

Технология

1. Технология структурного программирования. Преимущества и недостатки структурного программирования.
2. Структурное программирование: исходящая разработка, использование базовых логических структур, сквозной структурный контроль.
3. Технология ООП. Преимущества и недостатки ООП.
4. Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта. Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.
5. Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация. Рабочие продукты объектно-ориентированного анализа.
6. Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы.
7. Модель поведения объектов. Жизненный цикл и ДПС. Виды состояний. События, данные событий. Действия состояний. ТПС. Правила переходов.
8. Модель взаимодействия объектов. Диаграмма взаимодействия объектов в подсистеме. Типы событий. Схемы управления. Имитирование. Каналы управления.
9. Диаграмма потоков данных действий (ДПДД). Типы процессов: аксессоры, генераторы событий, преобразования, проверки. Таблица процессов состояний. Модель доступа к объектам.
10. Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.
11. Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.
12. Архитектурный домен. Паттерн КМС. Шаблоны для создания прикладных классов.
13. Структурные паттерны: адаптер, компоновщик, декоратор, заместитель, мост, фасад.
14. Порождающие паттерны: одиночка, фабричный метод, абстрактная фабрика,строитель, прототип, пул объектов.
15. Паттерны поведения: стратегия, шаблонный метод, посетитель, посредник, хранитель, команда.

C++

1. Структура программы на языках С и С++.
2. Классы и объекты в С++. Ограничение доступа к членам класса в С++. Члены класса и объекта. Методы. Схемы наследования.
3. Создание и уничтожение объектов в С++. Конструкторы и деструкторы. Виды конструкторов. Способы создания объектов.
4. Наследование в С++. Построение иерархии классов. Множественное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Неоднозначности при множественном наследовании.
5. Полиморфизм в С++. Понятие абстрактного класса. Дружественные связи.
6. Перегрузка операторов в С++.
7. Шаблоны функций и классов в С++. Специализация шаблонов частичная и полная.
8. Обработка исключительных ситуаций в С++. Пространства имен.

[9. «Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr. Использование weak_ptr на примере паттерна итератор.](#)

1. Технология структурного программирования. Преимущества и недостатки структурного программирования.

Структурное программирование - разбивка по действиям от сложного к простому. В основе структурного программирования лежит алгоритмическая декомпозиция – разбиение задачи на подзадачи по ДЕЙСТВИЮ, отвечая на вопросы «что нужно делать».

Три основные части технологии:

- Нисходящая разработка;
- Сквозной структурный контроль;
- Использование базовых логических структур.

Этапы (нисходящая разработка):

- Анализ (ТЗ, возможность формализации);
- Проектирование (разработка алгоритмов);
- Кодирование;
- Тестирование;
- Сопровождение;
- Модификация

От проектирования до модификации – нисходящий подход в структурном программировании.

Преимущества:

- Легко распределять работу между программистами;
- Естественные контрольные точки;
- Легко выявлять ошибки;
- Легко поддается тестированию (комплексное тестирование);
- Раннее начало процесса кодирования;
- Снижается вероятность допустить логическую ошибку;
- Возможен контакт с заказчиком на ранних стадиях, управление сроками;
- Упрощенное чтение кода;

Недостатки:

- Отсутствие гибкости системы. *После некоторого количества модификаций происходит смещение уровней абстракции, нарушается структура, что приводит к потере надежности (сопровождение затруднено и многое стоит);*
- Сложно изменить формы данных и структур;
- Сложно сопровождать программный продукт;

