дальше в представлении структуры и семантики приложения, останавливаясь только в том случае, когда для решения некоторой подзадачи необходимо написать некоторый программный код.

При определении библиотеки и конструктора приложений, как всегда, будем руководствоваться критериями повторного использования и расширяемости. В частности, это означает, что для каждой описываемой ниже абстракции данных (такой, как контекст, команда или состояние) конструктор приложения должен предоставить два средства:

- для повторного использования - каталог (событий, контекстов, состояний и т. д.), содержащий заранее определенные образцы данной абстракции, которые могут быть непосредственно включены в приложение;
- для расширяемости - редактор (контекстов, команд, состояний и т. д.), позволяющий разработчикам создавать собственные варианты либо с самого начала, либо, выбрав некоторый элемент из каталога и изменив его нужным образом.

Применение ОО-подхода

В ОО-подходе ключевым шагом является выбор правильных абстракций данных: типов объектов, характерных для данной проблемной области.

Для понимания графических интерфейсов пользователей и разработки хороших механизмов создания приложений требуется проанализировать соответствующие абстракции. Некоторые из них очевидны, другие окажутся более тонкими.

Каждая из перечисленных ниже абстракций будет давать хотя бы один библиотечный класс. Некоторые потребуют множества классов, являющихся потомками общего предка, описывающего их самые общие свойства. Например, в библиотеке имеется несколько классов, описывающих разные варианты понятия меню.

Сначала рассмотрим общую структуру переносимой графической библиотеки, затем - основные графические абстракции выводимых на экран геометрических объектов и "объектов взаимодействия", поддерживающих управляемые событиями диалоги, а в конце изучим более продвинутые абстракции, описывающие приложения: команды, состояния, само приложение.

Переносимость и адаптация к платформе

Некоторые разработчики приложений предпочитают переносимые библиотеки, позволяющие написать один исходный текст системы. Для переноса системы на другую платформу достаточно ее перекомпилировать, не внося никаких изменений. Другие хотели бы обратного: получить полный доступ ко всем специфическим элементам управления и прочим "штучкам" конкретной платформы, например, Microsoft Windows, но в удобном виде (а не на низком уровне стандартных библиотек). Третьи хотели бы понемногу и того и другого: переносимости по умолчанию и возможности, если потребуется, стать "родным" для данной платформы.

При аккуратном проектировании, основанном на двухуровневой структуре, можно попытаться удовлетворить все три группы:

Библиотека, не зависящая от платформы (Vision)

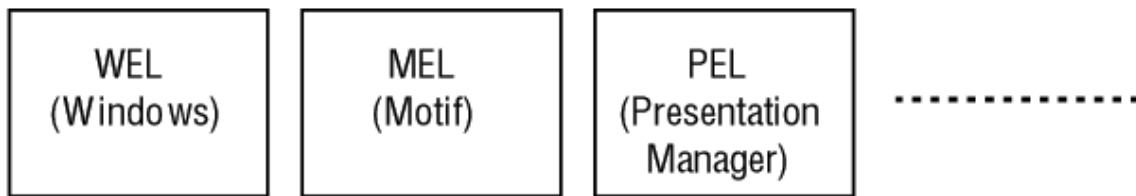


Рис. 14.1. Архитектура графической библиотеки

Для конкретизации на рисунке приведены имена соответствующих компонентов из окружения ISE, но идея применима к любой графической библиотеке. На верхнем уровне (Vision) находится переносимая графическая библиотека, а на нижнем уровне - специализированные библиотеки, такие как WEL для Windows, каждая из них приспособлена к "своей" платформе.

WEL и другие библиотеки нижнего уровня можно использовать непосредственно, но они также служат как зависящие от платформы компоненты верхнего уровня: механизмы Vision реализованы посредством WEL для Windows, посредством MEL для Motif и т. д. У такого подхода несколько преимуществ. Разработчикам приложений он дает надежду на совместимость понятий и методов. Разработчиков инструментальных средств он избавляет от ненужного дублирования и облегчает реализацию высокого уровня, базирующегося не на прямом всегда опасном интерфейсе с C, а на ОО-библиотеках, снабженных утверждениями и наследованием, таких как WEL. Связь между этими двумя уровнями основана на описателях (см. [лекцию 6](#)).

У разработчиков приложений имеется выбор:

- Для обеспечения переносимости следует использовать верхний уровень. Он также представляет интерес для разработчиков, которые, даже работая для одной платформы, хотят выиграть от более высокой степени абстракций, предоставляемых такими библиотеками высокого уровня, как Vision.
- Для получения прямого доступа ко всем специфическим механизмам некоторой платформы (например, многочисленным элементам управления, предоставляемым Windows NT), следует перейти на соответствующую библиотеку нижнего уровня.

Рассмотрим один тонкий вопрос. Как много специфических для данной платформы возможностей допустимо потерять при использовании переносимой библиотеки? Корректный ответ на него является результатом компромисса. Некоторые из первых переносимых библиотек использовали подход **пересечения** ("наименьшего общего знаменателя"), ограничивающий предлагаемые возможности теми, которые предоставляются всеми поддерживаемыми платформами. Как правило, этого недостаточно. Авторы библиотек могли использовать и противоположный подход - **объединения**: предоставить каждый из механизмов каждой из поддерживаемых платформ, используя точные алгоритмы для моделирования механизмов, первоначально отсутствующих на той или иной платформе. Такая политика приведет к огромной и избыточной библиотеке. Правильный ответ находится где-то посередине: для каждого механизма, присутствующего не на всех plataформах, авторы библиотеки должны отдельно решать, достаточно ли он важен для того, чтобы промоделировать его на всех plataформах. Результатом должна явиться согласованная библиотека, достаточно простая, чтобы использоваться без знаний особенностей отдельных платформ, и достаточно мощная для создания впечатляющих графических приложений.

Для разработчиков приложений еще одним критерием в выборе между двумя уровнями служит эффективность. Если основной причиной выбора верхнего уровня служит абстрактность, а не переносимость, то можете быть уверены - включение дополнительных классов приведет к потерям в памяти. Для правильно спроектированных библиотек потерями времени можно обычно пренебречь. Поэтому эффективность по памяти определяет, нужно ли это делать. Ясно, что библиотека для одной платформы (например, WEL) будет более компактной.

Наконец, заметим, что оба решения не являются полностью взаимоисключающими. Можно сделать основную часть работы на верхнем уровне, а затем добавить "бантики" для пользователей, работающих с библиотекой для самой продаваемой платформы. Конечно, это следует делать аккуратно, беззаботное смешение переносимых и непереносимых элементов быстро ликвидирует любую ожидаемую выгоду от переносимой разработки. Один элегантный образец проекта (используемый ISE в некоторых своих библиотеках) основан на попытке присваивания (см. [лекцию 16](#) курса "Основы объектно-ориентированного программирования"). Его идея в следующем. Рассмотрим некоторый графический объект, известный через сущность *m*, тип которой определен на верхнем уровне, например, MENU. Всякий актуальный объект, к которому она будет присоединяться во время исполнения, будет, конечно, специфичным для платформы, т. е. будет экземпляром некоторого класса нижнего уровня, скажем, WEL_MENU. Для применения специфических для платформы компонентов требуется некоторая сущность этого типа, скажем *wm*. Далее можно воспользоваться следующей схемой:

```
wm ?= m
if wm = Void then
... Мы не под Windows! Ничего не делать или заниматься другими делами...
else
```

```
... Здесь можно применить к wm любой специфический для Windows компонент WEL_MENU ...  
end
```

Можно описать эту схему, как путь в комнату Windows. Эта комната закрыта, не позволяя утверждать, если кто-нибудь вас в ней обнаружит, что вы попали туда случайно. Вам разрешается в нее войти, но для этого вы должны открыто и вежливо попросить ключ. Попытка присваивания является официальной просьбой разрешения войти в область специального назначения.

Графические абстракции

Многие приложения используют графические изображения для представления объектов внешней системы. Рассмотрим простое множество абстракций, покрывающее эти потребности.

Фигуры (изображения)

Прежде всего нам нужно подходящее множество абстракций для графической части интерактивного приложения. Для простоты мы будем рассматривать только двухмерную графику.

Прекрасную модель представляют географические карты. Карта (страны, области, города) дает визуальное представление некоторой реальности. Проектирование карты использует несколько уровней абстракции:

- Мы должны видеть реальность, стоящую за моделью (в почти абстрактном виде), как множество геометрических форм или фигур. На карте эти фигуры представляют реки, дороги, города и другие географические объекты.
- Карта описывает некоторое множество фигур, называемое миром.
- Карта показывает только часть мира - одну или более областей, называемых окнами. Окна имеют прямоугольную форму. Например, у карты может быть одно главное окно, посвященное стране, и вспомогательные окна, посвященные большим городам или удаленным частям (например, Корсике на картах Франции или Гавайям на картах США).
- Физически карта появляется на физическом носителе изображения, устройстве. Этим устройством обычно является лист бумаги, но им может быть и экран компьютера. Различные части устройства будут предназначены для разных окон.

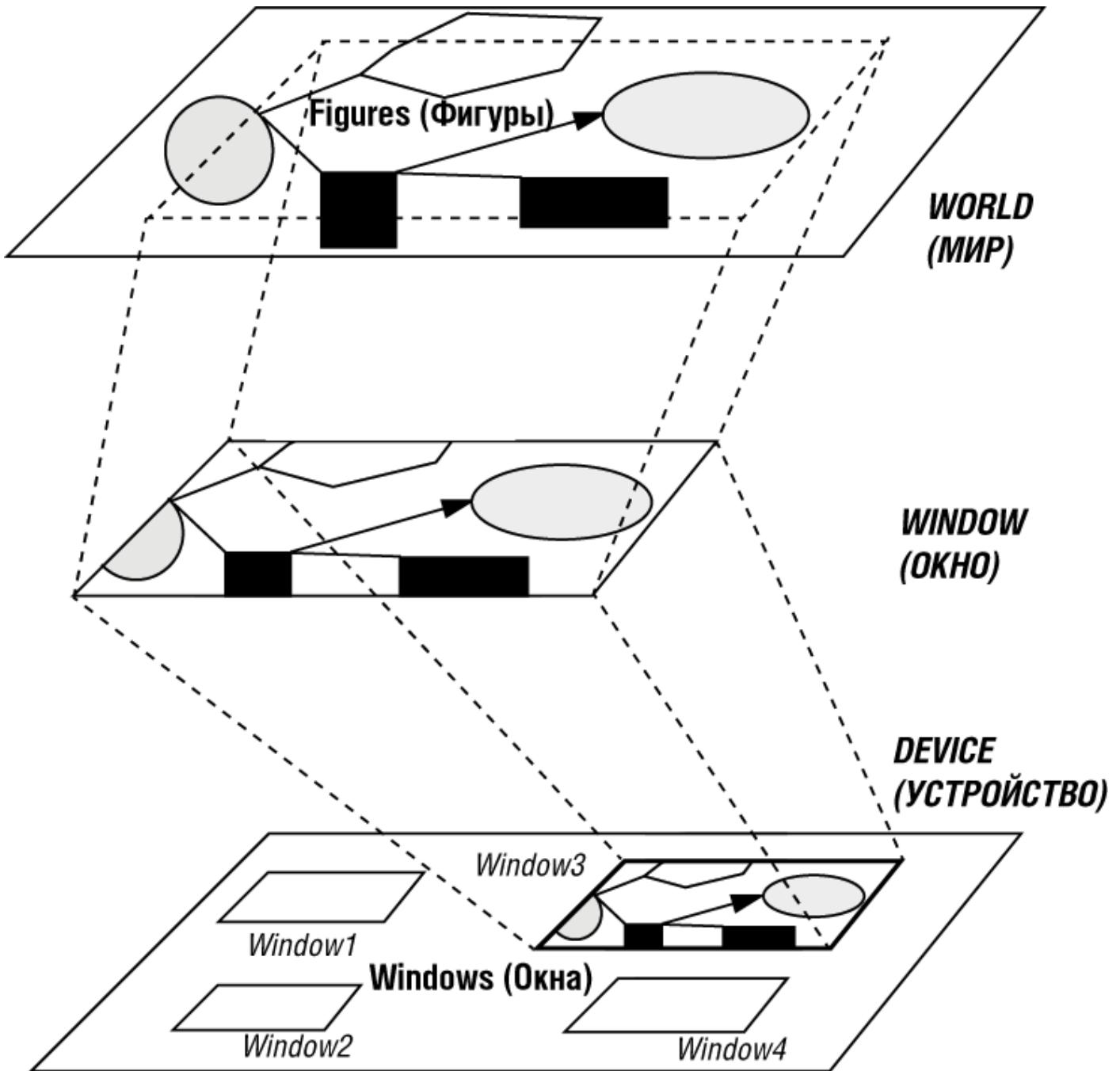


Рис. 14.2. Графические абстракции

Четыре базовых понятия - WORLD, FIGURE, WINDOW, DEVICE - легко переносятся на общие графические приложения, в которых мир может содержать произвольные фигуры, представляющие интерес для некоторого компьютерного приложения, а не только представления географических объектов. Прямоугольные области мира (окна) будут изображаться на прямоугольных областях устройства (экрана компьютера).

На [рис. 14.2](#) показаны три плоскости: мир (вверху), окно (посредине) и устройство (внизу). Понятие окна играет центральную роль, поскольку каждое окно связано как с некоторой областью мира, так и с некоторой областью устройства. С окнами также связано единственное существенное расширение базовых понятий - поддержка иерархически вложенных окон. У наших окон могут быть подокна (без всяких ограничений на уровень вложенности). (На рисунке вложенных окон нет.)

Координаты

Нам нужны две системы координат: координаты устройства и мировые координаты. Координаты устройства задают положения элементов, изображаемых на этом устройстве. На экранах компьютеров они часто измеряются в пикселях; пиксель (элемент изображения) - это размер маленькой точки, обычно самого малого из изображаемых элементов.

Стандартной единицы для мировых координат нет и быть не может - систему мировых координат лучше оставить разработчикам: астрономы могут пожелать работать со световыми годами, картографы - с километрами, биологи - с миллиметрами или микронами.

Так как окно отражает часть мира, то у него есть некоторое местоположение (определенное мировыми координатами x и y его верхнего левого угла) и некоторые размеры (длина по горизонтали и вертикали соответствующей части мира). Местоположение и размеры выражаются в единицах мировых координат.

Так как окно изображается на части устройства, то у него имеется некоторое местоположение на устройстве (определенное координатами *x* и *y* его верхнего левого угла на устройстве) и некоторые размеры на устройстве, все выражаемые в единицах координат устройства. Для окна, не имеющего родителя, местоположение определяется по отношению к устройству, а для подокна местоположение определяется по отношению к его родителю. Благодаря этому соглашению всякое приложение, использующие окна, может выполняться как внутри всего экрана, так и в предварительно размещенном окне.

Операции над окнами

Принимая во внимание иерархическую природу окон, мы сделаем класс WINDOW наследником класса TWO_WAY_TREE, реализующего деревья. В результате все иерархические операции легко доступны как операции на деревьях: добавить подокно (вершину-ребенка), переподчинить другому окружающему окну (другому родителю) и т. д. Для задания положения окна в мире и в устройстве будем использовать следующие процедуры (все с двумя аргументами):

Таблица 14.1. Установка позиций окна

	Установка абсолютного положения	Сдвиг относительно текущей позиции
Положение в мире	go	pan
Положение на устройстве	place_proportional place_pixel	move_proportional move_pixel

Процедуры _proportional интерпретируют значения своих аргументов как отношение высоты и ширины окна родителя, а аргументами остальных процедур являются абсолютные значения (в мировых координатах для go и pan, и в координатах устройства для процедур _pixel). Имеются аналогичные процедуры и для задания размеров окна.

Графические классы и операции

Все классы, представляющие фигуры, являются наследниками отложенного класса FIGURE, среди стандартных компонентов которого имеются display (**показать**), hide (**скрыть**), translate (**сдвинуть**), rotate (**поворнуть**), scale (**масштабировать**).

Безусловно, множество фигур должно быть расширяемым, позволяя разработчикам приложений (и, опосредованно, конечным пользователям графических средств) определять их новые типы. Мы уже видели, как это можно сделать: предоставить класс COMPOSITE_PICTURE, построенный с помощью множественного наследования из класса FIGURE и такого типа контейнера, как LIST [FIGURE].

Механизмы взаимодействия

Обратим теперь внимание на то, как наши приложения будут взаимодействовать с пользователями.

События

Современные интерактивные приложения управляются событиями: после того, как интерактивный пользователь своими действиями вызывает появление некоторых событий (их примеры - ввод текста с клавиатуры, движение мыши или нажатие кнопок), выполняются соответствующие им операции.

Хотя это описание выглядит вполне безобидно, в нем заключено главное отличие от традиционных стилей взаимодействия с пользователями. Программа, написанная в старом стиле (еще достаточно распространенном), получает ввод от пользователя, последовательно выполняя сценарий:

```
... Выполняет некоторые вычисления ...
print ("Введите, пожалуйста, значение параметра xxx.")
read_input
xxx := value_read
... Продолжает вычисление до тех пор, пока снова не потребуется получить
которое значение от пользователя ...
```

Когда вычислением управляют события, происходит перемена ролей: операции выполняются не оттого, что программа дошла до некоторого заранее заданного этапа своей работы, но потому, что какое-то событие, обычно инициированное интерактивным пользователем, вызвало выполнение некоторого компонента ПО. Входы определяют выполнение ПО, а не наоборот.

ОО-стиль проектирования ПО играет важную роль в реализации такой схемы. В частности, динамическое связывание позволяет программе вызывать компонент объекта, понимая, что тип объекта определяет, как он будет выполнять этот компонент. Вызов компонента может быть связан с событием.

Понятие события настолько важно в этом обсуждении, что заслуживает своей абстракции данных. Объект события (экземпляр класса EVENT) будет представлять действие пользователя, например, нажатие клавиши, движение мыши, щелчок кнопкой мыши, двойной щелчок и т. д. Эти предопределенные события будут частью каталога событий.

Кроме того, должна быть возможность определять в программах собственные события, сообщения о появлении которых компоненты ПО могут посыпать в явном виде с помощью процедуры вида raise(e).

Контексты и объекты интерфейса пользователя

Инструментальные средства GUI предлагают множество готовых "Объектов интерфейса пользователя": окна, меню, кнопки, панели. Вот пример кнопки ОК.



Рис. 14.3. Кнопка OK

По внешнему виду объект интерфейса пользователя - это просто некоторая фигура. Но в отличие от фигур, рассмотренных ранее, он, как правило, не имеет никакого отношения к окружающему миру: его роль ограничивается обработкой входа от пользователя. Точнее говоря, объект интерфейса пользователя представляет специальный случай **контекста**.

Для понимания необходимости контекста, заметим, что по одному событию в общем случае невозможно определить правильный ответ ПО. Например, нажатие кнопки мыши дает разные результаты в зависимости от того, где находится курсор мыши. Контекст - это условия, полностью определяющие отклик приложения на появление события.

Тогда в общем случае контекст - это просто логическое значение, т. е. значение, которое будет истинно или ложно в каждый момент выполнения ПО.

Наиболее общие контексты связаны с объектами интерфейса пользователя. Показанная выше кнопка задает логическое условие-контекст "курсор мыши на кнопке (внутри)?" Контексты такого рода будут записываться в виде IN (uiο), где uiο - это объект интерфейса пользователя.

Для каждого контекста с его отрицанием not с также является контекстом; not IN (uiο) называется также OUT (uiο). Контекст ANYWHERE всегда истинен, а его отрицание NOWHERE всегда ложно.

У нашего конструктора приложений будет каталог контекстов, в который будут входить ANYWHERE и контексты вида IN(uiο) для всех объектов интерфейса пользователя uiο. Кроме того, хочется предоставить разработчикам приложений возможность определять собственные контексты, для этой цели конструктор приложений предоставит специальный редактор. Среди прочего, этот редактор позволит получать контекст not с по любому с (в частности, и по с из каталога).

Обработка событий

Теперь у нас есть список событий и список контекстов, для которых эти события могут быть значимы. Ответы будут включать **команды** и **метки переходов**.

Команды

Выделение команд как важной абстракции является ключевым шагом в разработке хороших интерактивных приложений.

Это понятие было изучено в [лекции 3](#) в качестве учебного примера для операции отката Undo. Напомним, что объект команда содержит информацию, необходимую для выполнения запрошенной пользователем операции и, если поддерживается Undo, то и для отмены ее действия.

К обсужденным выше компонентам добавим еще атрибут exit_label, объясняемый ниже.

Базисная схема

Контексты, события и команды образуют базисные ингредиенты для определения основной операции интерактивного приложения, поддерживаемой конструктором приложений: разработчик приложения будет перебирать интересующие его пары вида контекст-событие (какие события распознаются в каких контекстах) и для каждой из них определять связанную с ней команду.

Эта важная идея может быть реализована как первая версия конструктора приложений. У него должны быть каталоги контекстов и событий (базирующиеся на имеющихся инструментальных средствах), а также каталог команд (предоставленный окружением разработки и допускающий расширение разработчиками приложений). Графическая метафора должна позволить выбрать нужную комбинацию контекст-событие, например, щелчок левой кнопкой мыши на некоторой кнопке, и команду, которая будет выполняться в ответ на эту комбинацию.

Состояния

Более полная схема включает дополнительный уровень абстракции, дающий модель **Контекст-Событие-Команда-Состояние (Context-Event-Command-State)** интерактивных графических приложений.

Вообще говоря, комбинация контекст-событие не всегда должна приводить к одинаковому действию в приложении. Например, во время сессии может оказаться ситуация, когда часть экрана выглядит так:

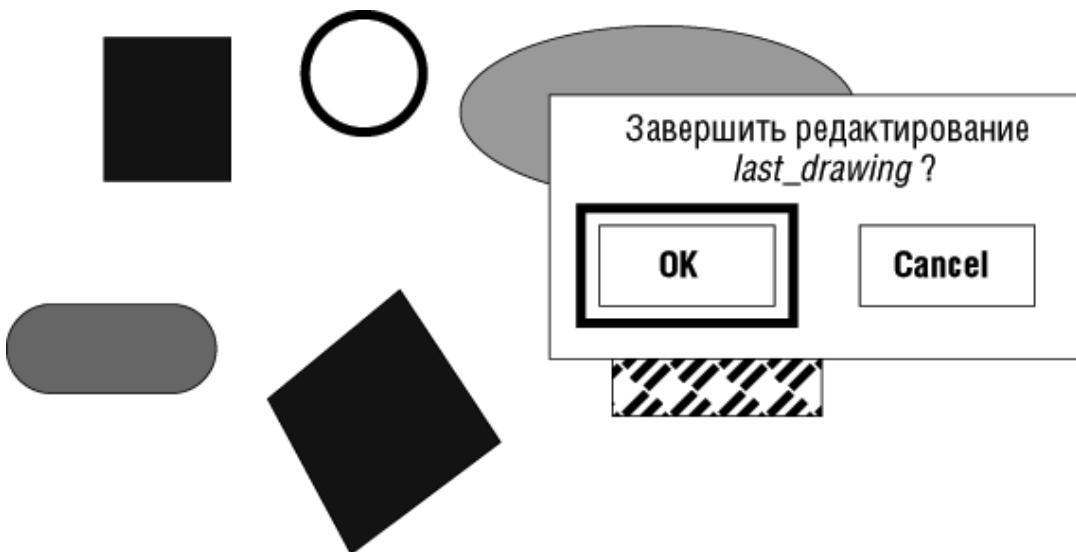


Рис. 14.4. Команда выхода

В этом состоянии приложение распознает различные события в различных контекстах; например, можно щелкнуть по фигуре, чтобы ее передвинуть, или запросить команду сохранения Save, щелкнув кнопку OK. В этом последнем случае появляется новая панель:

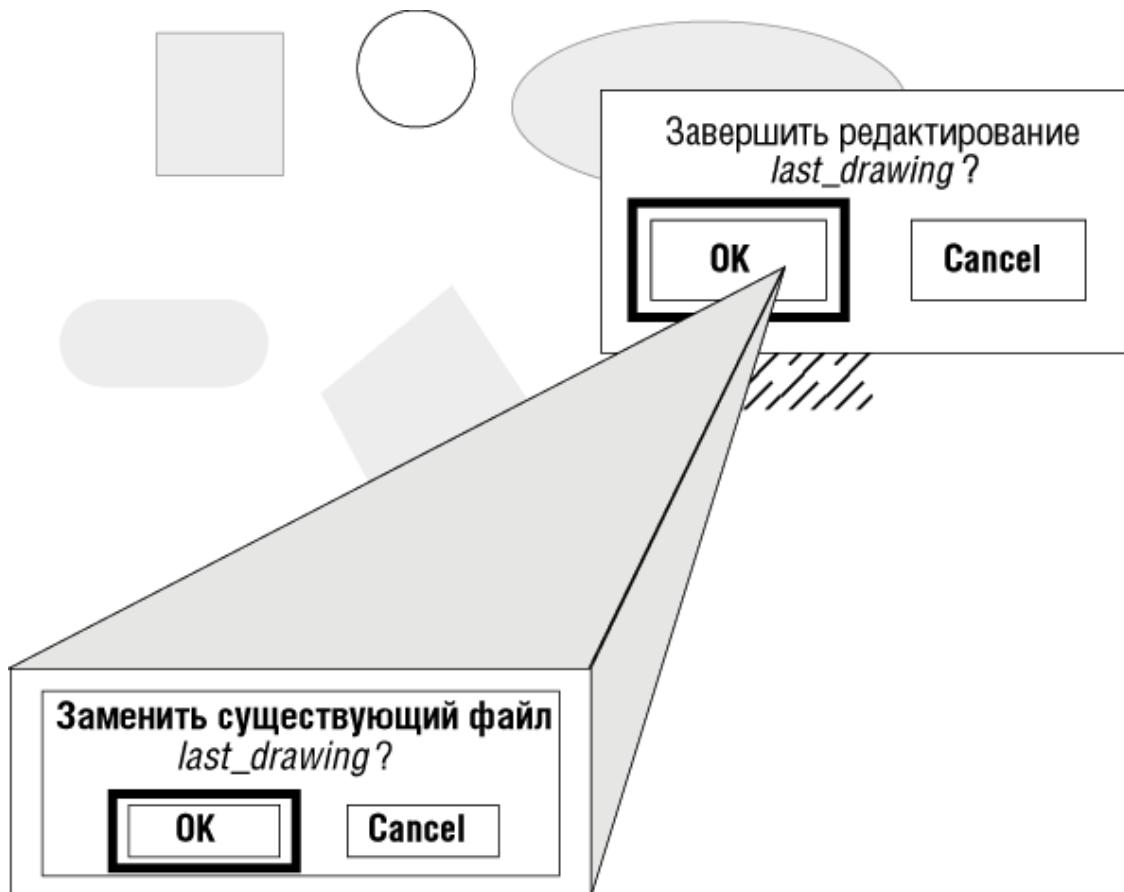


Рис. 14.5. Подтверждение команды

На этой стадии будут допустимы только две комбинации контекст-событие: щелчок кнопки OK или щелчок кнопки Cancel на новой панели. Приложение сделает остальную часть изображения "серой", напомнив тем самым, что все, кроме этих двух кнопок, временно не активно. Сессия перешла в новое состояние. Понятие состояния, также иногда называемого режимом, знакомо по дискуссиям об интерактивных системах, но редко определялось точно. Теперь у нас есть предпосылки для формального определения: состояние характеризуется множеством допустимых в нем комбинаций контекстов и событий и множеством команд; для каждой допустимой комбинации контекст-событие состояние задает связанную с ней команду. Ниже это будет переформулировано в виде математического определения.

У многих интерактивных приложений, не только графических, будет несколько состояний.

Типичным примером является хорошо известный редактор Vi, работающий под Unix. Поскольку это не графическое средство, то событиями являются нажатия на клавиши (каждой клавише клавиатуры соответствует свое событие), а контекстами являются различные возможные положения курсора (на некотором символе, в начале строки, в конце строки и т. п.). Грубый анализ показывает, что у Vi по крайней мере четыре состояния.

- В основном состоянии (которое является также начальным для конечного пользователя, вызывающего редактор на новом файле) нажатие на клавишу с буквой будет в большинстве случаев приводить к выполнению команды, связанной с этой буквой. Например, нажатие на `x` удаляет символ в позиции курсора, если таковой символ имеется, двоеточие переводит в командное состояние, нажатие на `i` переводит в состояние вставки, а нажатие `R` переводит в состояние замены. Некоторые символы не определяют события, например, нажатие `z` не имеет эффекта (если с ним не связан какой-либо макрос).
- В командном состоянии единственное, что допустимо, - это ввод команд в окне Vi, таких как "save" или "restart".
- В состоянии вставки в качестве событий допустимы нажатия клавиш с печатаемыми символами, при этом соответствующий символ вставляется в текст, вызывая сдвиг имеющегося текста вправо. Клавиша ESCAPE возвращает сессию в основное состояние.
- Состояние замены является вариантом состояния вставки, в котором печатаемые символы заменяют существующие, а не сдвигают их.

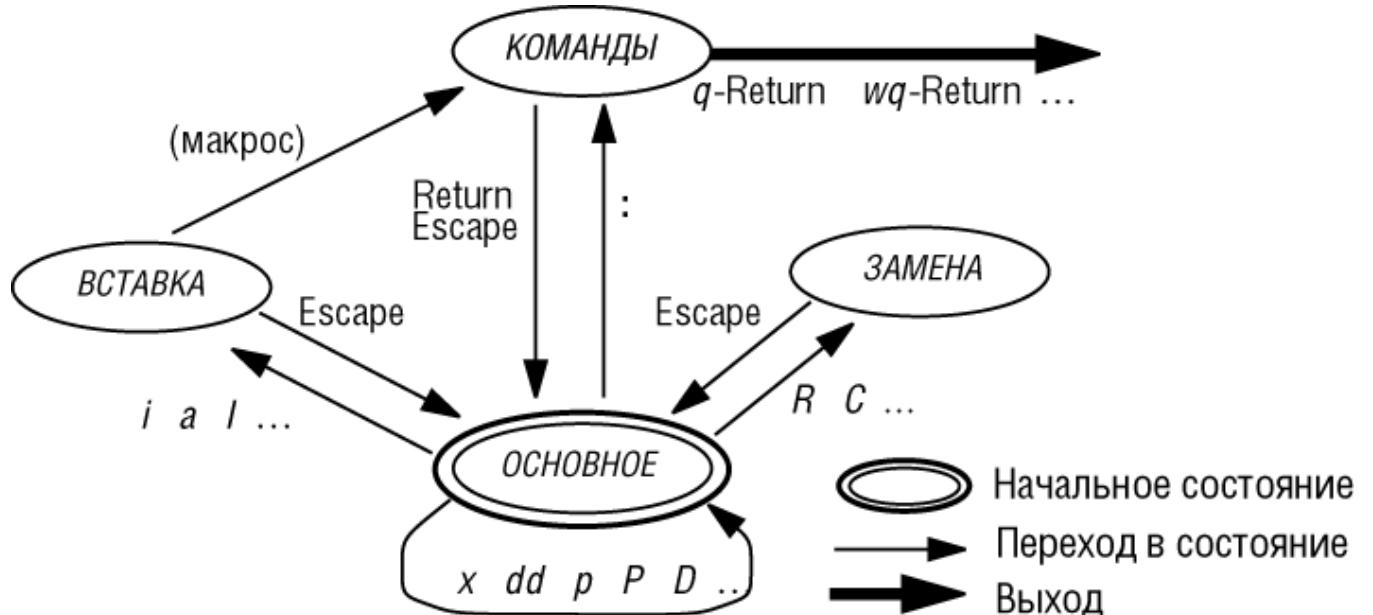


Рис. 14.6. Частичная диаграмма состояний для Vi

Литература по интерфейсам пользователей настроена критически к состояниям, потому что они могут вводить пользователей в заблуждение. В одной старой статье о пользовательском интерфейсе языка Smalltalk [Goldberg 1981] имеется фотография автора в футболке с надписью "Долой режимы!" ("Don't mode me in!"). Действительно, общий принцип разработки хорошего интерфейса пользователя состоит в том, чтобы обеспечить конечным пользователям на каждом этапе сессии возможность выполнять все имеющиеся в их распоряжении команды (вместо того, чтобы заставлять их изменять состояние для выполнения некоторых важных команд).

В соответствии с этим принципом хороший проект постарается минимизировать число состояний. Но этот принцип вовсе не означает, что всегда удастся обойтись одним состоянием. Такая крайняя интерпретация лозунга "долой режимы!" может, на самом деле, ухудшить качество интерфейса пользователя, так как чрезесчур большое количество одновременно доступных несвязанных между собой команд может запутать конечных пользователей. Более того, могут быть веские причины для ограничения числа доступных команд в некоторых ситуациях (например, когда приложению нужен неотложный ответ от своего конечного пользователя).

Как бы там ни было, состояния должны быть явно определены разработчиками и, как правило, должны быть также ясными для конечных пользователей. Это единственный способ, позволяющий разработчикам применять выбранную ими политику интерфейса - независимо от того, сильны ли их предубеждения против режимов или они относятся к ним более терпимо.

Поэтому наш конструктор приложений предоставит разработчикам в явном виде абстракцию STATE (**СОСТОЯНИЕ**); что касается других абстракций, то в нем будет каталог состояний, содержащий состояния, полезные для общего использования, и редактор состояний, позволяющий разработчикам определять новые состояния, часто получаемые с помощью модификации состояний, извлеченных из каталога.

Приложения

Последней из главных абстракций данных является понятие приложения.

Все предыдущие абстракции были внутренними средствами. Приложения - это то, что в действительности хотят построить разработчики. Примерами приложений являются системы обработки текстов, системы управления инвестициями, системы управления производством и др.

Для описания приложения необходимо задание множества состояний, переходов между состояниями, выделение одного состояния в качестве начального (с него начинаются все сессии). Мы уже видели, что состояние связывает некоторый отклик с каждой допустимой парой контекст-событие, включающей выполнение некоторой команды. Для полного построения приложения может также потребоваться включение в отклик указания на ту пару контекст-событие, приведшую к этому отклику, так что различные комбинации смогут инициировать переходы в разные состояния. Такую

информацию будем называть меткой перехода.

Имея состояния и метки переходов, можно построить диаграмму переходов, описывающую все приложение. На предыдущем рисунке показана часть такой диаграммы для Vi.

Контекст-Событие-Команда-Состояние: резюме

Определенные только что абстракции могут послужить основой для мощного конструктора интерактивных приложений, представляющего не только конструктора интерфейса, но средство, позволяющее разработчикам графически конструировать приложение; они будут использовать визуальные каталоги контекстов, событий и, что наиболее важно, команд, графически выбирать из них нужные элементы, будут устанавливать связи вида контекст-событие-команда с помощью простого механизма перетаскивания до тех пор, пока не получат полное приложение.

Так как простые приложения часто имеют лишь одно состояние, то наш конструктор приложений постарается сделать понятие состояния незаметным. Для более сложных приложений разработчики смогут использовать столько состояний, сколько потребуется, и последовательно получать новое состояние из уже имеющегося.

Математическая модель

Некоторые из неформально представленных в этой лекции понятий, в частности понятие состояния, имеют элегантное математическое описание, основанное на понятии **конечной функции** и математическом преобразовании, известном как **карринг (currying)**.

Поскольку эти результаты не используются в остальной части книги и представляют интерес в основном для читателей, которым нравится исследовать математические модели понятий, связанных с ПО, то соответствующие разделы вынесены на компакт-диск, сопровождающий эту книгу, в виде отдельной главы, названной "Математические основы"¹¹, взятой из [M 1995e].

Библиографические замечания

Идеи конструктора приложений, кратко описанного в этой лекции, взяты в значительной мере из конструктора приложений Build фирмы ISE, детально описанного в [M 1995e], где также приведено подробное описание лежащей в его основе математической модели.

¹¹) Эта дополнительная глава переведена на русский язык, но, следуя автору, помещена на CD. Здесь же отмечу, что под каррингом понимается понижение на единицу размерности функции. Поскольку чудес не бывает, то естественно на единицу возрастает размерность результата - если ранее он был числом, то после карринга он становится функцией, возвращающей число. Объектный подход весь основан на карринге. Вызов $x.f(a)$ объектом x метода $f(a)$ можно рассматривать как применение карринга к некоторой исходной функции от двух аргументов $F(x,a)$, так что $f(a) = \text{curry}(F(x,a))$. - Прим. ред.

Основы объектно-ориентированного проектирования

15. Лекция: ОО-программирование и язык Ada

Успехи методологии программирования 70-х годов привели к появлению нового поколения языков, сочетающих управляемые структуры Algol 60 и конструкции структурирования данных Algol W и Pascal с поддержкой скрытия информации. При различии свойств эти языки близки по сути, их принято называть инкапсулирующими языками. (Они также известны как "основанные на объекте" - терминология обсуждается в следующей лекции.) Хотя инкапсулирующих языков много, относительно широко используются всего несколько из них. Пять языков заслуживают особого внимания. Язык Modula-2, наследник языка Pascal, создан Никласом Виртом в Швейцарском Федеральном Институте Технологии - автором таких языков как Algol W, собственно Pascal и (позже) Oberon. Язык CLU был разработан в МИТ под руководством Барбары Лисков, хотя он и лишен наследования, но ближе в сих подошел к реализации ОО-концепций. Язык Mesa, разработанный в рамках проекта Xerox, особое внимание уделял описанию межмодульных отношений больших систем. Созданный Мэри Шоу, Вильямом Вулфом и Ральфом Лондоном из Carnegie-Mellon University язык Alphard включал механизм утверждений. В изучении ОО-подходов ограничимся языком Ada, поскольку он не только привлек самое большое внимание, но является самым законченным (и самым сложным), в оплещая основные черты других инкапсулирующих языков. Modula-2, например, не имеет таких возможностей, как универсальность или перегрузка.

Немного контекста

Создание языка Ada было реакцией на кризис середины 70-х годов, ощущимый для политики в области разработки ПО в Департаменте Обороны США (DoD). В отчете, предшествовавшем появлению языка Ada, отмечалось, что в военной отрасли в тот момент использовалось более 450 языков программирования, многие из которых технически устарели. Все это мешало управлению подрядными работами, обучению программистов, техническому прогрессу, разработке качественного ПО и контролю цен.

Помня об успехе языка COBOL, разработанного в 50-х годах по запросу DoD, был объявлен конкурс на разработку современного языка создания ПО. Одна из заявленных целей - возможность поддержки встроенных приложений в режиме реального времени. В результате были отобраны четыре, затем - два, и, наконец, в 1979 году, после действительно справедливого отбора, победителем оказался язык Green, созданный Жаном Ичбия (Jean D. Ichbiah) и его группой CII-Honeywell Bull. На основе опыта нескольких лет и первых промышленных реализаций язык был пересмотрен и в 1983 году был принят как стандарт ANSI.

Язык Ada (так был назван язык Green) начал новый этап в разработке языков. Никогда раньше язык не подвергался такому интенсивному испытанию перед выпуском. Никогда раньше создание языка не трактовалось как крупномасштабный инженерный проект. Лучшие эксперты многих стран в составе рабочих групп проводили недели, рассматривая предложения и делая - в те доинтернетовские дни - большое количество комментариев. Подобно языку Algol 60 в предыдущем поколении языков, Ada определил не только языковую перспективу, но и само понятие разработки языка.

Дальнейший пересмотр языка Ada привел к новой версии языка, официально называемой Ada 95, описываемой в конце данной лекции. В других частях курса название Ada без дальнейшего уточнения относится к версии Ada 83, широко используемой и сегодня.

Был ли язык Ada успешным? И да, и нет. Департамент Обороны получил то, что заказывал: благодаря строгому выполнению "поручения" язык Ada стал через несколько лет доминирующим техническим языком различных отраслей Американской военной промышленности и военных организаций некоторых других стран. Он используется в таких невоенных правительственные агентствах, как NASA и Европейское Космическое Агентство. Но, кроме некоторого проникновения в сферу обучения теории вычислительных систем - частично по инициативе Департамента Обороны, - этот язык имел лишь ограниченный успех в остальном мире ПО. Возможно, он бы распространился шире, если бы не конкуренция со стороны объектной технологии, внезапно появившейся на сцене, как раз тогда, когда язык Ada и промышленность созрели друг для друга.

По иронии судьбы разработчики языка Ada были хорошо знакомы с ОО-идеями. Хотя это не всем известно, Ичбия создал один из первых компиляторов для Simula 67 - первого ОО-языка. Позже, когда его спрашивали, почему он не представил ОО-проект Департаменту Обороны, он объяснял, что в контексте конкуренции такой проект посчитали бы настолько далеким от основного направления, что у него было бы шансов на победу. И он, без сомнения, прав. Действительно, до сих пор можно удивляться смелости проекта, принятого DoD. Было разумно ожидать, что процесс приведет к чему-то вроде усовершенствованной версии языка JOVIAL (языка военных приложений 60-х гг.). Но все четыре отобранных языка были основаны на языке Pascal, с его явным академическим привкусом. А Ada являлся воплощением новых смелых идей во многих областях, например, в обработке исключений, универсальности и параллелизме. Ирония состоит и в том, что язык Ada, направленный на поддержание соответствия проектов DoD прогрессу в разработках ПО, вытесняя **старые** подходы, в последующие годы невольно привел к задержке принятия **новой** (post-Ada) технологии в военном и космическом сообществе.

Уроки языка Ada остаются незаменимыми, и жаль, что многие ОО-языки 80-х и 90-х гг. не обращали большего внимания на акцент качества программной инженерии, характерный для языка Ada. Хотя в этой книге мы неоднократно будем противопоставлять решения, принятые в языке Ada, методам, принятым в объектной технологии, но эти замечания следует воспринимать не как укор, а как дань уважения к предшественнику, в

сравнении с которым должны оцениваться новые методы.

Пакеты

Любой инкапсулирующий язык предлагает модульную конструкцию для группирования логически связанных программных элементов. В языке Ada она называется пакетом, модулем - в Modula-2 и Mesa, кластером - в CLU.

Класс определяется и как структурный системный компонент - модуль, и как тип. Напротив, пакет - это только модуль. Ранее отмечалось, что пакеты являются чисто **синтаксическими** понятиями, а классы имеют и **семантическое** значение. Пакеты дают способ распределения элементов системы (переменных, подпрограмм ...) в согласованные подсистемы, но они нужны только для управляемости и удобочитаемости ПО. Декомпозиция системы на пакеты не затрагивает ее семантики: можно трансформировать многопакетную систему Ada в однопакетную систему, дающую те же самые результаты, посредством чисто синтаксической операции - сняв все границы пакетов, расширяя родовые порождения (это объясняется ниже) и разрешая конфликт имен посредством переименования. Классы являются семантической конструкцией, представляя одновременно единицу модульной декомпозиции, они описывают поведение объектов во время выполнения. Благодаря наследованию семантика обогащается полиморфизмом и динамическим связыванием.

Пакет языка Ada - это свободное соединение элементов программы. Он используется для различных целей. Разумное использование этого понятия включает создание пакета, содержащего:

- набор связанных констант (как в случае с наследованием возможностей);
- библиотеку подпрограмм, например, математическую библиотеку;
- набор переменных, констант и подпрограмм, описывающих реализацию одного абстрактного объекта, или фиксированное количество абстрактных объектов, доступных только через назначенные операции;
- реализацию абстрактного типа данных.

Последнее использование наиболее интересно для данного обсуждения. Оно будет изучаться на примере пакета, описывающего стеки, взятого из руководства по языку Ada.

Реализация стеков

Скрытие информации поддерживается в языке Ada двухъярусным объявлением пакетов. Каждый пакет состоит из двух частей, официально известных как "спецификация" и "тело". Первый термин - слишком сильный для конструкции, не поддерживающей формального описания семантики пакета (в форме утверждений или похожих механизмов), поэтому лучше использовать скромное слово "интерфейс".

Интерфейс перечисляет общедоступные свойства пакета: экспортированные переменные, константы, типы и подпрограммы. Для подпрограмм он дает только заголовки, перечисляя формальные аргументы и их типы, и тип результата для функции, например:

```
function item (s: STACK) return X;
```

Часть, содержащая тело пакета, обеспечивает реализацию подпрограмм и добавляет любые необходимые секретные элементы.

Простой интерфейс

Первую версию интерфейса пакета, задающего стек, можно выразить следующим образом. Заметим, что ключевое слово **package (пакет)** вводит интерфейс; тело, появляющееся позднее, вводится сочетанием **package body (тело пакета)**.

```
package REAL_STACKS is
    type STACK_CONTENTS is array (POSITIVE range <>) of FLOAT;
    type STACK (capacity: POSITIVE) is
        record
            implementation: STACK_CONTENTS (1..capacity);
            count: NATURAL := 0;
        end record;
    procedure put (x: in FLOAT; s: in out STACK);
    procedure remove (s: in out STACK);
    function item (s: STACK) return FLOAT;
    function empty (s: STACK) return BOOLEAN;
    Overflow, Underflow: EXCEPTION;
end REAL_STACKS;
```

Этот интерфейс перечисляет экспортированные элементы: тип STACK - для объявления стеков,

вспомогательный тип STACK_CONTENTS, используемый типом STACK, четыре открытые подпрограммы (процедуры и функции) и два исключения. Клиентские пакеты будут опираться только на интерфейс (предполагается, что создающие их программисты имеют представление о семантике, связанной с программами).

Этот пример наводит на несколько общих замечаний:

- Удивительно видеть все детали представления стека в объявлениях типов STACK и STACK_CONTENTS, появившихся в том, что должно быть чистым интерфейсом. Кратко рассмотрим причину этой проблемы и способ ее устранения.
- В отличие от класса, пакет не определяет тип. Тип STACK следует определить отдельно. Одним из следствий этого отделения для программиста, создающего пакет вокруг реализации абстрактного типа данных, является необходимость изобретения двух различных имен - одно для пакета, другое - для типа. Другое следствие состоит в том, что подпрограммы имеют еще один аргумент по сравнению со своими ОО-аналогами: здесь все они имеют первым аргументом стек s, в то время как для класса он задается неявно (см. предыдущие лекции).
- Объявление может определять не только тип сущности, но и ее исходное значение. Здесь объявление count в типе STACK предписывает исходное значение 0. Оно устраниет необходимость явной операции инициализации, задаваемой процедурой создания (конструктором) класса. Однако этот способ не работает, если требуется более сложная инициализация.
- Для понимания объявления типа следует привести некоторые детали языка Ada: POSITIVE и NATURAL обозначают подтипы INTEGER, включающие, соответственно, положительные и неотрицательные целые, спецификация типа вида array (TYPE range <>), где <> известно как Box-символ, описывает шаблон для типов массивов. Для получения действительного типа из такого шаблона нужно выбрать конечный отрезок TYPE. Здесь это делается при определении типа STACK, использующем интервал [1..capacity] типа POSITIVE. STACK является примером параметризованного типа. Любое объявление сущности типа STACK должно задавать фактическое значение емкости стека capacity, как в:

```
s: STACK (1000)
```

- В языке Ada каждый аргумент подпрограммы характеризуется статусом in, out или in out, определяющим права подпрограммы на использование фактических аргументов (только для чтения, только для записи, для обновления). В отсутствии явного ключевого слова состояние по умолчанию - in.
- Наконец, интерфейс определяет два имени исключений Overflow и Underflow. Исключение - это ситуация, когда из-за ошибок прерывается нормальный порядок вычислений. Интерфейс пакета должен перечислить любые исключения, которые могут возбуждаться в процессе работы подпрограмм пакета и передаваться для обработки клиентам. Подробно механизм исключений языка Ada описывается ниже.

Использование пакета

Приведем пример из клиентского пакета, использующего стек вещественных чисел:

```
s: REAL_STACKS.STACK (1000);
REAL_STACKS.put (3.5, s); ...;
if REAL_STACKS.empty (s) then ...;
```

Среда языка Ada должна иметь возможность компилировать такой клиентский код, располагая только интерфейсом REAL_STACKS, не имея доступа к его телу.

Синтаксически каждое использование сущности (здесь "сущности" включают имена программ и типов) повторяет имя пакета REAL_STACKS. Это утомительно - необходима неявная форма квалификации. Если включена директива:

```
use REAL_STACKS;
```

в начале клиентского пакета, то выражения записываются проще:

```
s: STACK (1000);
put (3.5, s); ...;
if empty (s) then ...;
```

Конечно, используется и полная форма для сущности, чье имя вступает в конфликт с именем, указанным в другом доступном пакете (скажем, объявленное в самом пакете или в пакете из списка в директиве use).

В литературе по языку Ada иногда встречается совет программистам вообще не использовать директиву use, поскольку она мешает ясности: неквалифицированная ссылка, например вызов empty (s), сразу не говорит

о поставщике empty (в нашем примере REAL_STACKS). Его аналог в ОО-подходе, s.empty, однозначно определяет поставщика через цель s.

В ОО-мире подобная проблема возникает из-за наследования: имя в классе может ссылаться на компонент, объявленный любым из предков. Техника, частично решающая проблему, - это плоская форма класса.

Реализация

Тело пакета REAL_STACKS может объявляться следующим образом. Полностью показана только одна подпрограмма.

```
package body REAL_STACKS is
    procedure put (x: in FLOAT; s: in out REAL_STACK) is
        begin
            if s.count = s.capacity then
                raise Overflow;
            end if;
            s.count := s.count + 1;
            s.implementation (count) := x;
        end put;
    procedure remove (s: in out STACK) is
        ... Реализация remove ...
    end remove;
    function item (s: STACK) return X is
        ... Реализация item ...
    end item;
    function empty (s: STACK) return BOOLEAN is
        ... Реализация empty ...
    end empty;
end REAL_STACKS;
```

Два свойства, показанные в этом примере, будут подробно обсуждаться ниже: использование исключений и необходимость повторения в теле большей части информации интерфейса (заголовков подпрограммы).

Универсальность

Пакет, в том виде как он появился, слишком специфичен. Он приложим к типу FLOAT, а хотелось бы задания произвольного типа. Чтобы сделать его универсальным, в языке Ada используется следующий синтаксис:

```
generic
    type G is private;
package STACKS is
    ... Все, как и ранее, заменяя все вхождения FLOAT на G ...
end STACKS;
```

Предложение generic синтаксически более тяжелое, чем наша ОО-нотация для универсальных классов (class C [G] ...), но зато в нем больше возможностей. В частности, параметры, объявляемые в generic, могут представлять не только типы, но и подпрограммы. В приложении В эти возможности обсуждаются при сравнении универсальности и наследования.

В теле пакета generic не повторяется, там достаточно конкретный тип FLOAT заменить родовым G.

Спецификация is private заставляет остальную часть пакета рассматривать G как закрытый тип. Это означает, что сущности этого типа могут использоваться только в операциях, применимых ко всем типам языка Ada: в качестве исходного или целевого объекта при присваивании, как операнд в проверке равенства, как фактический аргумент в подпрограмме, и в некоторых других специальных операциях. Это близко к соглашению для неограниченных формальных параметров универсальных классов нашей нотации. В языке Ada доступны и другие возможности. Можно ограничить операции, объявляя параметр как limited private, что запрещает все использования кроме фактических аргументов подпрограмм.

Называясь пакетом, универсально параметризованный модуль, такой как STACKS, в действительности является шаблоном пакета, поскольку клиенты не могут использовать его непосредственно; они должны получить из него действительный пакет, используя фактические родовые параметры. Новую версию нашего пакета стеков действительных величин можно определить через следующее родовое порождение:

```
package REAL_STACKS_1 is new STACKS (FLOAT);
```

Родовое порождение - главный механизм языка Ada адаптации модулей. Из-за отсутствия наследования он

менее гибок, поскольку можно выбирать только между универсальными модулями (параметризованными, но не используемыми непосредственно) или используемыми модулями (более не расширяемыми). Напротив, наследование допускает произвольные расширения существующих модулей, в соответствии с принципом Открыт-Закрыт. В приложении даются подробности сравнения.

Скрытие представления: частная история

Пакет STACKS в том виде, как он задан, не реализует принцип скрытия информации. Объявления типов STACK и STACK_CONTENTS, находясь в интерфейсе, позволяют клиентам непосредственный доступ к представлению стеков. Например, клиент может включить код вида:

```
[1]
use REAL_STACKS_1;...
s: STACK; ...
s.implementation (3) := 7.0; s.last := 51;
```

грубо нарушая основную спецификацию абстрактных типов данных.

Концептуально объявления типа должны находиться в теле. Почему их туда не помещают с самого начала? Объяснение находится вне языка и требует рассмотрения проблем программного окружения.

Одно из уже упомянутых требований к языку Ada состояло в возможности независимой компиляции пакета при наличии доступа к его интерфейсу, но не обязательно к его телу. Принятая технология предполагала построение сверху вниз: для продолжения работы над модулем достаточно знать спецификацию необходимых ему средств. Действительная реализация могла появиться значительно позже.

Если есть доступ к интерфейсу REAL_STACKS_1 (то есть к интерфейсу STACKS, REAL_STACKS_1 является просто его родовым порождением), можно компилировать любого из его клиентов. Такой клиент будет содержать объявления вида:

```
use REAL_STACKS_1;...
s1, s2: STACK; ...
s2 := s1;
```

Компилятор не сможет их хорошо обрабатывать, не зная размера объекта типа STACK. Но это может определяться только из объявлений типа для STACK и вспомогательного типа STACK_CONTENTS.

Отсюда концептуальная дилемма, стоявшая перед проектировщиками языка Ada: вопросы реализации требуют помещения объявлений типа в рай - интерфейс, в то время как им место в аду - теле пакета.

Пришлось создать чистилище: специальный раздел пакета, физически видимый в интерфейсе и компилируемый с ним, но такой, что клиенты не могут обращаться к его элементам. Чистилище - это закрытая часть интерфейса, она вводится ключевым словом `private`. Любое объявление, появляющееся здесь, недоступно клиентам. Эта схема иллюстрируется нашей последней версией интерфейса пакета, задающего стек:

```
generic
  type G is private;
package STACKS is
  type STACK (capacity: POSITIVE) is private;
  procedure put (x: in G; s: in out STACK);
  procedure remove (s: in out STACK);
  function item (s: STACK) return G;
  function empty (s: STACK) return BOOLEAN;
  Overflow, Underflow: EXCEPTION;
private
  type STACK_VALUES is array (POSITIVE range <>) of G;
  type STACK (capacity: POSITIVE) is
    record
      implementation: STACK_VALUES (1..capacity);
      count: NATURAL := 0;
    end record;
end STACKS;
```

Отметим, тип STACK теперь должен объявляться дважды: сначала в открытой части интерфейса, где он специфицируется как `private`, затем еще раз в закрытой части, где дается полное описание. Без первого объявления строка вида `s: REAL_STACK` не будет разрешенной в клиенте, поскольку доступ есть только к сущностям, объявляемым в открытой части. Первое объявление, специфицируя тип как `private`, запрещает

клиентам доступ к любым свойствам помимо универсальных операций: присваивания, проверки на равенство и использование в качестве фактических аргументов.

Заметьте, тип STACK_VALUES чисто внутренний и не нужен клиентам. Поэтому он не объявляется в открытой части интерфейса пакета.

Важно понять, что информация, помещаемая в закрытую часть интерфейса, должна быть в теле пакета и появляется в спецификации пакета только по причинам реализации языка. С новой формой STACKS клиентский код, выше помеченный как [1], имевший прямой доступ к представлению в клиенте, становится неправильным.

Авторы клиентских модулей могут **видеть** внутреннюю структуру экземпляров STACK, но они не могут **воспользоваться** ею в своих модулях. Это могло бы приводить разработчиков к танталовым мукам. (Хорошая среда языка Ada могла бы скрывать эту часть от клиента, также как это делает инструмент short, описанный в предыдущих лекциях.) Удивительная для новичков, эта политика не противоречит правилу скрытия информации. Как отмечалось ранее, цель скрытия не в том, чтобы не дать авторам клиента возможности прочитать скрытые подробности, а чтобы не дать им использовать эти подробности.

Те, кому хотелось бы все усложнить, могли бы подвести итог в двух предложениях (произнесенных очень быстро, чтобы произвести впечатление и на друзей, и на врагов): Закрытый раздел интерфейса пакета перечисляет реализацию тех концептуально закрытых типов, которые должны быть объявлены в интерфейсе, хотя их реализация недоступна для использования. В открытой части интерфейса эти типы объявлены закрытыми.

Исключения

Пакет STACKS определяет два исключения в своем интерфейсе: Overflow и Underflow. Язык Ada допускает как собственные исключения с произвольными именами, так и предопределенные исключения, запускаемые оборудованием или операционной системой.

Некоторые элементы механизма обработки исключений языка Ada были введены в [лекции 12](#) курса "Основы объектно-ориентированного программирования", так что здесь можно ограничиться коротким изучением исключений в подходе Ada к построению ПО.

Упрощение управляемой структуры

Исключения в языке Ada являются техникой исправления ошибок, не затрагивающей управляемой структуры процесса вычислений. Если рассматривать программу как выполнение ряда действий, каждое из которых может прерваться из-за сбоев, то ее структура могла бы выглядеть так:

```
action1;
if error1 then
    error_handling1;
else
    action2;
    if error2 then
        error_handling2;
    else
        action3;
        if error3 then
            error_handling3;
        else
            ...
        ...
    ...
else
    ...
end;
```

Механизм исключений в Ada предназначен для борьбы со сложностью подобной схемы - где элементы, выполняющие "полезные" задачи, выглядят как острова в океане кода, обрабатывающего ошибки программы. Ada отделяет обработку ошибок от их обнаружения. Конечно же, обработка ошибок должна включать тесты, определяющие тип возникшей ситуации. Единственным решением является в момент возникновения ошибки возбуждение определенного сигнала - исключения, обрабатывающегося далее где-то в другом месте.

Возбуждение и обработка исключений

Чтобы возбудить исключительную ситуацию, а не обрабатывать ошибки на месте, можно переписать текст следующим образом:

```
action1;
if error1 then raise exc1; end;
action2;
```

```

if error2 then raise exc2; end;
action3;
if error3 then raise exc3; end;
...

```

При выполнении команды `raise` нормальный порядок вычислений прерывается, и управление передается **обработчику исключений (exception handler)**, представленному специальным блоком подпрограммы и имеющему вид:

```

exception
when exc1, ...=> treatment1;
when exc2 ...=> treatment2;
...