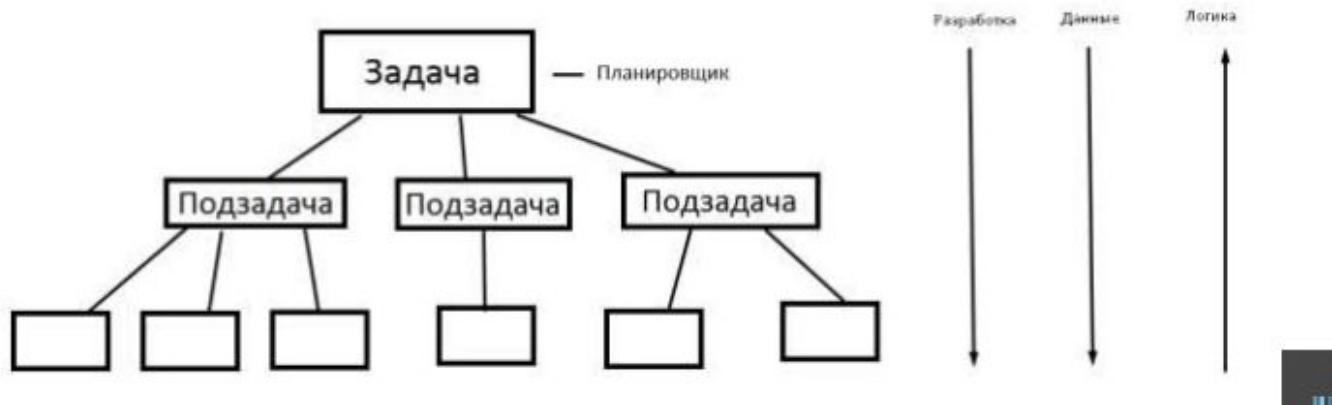
2. Структурное программирование: нисходящая разработка, использование базовых логических структур, сквозной структурный контроль.

1. Нисходящая разработка

Используется на этапах проектирования, кодирования и тестирования.

Разбивка на подзадачи, которые детерминизированы.

Разработка: сверху-вниз, логика: снизу-вверх, данные: сверху-вниз



Этапы создания программного продукта:

1. Анализ. (Оцениваем задачу, переработка ТЗ)
2. проектирование
3. кодирование
4. тестирование
5. сопровождение
6. модификация

2-4 используют нисходящий подход. Используются алгоритмы декомпозиции – разбиение задачи на подзадачи, выделенные подзадачи разбиваются дальше на подзадачи, формируется иерархическая структура (данные нисходящие, логика восходящая, разработка нисходящая). Логика поднимается на более высокий уровень. Данные на низком уровне, на высшем логика. Для каждой полученной подзадачи создаем отладочный модуль. Готовятся тестирующие пакеты (до этапа кодирования). Подзадача не принимает решения за модуль уровнем выше (функция отработала, вернула результат, а потом анализируется). Все данные должны передаваться явно. Блок, функция, файл – уровни абстракции. Ограничения вложенности – 3 (глубина вложенности), если больше то выделить подфункции.

2. Использование базовых логических структур

Майер: Любой алгоритм можно реализовать с помощью трех логических структур: следование, разветвление, ветвление.

Разветвление: выбор между двумя альтернативами, множественный выбор switch – ветви имеют const выражения.

Ветвление: while, until , for, безусловный цикл loop

Алгоритм любой сложности можно реализовать с использованием трёх базовых логических структур: условие (развилка), следование, цикл (повторение). Ввела IBM (*и*) Майер. [GO TO НЕЛЬЗЯ!](#)

Развилка:

if (между двумя), switch (множественный выбор)

Виды циклов:

- Цикл с постусловием do ... while();
- Цикл с предусловием while();
- Цикл со счетчиком for ();
- Безусловный цикл loop(цикл с выходом из тела цикла);

// А теперь я про циклы

Повторение:

- a) Цикл «до» («until») – сделали – проверяем
- b) Цикл «пока» («while»)
- c) Цикл «пока» с переменной («for», «для»)
- d) Безусловный цикл («loop»)

Замечание: выход из цикла должен быть 1!

3. сквозной структурный контроль

Сквозной структурный контроль представляет собой совокупность технологических операций контроля, позволяющих обеспечить как можно более раннее обнаружение ошибок в процессе разработки. Термин «сквозной» в названии отражает выполнение контроля на всех этапах разработки. Термин «структурный» означает наличие четких рекомендаций по выполнению контролирующих операций на каждом этапе.

Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовать специально приглашенные эксперты. Время между сессиями определяет объем материала, который выносится на сессию: при частых сессиях материал рассматриваются небольшими порциями, при редких - существенным фрагментами. Материалы для очередной сессии должны выдаваться участникам заранее, чтобы они могли их обдумать.

3. Технология ООП. Преимущества и недостатки ООП.

Идеи ООП (Хоар, 1966, “Совместное использование кода”):

1. **Инкапсуляция** (объединение данных и действий над ними, или как по лекции Маслова, для каждого типа данных – свои функции-действия);
2. **Наследование** (модификация развития программы за счет надстроек; вместо изменения написанного кода – делаем над ним надстройки);
3. Организация взаимодействия между объектами; перенесение взаимодействия объектов из физического мира в программирование.

Два вида взаимодействия:

- Аксессорное – вступление в контакт, получение информации от объектов (синхронное взаимодействие);
- Событийное взаимодействие – взаимодействие, связанное с изменением состояния объекта (асинхронное взаимодействие);

Преимущества ООП:

- Возможность легкой модификации (при грамотном анализе и проектировании);
- Возможность отката при наличии версий;
- Более легкая расширяемость;

«Более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями. Увеличивается показатель повторного использования кода.

Недостатки ООП:

- Требуется другая квалификация;
- Резко увеличивается время на анализ и проектирование систем;
- Увеличение времени выполнения;
- Размер кода увеличивается;
- Неэффективно с точки зрения памяти (мертвый код - тот, который не используется);
- Сложность распределения работ на начальном этапе;
- Себестоимость больше.

Производительность программ.

4. Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта.

Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.

Инкапсуляция — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя.

Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Объект – конкретная реализация абстрактного типа, обладающая следующими характеристиками: состояние, поведение и индивидуальность.

Состояние – один из возможных вариантов формы объекта.

Поведение – описание объекта в терминах изменения его состояния во время жизни или под воздействием других объектов (на его состояние могут влиять внутренние данные).

Индивидуальность – сущность объекта, отличающая его от других объектов.

Категории объектов:

1. Реальные объекты – абстракция фактического существующего объекта реального мира.
2. Роли – абстракции цели или назначения человека, части оборудования или организации.
3. Инциденты – абстракция чего-то произошедшего или случившегося (наводнение, скачок напряжения, выборы).
4. Взаимодействия – объекты получаемые из отношений между другими объектами (перекресток, договор, взятка).
5. Спецификации – используется для представления правил, критериев качества, стандартов (правила дорожного движения, распорядок дня).

Отношения между объектами:

Отношения использования (*старшинства*) - каждый объект включается в отношения. Может играть 3 роли:

1. Активный объект – объект может воздействовать на другие объекты, но сам не поддается воздействию (воздействующий).
2. Пассивный объект – объект может только подвергаться управлению, но не выступает в роли воздействующего (исполнитель).
3. Посредники – такой объект может выступать в роли воздействующего, так и в роли исполнителя (создаются для помощи воздействующим). Чем больше посредников тем легче модифицировать программу.

Отношения включения – один объект включает другие объекты.

Класс – такая абстракция множества предметов реального мира, что все предметы в этом множестве имеют одни и те же характеристики, все экземпляры подчинены и согласованы с одними и те же набором правил и линией поведения.

Отношения между классами:

Наследование – на основе одного класса, мы строим новый класс, путем добавления новых характеристик и методов.

Использование – один класс вызывает методы другого класса. (акцессор)

Наполнение – это когда один класс содержит другие классы.

Метаклассы – класс существующий для создания других классов.

Домены – отдельный, реальный, гипотетически и абстрактный мир населенный отчетливым набором объектов, которые ведут себя в соответствии с предусмотренным доменом правилами и линиями поведения. Каждый домен образует отдельное и связное единое целое. Класс определяется в одном соответствующем домене. Класс четко определен в одном домене. Классы в одном домене не требуют наличия классов в других доменах.

5. Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация. Рабочие продукты объектно-ориентированного анализа.

Этапы разработки ПО с использованием объектно-ориентированного подхода:

- 1.Анализ – строим модель нашей программы.
- 2.Проектирование – можно полностью автоматизировать.
- 3.Эволюция – процесс создания продукта.
- 4.Модификация – после того, когда мы получаем готовый продукт.

Преимущества эволюции

- 1.пользователю предоставляется обширная обратная связь.
- 2.предоставляются различные версии структур системы (обеспечивает плавный переход от старой системы к новой)
- 3.меньше возможности отмены проекта

Изменения на этапе эволюции

- 1.Добавление класса
- 2.изменение реализации класса
- 3.изменение представления класса
- 4.реорганизация структуры класса
- 5.изменение интерфейса класса

Анализ – построение модели системы.