```

При возбуждении исключения `exc` первым его обрабатывает захвативший его обработчик из динамической цепи вызовов - списка элементов, начинающегося подпрограммой, содержащей вызвавшее исключение предложение `raise`, и всеми вызывающими подпрограммами, как показано на [рис. 15.1](#):

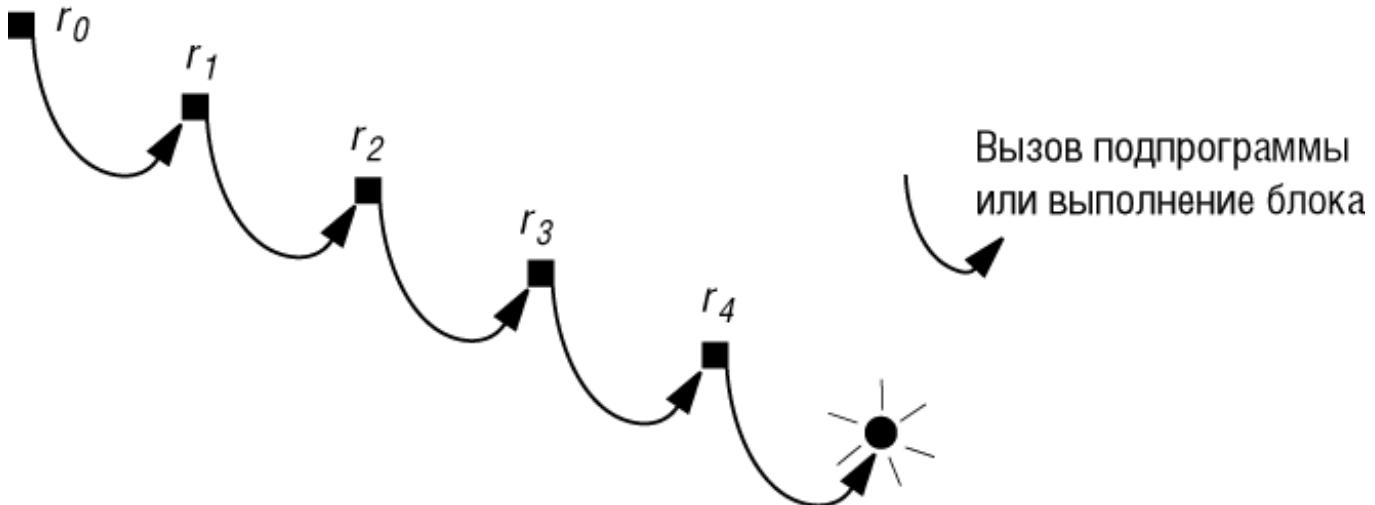


Рис. 15.1. Цепь вызовов (этот рисунок впервые появился в лекции 12 курса "Основы объектно-ориентированного программирования")

Говорят, что обработчик захватывает `exc`, если `exc` появляется в одном из его предложений `when` (или он содержит предложение вида `when others`). Такой обработчик выполняет соответствующие команды (после символа `=>`), после чего управление передается вызывающей программе или заканчивается в случае главной программы. (Ada имеет понятие главной программы.) Если никакой обработчик в динамической цепи не обрабатывает `exc`, выполнение приложения заканчивается, и управление возвращается к операционной системе, а она, вероятно, выведет системное сообщение об ошибке.

Обсуждение

Интересно сравнить механизм обработки исключений языка Ada с механизмом, разработанным выше в этом курсе в лекции, посвященной исключениям. Между ними есть технические различия и различия в методологии.

К техническим различиям можно отнести способы задания исключений. В одном случае используются множественные предложения `when`, в другом - наследование от класса `EXCEPTIONS`. Более важно включение в объектную нотацию возможности повторной попытки, что потребовало введения специального ключевого слова `retry`. Язык Ada не имеет подобной поддержки и требует для ее реализации использования `goto` или подобных управляющих структур.

Методологическое различие вытекает из принятой строгой политики, ведущей к принципу Дисциплинированной Обработки Исключений. Каждый обработчик исключений должен заканчиваться, кроме редких случаев "ложной тревоги", либо попыткой повторения, либо официальным отказом ("организованной паникой"). Язык Ada менее строг в этом отношении, что может приводить к некорректному использованию исключения, при котором вызывающая программа получит управление без устранения возникающих проблем.

Стоит повторить основное правило:

Правило исключений языка Ada

Выполнение любого обработчика исключений языка Ada должно заканчиваться либо выполнением команды `raise`, либо повтором охватывающей подпрограммы.

Исключения в Ada - это управляющие структуры, предназначенные для отделения обнаружения аварийных ситуаций от их обработки и сохранения простоты структуры ПО. Однако на практике этого часто не происходит.

Запись `raise some_exception` дает впечатление освобождения от запутанной и скучной задачи слежения за необычными ситуациями, позволяя сосредоточиться на самом алгоритме, имеющем дело с нормальной ситуацией. Но вызов исключения еще не решает задачи. Исключения в пакете `STACKS` типичны. Попытка поместить элемент в полный стек вызывает ошибку `Overflow`, а попытка доступа к пустому стеку вызывает `Underflow`. Как обрабатывать `Underflow`, ошибку, возникающую при вызове `remove` или `item` на пустом стеке? Обсуждение Проектирования по Контракту показало, что эти подпрограммы не могут знать, что следует делать в такой ситуации. Вся ответственность лежит на клиенте, вызвавшем эти подпрограммы, только он может решить, что следует делать. У него и должен содержаться код вида:

```
[2]
use REAL_STACKS;
procedure proc (...) is
    s: STACK; ...
begin
    ... remove (s); ...
exception
    when Underflow => action1;
    ...
end proc;
```

Клиент должен точно определить, что происходит в случае ошибки. Опустить оператор `when Underflow` было бы ошибкой проекта. Сравните это с обычной, не основанной на исключении, формой вызова:

```
[3]
if not s.empty then s.remove else action1 end
```

(или вариантом, определяющим ошибку апостериори). Форма [2], использующая исключения, отличается от формы [3] только двумя аспектами:

- код для обработки ошибки `action1` текстуально отделен от вызова, приведшего к ошибке;
- обработка ошибки одинакова для всех подобных вызовов.

Хотя и желательно избегать глубоко вложенных структур обработки ошибок `if... then... else...`, приведенных в начале лекции, то место в алгоритме, где обнаруживается ошибка, часто предоставляет наилучшую информацию для ее обработки. Если разделить обнаружение и обработку, то могут потребоваться сложные управляющие структуры для случаев, требующих повторного запуска или продолжения обработки.

Кроме того, для подпрограммы, содержащей несколько вызовов `remove`, способ работы с пустыми стеками вряд ли будет одним и тем же в каждом случае.

Существуют два общих стиля использования исключений. **Стиль управляющей структуры** рассматривает исключения как нормальный механизм для обработки всех случаев, отличающихся от обычных. **Стиль аварийных случаев** рассматривает их как непредсказуемые ситуации, когда все другие механизмы не работают. Объектный подход `rescue/retry`, описанный ранее, тяготеет к стилю аварийных случаев, хотя может использоваться и для первого стиля. Обработка исключений в языке Ada больше ориентирована на стиль управляющей структуры.

Каждый должен решить, какой стиль ему больше нравится. Но в любом случае, следует помнить, что не нужно наивно возлагать надежды на использование исключений. Есть механизм исключений или его нет, ошибки при выполнении программы - это факт жизни системы, и она должна их явно обрабатывать. Хороший методологический подход, поддерживаемый эффективным механизмом исключений, может быть полезным. Но проблеме обработки ошибок присуща природная сложность, и никакая волшебная палочка от нее не избавит.

Задачи

Кроме пакетов, Ada предлагает еще одну интересную модульную конструкцию - задачу. Задачи - это основной механизм Ada для обработки параллелизма. Лежащая в основе модель параллелизма близка к подходу ВПП, описанному в лекции о параллелизме. Задачи заслуживают упоминания за свои концепции модульности, которые ближе чем пакеты совпадают с объектными понятиями.

Синтаксически, задачи имеют много общего с пакетами. Главное различие в том, что задача - это не просто единица модульности, но и представление процесса, выполняемого параллельно с другими процессами. Поэтому подобно классу (и в отличие от пакета) задача является как синтаксической так и семантической конструкцией.

Как и пакет, задача имеет интерфейс и тело. Вместо подпрограмм, спецификация задачи вводит несколько входов (*entry*). Для клиента входы выглядят как процедуры, например, интерфейс задачи, управляющей буфером, может выглядеть так:

```
task BUFFER_MANAGER is
    entry read (x: out G);
    entry write (x: in G);
end BUFFER_MANAGER;
```

(Задачи не могут быть универсальными, так что тип G должен быть глобально доступным, или родовым параметром охватывающего пакета.) Только реализация входов отличает их от процедур: команды *accept*, появляющиеся в теле, будут специфицировать синхронизацию и другие ограничения на выполнение записей. Здесь, например, можно предписать, чтобы только одно *read* или *write* действовало в любой момент времени, чтобы *read* ожидало, пока буфер не станет непустым, а *write* - пока он не будет неполным.

Кроме отдельных задач, можно специфицировать **тип задачи** и использовать его для создания стольких задач - экземпляров типа задачи - сколько требуется во время выполнения. Это делает задачи похожими на классы без наследования. Действительно, можно представить реализацию в Ada ОО-концепций, представляющих классы типами задач, а объекты - их экземплярами (возможно, даже используя команды *accept* с различными условиями для эмуляции динамического связывания.) Поскольку в последовательном ОО-вычислении можно ожидать, что классы будут иметь много экземпляров, это упражнение представляет в основном академический интерес, учитывая издержки создания нового процесса в текущих операционных системах. Возможно, когда-нибудь наступит день, и широкомасштабные параллельные среды оборудования станут явью...

От Ada 83 к Ada 95

Версия языка Ada 95 предусматривает добавление ОО-концепций. В ней нет понятия класса в нашем смысле слова (модуль плюс тип), но есть поддержка наследования и динамического связывания для типов записей.

ОО-механизмы языка Ada 95: пример

Текст ниже приведенного пакета иллюстрирует некоторые технические приемы Ada 95. Его смысл должен быть достаточно ясен для читателя. Для получения нового типа с дополнительными полями (форма наследования Ada 95), нужно объявить уже существующий тип, такой как ACCOUNT, как **дескрипторный (tagged)**. Это, конечно, противоречит принципу Открыт-Закрыт, поскольку необходимо знать заранее, какие типы могут иметь потомков, а какие - нет. Множественное наследование отсутствует, так что тип new можно получить только из одного типа. Обратите внимание на синтаксис получения нового типа без добавления атрибутов (null record, к удивлению, без end).

```
package Accounts is
    type MONEY is digits 12 delta 0.01;
    type ACCOUNT is tagged private;
        procedure deposit (a: in out ACCOUNT; amount: in MONEY);
        procedure withdraw (a: in out ACCOUNT; amount: in MONEY);
        function balance (a: in ACCOUNT) return MONEY;
    type CHECKING_ACCOUNT is new ACCOUNT with private;
        function balance (a: in CHECKING_ACCOUNT) return MONEY;
    type SAVINGS_ACCOUNT is new ACCOUNT with private;
        procedure compound (a: in out SAVINGS_ACCOUNT; period: in Positive);
private
    type ACCOUNT is tagged
        record
            initial_balance: MONEY := 0.0;
            owner: String (1..30);
        end record;
    type CHECKING_ACCOUNT is new ACCOUNT with null record;
    type SAVINGS_ACCOUNT is new ACCOUNT with
        record
            rate: Float;
        end record;
end Accounts;
```

Дескрипторные типы по-прежнему объявляются как записи. Основное свойство большинства ОО-языков - операции над типом являются частью типа и фактически **определяют** тип - здесь не работает. Подпрограммы задаются вне объявления типа и принимают в качестве аргумента значение типа. (В ОО-языках, `deposit` и т. д. будут частью объявления `ACCOUNT`, а `compound` - частью `SAVINGS_ACCOUNT`, им не требуются их первые аргументы.) Здесь же все, что требуется, - так это объявление подпрограмм и типа как части одного и того же пакета; им даже не нужно находиться рядом друг с другом. В приведенном примере, только расположение показывает читателю, что определенные программы концептуально связаны с определенными дескрипторными типами записей.

Это отличается от обычного взгляда на создание ОО ПО. Хотя дескрипторный тип и связанные с ним подпрограммы с теоретической точки зрения являются частью одного абстрактного типа данных, они не образуют синтаксической единицы. Это противоречит принципу Лингвистических Модульных Единиц, предполагающему тесную связь между модульной концепцией и синтаксической структурой.

Появление нового объявления для `balance` в `SAVINGS_ACCOUNT` сигнализирует о переопределении. Процедуры `withdraw` и `deposit` не переопределяются. Как будет понятно, это означает, что Ada 95 использует механизм **перегрузки** для получения ОО-эффекта от **переопределения подпрограмм**. Не существует синтаксической метки (как `redefine`), сигнализирующей о переопределении. Чтобы увидеть, что функция `balance` в `SAVINGS_ACCOUNT` отличается от базовой версии в `ACCOUNT`, следует просмотреть весь текст пакета. В данном случае каждая версия подпрограммы находится рядом с соответствующим типом, с отступами для выделения этой связи, но это условность стиля, а не правило языка.

Дескрипторный тип может объявляться как `abstract`, соответствуя понятию отложенного класса. Подпрограмму также можно сделать `abstract`, не создавая для нее тела.

Функция, возвращающая результат абстрактного типа, должна сама быть абстрактной. Это правило сначала может показаться странным и помехой для написания эффективной функции, возвращающей, скажем, вершину стека фигур в предположении, что тип `FIGURE` абстрактный. В языке Ada, однако, результат такой функции обычно будет принадлежать не типу `FIGURE`, а "типу доступа", описывающему ссылки на экземпляры `FIGURE`. Так что можно будет написать эффективную функцию.

К сущностям дескрипторного типа можно применить динамическое связывание, как в следующем примере:

```
procedure print_balance (a: in ACCOUNT'Class) is
    -- Печать текущего баланса.
begin
    Put (balance (a));
    New_Line;
end print_balance;
```

Динамическое связывание следует задать явным образом. Подпрограмма объявляется как "выходящая за рамки класса" (`classwide operation`) заданием классификатора `'Class` для типа аргумента. Это напоминает объявление в C++ любой динамически связываемой функции как "виртуальной". Только здесь клиент выбирает статическое или динамическое связывание.

Ada 95 позволяет определить "дочерний пакет" A1. В существующего пакета A. Это дает новому пакету возможность получить свойства из A и добавить свои собственные расширения и модификации. (Это понятие, конечно, близко к наследованию, но отличается от него.) Вместо объявления трех типов счетов в одном пакете, возможно, лучше было бы разделить пакет на три, где `Accounts.Checking` представляет `CHECKING_ACCOUNT` и его подпрограммы, а `Accounts.Saving` делает то же для `SAVINGS_ACCOUNT`.

Ada 95 и объектная технология: оценка

Если рассматривать язык Ada 95 с позиций объектной технологии, то сначала он может привести в замешательство. Со временем, освоив различные языковые механизмы, можно добиться эффекта единичного наследования, полиморфизма и динамического связывания.

Однако цена этого - сложность. К сложному языку Ada 83 добавился новый набор понятий со многими внутренними связями и связями со старыми конструкциями. При сравнении с ОО-методом, где введено достаточно простое понятие класса, обнаружится, что в Ada 95 нужно изучить, по крайней мере, пять сложных понятий:

- пакеты, являющиеся модулями, но не типами, могут быть родовыми, предлагая нечто похожее на наследование: дочерние пакеты (как и ряд других возможностей, не описанных подробно выше, таких как возможность объявления дочернего пакета как `private`);
- дескрипторные типы записей, являющиеся типами, но не модулями и имеющие некоторую форму наследования, хотя в отличие от классов они не позволяют синтаксического включения подпрограмм в объявление типа;

- задачи, являющиеся модулями, но не типами и не имеющие наследования;
- типы задач, являющиеся модулями и типами, но без возможности быть родовыми (хотя они могут включаться в родовые пакеты) и не имеющие наследования;
- "защищенные типы" (понятие, до сих пор не встречавшееся), являющиеся типами и включающие подпрограммы, что делает их похожими на классы, но без наследования:

```
protected type ANOTHER_ACCOUNT_TYPE is
    procedure deposit (amount: in MONEY);
    function balance return MONEY;
private
    deposit_list: ...; ...
end ANOTHER_ACCOUNT_TYPE;
```

Комбинация возможностей взаимодействия поразительна. Например, пакеты имеют, в добавление к понятию дочернего пакета, механизмы Ada use и with. В одном из руководств дается следующее объяснение:

Закрытые потомки предназначены для "внутренних" пакетов, которые должны применять механизм with только к ограниченному числу пакетов. Закрытый потомок может применить механизм with только к телу своего родителя или к его потомкам. В обмен на такое ограничение потомок получает новые полномочия: его спецификация автоматически видима в открытых и закрытых частях спецификаций всех его предков.

Без сомнения, можно уловить смысл подобных объяснений. Но стоит ли результат усилий?

Интересно отметить, что Жан Ичбиа, создатель языка Ada, публично покинул аналитическую группу Ada 95 после тщетных попыток сохранить расширения простыми. В его пространном заявлении об уходе дается следующий комментарий: **дополнительные возможности приведут в результате к огромному увеличению сложности в 9X [позже Ada 95]... В 9X количество рассматриваемых взаимодействий приближается к 60000.**

Базовые понятия объектной технологии, при всей их силе, удивительно просты. В языке Ada 95 предпринята, возможно, самая амбициозная попытка сделать их сложными.

Обсуждение: наследование модулей и типов

При изучении языка Ada 95 попутно интересно отметить, что разработчики Ada 95 считали необходимым помимо механизма наследования для дескрипторных типов ввести понятие пакета потомка. Язык Ada, конечно, всегда разделял понятия модуля и типа, в то время как классы объединяют эти два понятия. Но методологии языка Ada 95 предлагают при введении типа наследника, такого как SAVINGS_ACCOUNT, объявлять его в целях ясности и модульности не в первоначальном пакете (Accounts), а в пакете потомка. Если обобщить этот совет, то дойдет до создания, наряду с иерархией типов, иерархии модулей, строго ему следующей.

У классов в объектной технологии такие вопросы не возникают. Классы являются модулями, и существует только одна иерархия.

Выбор, сделанный в Ada 95, является еще одним примером популярного взгляда, что "**следует отделять наследование типа от повторного использования кода**". Понимание же объектной технологии, начиная с языка Simula, заключается в **соединении** понятий - модуля и типа, подтипов и модульного расширения. Как и любое другое смелое соединение понятий, считавшихся ранее совершенно различными, эта идея могла временами пугать, но без нее мы бы лишились замечательного упрощения архитектуры ПО.

Вперед к ОО-языку Ada

Язык Ada 95 кажется сложным. Но это не значит, что сама идея создания ОО-языка Ada обречена. Просто следует ставить реальные цели и постоянно заботиться о простоте и состоятельности. Сообщество Ada может снова попытаться разработать ОО-расширение, сопровождающееся удалением некоторых возможностей. Возможны два общие направления:

- Первая идея, близкая по духу к замыслу Ada 95, состоит в сохранении пакетной структуры и введении понятия класса, обобщающего типы записей Ada, с поддержкой наследования и динамического связывания. Но это должны быть действительные классы, включающие применимые подпрограммы. Такое расширение, в принципе, подобно расширению, ведущему от C к C++. Оно должно стремиться к минимализму, пытаясь применять как можно шире уже существующие механизмы (такие как with и use для пакетов), не вводя новых возможностей, приводящих потом к проблемам взаимодействия, упоминаемых Ичбиа.
- Другой подход может строиться на замечании, сделанном при представлении задач в данной лекции. Отмечалось, что типы задач близки по духу к классам, поскольку они могут иметь экземпляры,

созданные во время выполнения. Структурно они обладают многими свойствами пакетов. Можно было бы ввести модуль, имеющий, грубо говоря, синтаксис пакетов и семантику классов. Можно думать о нем как о пакет-классе, или о типе задач, необязательно являющихся параллельными. Понятие "защищенного типа" может стать отправной точкой, будучи интегрировано в существующий механизм.

Упражнения в конце данной лекции предлагают исследовать эти возможности.

Ключевые концепции

- Язык Ada, изучаемый как представитель класса "инкапсулирующих языков", включающего также Modula-2, предлагает конструкции модульной декомпозиции - пакеты (и задачи).
- Внимание уделяется скрытию информации: интерфейс и реализация объявляются отдельно.
- Универсальность увеличивает гибкость пакетов.
- Конфликты между методологическими и реализацийными требованиями порождают "закрытый" раздел - концептуально секретный, но синтаксически включаемый в интерфейс.
- Пакет - это чисто синтаксический механизм. Модули отделены от типов. Невозможен никакой механизм наследования.
- Исключения отделяют обнаружение ошибок от их обработки, но не дают чудесного решения проблемы ошибок времени выполнения.
- Механизм исключений Ada должен использоваться только дисциплинированным путем; любое выполнение обработчика исключений должно приводить либо к повтору операции, либо к появлению исключения в вызывающей программе.
- Типы задач могут, в принципе, использоваться для реализации классов без наследования, но это решение непрактично в современном окружении.
- Ada 95 делает возможным определение нового типа, порожденного существующим типом с поддержкой переопределения подпрограмм, полиморфизма и динамического связывания.

Библиографические замечания

[Booch 1986a] обсуждает (под маркой "ОО-проектирование", но не используя классы, полиморфизм и т. д.) как достичь некоторых преимуществ объектной ориентации в Ada.

Официальное описание языка Ada [ANSI 1983] не рекомендуется ни для чтения в постели ни как начальный курс. Для этих целей доступна многочисленная литература.

Ссылки на другие модульные языки, упомянутые в начале этой лекции: [Mitcell 1979] - Mesa, [Wirth 1982] - Modula-2, [Liskov 1981, Liskov 1986] - CLU, [Shaw 1981] - Alphard.

Комментированный список учебников по Ada 95 приведен в [Feldman-Web]. Я благодарен Ричарду Рихли и Магнусу Кемке за прояснение ряда вопросов относительно Ada 95, конечно, высказанные здесь замечания принадлежат мне. Магнус Кемке указал мне на диссертацию Матса Вебера.

Упражнения

У15.1 Как выиграть, не используя скрытия

Проблема компиляции пакетов Ada, приведшая к появлению закрытого раздела в интерфейсе, в равной степени затрагивает и ОО-языки, если среда программирования поддерживает независимую компиляцию классов. В действительности, проблема кажется более серьезной из-за наследования: объявленная переменная типа C, может во время выполнения ссылаться на экземпляры не только типа C, но и любого класса-наследника. Поскольку любой наследник может добавить свои атрибуты, размер этих экземпляров различен. Если C - отложенный класс, невозможно даже присвоить его экземплярам размер по умолчанию. Объясните, почему, несмотря на эти замечания, ОО-нотация этой книги не нуждается в языковой конструкции, подобной механизму **private** языка Ada. (Подсказка: Ваши рассуждения должны рассматривать, в частности, следующие понятия: расширенные типы в сравнении со ссылочными типами, отложенные классы и технические приемы, используемые в нашем ОО-каркасе для создания спецификации абстрактных классов, не требующие от автора классов ш_написания двух отдельных частей модуля.) Обсудите компромиссы того и другого решения. Можете ли Вы предложить другие подходы к решению проблемы каркаса языка Ada?

У15.2 Родовые параметры подпрограммы

Родовые параметры пакетов Ada могут быть не только типами, но и подпрограммами. Объясните релевантность этой возможности для реализации ОО-понятий и ее ограничения. (См. также приложение B.)

У15.3 Классы как задачи (для программистов Ada)

Перепишите класс COMPLEX как тип задачи Ada. Приведите примеры, использующие результирующий тип.

У15.4 Добавление классов к Ada

(Это упражнение предполагает хорошее знание языка Ada.) Придумайте адаптацию Ada 83, сохраняющую понятие пакета, но расширяющую записи до классов с полиморфизмом, динамическим связыванием и наследованием (единичным или множественным), в соответствии с общими принципами ОО.

У15.5 Пакеты-классы

(Это упражнение предполагает хорошее знание Ada 83.) Используя в качестве образца типы задач, придумайте адаптацию Ada 83, поддерживающую пакеты, создающие экземпляры во время выполнения, а, следовательно, играющие роль классов с полиморфизмом, динамическим связыванием и наследованием.

Основы объектно-ориентированного проектирования

16. Лекция: Эмуляция объектной технологии в не ОО-средах

Языки Fortran, Cobol, Pascal, C, Basic, PL/I и даже ассемблер до сих пор составляют существенную часть создаваемого или обновляемого ПО. Ясно, что проект, использующий один из этих языков, не сможет получить все преимущества объектной технологии. Это потребовало бы введения нотации, подобной изучаемой в этой книге, и, соответственно, компилятора, среды и библиотек. Но те, кто вынужден использовать такие инструменты, чаще всего из-за ограничений не технического характера, все же могут, вдохновившись объектной технологией, использовать ее концепции для улучшения качества своего ПО. Данная лекция представляет технические приемы эмуляции объектов, дающие возможность приблизиться к объектной технологии. В частности, эти приемы рассматриваются для языков Fortran, Pascal и C, но они применимы не только для упомянутых языков. Ada и другие инкапсулирующие языки обсуждались в предыдущей лекции. В следующей лекции будут рассмотрены такие ОО-языки, как Simula, Smalltalk, Objective-C, C++ и Java. Заметьте: при использовании языка, не входящего в список, например, Basic или Cobol, перенос рассматриваемых концепций не должен вызвать особых трудностей; при использовании ОО-языка данное обсуждение может привести к лучшему пониманию новшеств объектной технологии и поддерживающей техники реализации.

Уровни языковой поддержки

Оценивая возможности поддержки ОО-концепций языками программирования, можно разделить их на три широкие категории (игнорируя самый низкий уровень, в основном, ассемблерных языков, не поддерживающих даже понятия подпрограммы):

- К функциональному уровню отнесем языки, где единицей декомпозиции является **подпрограмма**, функциональная абстракция, описывающая шаг обработки. Абстракция данных, если она есть, обрабатывается через определения структур данных, либо локальных для подпрограммы, либо глобальных.
- Языки **инкапсулирующего** уровня позволяют группировать подпрограммы и данные в синтаксической единице, называемой **модулем** или **пакетом**. Обычно такие единицы допускают независимую компиляцию. Довольно подробно это обсуждалось при рассмотрении языка Ada.
- На третьем уровне идут **ОО-языки**. Здесь не место обсуждать, что дает право языку на такое звание. Это вопрос детально рассмотрен в [лекции 2](#) курса "Основы объектно-ориентированного программирования", здесь же отметим необходимость поддержки классов, наследования, полиморфизма и динамического связывания.

Для категории инкапсулирующих языков, поддерживающих механизм абстракции данных, но не поддерживающих классы, наследование, полиморфизм и динамическое связывание, в литературе используется термин **основанный на объекте**, введенный в статье Питера Вегнера. Поскольку слова **основанный** и **ориентированный** близки и не отражают концептуальной разницы между языками, довольно трудно объяснить, особенно новичкам, суть термина "основанный на объекте". Поэтому я решил придерживаться выражений "инкапсулирующие языки" и "объектно-ориентированные языки".

Еще немного о терминологии. Термин "функциональные языки" двусмысленен, поскольку в литературе он применяется к классу языков, основанных на математических принципах и часто прямо или косвенно происходящих от Lisp, а этот язык использует функции, свободные от побочных эффектов, вместо императивных конструкций, таких как процедуры и присваивания. Во избежание путаницы в данной книге для обозначения этого стиля программирования всегда используется термин "аппликативный". В нашем толковании "функционального языка" слово функция противопоставляется не процедуре, а объекту. Путаница еще более усугубляется употреблением термина "процедурный язык" как синоним "не объектно-ориентированный". Для такой терминологии нет оснований - для нас "процедурный" является синонимом "императивный", в противоположность термину "аппликативный". Все обычные ОО-языки, включающие нотацию этой книги, явно процедурные.

Общее замечание по ОО-эмulation. В своей основе объектная технология - это "программирование с абстрактными типами данных". Даже на функциональном уровне можно применятьrudimentарную форму этой идеи, определив набор строгих методологических правил, требующих вызова подпрограмм для доступа к данным. Предполагается, что начинать надо с ОО-построения, определяющего АТД и его компоненты. Затем пишется набор подпрограмм, представляющих эти компоненты - `put`, `remove`, `item`, `empty`, как в нашем стандартном примере стека. Далее требуется, чтобы все клиентские модули использовали только эти подпрограммы. При отсутствии языковой поддержки, но при условии, что все в команде подчиняются навязанным правилам, это можно рассматривать как начало объектного подхода. Назовем эту технику дисциплинарным подходом.

ОО-программирование на языке Pascal?

Язык Pascal, введенный в 1970г. Никласом Виртом, много лет являлся доминирующим языком начального обучения программированию на факультетах информатики и дал большое влияние на построение последующих языков. Pascal - это функциональный язык в только что определенном смысле.

Собственно Pascal

Многое ли из ОО-подхода можно реализовать в Pascal?

К сожалению, немногое. Структура программы в Pascal основана на совершенно другой парадигме. Программа на языке Pascal состоит из последовательности описательных разделов, следующих в неизменном порядке: метки,

константы, типы, переменные, подпрограммы (процедуры и функции) и раздела выполняемых инструкции. Сами подпрограммы рекурсивно имеют ту же структуру.

Это простое правило облегчает однопроходную компиляцию. Но любая попытка использования ОО-техники обречена. Рассмотрим, что нужно для реализации ADT, например, стека, представленного массивом: несколько констант, задающих размер массива, тип, описывающий элементы стека, несколько переменных, таких как указатель на вершину стека и несколько подпрограмм, задающих операции АТД. В Pascal эти элементы будут разбросаны по разделам: все константы вместе, все типы вместе и т. д.

Результирующая программная структура противоположна ОО-проектированию. Использование Pascal противоречит принципу Лингвистических Модульных Единиц: любая политика модульности должна поддерживаться доступными конструкциями языка.

Итак, если рассматривать официальный стандарт Pascal, то сделать можно немногое, не считая использования дисциплинарного подхода.

Модульные расширения языка Pascal

За пределами стандарта Pascal многие коммерчески доступные версии снимают ограничения на порядок объявлений и включают поддержку модульности, включая независимую компиляцию. Такие модули могут содержать константы, типы и подпрограммы. Эти языки более гибкие и сильные, чем стандартный Pascal, сохраняют имя Pascal. Они не стандартизированы, и в действительности больше напоминают инкапсулирующие языки, такие как Modula-2 или Ada, обсуждаемые в предыдущей лекции.

ОО-расширения языка Pascal

Некоторые компании предложили ОО-расширения языка Pascal, широко известные как "Object Pascal". Две версии особенно значимы:

- версия Apple, происходящая от языка, первоначально называвшегося Clascal и используемого для компьютера Macintosh и его предшественника, - Lisa;
- версия Borland Pascal, адаптированная в среде Borland Delphi.

Наше обсуждение не затрагивает эти языки, так как реально их связь с Pascal проявляется только в имени, стиле синтаксиса и статически типизированном подходе. В частности, Borland Pascal - это ОО-язык с обработкой исключений. Он не поддерживает механизмы универсальности, утверждений, сборки мусора и множественного наследования.

Fortran

FORTRAN должен фактически устранить кодирование и отладку.

FORTRAN: Предварительный отчет, IBM, Ноябрь, 1954

Самый старый уцелевший язык программирования Fortran по-прежнему широко используется в сфере научных вычислений. Это может показаться удивительным для тех, кто использует такие "структурированные" языки как Pascal, но в Fortran легче достигнуть многих ОО-свойств, благодаря возможностям, которые считаются низкоуровневыми и предназначены для других целей.

Немного контекста

Язык Fortran был первоначально создан как инструмент для программирования на IBM 704 командой IBM во главе с Джоном Бэкусом (позже он принимал активное участие в описании языка Algol). Первая версия языка появилась в 1957 году. Затем последовал Fortran II, введший подпрограммы. Fortran IV появился в 1966 году и был стандартизован ANSI (Fortran III не получил широкого распространения). Дальнейший процесс развития привел к Fortran 77, одобренному в 1978 году, имевшему лучшие управляющие структуры и приятные упрощения. После длительного процесса пересмотра появился Fortran 90 и Fortran 95, они были по-разному приняты и до сих пор не заменили своих предшественников.

Для большинства специалистов в компьютерных науках Fortran устарел. Однако можно найти немало программистов Fortran еще с тех времен и тех, в чьих дипломах стоит "физик-теоретик", "прикладной математик", "инженер-механик", или даже "специалист по ценным бумагам", использующих Fortran повседневно и не только в старых проектах, но и в новых разработках.

Постороннему наблюдателю иногда кажется, что программирование в физике и других научных дисциплинах, где позиции Fortran наиболее сильны, остается в стороне от программной инженерии. Это отчасти верно, отчасти - нет. Низкий уровень языка и особая природа научного вычисления (ПО создаются людьми, хотя и учеными по образованию, но часто не имеющими формального программного образования) приводят изначально к невысокому качеству создаваемого ПО. Но некоторые из лучших и самых мощных программных систем созданы именно в этой области, включающей и передовое моделирование чрезвычайно сложных процессов, и поразительные инструменты научной визуализации. Такие продукты больше не ограничиваются изящными, но небольшими числовыми

алгоритмами. Как и их аналоги в других областях приложения, они часто манипулируют сложными структурами данных, опираются на технологию баз данных, включают множество компонентов пользовательских интерфейсов. И, хотя это может показаться удивительным, они все еще зачастую пишутся на языке Fortran.

Техника COMMON

Fortran система состоит из главной программы и ряда подпрограмм. Как обеспечить схожесть с абстракцией данных?

Возможная техника состоит в том, чтобы представить данные в так называемом общем блоке COMMON, а экспортруемые компоненты (например, `put` и т. д. для стеков) реализовать в виде независимых подпрограмм. Блок COMMON - это механизм Fortran, предоставляющий доступ к данным любой подпрограмме, желающей их получить. Вот набросок подпрограммы `put` для стека действительных чисел:

```
SUBROUTINE RPUT (X)
    REAL X
C
C      ВТАЛКИВАНИЕ X НА ВЕРШИНУ СТЕКА
C
    COMMON /STREP/ TOP, STACK (2000)
    INTEGER TOP
    REAL STACK
C
    TOP = TOP + 1
    STACK (TOP) = X
    RETURN
END
```

Эта версия не управляет переполнением (будет исправлено в следующей версии). Функция, возвращающая элемент вершины:

```
INTEGER FUNCTION RITEM
C
C      ВОЗВРАЩЕНИЕ ВЕРШИНЫ СТЕКА
C
    COMMON /STREP/ TOP, STACK (2000)
    INTEGER TOP
    REAL STACK
    RITEM = STACK (TOP)
    RETURN
END
```

Здесь также необходимо было бы проверять стек на пустоту. Подпрограммы `REMOVE` и другие строятся по тому же образцу. Имя общего блока - `STREP` - объединяет различные подпрограммы, дающие доступ к одним и тем же данным.

Ограничения очевидны: данная реализация описывает один абстрактный объект (один отдельный стек), а не абстрактный тип данных, из которого во время выполнения можно создать множество экземпляров. Мир Fortran статичен: необходимо указывать размеры всех массивов (в примере 2000 - произвольно выбранное число). Поскольку отсутствует универсальность, то в принципе, придется объявлять новый набор подпрограмм для каждого типа элементов стека. Отсюда имена `RPUT` и `RITEM`, где `R` означает `Real`. Можно справиться с этими проблемами, но не без значительных усилий.

Техника подпрограммы с множественным входом

Техника, основанная на блоке COMMON, как это видно, нарушает Принцип Лингвистических Модульных Единиц. В модульной структуре системы подпрограммы, являясь концептуально связанными, физически независимы.

Эту ситуацию можно улучшить (не убирая другие перечисленные ограничения) посредством особенностей языка, легализованной в Fortran 77 - множественными точками входа в одной подпрограмме.

Это расширение, введенное, возможно, для других целей, можно использовать во благо ОО-подхода. Клиентские подпрограммы могут вызывать точки входа, как если бы они были автономными подпрограммами, и разные входы могут иметь разные аргументы. Вызов входа начинает выполнение подпрограммы с этой точки. Все входы разделяют хранимые данные подпрограммы, появляющиеся с директивой `SAVE` и сохраняемые от одной активизации подпрограммы к другой. Понятно, куда мы клоним: эту технику можно использовать для определения модуля, инкапсулирующего абстрактный объект, почти как в инкапсулирующем языке. Здесь модуль моделируется подпрограммой, структура данных - набором объявлений с директивой `SAVE`, и каждый компонент соответствующего класса в ОО-языке - входом, заканчивающимся инструкцией `RETURN`:

```
ENTRY (arguments)
```

... Инструкции ...

RETURN

В отличие от предыдущего решения, основанного на COMMON, теперь все необходимое сосредоточено в единой синтаксической единице. В таблице 34.1 дан пример реализации стека действительных величин. Вызовы клиента выглядят следующим образом:

```
LOGICAL OK
REAL X
C
OK = MAKE ()
OK = PUT (4.5)
OK = PUT (-7.88)
X = ITEM ()
OK = REMOVE ()
IF (EMPTY ()) A = B
```

Взглянув на этот текст, можно почти поверить, что это использование класса, или, по крайней мере, объекта, через его абстрактный, официально определенный интерфейс!

Подпрограмма в Fortran и ее точки входа должны быть все либо подпрограммами, либо функциями. Здесь, поскольку EMPTY и ITEM должны быть функциями, все другие входы тоже объявляются как функции, включающие MAKE, чей результат бесполезен.

Таблица 16.1. Эмуляция модуля стек в Fortran

C --РЕАЛИЗАЦИЯ	C --УДАЛЕНИЕ ВЕРШИНЫ
C --АБСТРАКТНЫЙ СТЕК ЧИСЕЛ	C ENTRY REMOVE (X)
C	IF (LAST .NE. 0) THEN
INTEGER FUNCTION RSTACK ()	REMOVE = .TRUE.
PARAMETER (SIZE=1000)	LAST = LAST - 1
C	ELSE
--ПРЕДСТАВЛЕНИЕ	REMOVE = .FALSE.
C	END IF
REAL IMPL (SIZE)	RETURN
INTEGER LAST	C --ЭЛЕМЕНТ ВЕРШИНЫ
SAVE IMPL, LAST	C ENTRY ITEM ()
C	IF (LAST .NE. 0) THEN
--ВХОД С ОБЪЯВЛЕНИЯМИ	ITEM = IMPL (LAST)
C	ELSE
LOGICAL MAKE	CALL ERROR
LOGICAL PUT	(* ('ITEM: EMPTY STACK'))
LOGICAL REMOVE	END IF
REAL ITEM	RETURN
LOGICAL EMPTY	C -- ПУСТ ЛИ СТЕК?
C	C ENTRY EMPTY ()
REAL X	EMPTY = (LAST .EQ. 0)
C	RETURN
-- СОЗДАНИЕ СТЕКА	C END
C	
ENTRY MAKE ()	
MAKE = .TRUE.	
LAST = 0	
RETURN	
C	
-- ДОБАВЛЕНИЕ ЭЛЕМЕНТА	
C	
ENTRY PUT (X)	
IF (LAST .LT. SIZE) THEN	
PUT = .TRUE.	
LAST = LAST + 1	
IMPL (LAST) = X	
ELSE	
PUT = .FALSE.	
END IF	
RETURN	

Этот стиль программирования может успешно применяться для эмуляции инкапсуляции Ada или Modula-2 в контекстах, где нет другого выбора, кроме использования Fortran. Конечно, он страдает от жестких ограничений:

- Не разрешаются никакие внутренние вызовы: в то время как подпрограммы в ОО-классе обычно опираются друг на друга для реализации, вызов выхода той же подпрограммы будет понят как рекурсия - проклятие для Fortran - и бедствие во время выполнения во многих реализациях.

- Как отмечалось, этот механизм строго статичен, поддерживая только один абстрактный объект. Он может быть обобщен преобразованием каждой переменной в одномерный массив. Но не существует переносимой поддержки для создания динамического объекта.
- На практике некоторые среди Fortran не слишком хорошо работают с множественными входами подпрограмм.
- Наконец, сама идея использования языкового механизма для целей, отличных от проектируемых, порождает опасность путаницы.

ОО-программирование и язык С

Созданный в тиши кабинета язык С быстро стал известным. Большинство людей, интересующиеся и С, и объектной технологией, перешли к ОО-расширениям С, обсуждаемым в следующей лекции (C++, Objective-C, Java). Но по-прежнему интересно, как можно заставить сам С эмулировать ОО-концепции.

Немного контекста

Язык С создавался в AT&T's Bell Laboratories как машинонезависимый язык для написания операционных систем. Первая версия ОС Unix была написана на языке ассемблера, но вскоре потребовалась ее переносимая версия. Для решения этой задачи в 1970 г. и был создан язык С. Он вырос из идей языка BCPL. Как и С, это язык высокого уровня (благодаря управляющим структурам, похожим на структуры в Algol или Pascal), машинно-ориентированный (из-за возможности манипулировать данными на самом низком уровне через адреса, указатели и байты) и переносимый (поскольку машинно-ориентированные концепции охватывают широкий круг типов компьютеров). Язык С появился вовремя. В конце 70-х операционная система Unix использовалась во многих университетах, и вместе с ней распространялся С. В 80-х гг. началась революция микроКомпьютеров, и С был готов служить ей как *lingua franca* (**язык франков**), поскольку был более масштабируемым, чем Basic, и более гибким, чем Pascal. Система Unix тоже была коммерчески успешна, и с ней рядом шел С. Через несколько лет он стал доминирующим языком в больших и самых активных сегментах компьютерной индустрии.

Все, кто интересуется развитием языков программирования, и даже те, кто не очень обращает внимание на сам язык, должны быть признательны языку С по крайней мере по двум причинам.

- Язык С покончил с состоянием закостенелости, существовавшим в мире языков программирования приблизительно до 1980 г. В то время никто не хотел слышать (особенно после коммерческой неудачи Algol) ни о чем другом, кроме священной тройки: Fortran в науке, Cobol в бизнесе и PL/I в сфере могущества IBM. Вне академических кругов любые попытки предложить другие решения встречались как предложения еще одного сорта Колы. Язык С разрушил это состояние, дал возможность думать о языке программирования как о чем-то, выбираемом из большого каталога. (Несколько лет спустя сам С настолько укрепил позиции, что в некоторых кругах выбор свелся к нему одному, но такова судьба многих успешных ниспровергателей.)
- Переносимость языка С и его близость к машине делали его привлекательным в качестве языка написания компиляторов. Этот подход использовался при создании компиляторов для языков C++ и Objective-C. За ними последовали компиляторы для других языков. Преимущества для создателей компиляторов и их пользователей очевидны: переносимость (компиляторы С есть почти для любой компьютерной архитектуры), эффективность (оптимизация, реализуемая хорошим компилятором), и простота интеграции с универсальными инструментами и компонентами, основанными на С.

Стандарт ANSI для С впервые был опубликован в 1990 году. Более ранняя версия языка известна как K&R (по инициалам авторов первой книги по С - Кернигана и Ритчи). Со временем противоречие между двумя взглядами на С - как на язык программирования высокого уровня и переносимый язык ассемблера - стало более явным. Эволюция стандарта сделала язык более типизированным и, следовательно, менее удобным для использования в качестве целевого кода компилятора. Было даже объявлено, что следующие версии будут иметь понятие класса, сглаживая отличие от C++ и Java.

Хотя, возможно, стоило бы иметь более простое, чем C++ и Java, ОО-расширение С, но не ясно, является ли эта разработка правильной для С. ОО-язык, основанный на С, всегда будет казаться странным изобретением. А идея простого, переносимого, универсального, эффективно компилируемого, машинно-ориентированного языка остается по-прежнему полезной. Он может служить и в качестве целевого языка для компиляторов высокого уровня, и инструмента низкого уровня для написания коротких подпрограмм для доступа к операционной системе (то есть, для того же, что язык ассемблера обычно делал для С, только на следующем уровне).

Основные положения

Дисциплинарный подход применим к языку С, как и к любому другому языку. За его пределами для реализации модульности можно использовать понятие файла. Файл - это понятие языка С, балансирующее на границе между языком и операционной системой. Файл - единица компиляции, он может содержать функции и данные. Некоторые функции могут быть скрытыми от других файлов, другие - общедоступны. Это прямой путь к инкапсуляции: файл может содержать все элементы, относящиеся к реализации одного или более абстрактных объектов, или абстрактного типа данных. Благодаря понятию файла, С достигает уровня инкапсулирующего языка, как Ada или Modula-2. В сравнении с Ada здесь нет универсальности и отличия между интерфейсом и реализацией.

Обычная техника программирования на С не расположена к ОО-принципам. Большинство программ С используют "файлы заголовков", описывающих разделяемые данные. Любой файл, нуждающийся в данных, получает доступ к

ним через директиву "include" (управляемую встроенным препроцессором С):

```
#include <header.h>
```

где header.h - это имя файла заголовка (h - обычный суффикс для таких имен файлов). Это эквивалентно копированию файла заголовка в точке появления директивы. В результате, традиция С, если не сам язык, дает возможность модулям клиента получить доступ к структурам данных через их физические представления, что явно противоречит принципам скрытия информации и абстракции данных. Однако возможно использовать файлы заголовка более дисциплинированным путем, скорее насаждая, а не нарушая абстракцию данных. Они могут даже помочь продвинуться к определению модулей интерфейса в стиле, обсуждаемом для языка Ada в предыдущей лекции.

Эмуляция объектов

Помимо инкапсуляции, для эмуляции продвинутых свойств настоящего ОО-подхода можно использовать одно из наиболее специализированных свойств языка - возможность манипуляций с указателями на функции. Этот механизм заслуживает внимания, хотя он требует аккуратного обращения и его стоит рекомендовать в первую очередь разработчикам компиляторов, а не обычным программистам.

С внешней точки зрения "каждый объект имеет доступ к операциям, применимым к нему". Возможно, это немного наивно, но не является концептуально неверным. Язык С буквально поддерживает это понятие! Экземпляр "структурой" языка С (эквивалент записи в Pascal) может содержать среди своих полей указатели на функции.

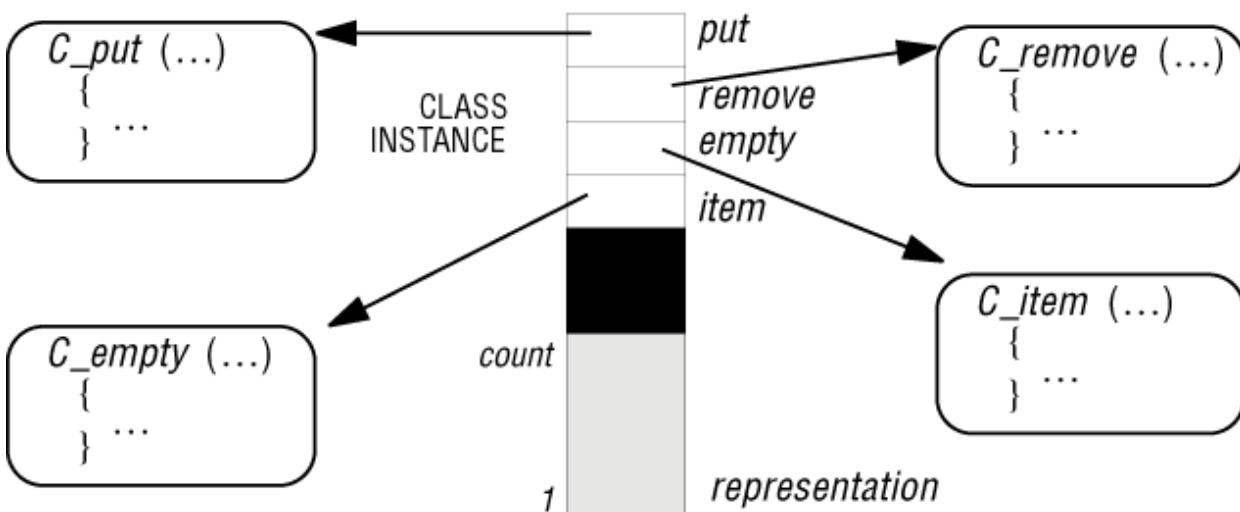


Рис. 16.1. Объект С со ссылками на функцию

Например, структурный тип REAL_STACK можно объявить так:

```
typedef struct
{
    /* Экспортируемые компоненты */
    void (*remove)();
    void (*put)();
    float (*item)();
    BOOL (*empty)();
    /* Закрытые компоненты (реализация) */
    int count;
    float representation [MAXSIZE];
}
```

REAL_STACK;

Фигурные скобки { . . . } ограничивают компоненты структуры; float задает вещественный тип; процедуры объявляются как функции с типом результата void; комментарии берутся в скобки /* и */. Важный символ *? служит для разыменования указателей. В практике программирования на С, чтобы все работало, принято добавлять достаточное количество указателей, если это не помогает, то всегда можно попробовать добавить один или парочку символов &. Если и это не дает результата, всегда найдется кто-нибудь, кто сможет помочь.

В структурном типе REAL_STACK два последних компонента - переменная и массив, остальные - ссылки на функции. В данном тексте комментарии предупреждают об экспортных и закрытых компонентах эмулируемого класса, но на уровне языка клиентам доступно все.

Каждый экземпляр типа должен инициализироваться так, чтобы поля ссылок указывали на соответствующие функции. Например, если my_stack является переменной этого типа, а C_remove - функция, реализующая выталкивание из стека, то можно присвоить полю remove объекта my_stack ссылку на эту функцию таким образом:

```
my_stack.remove = C_remove
```

В эмулируемом классе `remove` не имеет необходимого для нее аргумента. Для доступа к соответствующему стеку следует объявить функцию `C_remove` так:

```
C_remove (s)
REAL_STACK s;
{
... Реализация операции remove ...
}
```

Тогда клиент сможет применить `remove` к стеку `my_stack`:

```
my_stack.remove (my_stack)
```

В общем случае, подпрограмма `rout`, имеющая n аргументов в эмулируемом классе, порождает функцию `C_rout` с $n+1$ аргументами. Вызов ОО-подпрограммы:

```
x.rout (arg1, arg2, ..., argn)
```

Эмулируется как:

```
x.C_rout (x, arg1, arg2, ..., argn)
```

Эмулирующие классы

Описанная техника будет работать в определенных пределах. Ее даже можно расширить для эмуляции наследования.

Но она неприменима к серьезным разработкам, что иллюстрируется на [рис. 16.1](#). Каждый экземпляр любого класса должен физически содержать ссылки на все применимые к нему подпрограммы. Это приведет к существенным потерям памяти, особенно при наследовании.

Для снижения потерь заметим, что подпрограммы одинаковы для всех экземпляров класса. Поэтому для каждого класса можно ввести структуру данных периода выполнения, дескриптор класса, содержащий ссылки на подпрограммы. Его можно реализовать как связный список или массив. Требования к пространству значительно уменьшаются: вместо n^m указателей можно иметь их $n+m$, где n - число подпрограмм, а m - число объектов, как показано на [рис. 16.2](#).

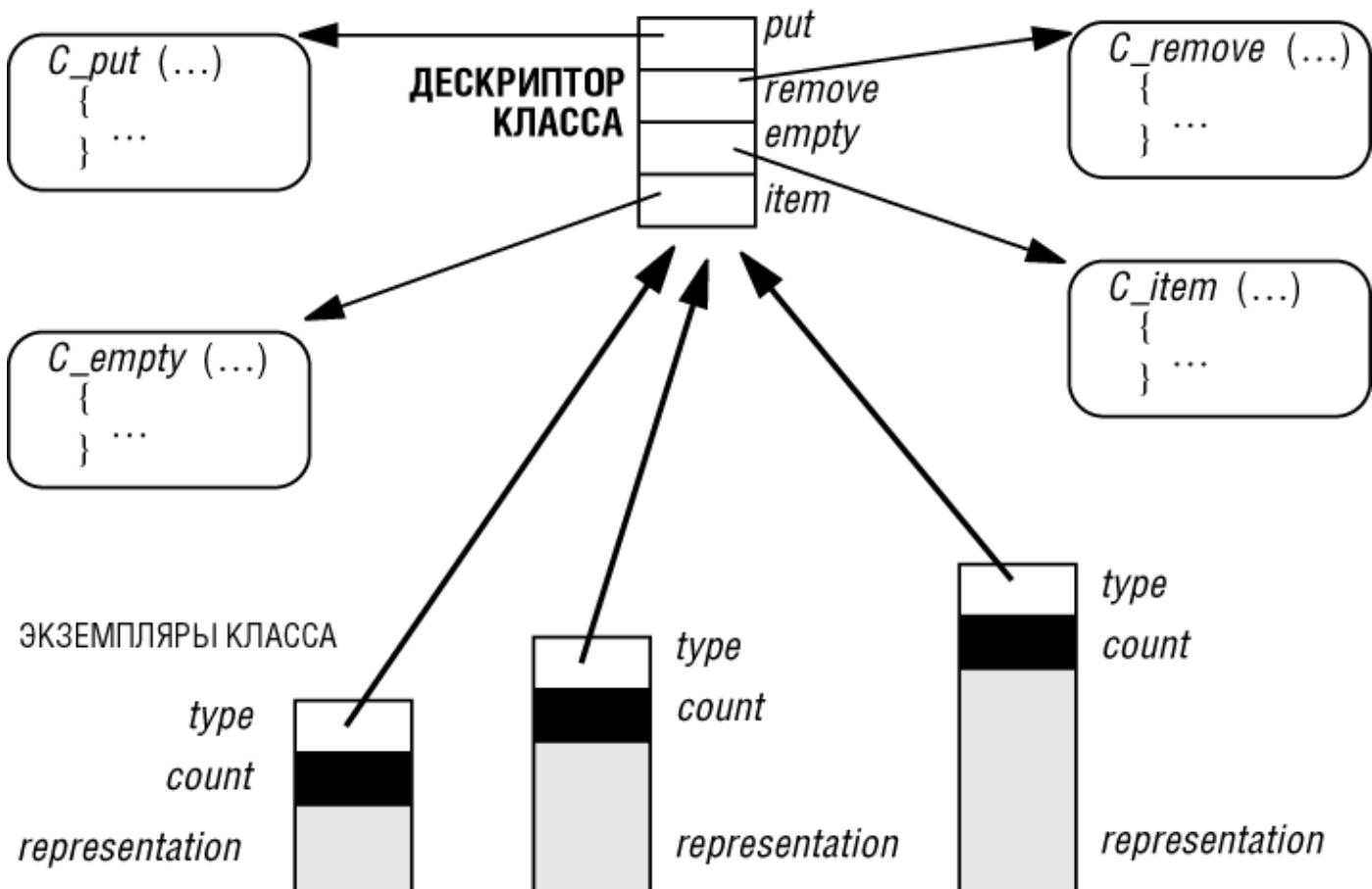


Рис. 16.2. Объекты С, разделяющие дескриптор класса

Это приводит к незначительным временным потерям, но экономия пространства и простота стоят этого.

В этой технике нет секрета. Именно она сделала С полезным в качестве средства реализации для компиляторов ОО-языков, начиная с Objective-C и C++ в начале 80-х. Способность использовать указатели функций, в сочетании с идеей группирования этих указателей в дескриптор класса, разделяемый произвольным числом экземпляров, служит первым шагом к реализации ОО-техники.

Конечно, это только первый шаг, и нужно еще найти способы реализации наследования (особенно непросто множественное наследование), универсальности, исключений, утверждений и динамического связывания. Объяснения потребовали бы отдельной книги. Отметим лишь одно важное свойство, выводимое из всего, что мы видели до сих пор. Реализация динамического связывания требует доступа к типу каждого объекта во время выполнения для нахождения нужного варианта компонента f в динамически связываемом вызове $x.f(\dots)$ (написанном здесь в ОО-нотации). Другими словами: в дополнение к официальным полям объекту необходимо дополнительное внутреннее поле, порождаемое компилятором и указывающее на **тип** объекта. Описанный подход показывает возможную реализацию такого поля - как указателя на дескриптор класса. По этой причине на [рис. 16.2](#) для такого поля используется ярлык `ture`.

ОО С: оценка

Обсуждение показало, что в С есть технические способы введения ОО-идей. Но это еще не значит, что программисты должны их использовать. Как и в случае с языком Fortran, эмуляция - это некоторое насилие над языком. Сила языка С - в его доступности как "структурного языка ассемблера" (последователя BCPL и PL/360, созданного Виртом), переносимого, разумно простого и эффективно интерпретируемого. Его базисные понятия далеки от ОО-проектирования.

Опасность попыток навязывания языку С ОО-идей может привести к несостойтельной конструкции, ухудшающей процесс разработки ПО и качество получаемых продуктов. Лучше использовать С для того, что он может делать хорошо: создания интерфейсов для оборудования и операционных систем, и как машинно-генерируемый целевой код. Для применения объектной технологии лучше использовать инструмент, созданный для этой цели.

Библиографические замечания

Способы написания пакетов в Fortran, основанные на принципах абстракции данных, описаны в [M1982a]. Они используют подпрограммы, разделяющие блоки COMMON, а не подпрограммы с множественным входом, и идут дальше в реализации ОО-концепций, чем способы, описываемые в данной лекции. Это становится возможным благодаря использованию специфических библиотечных механизмов, эквивалентных динамически размещаемым экземплярам классов. Однако такие механизмы требуют значительных вложений и должны переноситься на каждый тип платформы.

Я благодарен Полю Дюбуа за замечание, что техника Fortran со множественными входами, являясь частью стандарта, не всегда хорошо поддерживается текущими компиляторами.

[Cox 1990] содержит обсуждение технических приемов С для реализации ОО-концепций.

Ссылки на историю классических языков программирования даны в трудах конференции [Wexelblat 1981], см. также [Knuth 1980].

Упражнения

У16.1 Графические объекты (для программистов на Fortran)

Напишите на Fortran набор подпрограмм с множественными входами, реализующих основные графические объекты (точки, окружности, многоугольники). Для спецификации имеющихся абстракций и соответствующих операций можно опираться на графический стандарт GKS.

У16.2 Универсальность (для программистов на С)

Как бы Вы преобразовали на С эмуляцию класса "real stack" в эмуляцию с родовыми параметрами, адаптируемую к стекам любого типа G, а не просто float?

У16.3 ОО-программирование на С (семестровый проект)

Постройте и реализуйте простое ОО-расширение С, используя идеи этой лекции. Вы можете написать либо препроцессор, переводя расширенную версию языка на С, либо функциональный пакет, не изменяющий самого языка.

Подойдите к задаче через три последовательные уточнения:

- сначала реализуйте механизм, позволяя объектам содержать их собственные ссылки на имеющиеся подпрограммы;

- затем посмотрите, как факторизовать ссылки на уровне класса;
- наконец, изучите, как добавить механизм единичного наследования.

Основы объектно-ориентированного проектирования

17. Лекция: От Simula к Java и далее: основные ОО-языки и окружения

Под влиянием языка Simula, введенного в 1967 г., появился ряд ОО-языков, отличающихся разнообразием подходов. Эта лекция описывает некоторые из них, привлекшие самое большое внимание: Simula, Smalltalk, C++ и другие ОО-расширения С, Java. В литературе до сих пор нет глубокого сравнительного описания важнейших ОО-языков. Цель этой лекции гораздо скромнее. Потому некоторые наиболее популярные языки описываются весьма кратко. Наша цель - изучить проблемы и концепции, находя их, где это возможно, даже если придется обратиться к менее популярным подходам. Для языка, имеющего практическую значимость, невелик риск пропустить что-то главное, поскольку многочисленные статьи и книги описывают его достаточно подробно. Настоящий риск таится в обратном: риск пропустить перспективную идею, просто потому, что поддерживающий ее язык (скажем, Simula) сейчас не так популярен. Таким образом, описание будет следовать не принципу равных возможностей в выборе примечательных языковых свойств, а принципу позитивных действий. В официальных описаниях языков для одних и тех же или схожих концепций используются различные термины. Мы старались применять "родные" термины, когда они отражают особенности языка. Однако для простоты и согласованности используется и терминология, принятая в книге (как попытка унификации), когда различия не так важны. Например, мы говорим о подпрограммах, процедурах и функциях в языке Simula, хотя соответствующие термины в официальном использовании Simula - процедура, нетипизированная процедура и типизированная процедура.

Simula

Simula - это несомненный основатель Дома Классов (Дворца Объектов). Создание его было завершено (если не принимать во внимание небольшие более поздние обновления) в 1967 г. В это, возможно, трудно поверить: оформившийся ОО-язык существовал и был реализован до структурного программирования, до публикации Парнасом статей по скрытию информации, задолго до появления фразы "абстрактный тип данных". Война во Вьетнаме еще освещалась в газетах, мини-юбки еще могли возбуждать, а на северных берегах Балтики несколько удачливых разработчиков ПО, ведомые горсткой мечтателей, уже использовали силу классов, наследования, полиморфизма, динамического связывания и других прелестей ОО.

Основные понятия

Simula, по существу, является второй попыткой. В начале 60-х был разработан язык, известный как Simula 1, для поддержки моделирования дискретных событий. Хотя он не был ОО-языком в полном смысле термина, но суть он уловил. Собственно Simula - это Simula 67, созданный в 1967 г. Далее и Нигардом (Kristen Nygaard, Ole-Johan Dahl) из Университета Осло и Норвежского Компьютерного Центра (Norsk Regnesentral). Нигард потом объяснял, что решение сохранить название отражало связь с предыдущим языком и с сообществом его пользователей. К несчастью, это название долгое время для многих людей создавало образ языка, предназначенного только для моделирования событий, что было довольно узкой областью приложения, в то время как Simula 67 - это общечелевой язык программирования. Единственные его компоненты моделирования - это набор инструкций и библиотечный класс SIMULATION, используемый небольшим числом разработчиков Simula.

Название было сокращено до Simula в 1986 г., текущий стандарт датируется 1987 г.

Доступность

Simula часто представляется как респектабельный, но более не существующий предок. В действительности он еще жив и используется небольшим, но восторженным сообществом. Определение языка поддерживается Группой Стандартов Simula (Simula Standards Group). Существуют компиляторы и ПО от нескольких компаний, в основном скандинавских.

Основные черты языка

Рассмотрим в общих чертах основные свойства Simula. Автор не обидится, если читатель перейдет к следующему разделу о Smalltalk. Но чтобы полностью оценить объектную технологию, стоит потратить время на изучение Simula. Концепции представлены в их первоначальной форме, и некоторые возможности еще и теперь, спустя тридцать лет, не полностью использованы.

Simula - ОО-расширение языка Algol 60. Большинство правильных программ на Algol также являются правильными на Simula. В частности, основные структуры управления такие же, как в Algol: цикл, условный оператор, переключатель (низкоуровневый предшественник команды case в Pascal). Основные типы данных (целые, действительные и т. д.) тоже взяты из Algol.

Как и Algol, Simula использует на самом высоком уровне традиционную структуру ПО, основанную на понятии главной программы. Выполняемая программа - это главная программа, содержащая ряд программных единиц (подпрограмм или классов). Программная среда Simula поддерживает независимую компиляцию классов.

Simula использует структуру блока в стиле Algol 60: программные единицы, такие как классы, могут быть вложены друг в друга.

Все реализации Simula поддерживают автоматическую сборку мусора. Есть маленькая стандартная библиотека, включающая, в частности, двусвязные списки, используемые классом SIMULATION, изучаемым далее в этой лекции.

Как и в нотации этой книги, большинство общих сущностей, не относящихся к встроенным типам, обозначают ссылки на экземпляры класса, а не сами экземпляры. Однако это их явное свойство, подчеркиваемое нотацией. Тип такой

сущности объявляется как ссылочный ref(C), а не просто C, для некоторого класса C. Для них используются специальные символы для присваивания, проверки на равенство и неравенство (: -, ==, /=), в то время как целочисленные и действительные операнды используют другие символы для этих целей (:=, =, /=). Выше в одной из лекций даны обоснования за и против этого соглашения.

Для создания экземпляра используется выражение new, а не команда создания:

```
ref (C) a; ...;a :- new C
```

Выражение new создает экземпляр C и возвращает ссылку на него. Класс может иметь аргументы (играющие роль аргументов процедур создания в нашей нотации):

```
class C (x, y); integer x, y  
begin ... end;
```

В этом случае при вызове new следует передать соответствующие фактические аргументы:

```
a :- new C (3, 98)
```

Аргументы могут использоваться в подпрограммах класса. Но в отличие от возможности использования нескольких команд создания (конструкторов), данный подход дает только один механизм инициализации.

Кроме подпрограмм и атрибутов, класс может содержать последовательность инструкций, тело класса. Если тело существует, то вызов new будет выполнять его. Мы увидим, как использовать эту возможность, чтобы заставить классы представлять не пассивные объекты, как в большинстве других ОО-языков, а активные элементы, подобные процессам.

Механизм утверждений в языке не поддерживается. Simula поддерживает единичное наследование. Вот как класс B объявляется наследником класса A:

```
A class B;  
begin ... end
```

Для переопределения компонента класса в классе наследника, нужно просто задать новое объявление. Оно имеет приоритет над существующим определением (эквивалента оператора redefine нет).

Первоначальная версия Simula 67 не имела явных конструкций скрытия информации. В последующих версиях, компонент, объявленный как protected, недоступен клиентам. Защищенный компонент, объявляемый как hidden, недоступен потомкам. Незащищенный компонент может быть защищен потомком, но защищенный компонент не может экспортиться потомками.

Отложенные компоненты задаются в форме "виртуальных подпрограмм", появляющихся в параграфе virtual в начале класса. Нет необходимости объявлять аргументы виртуальной подпрограммы, как следствие, разные эффективные определения виртуальной подпрограммы могут иметь разное число и типы аргументов. Например, класс POLYGON может начинаться так:

```
class POLYGON;  
    virtual: procedure set_vertices  
begin  
    ...  
end
```

позволяя потомкам задавать различное число аргументов типа POINT для set_vertices: три - для TRIANGLE, четыре - для QUADRANGLE и т. д. Эта гибкость подразумевает некоторую проверку типов во время выполнения.

Пользователям C++ следует опасаться возможной путаницы: хотя C++ был инспирирован Simula, он использует другую семантику virtual. Функция C++ объявляется виртуальной, если целью является динамическое связывание (как отмечалось, это один из самых противоречивых аспектов C++, разумнее динамическое связывание подразумевать по умолчанию). Виртуальным процедурам Simula соответствуют "чистые виртуальные функции" C++.

Simula поддерживает полиморфизм: если B - потомок A, присваивание a1 :- b1 корректно для a1 типа A и b1 типа B. Довольно интересно, что попытка присваивания почти рядом: если тип b1 является предком типа a1, присваивание будет работать, если во время выполнения объекты имеют правильное отношение соответствия - источник является потомком цели. Если соответствия нет, то результатом будет ошибка во время выполнения, а не специальная величина, обнаруживаемая и обрабатываемая ПО (как при попытке присваивания). По умолчанию связывание статично, за исключением виртуальных подпрограмм. Поэтому если f - не виртуальный компонент, объявленный в классе A, a1.f будет обозначать A версию f , даже если есть другая версия в B. Можно при вызове насилино задать динамическое связывание через конструкцию qua¹¹, как в:

```
(a1 qua B). f
```

Конечно, теряется автоматическая адаптация операции к ее целевому объекту. Однако можно получить желаемое поведение динамического связывания (его можно считать изобретением Simula), объявляя полиморфные

подпрограммы как виртуальные. Во многих рассмотренных примерах полиморфная подпрограмма не была отложенной, но имела реализацию по умолчанию с самого начала. Для достижения того же эффекта разработчик Simula добавит промежуточный класс, где подпрограмма виртуальна.

В качестве альтернативы использования ции, инструкция inspect дает возможность выполнять различные операции на сущности a1, в зависимости от фактического типа соответствующего объекта, обязательно представляющего собой потомка типа A, объявленного для a1:

```
inspect a1
when A do ...;
when B do ...;
...
```

Этим достигается нужный эффект, но лишь при замороженном множестве потомков класса, что вступает в конфликт с принципом Открыт-Закрыт.

Пример

Следующие фрагменты классов показывают общий колорит Simula. Они соответствуют классам системы управления панелями, проектирование которой рассмотрено в [лекции 2](#).

```
class STATE;
virtual:
    procedure display;
    procedure read;
    boolean procedure correct;
    procedure message;
    procedure process;
begin
    ref (ANSWER) user_answer; integer choice;
    procedure execute; begin
        boolean ok;
        ok := false;
        while not ok do begin
            display; read; ok := correct;
            if not ok then message (a)
        end while;
        process;
    end execute
end STATE;
class APPLICATION (n, m);
    integer n, m;
begin
    ref (STATE) array transition (1:n, 0:m-1);
    ref (STATE) array associated_state (1:n);
    integer initial;
    procedure execute; begin
        integer st_number;
        st_number := initial;
        while st_number /= 0 do begin
            ref (STATE) st;
            st := associated_state (st_number); st.execute;
            st_number := transition (st_number, st.choice)
        end while
    end execute
    ...
end APPLICATION
```

Концепции сопрограмм

Наряду с базовыми ОО-механизмами язык Simula предлагает интересное понятие - сопрограмма.

Понятие сопрограммы рассматривалось при обсуждении параллелизма. Дадим краткое напоминание. Сопрограммы моделируют параллельные процессы, существующие в операционных системах или системах реального времени. У процесса больше концептуальной свободы, чем у подпрограммы. Например, драйвер принтера полностью ответственен за то, что происходит с принтером, им управляемым. Он не только ответственен за абстрактный объект, но и имеет собственный алгоритм жизненного цикла, часто концептуально бесконечный. Форма процесса принтера может быть приблизительно такой:

```
from some_initialization loop forever
    "Получить файл для печати"; "Напечатать его"
end
```

В последовательном программировании связь между единицами программы асимметрична: когда один программный блок вызывает другой, то последний выполняется, после чего возвращает управление вызывающему блоку в точке вызова. Процессы равноправны: каждый процесс выполняется сам по себе, прерываясь временами для предоставления информации другому процессу или ожидая ее получения.

Сопрограммы спроектированы подобным же образом, но для выполнения в одном потоке управления. (Последовательная эмуляция параллельного выполнения называется **квазипараллелизмом**.) Сопрограмма прерывает свое собственное выполнение и предлагает продолжить выполнение (resume) другой сопрограмме в ее последней точке прерывания; прерванная сопрограмма позже может продолжиться сама.

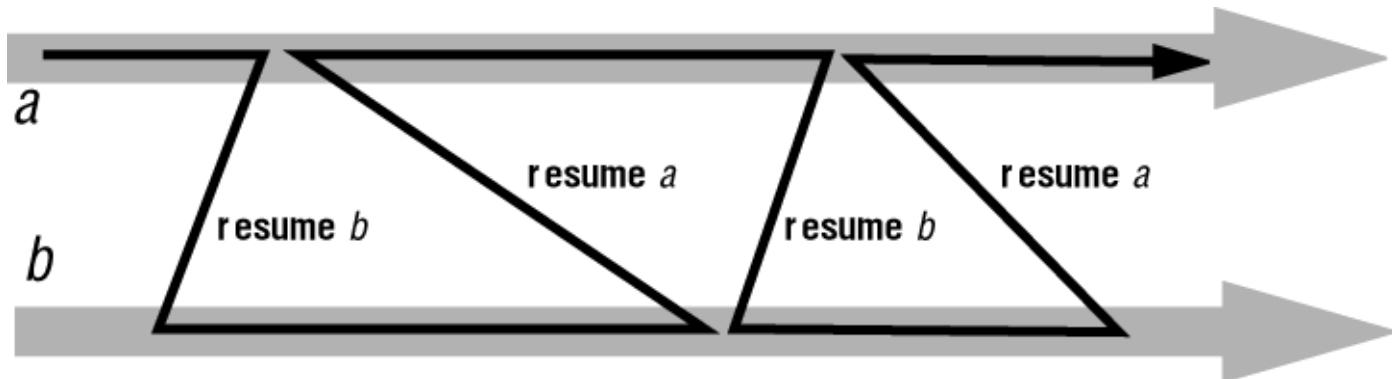


Рис. 17.1. Последовательное выполнение сопрограмм

Сопрограммы особенно полезны, когда каждая из нескольких связанных деятельности имеет собственную логику. Каждая из них может быть задана последовательным процессом, и отношение "хозяин-слуга", характерное для обычных подпрограмм, является неадекватным. Типичным примером является преобразование входных данных в выходные, где на структуру входных и выходных файлов накладываются различные ограничения. Такой случай будет обсуждаться ниже.

Simula представляет сопрограммы как экземпляры классов. Это уместно, поскольку сопрограммы почти всегда нуждаются в длительно хранимых данных, и с ними ассоциируется абстрактный объект. Как отмечалось выше, класс в Simula может иметь **тело**. В классе, представляющем абстракцию пассивных данных, оно будет служить только для инициализации экземпляров классов (эквивалент нашей процедуры создания). В сопрограмме оно будет описанием процесса. Тело сопрограммы - это обычно цикл вида

```
while continuation_condition do begin
    ... Действия ...
    resume other_coroutine;
    ... Действия ...
end
```

Для некоторых сопрограмм условием `continuation_condition` часто является `True`, что эквивалентно бесконечному процессу (несмотря на то, что хотя бы одна сопрограмма должна завершиться).

Система, основанная на сопрограммах, обычно имеет основную программу, сначала создающую ряд объектов - сопрограмм, а затем продолжает одну из них:

```
corout1 :- new C1; corout2 :- new C2; ...
resume corouti
```

Каждое выражение `new` создает объект и приступает к выполнению его тела. Но квазипараллельная природа сопрограмм (в отличие от истинного параллелизма процессов) поднимает проблему инициализации. Для процессов каждое `new` порождает новый процесс, запускает его, возвращая тут же управление исходному процессу. Но здесь только одна сопрограмма может быть активной. Если выражение `new` запустило основной алгоритм сопрограммы, то исходный процесс не получит вновь управление - у него не будет возможности создать `C2` после порождения `C1`.

Simula решает эту проблему посредством инструкции `detach`. Сопрограмма может выполнить `detach`, возвращая управление блоку, создавшему его посредством `new`. Тела сопрограмм почти всегда начинаются с `detach` (если необходимо, после инструкции инициализации), а дальше обычно следует цикл. После выполнения своего `detach` сопрограмма приостановится до тех пор, пока главная, или другая, сопрограмма не продолжит ее выполнение.

Пример сопрограммы

Приведем пример некоторой ситуации, где сопрограммы могут оказаться полезными. Вам предлагается напечатать последовательность действительных чисел в качестве ввода, но каждое восьмое число в выводе нужно опустить. Вывод должен представлять собой последовательность строк из шести чисел (кроме последней строки, если для ее заполнения чисел не хватает). Если i_n обозначает n -й элемент ввода, вывод выглядит так:

i_1	i_2	i_3	i_4	i_5	i_6
i_7	i_9	i_{10}	i_{11}	i_{12}	i_{13}

i₁₄ i₁₅ i₁₇ и т. д.

Наконец, вывод должен включать только 1000 чисел.

Эта задача характерна для использования сопрограмм. Она включает три процесса, каждый со своей специфической логикой: ввод, где требуется пропускать каждое восьмое число, вывод с ограничением строки до шести чисел, главная программа, где требуется обработать 1000 элементов. Традиционные структуры управления неудобны при сочетании процессов с разными ограничениями. Решение, основанное на сопрограммах, будет проходить гладко.

Введем три сопрограммы: producer (ввод), printer (вывод) и controller. Общая структура такова:

```
begin
    class PRODUCER begin ... См. далее ... end PRODUCER;
    class PRINTER begin ... См. далее ... end PRINTER;
    class CONTROLLER begin ... См. далее ... end CONTROLLER;
    ref (PRODUCER) producer; ref (PRINTER) printer;
    ref (CONTROLLER) controller;
    producer :- new PRODUCER; printer :- new PRINTER;
    controller :- new CONTROLLER;
    resume controller
end
```

Это главная программа, в обычном смысле этого слова. Она создает экземпляр каждого из трех классов - соответствующую сопрограмму и продолжает одну из них - контроллер. Классы приведены далее:

```
class CONTROLLER; begin
    integer i;
    detach;
    for i := 1 step 1 until 1000 do resume printer
end CONTROLLER;
class PRINTER; begin
    integer i;
    detach;
    while true do
        for i := 1 step 1 until 8 do begin
            resume producer;
            outreal (producer.last_input);
            resume controller
        end;
        next_line
    end
end PRINTER;
class PRODUCER; begin
    integer i; real last_input, discarded;
    detach;
    while true do begin
        for i := 1 step 1 until 6 do begin
            last_input := inreal; resume printer
        end;
        discarded := inreal
    end
end PRODUCER;
```

Тело каждого класса начинается с detach, что позволяет главной программе продолжать инициализацию других сопрограмм. Функция inreal возвращает число, прочитанное из входного потока, процедура outreal его печатает, процедура next_line обеспечивает переход на следующую строку ввода.

Сопрограммы хорошо соответствуют другим понятиям ОО-построения ПО. Заметим, насколько децентрализована приведенная схема: каждый процесс занимается своим делом, вмешательство других ограничено. Producer заботится о создании элементов ввода, printer - о выводе, controller - о том, когда начинать и заканчивать. Как обычно, хорошей проверкой качества решения является простота расширения и модификации; здесь явно надо добавить сопрограмму, проверяющую конец ввода (как просит одно из упражнений). Сопрограммы расширяют децентрализацию еще на один шаг, что является признаком хорошей ОО-архитектуры.

Архитектуру можно сделать еще более децентрализованной. В частности, процессы в описанной структуре должны все же активизировать друг друга по имени. В идеале им не нужно ничего знать друг о друге, кроме передаваемой информации (например, принтер получает last_input от producer). Примитивы моделирования, изучаемые далее, позволяют это. После этого решение может использовать полный механизм параллелизма, описанный в одной из лекций. Его независимость от платформы означает, что он будет работать для сопрограмм, так же как истинный параллелизм.

Последовательное выполнение и наследование

Даже если класс Simula не использует механизмы сопрограмм (`detach`, `resume`), он помимо компонентов имеет тело (последовательность инструкций) и может вести себя как процесс в дополнение к своей обычной роли реализации АТД. В сочетании с наследованием это свойство ведет к более простой версии того, что в обсуждении параллелизма называлось **аномалией наследования**. Язык Simula, благодаря ограничениям (наследование единичное, а не множественное; сопрограммы, а не полный параллелизм), способен обеспечить языковое решение проблемы аномалии.

Пусть `body_C` - это последовательность инструкций, объявленная как тело C, а `actual_body_C` - последовательность инструкций, выполняемая при создании каждого экземпляра C. Если у C нет предка, `actual_body_C` - это просто `body_C`. Если у C есть родитель A (один, поскольку наследование одиночное), то `actual_body_C` - по умолчанию имеет вид:

```
actual_body_A; body_C
```

Другими словами, тела предков выполняются в порядке наследования. Но эти действия по умолчанию, возможно, не то, что нужно. Для изменения порядка действий, заданных по умолчанию, Simula предлагает инструкцию `inner`, обозначающую подстановку тела наследника в нужное место тела родителя. Тогда действия по умолчанию эквивалентны тому, что `inner` стоит в конце тела предка. В общем случае тело A выглядит так:

```
instructions_1; inner; instructions_2
```

Тогда, если предположить, что само A не имеет предка, `actual_body_C` имеет вид:

```
instructions_1; body_C; instructions_2
```

Хотя причины введения подобной семантики ясны, соглашение выглядит довольно неуклюже:

- во многих случаях потомкам необходимо создать свои экземпляры не так, как их предкам (вспомните `POLYGON` и `RECTANGLE`);
- тела родителей и потомков, как, например C, становится трудно понять: прочтение `body_C` еще ничего не говорит о том, что будет делаться при выполнении `new`;
- соглашение не переносится естественным образом на множественное наследование (хотя это не прямая забота Simula).

Трудности с `inner` - типичное следствие активности объектов, о чем говорилось при обсуждении параллелизма.

Почти все ОО-языки после Simula отказались от соглашения `inner` и рассматривали инициализацию объекта как процедуру.

Моделирование

Верный своему прошлому язык Simula содержит набор примитивов для моделирования дискретных событий. Конечно, неслучайно, что первый ОО-язык создавался для моделирования внешнего мира. Сила объектного подхода проявляется особенно ярко именно в этой области.

Моделирующее ПО анализирует и предсказывает поведение некоторой внешней системы: линии сборки, химической реакции, компьютерной операционной системы.

Особенностью **моделирования дискретных событий** является то, что внешняя система представлена своими **состояниями**, способными изменяться в ответ на **события**, происходящие в дискретные моменты времени. При **непрерывном** моделировании жизнь системы рассматривается как непрерывный процесс, как непрерывно развивающееся состояние. Какой из подходов является лучшим для данной внешней системы, зависит не столько от природы системы - непрерывной или дискретной (часто такая постановка бессмысленна), сколько от моделей, для нее создаваемых.

Еще одним конкурентом моделирования дискретных событий является **аналитическое** моделирование, где строится математическая модель внешней системы, а затем решаются соответствующие уравнения. При моделировании дискретных событий для предсказания поведения системы на сколько-нибудь значимом периоде времени приходится увеличивать время моделирования и время работы программной системы. Аналитические методы позволяют получать решение на любой заданный момент времени и, следовательно, более эффективны. Однако, как правило, физические системы слишком сложны, чтобы можно было построить реалистичную математическую модель, допускающую аналитическое решение. Тогда моделирование остается единственной возможностью.

Многие внешние системы естественно укладываются в схему моделирования дискретных событий. Примером может служить линия сборки, где типичные события могут включать появление на линии новых деталей, рабочих или машин, выполняющих определенную операцию над деталями, снятие с линии готового продукта, сбой, приводящий к остановке. Моделирование можно использовать для нахождения ответов на вопросы о моделируемых физических системах. Сколько времени (в среднем, минимально, максимально, среднее отклонение) потребуется для производства конечного продукта? Как долго данный механизм остается неиспользованным? Каков оптимальный уровень запасов?

Исходные данные для моделирования - это последовательность событий и частота их появлений. Данные можно

получить с помощью измерений, производимых на внешней системе. Зачастую их получают в соответствии с заданными статистическими законами, генерируя соответствующие последовательности случайных чисел.

При моделировании вводится понятие **модельного времени**. Его протекание определяет моменты совершения определенных событий, продолжительность выполнения определенной операции на определенной детали. Модельное время не следует путать со **временем вычисления**. Для моделирующей системы модельное время - это переменная, значение которой в программе дискретно изменяется. Доступный в языке Simula запрос `time` позволяет в период выполнения системы получать текущее время и управлять временными событиями.

Компонент `time` и другие компоненты моделирования содержатся в библиотечном классе `SIMULATION`, он может использоваться как предок любого класса. Будем называть "классом моделирования" любой класс, являющийся потомком `SIMULATION`.

В Simula наследование можно применять к блокам: блок, написанный в форме: С `begin... end` имеет доступ ко всем компонентам, объявленным в классе С. Класс `SIMULATION` часто используется таким образом как родитель всей программы, а не просто класса. Поэтому можно говорить о "моделирующей программе".

Класс `SIMULATION` содержит объявление класса `PROCESS`. (Как уже отмечалось, объявления классов в Simula могут быть вложенными.) Его потомки - классы моделирования - могут объявлять потомков `PROCESS`, их будем называть "классами процессов", а их экземпляры - просто "процессами". Экземпляр `PROCESS` задает при моделировании процесс внешней системы. Наряду с другими свойствами процессы могут объединяться в связанный список (что означает, что `PROCESS` - это потомок некоторого класса Simula, являющегося эквивалентом класса `LINKABLE`). Процесс может находиться в одном из четырех состояний:

- **активный** - выполняемый в данный момент;
- **приостановленный** - ждущий продолжения;
- **бездействующий** - холостой, или не являющийся частью системы;
- **завершенный**.

Любое моделирование (то есть любой экземпляр потомка `SIMULATION`) поддерживает **список событий (event list)**, содержащий **уведомления о событиях (event notices)**. Каждое уведомление - это пара `<process, activation_time>`, где `activation_time` означает время активизации процесса `process`. (Здесь и далее любое упоминание о времени, так же как слова "когда" или "в настоящее время", относится к модельному времени - времени внешней системы, доступному через `time`.) Список событий сортируется по возрастанию `activation_time`; первый процесс активный, все остальные приостановлены. Незавершенные процессы, которых нет в списке, являются бездействующими.

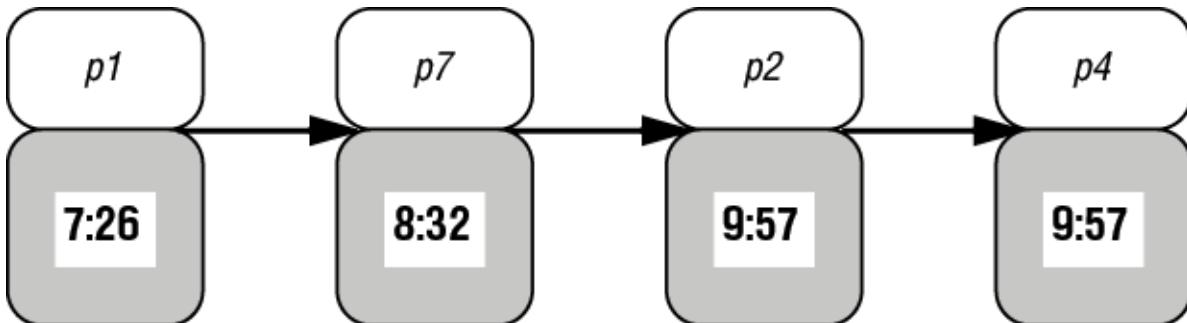


Рис. 17.2. Список событий

Основная операция над процессами - активизация, она планирует активизацию процесса в определенное время, помещая уведомление о событии в список событий. Видимо по синтаксическим причинам эта операция не является вызовом процедуры класса `SIMULATION`, а специальной инструкцией, использующей ключевое слово `activate` или `reactivate`. (Вызов процедуры был бы более согласованным подходом, тем более что фактически стандарт определяет семантику `activate` в процедурных терминах.) Основная форма инструкции такова:

```
activate some_process scheduling_clause
```

где `some_process` - непустая сущность типа `PROCESS`. Необязательный параметр `scheduling_clause` задается одной из следующих форм:

```
at some_time  
delay some_period  
before another_process  
after another_process
```

Первые две формы указывают на позицию нового уведомления о событии, задавая время его активизации, вычисляемое как `max (time, some_time)` для формы `at` и `max (time, time + some_period)` в форме `delay`. Новое уведомление о событии будет внесено в список событий после любого другого события, уже находящегося в перечне с меньшим или таким же временем активизации, если оно не помечено `prior`. Последние две формы определяют позицию по отношению к другому процессу в перечне. Отсутствие `scheduling_clause` эквивалентно `delay 0`.

Процесс может активизировать себя в более позднее время, указав себя как целевой процесс - some_process. В этом случае ключевое слово должно быть reactivate. Это полезно при запуске задачи внешней системы, требующей на свое выполнение некоторого модельного времени. Если запускается задача, решение которой занимает 3 минуты (180 сек.), то для соответствующего исполнителя - процесса worker - можно задать инструкцию:

```
reactivate worker delay 180
```

Эта ситуация настолько типична, что для нее введен специальный синтаксис, позволяющий избежать явного вызова самого себя:

```
hold (180)
```

с точно тем же эффектом.

Вы, вероятно, уже догадались, что процессы реализуются как сопрограммы. Примитивы моделирования внутренне используют рассмотренные выше примитивы сопрограмм. Эффект hold (some_period) можно приблизительно описать (в синтаксисе, похожем на нотацию этой книги, но с расширением resume) как:

```
-- Вставка нового уведомления о событии в список событий в требуемую позицию:  
my_new_time := max (time, time + some_period)  
create my_reactivation_notice.make (Current, my_new_time)  
event_list.put (my_reactivation_notice)  
-- Получить первый элемент списка событий и удалить его:  
next := event_list.first; event_list.remove_first  
-- Активизировать выбранный процесс, изменяя время при необходимости:  
time := time.max (next.when); resume next.what
```

предполагая следующие объявления:

```
my_new_time: REAL; my_reactivation_notice, next: EVENT_NOTICE  
class EVENT_NOTICE creation make feature  
    when: REAL - т.е. время  
    what: PROCESS  
    make (t: REAL; p: PROCESS) is  
        do when := t; what := p end  
    end
```

Если процесс приостанавливается, задавая время своей последующей активизации, то выполнение продолжает приостановленный процесс с наиболее ранним временем активизации. Если указанное время активизации этого процесса позже текущего времени, то соответственно изменяется (увеличивается) текущее время.

Примитивы моделирования, хотя они и основаны на примитивах сопрограмм, принадлежат к более высокому уровню абстракции, потому лучше использовать их, а не полагаться непосредственно на механизмы сопрограмм. В частности, можно рассматривать hold (0) как форму resume, благодаря которой можно не определять явным образом процесс для продолжения, а поручить его выбор механизму списка событий.

Пример моделирования

Классы процессов и примитивы моделирования дают элегантный механизм моделирования процессов внешнего мира. Рассмотрим в качестве иллюстрации исполнителя, которому предлагается выполнять одну из двух задач. Обе требуют некоторого времени; вторая требует включения машины m, работающей 5 минут, и ожидания, пока машина выполнит свою работу.

```
PROCESS class WORKER begin  
    while true do begin  
        "Получить следующую задачу типа i и время ее выполнения d";  
        if i = 2 then  
            activate m delay 300; reactivate this WORKER after m;  
        end;  
        hold (d)  
    end while  
end WORKER
```

Операция "получить тип и продолжительность следующей задачи" обычно получает запрашиваемые величины от генератора псевдослучайных чисел, используя определенное статистическое распределение. Библиотека Simula включает ряд генераторов для типичных законов распределения. Предполагается, что в данном примере m - это экземпляр некоторого класса процесса MACHINE, представляющий поведение машин. Все действующие субъекты моделирования равным образом представляются классами процессов.

Simula: оценка

Как и Algol 60, язык Simula знаменателен не столько своим коммерческим успехом, сколько интеллектуальным

влиянием. Это очевидно и в теории (абстрактные типы данных), и в практике, где большинство языковых разработок последних двух десятилетий является его потомками - либо детьми, либо внуками его идей. Большой коммерческий успех не пришел по ряду причин, но самая важная и очевидная, заслуживающая лишь сожаления, состоит в том что, как и многие значительные изобретения до него, Simula опередил свое время. Хотя многие сразу увидели потенциальную ценность его идей, в целом программистское сообщество не было к нему готово.

Спустя тридцать лет, как ясно из предыдущего описания, многие идеи языка все еще остаются актуальными.

Smalltalk

Идеи языка Smalltalk были заложены в 1970 г. Аланом Кейем в Университете Юты, в то время его выпускником и членом группы, занимающейся графикой. Алана попросили познакомиться с компилятором с расширением языка Algol 60, только что доставленного из Норвегии. (Расширением языка Algol был, конечно, язык Simula.) Изучая его, он понял, что компилятор в действительности выходит за пределы Algol и реализует ряд идей, непосредственно относящихся к его работе над графикой. Когда Кей позднее стал сотрудником Xerox Palo Alto Research Center - PARC, он заложил те же принципы в основу своего видения современной среды программирования на персональных компьютерах. В первоначальную разработку Smalltalk в центре Xerox PARC также внесли вклад А. Гольдберг и Д. Инголс (Adele Goldberg, Daniel Ingalls).

Smalltalk-72 развился в Smalltalk-76, затем в Smalltalk-80. Были разработаны версии для ряда машин - вначале для Xerox, а затем как промышленные разработки. Сегодня реализации Smalltalk доступны на большинстве известных платформ.

Языковой стиль

Язык Smalltalk сочетает в себе идеи Simula и свободный, бестиповый стиль языка Lisp. Статическая проверка типов не производится, что противоречит подходу, предлагаемому в данной книге. Основное внимание в языке и окружении уделяется динамическому связыванию. Решение о том, можно ли применить подпрограмму к объекту, происходит во время выполнения.

У Smalltalk своя терминология. Подпрограмма называется "методом", применение подпрограммы к объекту называется "посланием сообщения" объекту (чей класс должен находить соответствующий метод для обработки сообщения).

Другой важной чертой отличия стиля Smalltalk от изучаемого в этой книге является отсутствие ясного различия между классами и объектами. В системе Smalltalk все - объекты, включая и сами классы. Класс рассматривается как объект класса более высокого уровня, называемого метаклассом. Это позволяет иерархии классов включать все элементы системы, в корне иерархии находится класс самого высокого уровня, называемый *объект*. Корень поддерева, содержащий только классы, - это метакласс *class*. Аргументация для этого подхода такова:

- Согласованность: все в Smalltalk идет от единого понятия, объекта.
- Эффективность окружения: классы становятся частью контекста во время выполнения, это облегчает разработку символьических отладчиков, браузеров и других инструментов, нуждающихся в доступе к текстам классов во время выполнения.
- Методы класса: можно определить статические методы, применяемые к классу, а не к его экземплярам. Методы класса можно использовать для реализации таких стандартных операций, как *new*, размещающих экземпляры класса.

Наше обсуждение в предыдущих лекциях рассматривало аргументацию в пользу других, статичных подходов, показывая другие способы получения тех же результатов.

Сообщения

Smalltalk определяет три основные формы сообщений (и связанных с ними методов): унарные, бинарные и заданные ключевым словом. **Унарные** сообщения задают вызовы подпрограмм без аргументов:

```
acc1 balance
```

Здесь сообщение *balance* посыпается объекту, связанному с *acc1*. Запись эквивалентна нотации *acc1.balance*, используемой в Simula и в данной книге. Сообщения могут, как в данном случае, возвращать значения. Сообщения с **ключевыми словами** вызывают подпрограммы с аргументами:

```
point1 translateBy: vector1  
window1 moveHor: 5 Vert: -3
```

Заметьте, используется ключевой способ при передаче аргументов. При этом частью установленного стиля Smalltalk является объединение имени вызываемого сообщения и первого аргумента, что порождает такие идентификаторы, как *translateBy* или *moveHor*. Соответствующая запись в Simula или нашей нотации была бы *point1.translate* (*vector1*) и *window1.move* (5, -3).

Бинарные сообщения, похожие на инфиксные функции Ada и нотацию этой книги, служат для примирения подхода "все является объектом" с традиционными арифметическими нотациями. Большинство людей, по крайней мере старшего поколения, изучавших арифметику до объектной технологии, скорее напишут 2+3, чем:

```
2 addMeTo: 3
```

Бинарные сообщения Smalltalk дают первую форму записи как синоним второй. Однако здесь есть заминка: приоритет операций. Выражение $a + b * c$ здесь означает не то, что вы думаете - $(a + b) * c$. Для изменения порядка разработчики могут использовать скобки. Унарные сообщения предшествуют по старшинству бинарным, так что `window1 height + window2 height` имеет ожидаемое значение.

В отличие от Simula и нотации данной книги классы Smalltalk могут экспортить только методы (подпрограммы). Для экспорта атрибутов, являющихся закрытыми, необходимо написать функцию, дающую доступ к их значениям. Типичным примером является:

```
x | |
↑xx
y | |
↑yy
scale: scaleFactor | |
xx <- xx * scaleFactor
yy <- yy * scaleFactor
```

Методы `x` и `y` возвращают значения переменных экземпляров (атрибутов) `xx` и `yy`. Стрелка вверх означает, что следующее выражение - это величина, возвращаемая методом отправителю соответствующего сообщения. Метод `scale` имеет аргумент `scaleFactor`. Вертикальные полосы `| |` будут ограничивать локальные переменные, если они есть.

Наследование - важная часть подхода Smalltalk, но кроме некоторых экспериментальных реализаций, оно ограничивается единичным наследованием. Чтобы дать возможность при переопределении метода вызывать оригинальную версию, Smalltalk позволяет разработчику ссылаться на объект, рассматриваемый как экземпляр класса родителя, посредством имени `super`, как в:

```
aFunction: anArgument | ... |
... super aFunction: anArgument ...
```

Интересно сравнить этот подход с техникой, основанной на `Precursor`, и дублирующим наследованием.

Всякое связывание в Smalltalk - динамическое. В отсутствии статического связывания, ошибки, вызванные пересылкой сообщения объекту, не снабженному соответствующим методом для его обработки, не будут обнаружены компилятором и приведут к сбою во время выполнения.

Динамический контроль типов делает неуместными некоторые концепции, развитые ранее в этой книге: Smalltalk не нуждается в языковой поддержке универсальности, поскольку структуры, подобные стеку, могут содержать элементы любого типа без какой бы то ни было статической проверки согласованности. Ставятся ненужными отложенные подпрограммы, поскольку для вызова `x f` (эквивалент `x . f`) нет статического правила, требующего, чтобы определенный класс обеспечивал метод `f`. Если класс `C` получает сообщение, соответствующее методу, чьи эффективные определения появляются только в потомках `C`, то Smalltalk лишь обеспечит возбуждение ошибки во время выполнения. Например, в классе `FIGURE` можно реализовать `rotate` таким образом:

```
rotate: anAngle around: aPoint | |
self shouldNotImplement
```

Метод `shouldNotImplement` включается в общий класс **объект** и возвращает сообщение об ошибке. Нотация `self` означает текущий объект.

Окружение и производительность

Многие из достоинств Smalltalk пришли из поддерживающей среды программирования. Он одним из первых включил инновационную для того времени интерактивную технику. Многое пришло из других проектов Xerox PARC, разрабатываемых одновременно со Smalltalk. Ставшие теперь обычными окна, значки, соединение текста и графики, выпадающие контекстные меню, использование мыши - все это идет из Palo Alto тех лет. Такие современные инструменты ОО-среды, как браузеры, инспекторы и ОО-отладчики восходят корнями к окружению Smalltalk.

Как и в Simula, все коммерческие реализации языка поддерживают сборку мусора. Smalltalk-80 и последующие реализации признаны за их библиотеки базовых классов, охватывающие важные абстракции, такие как "коллекции" и "словари", ряд графических понятий.

Отсутствие статической типизации оказалось большим препятствием на пути к эффективности систем, разрабатываемых на Smalltalk. Современная среда Smalltalk в отличие от ранних версий предоставляет не только интерпретатор, но и компилятор языка. Однако непредсказуемость типов лишает ряда важнейших оптимизаций, доступных для статически типизированных языков. Неудивительно, что многие проекты, реализованные на Smalltalk, имели проблемы с эффективностью. В действительности, неверное представление о проблемах с эффективностью, характерных для объектной технологии, можно отчасти объяснить практикой Smalltalk.

Smalltalk: оценка

Smalltalk явился инструментом, соединившим интерактивную технику с концепциями объектной технологии, превратив абстрактные объекты Simula в визуальные объекты, вдруг ставшие понятными и привлекательными для публики. Simula повлиял на методологию программирования и произвел впечатление на экспертов языков; Smalltalk со временем знаменитого выпуска **Byte** в августе 1981 г. поразил массы.

Учитывая, насколько идеи Smalltalk современны сегодня, поражаешься коммерческому успеху языка в начале 90-х гг. Этот феномен отчасти можно объяснить двумя независимыми явлениями, оба из которых имеют природу "**от противного**":

- Эффект "испытай следующего в списке". Многие, привлеченные к объектной технологии элегантностью концепций, были разочарованы смешанными подходами, существующими, например, в C++. В поисках лучшего воплощения концепций они часто обращались к подходу, представляющему в компьютерных публикациях как чистый ОО-подход: Smalltalk. Многие разработчики Smalltalk - те, кто "просто говорят нет" С и похожим на С разработкам.
- Упадок Lisp. Долгое время многие компании полагались на варианты Lisp (язык Prolog и другие подходы, основанные на искусственном интеллекте) для проектов, включающих быструю разработку прототипов и проведение экспериментов. Начиная с середины 70-х, однако, Lisp исчез со сцены; Smalltalk естественно занял образовавшуюся пустоту.

Последнее замечание дает хорошее представление о месте Smalltalk. Smalltalk - отличный инструмент для прототипирования и экспериментов, особенно с визуальными интерфейсами (в этом он конкурирует с современными инструментами, такими как Delphi от Borland или Visual Basic от Microsoft). Но он во многом остался в стороне от более поздних разработок в методологии инженерии программ. Об этом свидетельствует отсутствие статической типизации, механизмов утверждений, дисциплинированной обработки исключений, отложенных классов - все это имеет значение для систем, решающих критически важные задачи, или просто любой системы, чье правильное поведение во время выполнения важно для разработавшей ее организации. Остаются и проблемы эффективности.

Урок ясен: было бы неразумным, по моему мнению, сегодня использовать Smalltalk для серьезных разработок.

Расширения Lisp

Как и многие необъектные языки, Lisp послужил основой для нескольких ОО-расширений. После Simula и Smalltalk многие ОО-языки строились на основе Lisp или по его подобию. Это неудивительно, поскольку Lisp и его реализации долгое время предлагали механизмы, непосредственно помогающие в реализации ОО-концепций, и отсутствующие в языках и окружениях, относящихся к основному направлению развития программирования. К таким механизмам можно отнести:

- высоко динамичный подход к созданию объектов;
- автоматическое управление памятью со сборкой мусора;
- доступная реализация древовидных структур данных;
- среды разработки с широкими возможностями, как, например, Interlisp в 70-х гг. и его предшественники в предыдущее десятилетие;
- выбор операций во время выполнения, облегчающих реализацию динамического связывания.

Концептуальный путь от Lisp к ОО-языку короче пути, идущему от C, Pascal или Ada. Термин "гибридный", обычно используемый для ОО-расширений этих языков, менее уместен для расширений Lisp.

Приложения искусственного интеллекта - главная область применения Lisp, Prolog и подобных языков, нашли в ОО-концепциях преимущества гибкости и масштабируемости. Они используют Lisp-преимущества унифицированного представления программ и данных, расширяя ОО-парадигму такими понятиями, как "протокол мета-объектов" и "вычисляемое отражение". Теперь некоторые ОО-принципы применяются не только к описанию структур времени выполнения (объектов), но также к самой структуре ПО (классам), обобщая Smalltalk-понятие метакласса и продолжая Lisp-традицию самомодифицируемого ПО. Однако для большинства разработчиков эти возможности далеки от потребностей практики. Они плохо сочетаются с подходом программной инженерии, пытающимся разделить статическую и динамическую ипостась ПО.

Три главных соперника соревновались в 80-х гг. за внимание к ним в мире ОО Lisp: Loops, разработанный в Xerox первоначально для среды Interlisp, Flavors, разработанный в MIT, доступный на нескольких Lisp-ориентированных архитектурах, Ceux, разработанный в INRIA. В Loops было введено интересное понятие "программирования, ориентированного на данные", где можно присоединить подпрограмму к элементу данных (такому как атрибут). Выполнение подпрограммы будет инициировано не только явным вызовом, но всякий раз, когда элемент становится доступным или модифицируется. Это открывает путь к вычислению, управляемому событиями, что является дальнейшей ступенью к децентрализации архитектур ПО.

Унификация различных подходов пришла с расширением Common Lisp (Common Lisp Object System или CLOS), ставшим первым ОО-языком, получившим стандарт ANSI.

Расширения С

Трансформацию объектной технологии в 1980-х гг. от привлекательной идеи к промышленной практике во многом можно объяснить появлением и огромным коммерческим успехом языков, добавивших ОО-расширения к стабильному

и широко распространенному языку С. Первой такой попыткой, привлекшей широкое внимание, был язык Objective-C, а самой известной - C++.

Эти расширения отражают два радикально различных подхода к проблеме проектирования "гибридных" языков, называемых так по той причине, что при расширении приходится сочетать ОО-механизмы с механизмами языка, основанного совсем на других принципах. (Примерами других гибридных языков являются Ada 95 и Borland Pascal.) Язык Objective-C при построении объектного расширения иллюстрирует **ортогональный** подход: добавляя ОО-слой к существующему языку, сохраняя при этом обе части как можно более независимыми. Язык C++ иллюстрирует подход **слияния**, сближая, насколько это возможно, концепции. Потенциальные преимущества каждого стиля ясны: ортогональный подход облегчает переход, избегая непредвиденных взаимных влияний, а подход слияния ведет к более согласованному языку.

Фундаментом успеха в обоих случаях был язык С, ставший к тому времени одним из доминирующих языков в промышленности. Призыв к менеджерам был понятен - превратить С-программистов в ОО-разработчиков без особого "культурного" шока. Моделью такого подхода, востребованной Бредом Коксом, была модель препроцессоров С и Fortran, например Ratfor, позволившая познакомить в 70-х гг. часть ПО сообщества с концепциями "структурного программирования", оставаясь в рамках привычного языка.

Objective-C

Созданный в корпорации Stepstone (первоначально Productivity Products International) Бредом Коксом (Brad Cox) язык Objective-C представлял ортогональное дополнение концепций Smalltalk к языку С. Это был базовый язык для рабочей станции и операционной системы NEXTSTEP. Хотя успех C++ отчасти затмил популярность этого языка, Objective-C все же сохранил активное сообщество пользователей.

Как и в Smalltalk, акцент делается на полиморфизм и динамическое связывание, но современные версии Objective-C предлагают статическую типизацию, а некоторых из них и статическое связывание. Вот пример синтаксиса Objective-C:

```
= Proceedings: Publication {id date, place; id articles;
+ new {return [[super new] initialize]}
- initialize {articles = [OrderedCollection new]; return self;}
- add: anArticle {return [contents add: anArticle];}
- remove: anArticle {return [contents remove:anArticle];}
- (int) size {return [contents size];}
=:
```

Класс Proceedings определяется как наследник Publication (Objective-C поддерживает только единичное наследование). Скобки вводят атрибуты ("переменные экземпляра"). Далее описываются подпрограммы; self, как и в Smalltalk, обозначает текущий экземпляр. Имя id обозначает для варианта без статической типизации общий тип всех не-С объектов. Подпрограммы, вводимые знаком +, являются "методами класса". Здесь таким методом является конструктор new. Другие подпрограммы, вводимые знаком -, являются нормальными "методами объектов", посылающими сообщения экземплярам класса.

Objective-C корпорации Stepstone оснащен библиотекой классов, первоначально построенных по образцу аналогов Smalltalk. Для NEXTSTEP также доступны многие другие классы.

C++

Язык C++ создан примерно в 1986 г. Бьерном Страуструпом в AT&T Bell Laboratories (организации, известной помимо других достижений разработкой Unix и С). Он быстро развивался и занял лидирующую позицию в промышленных разработках, стремившихся получить преимущества объектной технологии при сохранении совместимости с языком С. Язык остался почти полностью снизу вверх совместимым (корректная программа на С является в нормальных обстоятельствах корректной программой на C++) .

Первые реализации C++ были простыми препроцессорами, преобразующими ОО-конструкции в обычный С, основываясь на технике, описанной в предыдущей лекции. Современные компиляторы, однако, являются "родными" реализациями C++. Теперь трудно найти компилятор С, становящийся одновременно компилятором C++ при включении специального параметра компиляции "C++ конструкции". Это один из показателей успеха. Компиляторы C++ доступны практически для большинства платформ.

Первоначально C++ представлял улучшенную версию С благодаря конструкции класса и строгой формы типизации. Вот пример класса:

```
class POINT {
    float xx, yy;
public:
    void translate (float, float);
    void rotate (float);
    float x ();
    float y ();
    friend void p_translate (POINT*, float, float);
    friend void p_rotate (POINT*, float);
    friend float p_x (POINT*);
```

```
    friend float p_y (POINT*);  
};
```

Первые четыре подпрограммы задают привычный ОО-интерфейс класса. Как показывает этот пример, объявление класса содержит только заголовки подпрограмм, а не их реализации, определяемые отдельно. В связи с этим возникают вопросы области действия объявлений, важные и для компиляторов, и для читателей.

Другие четыре подпрограммы - это примеры "дружественных" подпрограмм. Это понятие характерно для C++ и дает возможность вызова подпрограмм C++ из нормального кода С. Дружественные подпрограммы нуждаются в дополнительном аргументе, задающем объект, к которому применяется операции. Здесь этот аргумент имеет тип POINT*, означающий указатель на POINT.

C++ предлагает широкий набор мощных механизмов:

- Скрытие информации, включая способность скрывать компоненты от собственных наследников.
- Поддержка наследования. Первоначальные версии поддерживали только единичное наследование, но теперь язык включает множественное наследование. Дублируемое наследование не обладает покомпонентной гибкостью. (В лекции, посвященной множественному наследованию, отмечалась важность этого свойства.) Вместо этого, разделяется или дублируется весь набор методов дублируемых предков.
- По умолчанию предлагается статическое связывание, для динамического связывания функция должна быть определена как виртуальная. Подход C++ к этому вопросу подробно обсуждался.
- Понятие "чистой виртуальной функции" напоминает отложенные методы.
- Введена более строгая типизация, чем в языке С, но все же разрешающая преобразования типа (кастинг).
- Сборка мусора обычно отсутствует (из-за приведений типа и использования указателей для массивов и подобных структур), хотя доступны некоторые инструменты для надлежаще ограниченных программ.
- Из-за отсутствия автоматического управления памятью введено понятие **деструктора** для явного удаления объектов (понятие, дополняющее понятие **конструктора** класса - процедуры создания).
- Обработка исключений не входила в первоначальное определение, но теперь поддерживается большинством компиляторов.
- Введена некоторая форма попытки присваивания - downcasting.
- Введена универсальность - "шаблоны". У них два ограничения: отсутствует ограниченная универсальность, и при конкретизации шаблона велика нагрузка на работу во время компиляции (известная в C++ как проблема).
- Разрешена перегрузка операторов (знаков операций).
- Введена инструкция assert для отладки, но отсутствуют утверждения для поддержки Проектирования по Контракту (предусловия, постусловия, инварианты классов), соединенные с ОО-конструкциями.
- Библиотеки, доступны от различных поставщиков, например библиотека MFC (Microsoft Foundation Classes).

Сложность

Размер C++ значительно вырос в сравнении с первой версией языка, и многие жалуются на его сложность. Для этого есть все основания. Для примера можно привести маленький отрывок из статьи учебного характера признанного авторитета в С и C++, председателя комитета по стандартам С Американского Института Национальных Стандартов (ANSI), автора словаря (Dictionary of Standard C) и нескольких уважаемых книг по C++. (Я надеялся научиться у него разнице между ссылкой и указателем в C++):

Хотя ссылка похожа на указатель, но указатель - это объект, занимающий память и имеющий адрес. Не константные указатели могут также быть применены для указания на различные объекты во время выполнения. С другой стороны, ссылка - это еще одно имя (псевдоним) объекта и сама не занимает памяти. Ее адрес и значение - это адрес и значение объекта, именуемого ею. И хотя вы можете иметь ссылку на указатель, но не можете иметь указатель на ссылку или массив ссылок, или объект некоторого ссылочного типа. Ссылки на тип void также запрещены.

Ссылки и указатель не взаимозаменяемы. Ссылка на int не может, например, быть присвоена указателю на int, и наоборот. Однако ссылка на указатель на int может быть присвоена указателю на int.

Клянусь, что я пытался понять. Я был почти уверен, что уловил суть, хотя, возможно, еще не готов к семестровому экзамену. (Приведите убедительные примеры случаев, когда уместно использовать: (1) только указатель, (2) только ссылку, (3) или то, или другое, (4) ни то, ни другое.) Потом я заметил, что пропустил начало следующего абзаца:

Из всего этого следует, что неясно, почему ссылки на самом деле существуют.

Зашитники C++ несомненно заявят, что большинство пользователей могут игнорировать такие тонкости. Сторонники другой школы считают, что язык программирования, главный инструмент разработчиков ПО, должен основываться на разумном количестве надежных, мощных, полностью понятных концепций. Другими словами, каждый серьезный пользователь должен знать **все** о языке, и доверять **всему**. Но может быть невозможно примирить этот взгляд с самим понятием гибридного языка - понятием, в случае C++ непреодолимо напоминающим транскрипцию Листа восхитительной шубертовской "Фантазии странника", - трудно добавить целый симфонический оркестр и **сохранить звучание фортепиано**.

C++: оценка

Язык C++ мало кого оставляет безразличным. Известный автор Гради Буч называет его в интервью "Geek Chic", "языком моего предпочтения". Зато, по словам Дональда Кнута, Эдсгера Дейкстру **"сама мысль о программировании на C++ сделала бы больным"**.

В данном случае для C++ подходит ответ Юнии, данный Нерону в "Британике" Ж. Расина:

За что такая честь?
За что такой позор?
(пер. Э. Л. Линецкой)

Разочарование C++ является следствием преувеличеннных надежд. Предыдущие обсуждения в данной книге тщательно анализировали некоторые наиболее противоречивые концепции языка - особенно в области типизации, управления памятью, соглашений по наследованию и динамического связывания - и показали возможность лучших решений. Но никто не может критиковать C++, полагая, что это первый и единственный ОО-язык. Язык C++ появился в нужное время и в нужном месте, ему удалось вне всяких сомнений поймать тот особенный момент в истории ПО, когда многие профессионалы и менеджеры были готовы использовать объектную технологию, но **не** были готовы отбросить существующую практику. C++ был почти магическим ответом: языка С достаточно, чтобы еще не напугать менеджеров, ОО уже достаточно, чтобы привлечь передовых специалистов. Уловив это обстоятельство, C++ просто следовал примеру С, который пятнадцать лет назад тоже был продуктом совпадающих возможностей - необходимости переносимого машинно-ориентированного языка, разра ботки Unix, появления персональных компьютеров, и наличия нескольких списанных машин в лаборатории Bell. Заслуга C++ в том, что он способствовал историческому подъему в развитии объектной технологии, представив ее всему сообществу, возможно не принявшему бы эти идеи в менее общепринятое облачении.

Тот факт, что C++ не является идеальным ОО-языком, о чем часто говорят авторы и лекторы и что ясно всем, изучавшим эти концепции, не должен умалять его заслугу. Не следует смотреть на C++, как будто бы ему суждено остаться основным инструментом программистского сообщества и в 21 веке, поскольку тогда он переживет себя. Тем временем C++ восхитительно играет свою роль - роль переходной технологии.

Java

Созданный в корпорации Sun Microsystems, язык Java привлек к себе большое внимание уже в первые месяцы своего появления в начале 1996 г. как способ, помогающий приручить Интернет. Журнал **ComputerWorld** отмечал, что количество упоминаний о Java в прессе в первой половине 1996 г. составляло 4325 (что можно увеличить в 2-3 раза, поскольку имелась в виду только американская пресса). Кстати, для сравнения, Билл Гейтс упоминался только 5076 раз.

Основной вклад Java связан с технологией реализации. Сама идея уже существовала во многих других средах, но теперь была реализована на новом уровне. Текст Java-программ транслируется в специальный **байт-код** - низкоуровневый переносимый интерпретируемый формат. Спецификация этого кода общедоступна и хранится в Интернете. Для большинства платформ разработана **виртуальная Java-машина**, выполняющая интерпретацию программы на байт-коде. Виртуальная машина - это просто программа с доступными версиями для многих различных платформ, свободно загружаемая через Интернет. Это дает возможность почти всем выполнять программы на байт-коде, написанные почти ком угодно. Виртуальная машина поддерживается сетевыми браузерами, распознающими ссылки на программы на байт-коде. Например, ссылки, встроенные в Web-страницу, автоматически загружают программу и тут же выполняют ее.

Взрывной интерес к Интернету дал этой технологии огромный толчок, и корпорация Sun смогла убедить многих других основных игроков производить основанные на ней инструменты. Поскольку байт-код отделен от языка Java, он имеет хорошие шансы стать выходным языком компиляторов независимо от исходного языка. Создатели компиляторов для ОО-расширений Pascal и Ada, а также нотации этой книги сразу увидели возможность разработки ПО, способного работать без всяких изменений, и даже без необходимости перекомпиляции, на различных промышленных платформах.

Java - одна из инновационных разработок, дающая много причин восхищаться ею. Но язык Java не является и вряд ли станет основным языком разработки. Будучи ОО-расширением С, язык не учитывает уроки, уже усвоенные с 1985 г. сообществом C++. Как в самой первой версии C++, в нем нет универсальности, и поддерживается только единичное наследование. Исправление этих упущений в C++ было долгим и болезненным процессом, годами создававшим неразбериху, поскольку компиляторы никогда полностью не поддерживали один и тот же язык, книги никогда не давали точной информации, преподаватели никогда не давали один и тот же материал, и программисты никогда не знали, что обо всем этом думать.

Как и все в мире C++, Java не стоит на месте. Этот язык имеет одно значительное преимущество перед C++: убрав понятие произвольного указателя, особенно для описания массивов, он, наконец, стал поддерживать сборку мусора. В остальном, он, кажется, не обращает внимания на современные идеи программной инженерии: нет поддержки утверждений (более того, Java дошел до устранения скромной инструкции assert С и C++), он лишь частично полагается на проверку типов во время выполнения, сбивает с толку модульная структура с тремя взаимодействующими понятиями (классы, вложенные пакеты, исходные файлы). Затемнен синтаксис, унаследованный от С. В качестве примера приведем несколько строк, взятых из книги по языку, написанной его разработчиками:

```
String [ ] labels = (depth == 0 ? basic : extended);
while ((name = getNextPlayer()) != null) {
```

В них вы видите функции, создающие побочные эффекты: использование присваивания =, конфликтующее с традицией математики, точка с запятой, иногда необходимая, иногда неправомерная, и т. д.

Однако тот факт, что в языке есть невдохновляющие свойства, не умаляет вклада технологии Java в разработку переносимого ПО. Если Java сумеет решить текущие проблемы эффективности, он может стать лучшим приближением к старой мечте компьютерной промышленности - действительно универсальной машине. И эта машина будет основываться на ПО, а не на оборудовании, хотя поговаривают о "чипах Java".

Другие ОО-языки

До сих пор описывались широко известные языки, но не только они привлекали внимание. Отметим еще несколько ОО-языков, каждый из которых заслуживает отдельной лекции. Ссылки на эти языки можно найти в разделе библиографии.

- Oberon - это ОО-последователь Modula-2, созданный Виртом, является частью проекта, включающего среду программирования и поддержку оборудования.
- **Modula-3**, созданный в исследовательской лаборатории Digital Equipment (DEC Research), является модульным языком с типами, похожими на класс, также основанный на Modula-2.
- **Trellis**, тоже созданный в лаборатории DEC Research, был среди первых языков, предлагающих универсальность и множественное наследование.
- **Sather**, частично возникший из концепций первого издания этой книги, в частности, широко использует утверждения. Его версия **pSather** дает интересный механизм параллелизма.
- **Beta** - это прямой потомок Simula, созданный в Скандинавии при сотрудничестве с Нигардом (одним из первых авторов Simula). Он вводит конструкцию pattern для унификации понятий класса, процедуры, функции, типа и сопрограммы.
- **Self** основан не на классах, а на "прототипах", поддерживающая наследование как отношение между объектами, а не типами.
- Ada 95 обсуждался в лекции, посвященной Ada.
- **Borland Pascal** и другие ОО-расширения Pascal упоминались при обсуждении Pascal.

Библиографические замечания

Simula

[Dahl 1966] описывает первую версию Simula, впоследствии ставшую известной как Simula 1. Язык Simula, известный как Simula 67, впервые описан в [Dahl 1970], где за основу принимался Algol 60, и описывались расширения Simula. Одна из лекций в известной книге "Структурное программирование" (авторы: Дал, Дейкстра, Хоар) [Dahl 1972] донесли эти понятия до более широкого круга читателей. Описание языка было пересмотрено в 1984 г., оно включало элементы Algol 60. Официальная ссылка - Шведский национальный стандарт [SIS 1987]. Описание истории Simula, данное его проектировщиками, см. в [Nygaard 1981].

Самая известная книга по Simula - [Birtwistle 1973]. Она остается отличным введением. Более современное издание - [Pooley 1986].

Smalltalk

Ссылки на самые первые версии Smalltalk (72 и 76) см. в [Goldberg 1981] и [Ingalls 1978].

Специальный выпуск Byte, посвященный Smalltalk - [Goldberg 1981] - стал ключевым событием, обратившим внимание на Smalltalk задолго до появления широко доступных поддерживающих сред. Основная ссылка на язык, [Goldberg 1983], служит и как педагогическое описание, и как ссылка. Дополняет ее [Goldberg 1985], описывая среду программирования

Хорошим современным введением и в язык Smalltalk, и в среду VisualWorks служит [Hopkins 1995]; подробности даются в двухтомном [Lalonde 1990-1991].

История изначального влияния Simula на Smalltalk ("Algol компилятор из Норвегии") отражена в интервью Алана Кея в TWA Ambassador (да, в журнале авиалиний), точный номер выпуска забыт - в начале или середине 80-х. Я в долгу перед Бобом Маркусом за то, что он отметил связь между упадком Lisp и возрождением Smalltalk.

Расширения С: Objective-C, C++

Objective-C описан его создателем в статье [Cox 1984] и в книге [Cox 1990] (первое ее издание относится к 1986 г.). Пинсон и Винер написали введение в ОО-концепции, основанные на Objective-C [Pinson 1991].

Есть сотни книг по C++. (См. описание истории языка его создателем в [Stroustrup 1994].) Первая статья была [Stroustrup 1984], она расширена в книгу [Stroustrup 1986], позже переработанную в [Stroustrup 1991], содержащую много учебных примеров и полезной информации. Справочник - [Ellis 1990].

Ян Йонер опубликовал книгу "C++ критика" [Joyner 1996], доступную на нескольких Интернет-сайтах и содержащую подробные сравнения с другими ОО-языками.

Расширения Lisp

Loops: [Bobrow 1982]; Flavors: [Cannon 1980], [Moon 1986]; Ceyx: [Hullot 1984] CLOS: [Paepske 1993].

Java

За несколько месяцев после выпуска Java появилось много книг с его описанием. Книги, написанные его разработчиками, включают: [Arnold 1996] как учебное пособие, [Gosling 1996] как справочник и [Gosling 1996a] с описанием базовых библиотек.

Обсуждение отсутствия в Java утверждений в стиле этой книги (то есть поддерживающих принципы Проектирования по Контракту), проведенном по Usenet в августе 1995, см. <http://java.sun.com/archives/java-interest/0992.html>.

Другие языки

Oberon: [Wirth 1992], [Oberon-Web]. Modula-3: [Harbison 1992], [Modula-3-Web]. Sather: [Sather-Web]. Beta: [Madsen 1993], [Beta-Web]. Self: [Chambers 1991], [Ungar 1992].

Упражнения

У17.1 Остановимся на коротких файлах

Адаптируйте пример сопрограммы Simula (Printer-Controller-Producer), чтобы она останавливалась, если вход исчерпан до получения 1000 элементов выхода. (**Подсказка:** один из возможных приемов - добавить четвертую сопрограмму, "читателя".)

У17.2 Неявный вызов

(Это упражнение связано с концепциями Simula, но можно использовать нотацию, принятую в книге, расширенную примитивами моделирования, описанными в этой лекции.) Перепишите предыдущий пример так, чтобы каждая сопрограмма не нуждалась в `resume` явным образом. Вместо этого объявите классы сопрограммы потомками `PROCESS` и замените явные инструкции `resume` на инструкции `hold (0)`. **Подсказка:** вспомните, что уведомления о событиях с одним и тем же временем активизации появляются в перечне событий в порядке их создания. Свяжите с каждым процессом условие, необходимое для продолжения процесса.

У17.3 Эмулирующие сопрограммы

Придумайте механизм эмуляции сопрограмм в выбранном вами ОО-языке, не обеспечивающем поддержку сопрограмм. Примените ваше решение к примеру, рассматриваемому в У17.1.

Подсказка: напишите процедуру `resume`, реализуемую как цикл, содержащий разбор случаев с отдельной ветвью для каждого `resume`. Для этого упражнения нельзя использовать механизм параллелизма [лекция 12](#), поддерживающий сопрограммы.

У17.4 Моделирование

Напишите классы для моделирования дискретных событий по образцу классов Simula: `SIMULATION`, `EVENT_NOTICE`, `PROCESS`.

Подсказка: можно использовать технику, разработанную для предыдущего упражнения.

У17.5 Ссылка на версию предка

Обсудите заслуги техники super Smalltalk в сравнении с методами, введенными в этой книге, дающими возможность при переопределении использовать первоначальную версию: конструкцию `Precursor` и, когда это уместно, дублируемое наследование.

¹⁾ Qua (лат.) - где, через, с помощью.

Основы объектно-ориентированного проектирования

18. Лекция: Объектно-ориентированная среда

В заключительной части Девятой симфонии Бетховена баритон прерывает поток изумительной инструментальной музыки, призывая нас к чему-то возвышенному: О, друзья! Не эту мелодию! Пусть начнет звучать Более радостная. В предыдущих лекциях был дан обзор некоторых известных средств ОО-разработки. Не принижая их значения, мы завершим обсуждение рассмотрением современного и комплексного подхода (первые три части Девятой симфонии тоже прелестны, хоть в окончании в них и отсутствует). Данная лекция представляет программную среду (ISE EiffelStudio), реализующую принципы, изложенные в данной книге и делающую их непосредственно доступными разработчикам ОО-ПО. Завершенная схема среды будет приведена позже ([рис. 18.2](#)). Некоторые ее важнейшие компоненты помещены на CD, прилагаемом к книге. Цель этого представления состоит в том, чтобы показать, как поддержка среды может сделать ОО-концепции удобными для практического использования. Предосторожение: обсуждаемая среда никоим образом не является совершенной (фактически она все еще развивается). Это просто пример современной ОО-среды. Другие, упомянем, например, Borland Delphi, завоевали широкий и заслуженный успех. Но мы должны изучить одну среду более глубоко, чтобы понять связь между принципами ОО-метода и их ежедневным применением разработчиком у терминала. Автор надеется, что многие концепции будут полезны и для читателей, использующих другие инструментальные средства.

Компоненты среды

Среда¹ объединяет следующие элементы:

- лежащий в основе **метод**: ОО-метод, описанный в этой книге;
- **язык** - нотацию, представленную в этой книге и используемую на этапах анализа, проектирования и реализации;
- набор инструментальных **средств**, необходимых для использования метода и языка: средства компиляции, просмотра, документирования, проектирования;
- **библиотеки** программных компонент повторного использования.

Следующие разделы содержат обзор этих элементов за исключением первого, составляющего предмет данной книги.

Язык

Язык - это нотация, введенная в лекциях 7-18 курса "Основы объектно-ориентированного программирования" и применяемая в ней. Мы по существу полностью ее рассмотрели за исключением нескольких технических деталей, таких как представление специальных символов.

Развитие

Первая реализация языка выпущена в конце 1986 г. Единственный существенный пересмотр (1990 г.) не затронул никаких фундаментальных концепций, но упростило выражение некоторых из них. С тех пор ведется постоянная работа по уточнению и упрощению, затрагивающая только детали. Два недавних расширения связаны с механизмом параллелизма (рассмотренным в [лекции 12](#), где добавлено единственное ключевое слово `separate`) и конструкцией `Precursor` для облегчения переопределения. Стабильность языка, редкое явление в этой области, была для пользователей среды одним из важных преимуществ.

Открытость

Одно из предназначений языка программирования состоит в использовании его для обертывания компонентов, написанных на других языках. Механизм включения внешних элементов с помощью предложения `external` был описан ранее. Библиотека `Cecil` позволяет внешнему ПО использовать ОО-механизмы: создавать экземпляры классов и вызывать компоненты этих объектов через динамическое связывание (конечно, при ограниченном статическом контроле типов).

Особый интерес представляют интерфейсы с языками C и C++. Для C++ доступно средство под названием `Legacy++`, позволяющее на основе существующего класса C++ создать обертывающий класс (wrapper class), автоматически инкапсулирующий все экспортные компоненты оригинала. Это особенно полезно для организаций, которые использовали C++ как первую остановку на пути к ОО и теперь хотят без потерь инвестиций перейти к законченной и систематической форме объектной технологии. Инструментарий `Legacy++` сглаживает такой переход.

Технология компиляции

Первой задачей среды разработки является выполнение ПО.

Требования к компиляции

Технология компиляции разрабатывалась и совершенствовалась многие годы для решения следующих задач:

- **C1** Эффективность генерированного кода должна быть сопоставимой с эффективностью, достигаемой при использовании классического языка типа C. Нет никаких причин платить за ОО-методы снижением производительности.
- **C2** Время перекомпиляции после внесения изменений должно быть коротким. Точнее, оно должно быть пропорционально объему изменений, а не размеру полной системы. Важнейшим требованием для разработчиков, создающих большие системы, является возможность немедленно увидеть результаты сразу после внесения изменений.
- **C3** Третье требование появилось позже, но становится важным для пользователей: поддержка быстрой доставки приложений через Internet для непосредственного выполнения.

Согласовать два первых требования очень трудно. Требование C1 обычно обеспечивается путем экстенсивной оптимизации, в результате приводящей к замедлению перекомпиляции и компоновки. Интерпретирующие среды хорошо соответствуют C2, выполняя ПО "на лету" после минимальной обработки, но приносят в жертву производительность (C1) и статический контроль типов.

Технология тающего льда

Для решения указанных проблем технология компиляции, известная как **Технология тающего льда (Melting Ice Technology)**, использует сочетание дополняющих друг друга методов. Откомпилированную систему называют **замороженной**, уподобляя ее куску льда в морозильной камере. Образно говоря, чтобы начать работу над системой, ее нужно достать из холодильника и немножко подогреть. Растворившиеся элементы представляют собой изменения. Эти элементы **не станут** причиной цикла "перекомпиляция - сборка" для удовлетворения требования C2. Вместо этого "растаявший" код будет непосредственно обрабатываться исполняющей машиной, встроенной в окружение.

Подобная технология для разработчиков компилятора сложна тем, что нужно обеспечить возможность совместной работы различных компонентов. Сможет ли замороженный код вызывать растворившиеся элементы, ведь при замораживании не было известно, что они позже растают! Но, если ее реализовать, то результат стоит того:

- Быстрая перекомпиляция. Типичное время ожидания - несколько секунд.

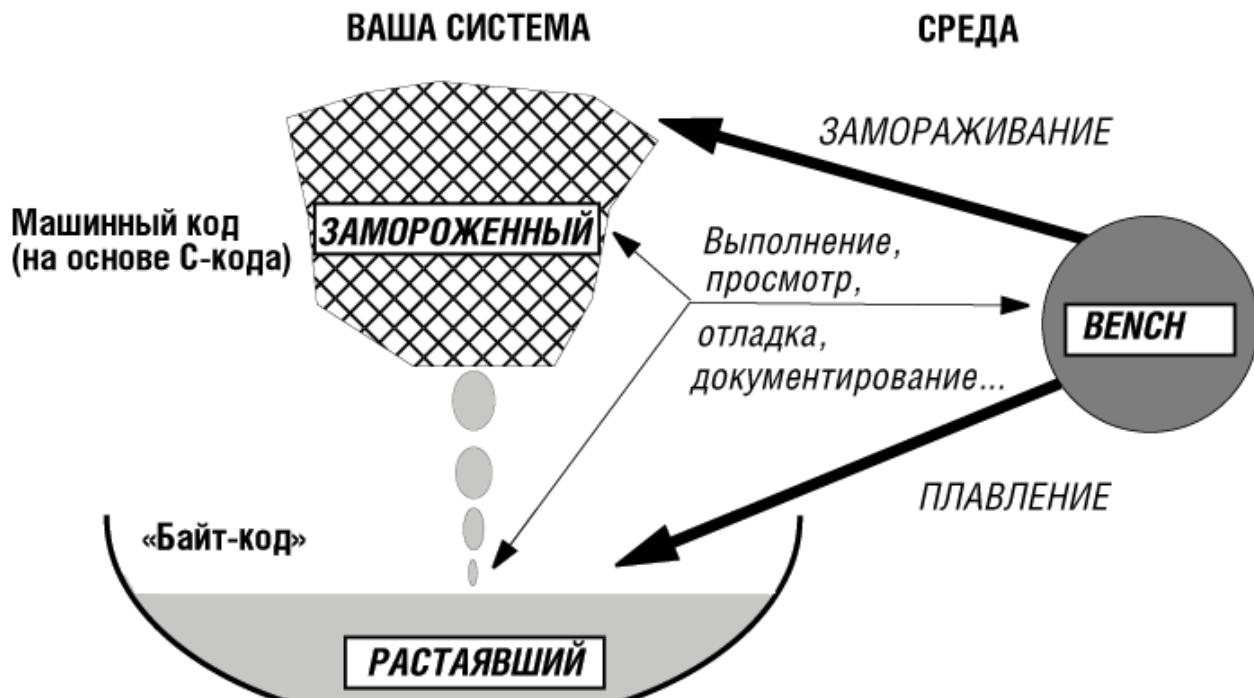


Рис. 18.1. "Замороженная" и "растаявшая" части системы

- Это по-прежнему компиляционный подход: при любой перекомпиляции выполняется полный контроль типов (без чрезмерных временных потерь, потому что проверка, подобно компиляции, является возрастающей - проверке подлежат только изменяемые части кода).
- Скорость выполнения остается приемлемой, потому что для нетривиальной системы типичная модификация затронет лишь небольшую часть кода, которая и будет запускаться на машине выполнения, а все остальное будет выполняться в его откомпилированной форме. (Для максимальной эффективности используется рассмотренная ниже форма компиляции - finalization.)

При увеличении числа изменений доля растаявшего кода будет расти и через некоторое время снижение производительности может стать заметным. Разумно "замораживать" всю систему полностью каждые несколько дней. Поскольку замораживание подразумевает компиляцию и компоновку, типично на это требуется несколько минут (или даже часов после нескольких дней обширных изменений). Можно запустить эту задачу в фоновом режиме или ночью.

Анализ зависимостей

Как это и должно быть в любой современной среде разработки, процесс перекомпиляции является автоматическим. Вы просто нажимаете кнопку Melt в инструментарии Project Tool, описанном ниже, и механизмы тихо определят набор элементов, подлежащих перетрансляции. Нет никакой потребности в файлах Make, а нотация не содержит понятия "include file".

Для выявления фрагментов, требующих перекомпиляции, инструментальные средства среды сначала выясняют, какие сделаны изменения. Изменения могут делаться редактором классов, встроенным в среду, либо обычным внешним текстовым редактором. Поскольку исходный текст класса хранится в отдельном файле, имеющем временную отметку, то она обеспечивает нас нужной информацией. Далее анализируются отношения между классами - клиентские и наследования - для определения того, что еще затронуто и требует перекомпиляции. В случае клиентских отношений скрытие информации позволяет минимизировать подобные действия: если изменения затрагивают только секретные компоненты класса, то его клиенты не нуждаются в перекомпиляции.

Для снижения времени в качестве единицы перекомпиляции выбирается не класс, а отдельная подпрограмма.

Заметим, что если добавлен внешний элемент, например функция C, то потребуется замораживание. Необходимость этого определяется автоматически.

Предкомпиляция

Понимая всю значимость повторного использования, разработчикам дается возможность объединять тщательно отработанные наборы компонентов в библиотеки, откомпилированные раз и навсегда. Другие разработчики будут просто включать их в свои системы, ничего не зная о внутренней организации компонентов.

Эта цель достигается с помощью механизма предкомпиляции набора классов. Такую откомпилированную библиотеку можно с помощью файла Ace включить в новую систему.

В новую систему можно включить неограниченное число откомпилированных библиотек. Механизм объединения таких библиотек поддерживает совместное использование. Если две откомпилированные библиотеки В и С ссылаются на А (например, графическая библиотека Vision и клиент-серверная библиотека Net, обсуждаемые далее, обе используют библиотеку структур данных и фундаментальных алгоритмов Base), то только одна копия А включается в систему.

Автор библиотеки может запретить клиентам доступ к ее исходному тексту (все за и против такой политики обсуждаются в гл. 4). Поскольку используется предкомпиляция, то запрет реализовать просто. В таких случаях пользователи смогут просматривать **краткую форму** и **плоско-краткую форму** классов библиотеки, представляющих интерфейс классов, тогда как полный текст классов останется недоступным.

Удаленное выполнение

Интерпретируемый код, сгенерированный после таяния, традиционно известный как **байт-код**, является независимым от платформы. Для выполнения байт-кода достаточно иметь копию Исполняющей Машины (Execution Engine), известной как ЗЕ и свободно загружаемой через Интернет.

Установка ЗЕ в качестве **дополнительного модуля (plug-in)** Web-браузера дает возможность непосредственного выполнения кода. ЗЕ автоматически выполнит соответствующий код при активизации пользователем гиперссылки, соответствующей байт-коду. Этот механизм удаленного выполнения стал популярен благодаря Java.

Существует два варианта ЗЕ, отличающиеся набором библиотек. Первый предназначен для использования в Интернете и отличается повышенной безопасностью, он допускает только терминальный ввод-вывод. Второй, предназначенный для Инtranет (корпоративных сетей), обеспечивает полноценную поддержку ввода-вывода и ряд других возможностей.

Ведется работа над реализацией средств перевода байт-кода в байт-код Java, что обеспечит дополнительную возможность выполнения на виртуальной машине Java.

Оптимизация

Для максимальной реализации цели С1 одного замораживания кода недостаточно. Для получения законченной, устойчивой системы требуется дополнительная оптимизация:

- **Удаление мертвого кода**, то есть любых подпрограмм, которые никогда не вызываются, прямо или косвенно, из корневой процедуры создания системы. Это особенно важно, если использовано много предкомпилированных библиотек. Выигрыш в размере системы нередко составляет около 50%.
- **Статическое связывание**, автоматически выполняемое компилятором для компонентов, не являющихся полиморфными или не переопределяемых потомками.
- **Подстановка кода** подпрограмм.

Пока в систему еще вносятся изменения, оптимизация не имеет смысла, так как очередное редактирование сводит на нет усилия компилятора. Например, добавление единственного запроса может реанимировать мертвую подпрограмму, а переопределение подпрограммы потребует динамическое, а не статическое связывание. Более того, такая оптимизация может требовать полного прогона системы и, следовательно, невозможна на промежуточных этапах разработки.

В результате, эта оптимизация должна быть частью третьей формы компиляции - **заключительной (finalization)**, дополняя две другие (оттаивание и замораживание). Для большой системы заключительная компиляция может продолжаться несколько часов, но в результате не остается ни одного неперевернутого камня, удаляется все лишнее и ускоряется все, что не оптимально. Результат - максимальная эффективность исполняемого кода системы.

Очевидно, что заключительная компиляция необходима перед поставкой системы или выпуском промежуточной версии. Однако многие руководители проектов любят выполнять эту операцию в конце каждой недели.

Инструментальные средства

На [рис. 18.2](#) представлена общая организация среды. Сама среда, конечно, написана в ОО-нотации (за исключением некоторых элементов системы поддержки выполнения), это делает ее превосходной системой отладки технологии и живым доказательством масштабируемости и реализуемости больших, амбициозных систем. Конечно, мы не хотели бы их разрабатывать иным способом!

Bench и процесс разработки

Центральное место занимает **Bench** - графическое рабочее место для компиляции, просмотра классов и их компонентов, документирования, выполнения, отладки. Разработчик системы постоянно взаимодействует с Bench.

Пока Вы плавите и замораживаете, Вы можете оставаться в Bench. При выполнении заключительной компиляции (она запускается нажатием соответствующей кнопки, хотя для этой операции и многих других доступны и неграфические команды) на выходе формируется программа на С, компилируемая далее в машинный код для соответствующей платформы. Замораживание также использует промежуточный код на С. Использование С имеет несколько преимуществ. Язык С доступен практически на всех plataформах. Низкий уровень языка позволяет писать код, учитывающий возможности реализации на разных plataформах. Компиляторы С производят собственную обширную оптимизацию. Два других преимущества заслуживают особого внимания:

- Благодаря исходному коду на С среду можно использовать для **кросс-платформенной разработки**, компилируя код на другой платформе. Это особенно полезно для создания встроенных систем, для которых типично использование различных платформ для разработки и для выполнения.
- Использование С способствует реализации открытости, обсуждаемой ранее, в особенности упрощаются интерфейсы к программам, написанным на С, C++ и производных от них.

Откомпилированный завершенный С код должен быть скомпонован. На этой стадии используется система поддержки выполнения, набор подпрограмм, обеспечивающих интерфейс с операционной системой: файловый доступ, обработка сигналов, распределение памяти.

В случае кросс-разработки встроенной системы можно обеспечить минимальный размер системы, исключив, например, ввод-вывод.

На [рис. 18.2](#) показана общая схема среды разработки, где в центре находится рабочее место разработчика.

Инструментальные средства высокого уровня

В верхней части [рис. 18.2](#) присутствуют два инструментальных средства высокого уровня.

Build - интерактивный генератор приложений, основанный на модели Контекст-Событие-Команда-Состояние (см. [лекцию 14](#)). Его можно использовать для визуальной разработки графического интерфейса пользователя (GUI) в интерактивном режиме.

Case - средство анализа и проектирования, обеспечивающее возможность рассмотрения системы на высоком уровне абстракции с привлечением графических представлений. В соответствии с принципами бесшовности и обратимости Case позволяет:

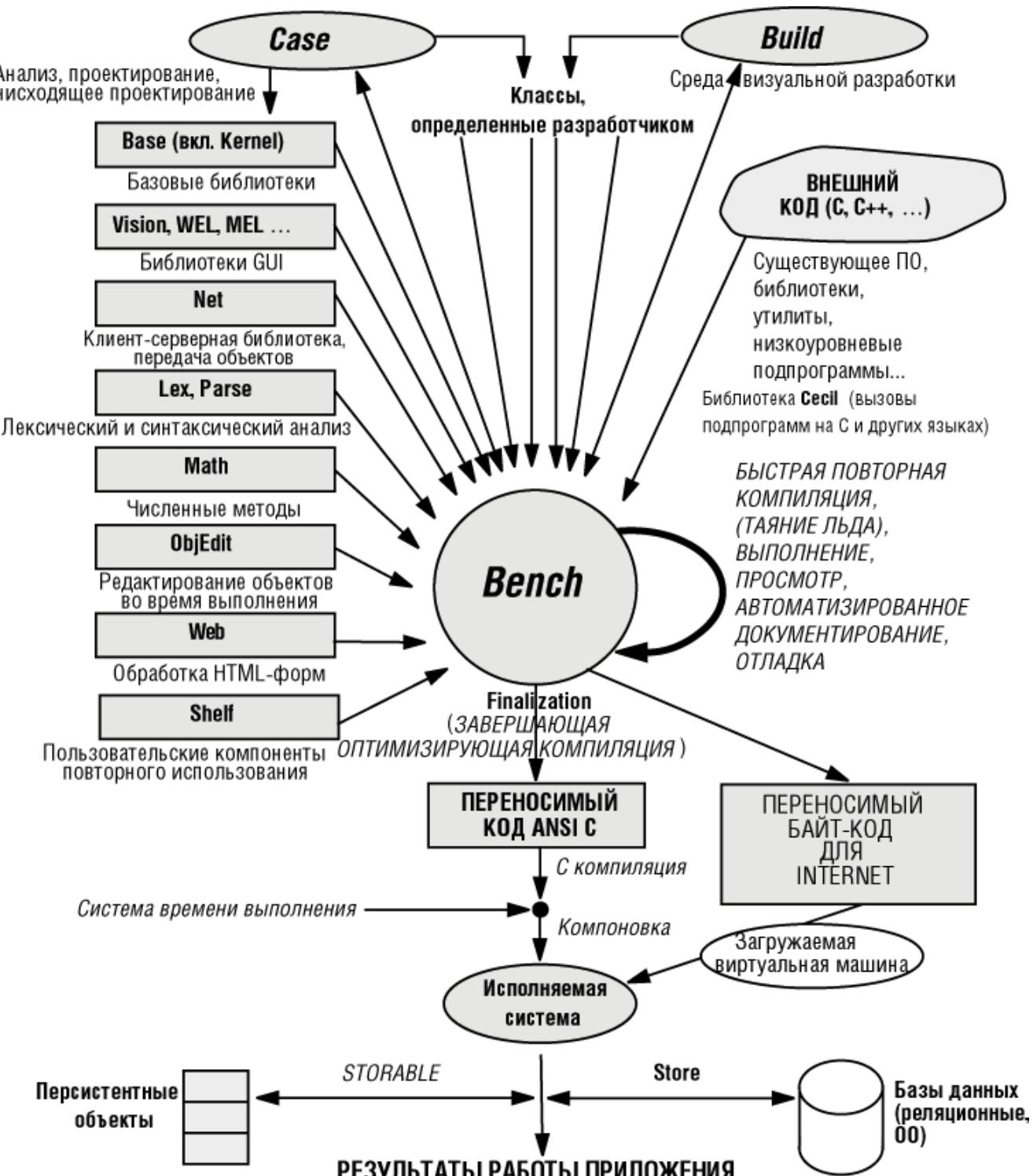


Рис. 18.2. Общая структура среды

- Разрабатывать системные структуры в графической среде, создавая визуальные представления классов ("пузырьки") и определяя их клиентские отношения и отношения наследования с помощью стрелок с последующей группировкой их в кластеры. В конце Case генерирует соответствующие программные тексты (**прямое проектирование - forward engineering**).
- Обработать существующий текст класса и воспроизвести соответствующее графическое представление, облегчая анализ и реструктурирование (**обратное проектирование - reverse engineering**).

Особенно важно убедиться, что разработчики могут свободно переключаться между прямым и обратным проектированием. Вне зависимости от того, в текстовой или в графической форме вносятся изменения, Case обеспечивает механизм согласования, объединяющий эти изменения. В конфликтных ситуациях Case последовательно демонстрирует разработчику конфликтующие версии и предлагает принять решение о том, какая из них будет сохранена. Это ключевой фактор поддержки истинной обратимости, позволяющий разработчикам выбирать на каждом этапе наиболее приемлемый уровень абстракции и переключаться между графической и текстовой нотацией.

Обозначения в Case заимствованы из BON (см. [лекцию 9](#)). BON поддерживает возможность **изменения масштаба (zooming)**. Это существенно для больших систем, разработчики могут работать со всей системой, подсистемой, с одним небольшим кластером, точно выбирая необходимый уровень абстракции.

На [рис. 18.3](#) приведен пример работы с Case, показан кластер описания химического предприятия, свойства одного из его классов (VAT) и свойства одного из компонентов этого класса (fill).

Библиотеки

Перечень библиотек приведен на [рис. 18.2](#). Они играют значительную роль в процессе разработки ПО, обеспечивая разработчиков богатым набором (несколько тысяч классов) компонентов повторного использования. Набор библиотек содержит:

- Библиотеки Base, включающие около 200 классов. Они содержат фундаментальные структуры данных - списки, таблицы, деревья, стеки, очереди, файлы и т. д. Наиболее фундаментальные классы составляют библиотеку Kernel, регулируемую международным стандартом (ELKS).
- Графические библиотеки: Vision для независимого от платформы графического интерфейса пользователя, WEL для Windows, MEL для Motif, PEL для OS/2-Presentation Manager.
- Net для клиент-серверных разработок позволяет передавать по сети объекты произвольной сложности. Платформы могут быть одинаковыми или разными (использование `independent_store` делает формат независимым от платформы).
- Lex, Parse для анализа языка. Parse, в частности, обеспечивает интересный подход к синтаксическому анализу, основанному на последовательном приложении ОО-концепций к синтаксическому анализу (каждое правило моделируется с помощью класса, см. библиографию). Общедоступное средство YOCCC служит препроцессором для Parse.
- Math - библиотека, обеспечивающая ОО-представление фундаментальных численных методов. Она основана на библиотеке NAG и охватывает большой набор средств. Некоторые из ее концепций были представлены в [лекции 13](#) курса "Основы объектно-ориентированного программирования", как пример ОО-изменения архитектуры не ОО-механизмов.

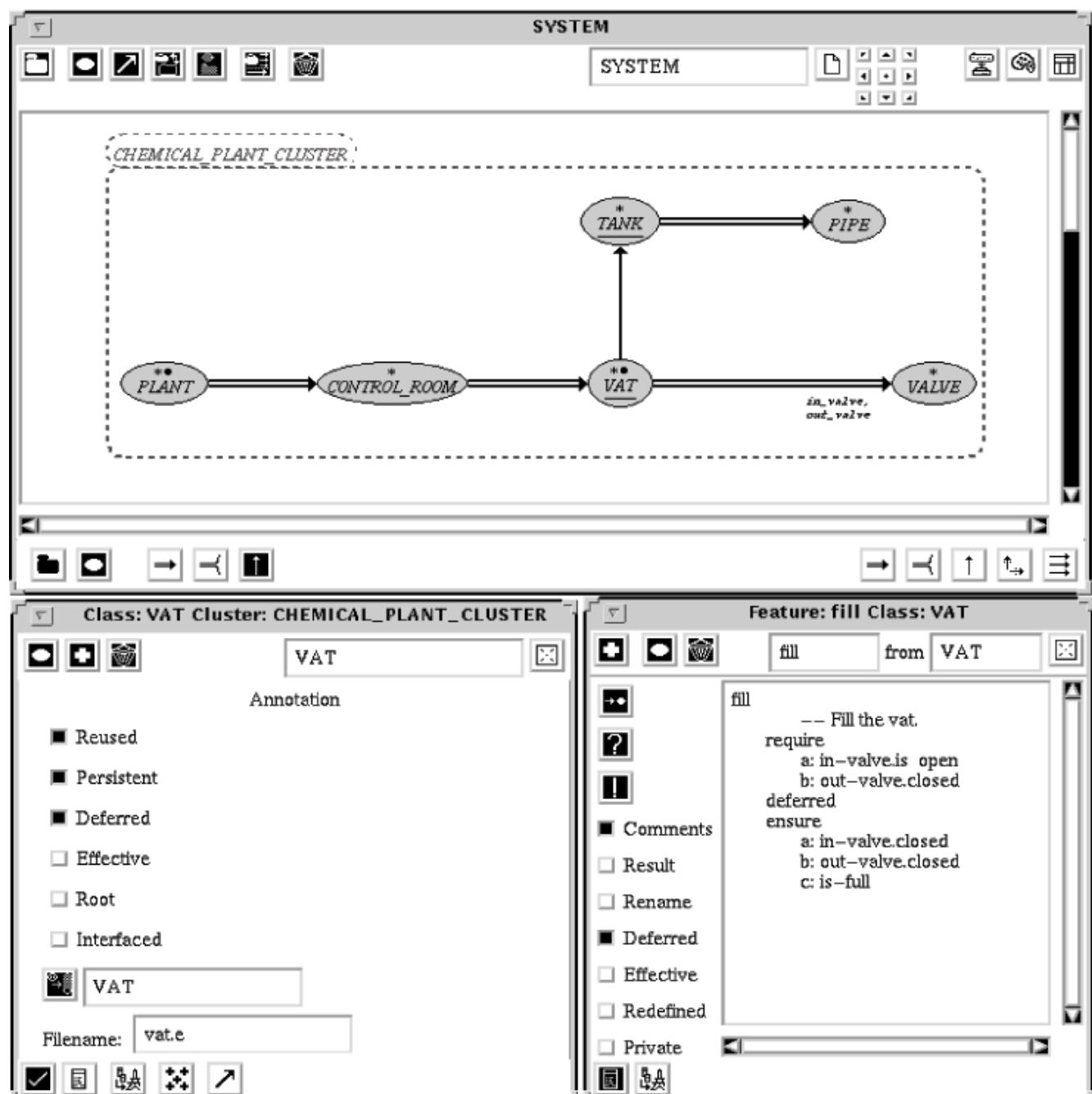


Рис. 18.3. Работа с кластером, классом и компонентом в Case (Вариант для Sun Sparcstation с Motif, доступны версии для Windows и других операционных систем)

- ObjEdit обеспечивает возможность редактирования объектов в интерактивном режиме в процессе выполнения.

- Web поддерживает обработку форм, отправленных клиентами на Web-сервер с целью создания CGI-скриптов.

В нижней части [рис. 18.2](#) показаны библиотеки, используемые для поддержки сохраняемости во время выполнения. Класс STORABLE и дополнительные инструментальные средства, обсуждаемые ранее, поддерживают хранение, поиск и передачу по сети объектных структур в соответствии с принципом Замыкания Сохраняемости. Библиотека Store обеспечивает интерфейс с базами данных, реализуя механизмы доступа и сохранения данных в реляционных (Oracle, Ingres, Sybase) и ОО-базах данных.

Этот список не является исчерпывающим, постоянно разрабатываются новые компоненты. Ряд коммерческих и свободно распространяемых библиотек создан пользователями среды.

Особый интерес представляет совместное использование Net, Vision and Store для формирования клиент-серверных систем. Сервер обеспечивает работу базы данных с помощью Store и выполняет громоздкие вычисления, используя Base, Math и т. д. Тонкие клиенты, использующие Vision (или одну из библиотек для конкретных платформ), обеспечивают практически только интерфейс пользователя.

Реализация интерфейса

Для поддержки обозначенных концепций среда обеспечивает визуальный интерфейс, основанный на анализе потребностей разработчиков и требований различных платформ.

Это лишь краткий обзор наиболее оригинальных аспектов среды. Читателю, знакомому с другими современными средами разработки не составит труда представить себе не описанные здесь средства и возможности. Деталям посвящена обширная литература (см. библиографию).

Платформы

Приведенные иллюстрации получены во время сеанса работы на Sun Sparcstation исключительно по причине удобства. На время написания книги поддерживались другие платформы, включая Windows 95 и Windows NT, Windows 3.1, OS/2, Digital VMS (Alpha и Vax) и все основные версии Unix (SunOS, Solaris, Silicon Graphics, IBM RS/6000, Unixware, Linux, Hewlett-Packard 9000 Series и т. д.).

Хотя общие концепции идентичны для всех платформ и среда поддерживает переносимость исходного кода, точные настройки приспособлены к соглашениям каждой платформы, особенно для Windows, отличающейся собственной культурой.

На [рис. 18.4](#) представлен набор окон среды во время работы. Рисунок черно-белый, но в реальной среде активно используется цвет, особенно для синтаксического выделения различных частей текстов класса. По умолчанию ключевые слова выделены синим цветом, идентификаторы - черным, комментарии - красным. Пользователь может изменить эти настройки.

Инструментальные средства

Среда разработки зачастую состоит из инструментальных средств, построенных на основе функционального подхода, когда каждое средство выполняет определенные функции: просмотр, отладку или форматированную печать исходных текстов. Например, среда Sun Java Workshop (продемонстрированная в сентябре 1996 г.) соответствует этому традиционному образцу, так для поиска предков класса необходимо запустить специальный браузер.

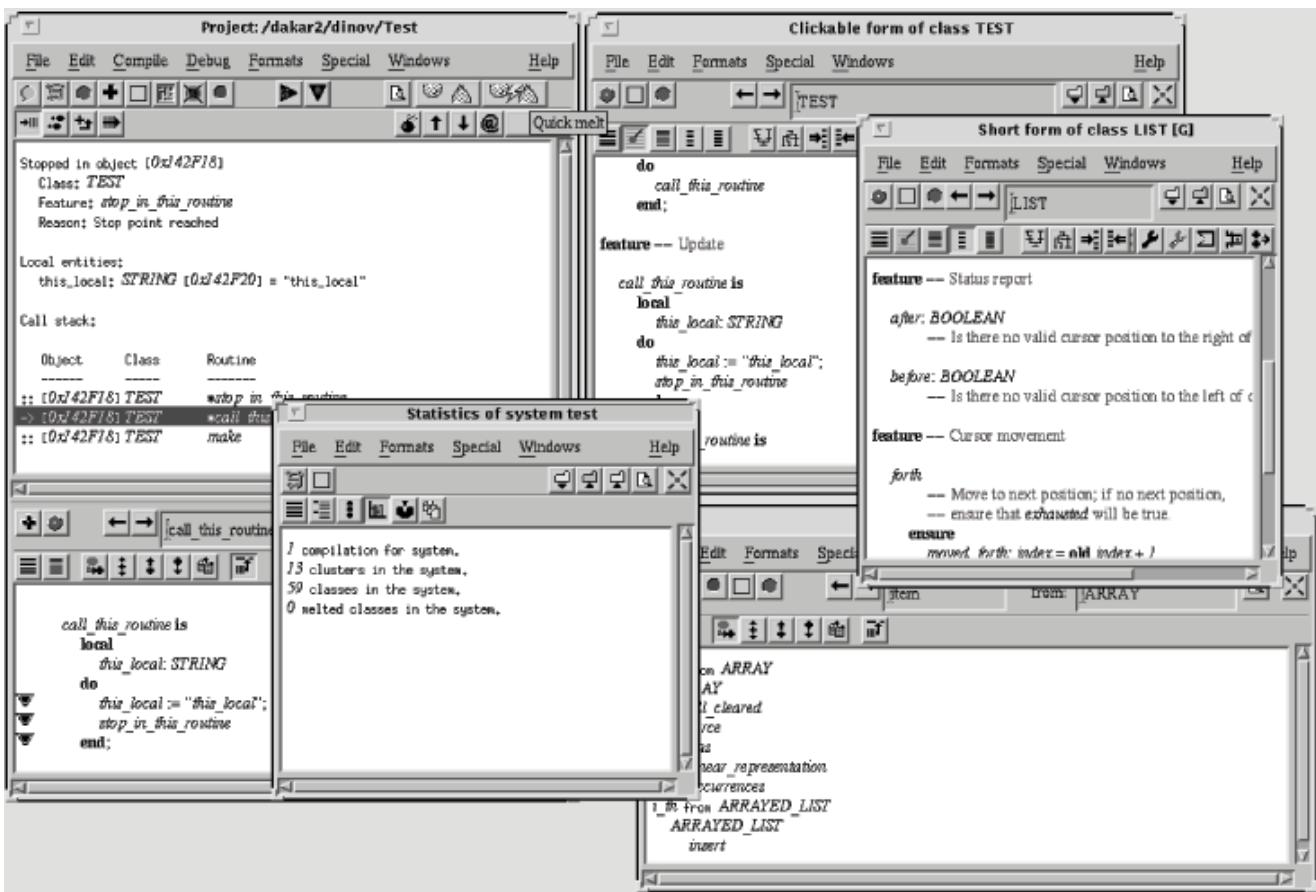


Рис. 18.4. Инstrumentальные средства

Недостаток такого подхода в его модальности. Сначала необходимо выбрать, что Вы хотите сделать, а затем соответствующий инструмент. Практика разработки программного обеспечения выглядит иначе. В течение сеанса отладки, может внезапно потребоваться средство просмотра: например, обнаруживается, что новая версия подпрограммы вызывает ошибки и необходимо посмотреть оригинал. При просмотре оригинала, возможно, захочется посмотреть класс включения, его краткую форму и т.д. Модальные среды не позволяют этого делать: нужно перейти из "отладчика" в "браузер" и опять искать интересующий элемент (подпрограмму) несмотря на то, что он присутствует в другом окне.

Тем же самым способом, каким мы учились доверять типам объектов, а не функциям, описывающим программную архитектуру, можно создавать инструментальные средства в соответствии с используемыми **объектами разработки (development objects)**. Вместо отладчика или окна браузера необходимы Инструмент Класса (Class Tool), Инструмент Компонента (Feature Tool), Системный Инструмент (System Tool), Инструмент Проекта (Project Tool), Объектный Инструмент (Object Tool) в соответствии с абстракциями, используемыми в ОО-разработке: классами, компонентами, системами (наборами классов), проектами и экземплярами класса во время выполнения ("объектами" в строгом смысле).

Project Tool, например, будет полностью следить за проектом. Он используется для выполнения Melt, Freeze или Finalize. На [рис. 18.2](#) показан Project Tool в процессе компиляции, показывающий процент выполненной работы.

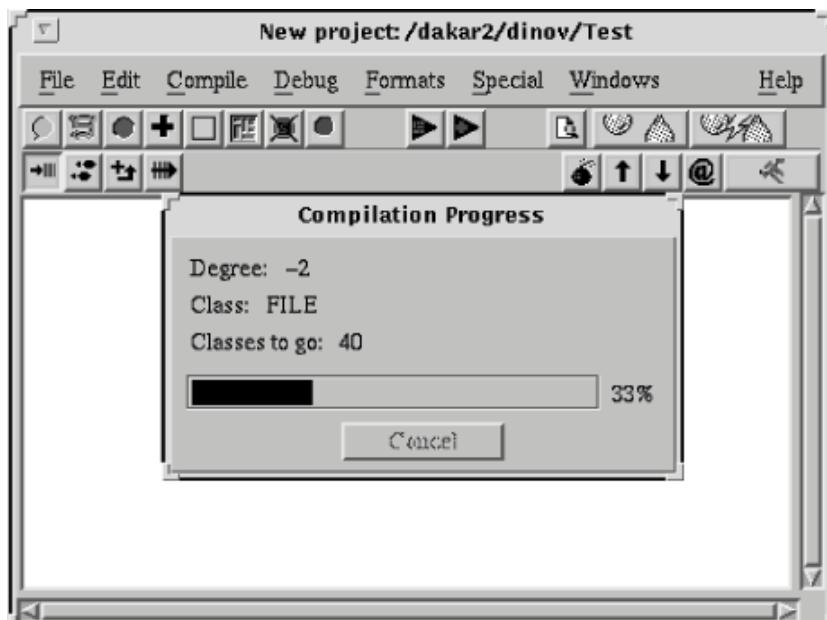


Рис. 18.5. Project Tool в процессе компиляции

Class Tool может быть нацелен на конкретный класс, например, LIST ([рис. 18.6](#)).

```
Clickable form of class LIST [G]

File Edit Formats Special Windows Help
LIST
indexing
description: "Sequential lists, without commitment to a particular representation";
status: "See notice at end of class";
names: list, sequence;
access: index, cursor, membership;
contents: generic;
date: "$Date: $";
revision: "$Revision: $""

deferred class LIST [G]

inherit
CHAIN [G]
redefine
forth
end

feature -- Cursor movement

forth is
-- Move to next position; if no next position,
-- ensure that exhausted will be true.
deferred
ensure
moved forth: index = old index + 1
end;

feature -- Status report

after: BOOLEAN is
```

Рис. 18.6. Class Tool, вид по умолчанию

Feature Tool совместно с Project Tool во время сеанса отладки (рис. 18.7) показывают компонент и ход выполнения с механизмами пошаговой отладки, отображают состояние стека (см. значения локальных сущностей в Project Tool). Feature Tool нацелен на компонент call_this_routine класса TEST.

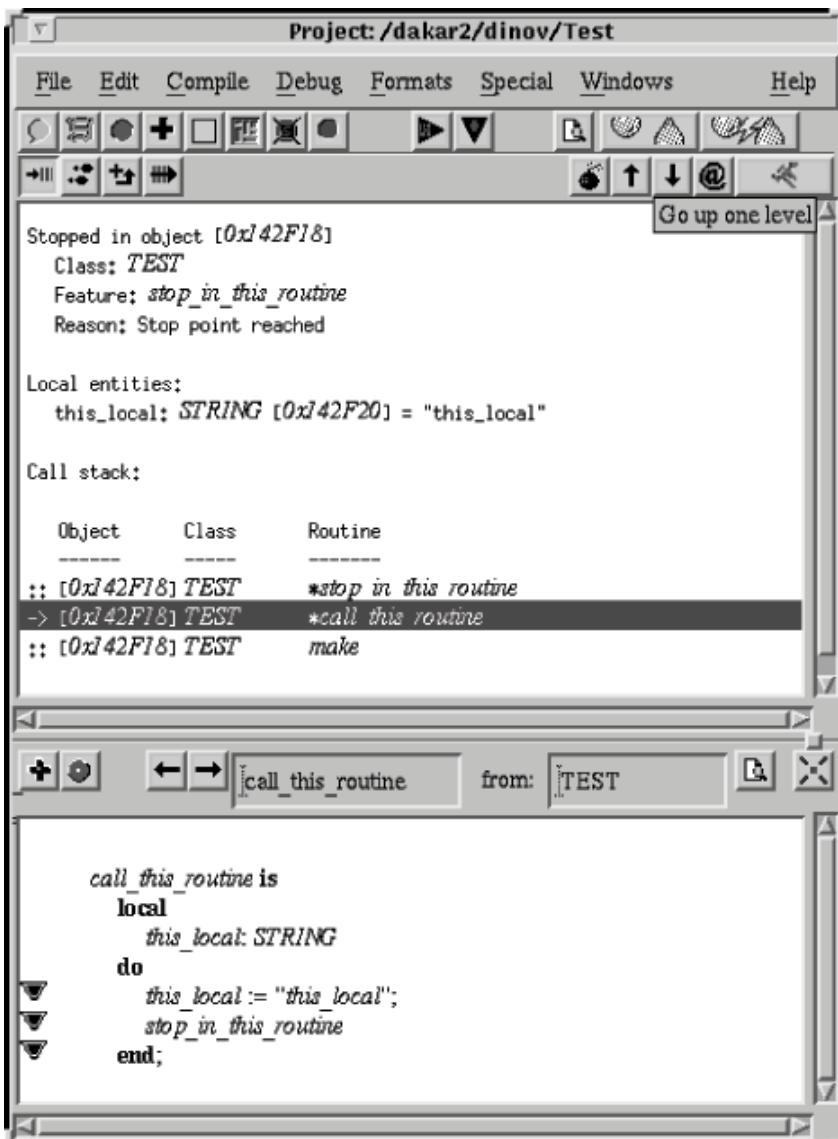


Рис. 18.7. Отладка в Project и Feature Tool

В процессе выполнения можно следить за отдельным объектом с помощью инструментария Object Tool, показанного на [рис. 18.8](#).

В окне отображаются различные поля объектов. Одно из них, *guardian*, содержит непустую ссылку, следуя которой можно просмотреть соответствующий объект, как вскоре будет показано.

Можно использовать столько экземпляров Class Tool, Feature Tool и Object Tool, сколько необходимо, но только один System Tool и один Project Tool доступен в течение сеанса.

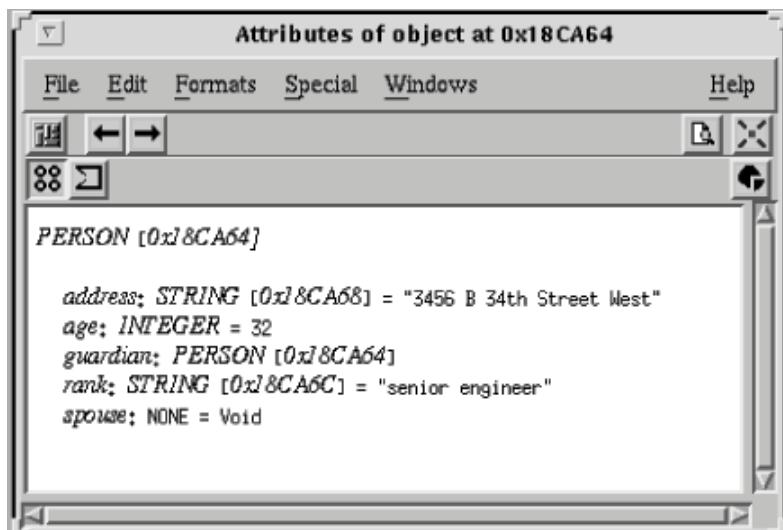


Рис. 18.8. Объект и его поля в процессе выполнения

Перенастройка и просмотр

Существуют разные способы перенастройки инструмента, например перенастройка Class Tool с LIST на ARRAY. Можно просто

ввести новое имя класса в соответствующее поле (если Вы точно его не помните, то можно использовать символ подстановки "*" - ARR* для получения меню со списком соответствующих имен).

Можно также использовать кратко представленный ранее механизм pick-and-throw1 ("выбрать и перетащить") (см. [лекцию 15](#) курса "Основы объектно-ориентированного программирования"). Если щелкнуть правой кнопкой мыши на имени класса, например, на CHAIN в Class Tool настроенном на класс LIST, то курсор превратится в "камешек" (**pebble**) в форме эллипса, показывая, что выбран класс. Далее нужно выбрать "лунку" (**hole**) такой же формы в Class Tool (тот же самол или другом) и положить камешек в лунку, щелкнув правой кнопкой мыши. В качестве лунки может выступать кнопка на панели инструментов или клиентская область окна соответствующего инструментального средства.



Рис. 18.9. Пример pick-and-drop ("выбрать и переложить")

Механизм pick-and-drop - обобщение drag-and-drop. Вместо необходимости постоянно удерживать нажатую кнопку операция разбивается на три шага. Сначала выбирается объект щелчком правой кнопки мыши, появляется камешек. Далее в режиме перетаскивания камешек постоянно связан нитью с выбранным элементом.



([рис. 18.9](#)). Наконец, еще один щелчок правой кнопкой мыши уже в целевой лунке. Можно назвать три преимущества по сравнению с обычным drag-and-drop:

- Нет необходимости постоянно держать нажатой кнопку мыши. При частом выполнении операций drag-and-drop в конце рабочего дня возникает значительная мышечная усталость.
- При ослаблении давления на кнопку на долю секунды операция может завершиться в неправильном месте, часто с неприятными или катастрофическими последствиями. (Это случилось с автором в Windows 95 при перетаскивании значка, представляющего файл. Потом пришлось долго выяснять, что с этим файлом произошло.)
- Обычный drag-and-drop не позволяет отменить операцию! Как только объект выбран, с ним необходимо что-то сделать. В механизме pick-and-drop щелчком левой кнопки операцию можно отменить в любой момент.
- Следует особо отметить, что механизм типизирован, камешек можно положить только в соответствующую лунку. Допускаются некоторые отклонения: аналогично тому, как полиморфизм позволяет присоединить объект RECTANGLE к сущности POLYGON, можно положить компонент в лунку класса и увидеть соответствующий класс с подсвеченным компонентом. Это еще один пример непосредственного применения концепций метода при построении среды. (Здесь различие с механизмами drag-and-drop не является критическим, поскольку они также могут быть ограниченно типизированными.)

Однако все это связано только с интерфейсом пользователя. Более важная роль pick-and-drop проявляется в соединении с

другими механизмами среды - поддержка интегрированного набора механизмов для всех задач разработки ПО. Если вновь обратиться к Class Tool, отображающем отложенный класс LIST библиотеки Base (рис. 18.10), то второй сверху ряд кнопок позволяет выбрать формат вывода. Возможные варианты:

- class text (текст класса) ;



- ancestors (предки) ;



- short form (краткая форма) ;



- routines (подпрограммы) ;



- deferred routines (отложенные подпрограммы)



и так далее. Щелчок на одной из них отобразит текст класса в соответствующем формате. Например, если нажимается кнопка Ancestors (Предки), то Class Tool отобразит структуру наследования (рис. 18.10).

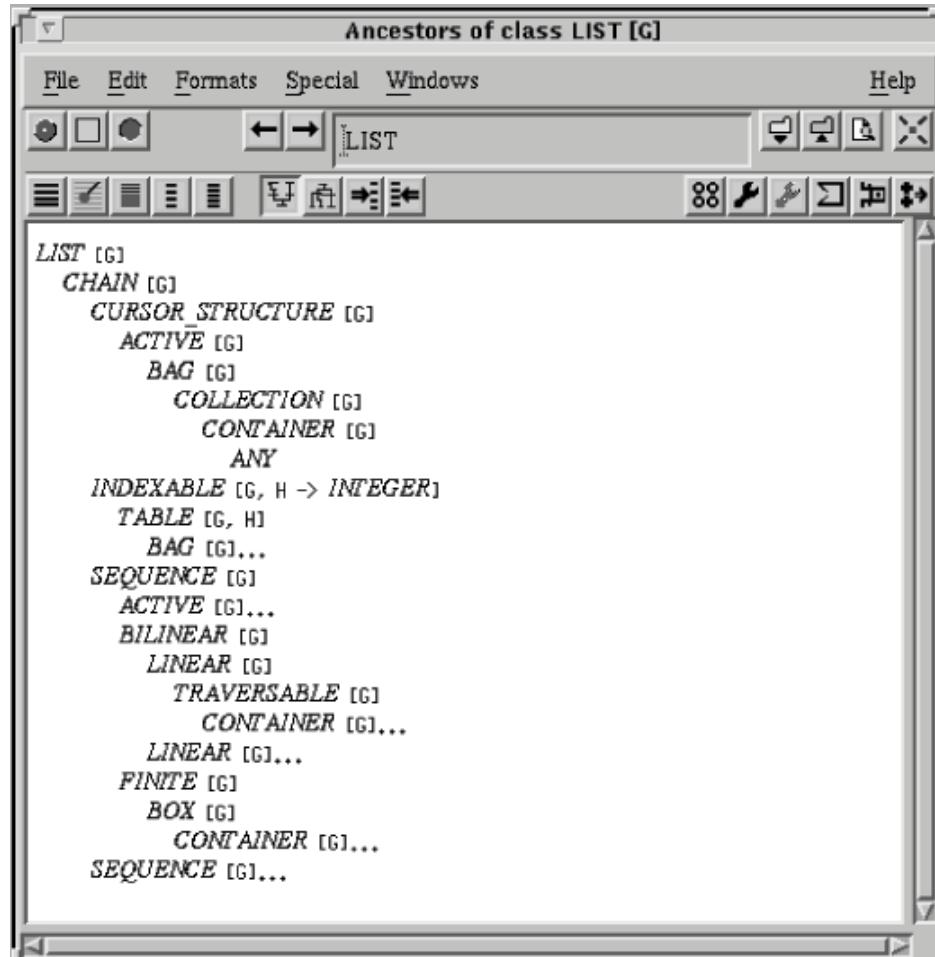


Рис. 18.10. Родословная класса

В любом окне инструментальных средств **все важные элементы интерактивны (clickable)**. Это означает, что для получения информации о классе CURSOR_STRUCTURE достаточно щелкнуть на нем правой кнопкой мыши и использовать pick-and-drop для перенастройки этого или другого инструментального средства на выбранный класс. После этого можно выбрать другой формат, например краткую форму. Далее можно снова применить pick-and-drop и настроить Feature Tool на интересующую Вас подпрограмму. В Feature Tool можно просмотреть предысторию, то есть все приключения компонента в играх наследования: все версии после переименования, переопределения и т. д. Для любого упомянутого класса и компонента можно вновь использовать pick-and-drop.

В процессе сеанса отладки, показанного ранее (рис. 18.7), необходимую информацию можно также получить с помощью pick-and-drop. Щелчок правой кнопкой на объекте 0X142F18 (внутренний идентификатор, сам по себе ничего не говорящий, но интерактивный) позволяет запустить Object Tool, использованный для отображения экземпляра PERSON (рис. 18.8). Этот

инструментарий обеспечит просмотр всех полей и ссылок объекта, также интерактивных. Так можно легко исследовать структуры данных во время выполнения.

Можно осуществить вывод в каждом из доступных форматов (HTML, TEX, RTF, FrameMaker MML, troff), причем компактный язык описаний позволяет определить собственные форматы или модифицировать существующие. Вывод может быть отображен, сохранен с файлами класса или в отдельном каталоге для подготовки документации проекта или кластера.

Механизмы просмотра не делают никаких различий между встроенными библиотеками и классами, определенными разработчиком. Если используется базовый класс INTEGER, то его точно так же можно просматривать в Class Tool в любом доступном формате. Автор библиотеки может закрыть доступ к исходному тексту, но краткая и плоско-краткая формы доступны и остаются интерактивными. Это вполне соответствует общим принципам однородности и бесшовности. В течение всех этапов разработки ПО используются единые концепции, насколько это возможно.

Автор пробовал в вышеупомянутой демонстрационной версии Java Workshop получить информацию о переопределенному компоненте класса, выбранного наугад, но получил сообщение, что браузер не может этого сделать. Оказалось, что это компонент предопределенной графической библиотеки. Для получения информации пришлось с помощью другой программы обратиться к документации, содержащей краткое описание компонента. Просмотр INTEGER был невозможен, поскольку базовые типы Java не являются классами.

Все механизмы времени выполнения, в частности средства отладки (выполнение в пошаговом режиме, точки останова и т. д.), следуют из основных концепций. Например, для добавления точки останова достаточно "переложить" инструкцию или подпрограмму в лунку Stop Point.



Некоторые лунки, известные как "кнопки-лунки" (buttonholes), одновременно выполняют функции кнопки. Например, щелчок левой кнопкой на лунке Stop Point приведет к отображению в Project Tool информации о точках останова. Это представление тоже интерактивно и позволяет легко удалить существующие точки останова или добавить новые.

Систематическое применение этих методов обеспечивает механизм просмотра, при котором все представляющие интерес элементы являются гиперссылками. Такое решение гораздо предпочтительнее модальных сред, постоянно вынуждающих Вас задавать себе вопросы: "Я просматриваю? О нет, я отлаживаю, так что придется запустить браузер. А какой инструментарий нужно запустить для получения документации?"

Нет ни отладки, ни просмотра, ни документирования, ни редактирования. Есть единый процесс создания ПО, и инструментальные средства должны в любой момент обеспечить любые необходимые действия с объектами.

Библиографические замечания

Обзор преимуществ среды дан в [M 1996b], он также доступен в Internet [M-Web] наряду со многими другими техническими документами и описаниями реальных проектов.

В коллективной монографии [M 1993] обсуждается ряд приложений, разработанных в данной среде. Отдельные лекции написаны руководителями соответствующих проектов.

В публикациях [M 1985c], [M 1987b], [M 1987c], [M 1988], [M 1988a], [M 1988d], [M 1988f], [M 1989], [M 1993d], [M 1997] обсуждаются различные аспекты среды на различных стадиях ее развития.

Описание языка дано в [M 1992]. Книга **Reusable Software** [M 1994a] содержит наряду с обсуждением принципов построения библиотек детальное описание библиотек Base.

Другая книга [M 1994] представляет среду в целом. [M. 1995c] описывает анализ Case и среду проектирования, [M. 1995e] - конструктор графических приложений Build [M 1995e]. Принципы построения интерфейса представлены в [M 1993d].

Генератор YOOC разработан Christine Mingins, Jon Avotins, Heinz Schmidt и Glenn Maughan из Monash University [Avotins 1995] и доступен на FTP-сервере Monash University. ОО-методы синтаксического анализа библиотеки Parse представлены в [M 1989d] и [M 1994a].

Библиотека Math разработана Полем Дюбуа и описана в [Dubois 1997].

В разработке среды участвовало много людей. Принципиальный вклад внесли Eric Bezault (я благодарен ему за корректуру этой части книги), Reynald Bouy, Fred Deramat, Fred Dernbach (создатели оригинальной архитектуры текущего компилятора), Sylvain Dufour, Fabrice Franceschi, Dewi Jonker, Patrice Khawam, Vince Kraemer, Philippe Lahire, Frederic Lalanne, Guus Leeuw, Olivier Mallet, Raphael Manfredi (разработка основ системы времени выполнения), Mario Menger, Joost De Moel, David Morgan, Jean-Marc Nerson (особенно начальные версии), Robin van Ommeren, Jean-Pierre Sarkis, Glen Smith, Philippe Stephan (принципы построения интерфейса), Terry Tang, Dino Valente, Xavier Le Vourch, Deniz Yuksel. Невозможно перечислить всех пользователей среды, приславших свои замечания и предложения.

¹⁾ При чтении этой лекции следует учитывать, что за годы, прошедшие после написания этого текста, сама среда (но не язык) существенно изменилась. На диске, прилагаемом к книге, представлена одна из ее последних версий Envision, позволяющая работать на Eiffel в среде Visual Studio.Net. Хотя данный текст описывает устаревшую версию, но общие идеи проектирования подобной среды сохранили свое значение. В предисловии редактора и автора даются ссылки на сайт, позволяющий познакомиться с последними достижениями в этой области.