Действия при анализе и результаты (рабочие продукты):

1. Разбиваем задачу на домены:
 - a. Схема доменов;
 - b. Проектная матрица.
2. Разбиваем домены на подсистемы:
 - a. Модель взаимодействия подсистемы;
 - b. Модель связей подсистемы;
 - c. Модель доступа к подсистемам.
3. Для каждой подсистемы получаем:
 - a. Информационная модель (получаем описание классов и их атрибутов, а также описание их связей);
 - b. Модель взаимодействия объектов (получаем список событий в подсистеме);
 - c. Модель доступа к объектам (получаем таблицу процессов состояний).
4. Для каждого объекта получаем *модель переходов состояний*;
5. Для каждого состояния каждой модели состояния строим *диаграмму потоков данных действий*;
6. Для каждого процесса получаем *описание процесса*.

Не путать домен (мир) с сервисом (функционал)!

На основе полученных в ОOA документов мы приходим к проектированию.

Четыре основных рабочих продукта:

1. Диаграмма класса (проектируется вокруг объекта класса и класса).
2. Схема структуры класса (для внутренней структуры класса).
3. Диаграмма зависимостей – схема использования.
4. Диаграмма наследований – схема наследования классов.

6. Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы.

Концепция:

- Выделение физических объектов
- Короткое описание классов (чтобы установить, является ли объект экземпляром класса)
- Выделение характеристик объектов и идентификаторов (множество из одного или нескольких атрибутов, однозначно определяющих экземпляр класса)
- Графическое обозначение класса на информационной модели:

<номер><имя><ключевой литерал (краткое имя)>
Атрибуты *<> - идентифицирующие (выделяем привилегированный идентификатор) V } - неидентифицирующие

- Строим таблицу атрибутов: для каждого объекта должен быть определен однозначно атрибут (<имя>(<иден.>,<список атрибутов>) - текстовое описание)

Информационное моделирование. Атрибуты класса, связи.

- Выделение сущностей, с которыми мы работаем
- Описание или понятие этой сущности.
 - Выделение атрибутов, каждая характеристика которая является общей для всех возможных экземпляров классов выделяется как отдельный атрибут.
 - Идентификатор – это множество из одного или множества атрибутов, которое определяет класс.
 - Выделение привилегированных атрибутов.
- Графическое представление.
 - Все сущности мы номеруем. Для классов, связей и проч. мы создаём ключевой литерал (1-3 буквы)

Атрибуты класса - содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса.

Типы атрибутов:

- Описательные – факты, внутренне присущие каждому объекту (если меняется, то только какой-то аспект, сам объект остается прежним)
- Указывающие (идентифицирующие) – использующиеся как идентификатор или часть идентификатора (если меняется, то объект сменил имя, но сам остался)
- Вспомогательные – выделяемые из состояния объекта или его отношений, связей с другими объектами (при изменении соответственно меняются связи или состояние объекта)

Если описательный, то описание – информационная строка, которая показывает реальную характеристику, как определяется характеристика и почему она уместна для данного объекта.

Если указывающий, то описание – форма указания, кто назначает атрибуты использующиеся в идентификаторе.

Если вспомогательный, то описание – какое отношение или состояние сохраняются.

Правила:

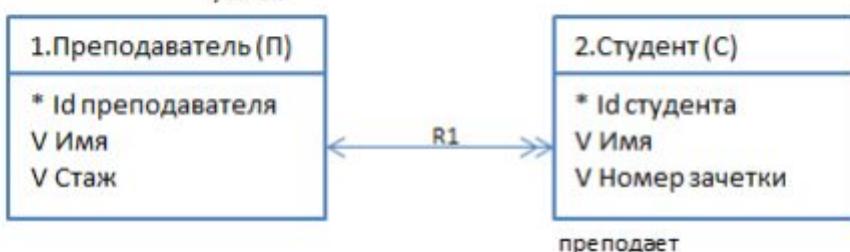
- 1) Каждый объект выделенного класса имеет в каждый момент времени одно единственное значение.
- 2) Атрибут не должен содержать внутренней структуры (она нам не важна, мы рассматриваем атрибут как объект)
- 3) Когда объект имеет составной идентификатор, каждый атрибут, являющийся частью этого идентификатора, представляет характеристику всего объекта, а не части объекта и, тем более, не чего-то другого.
- 4) Каждый атрибут, не являющийся частью идентификатора, представляет характеристику объекта, а не другого атрибута или чего-то другого.

Связи:

Связь – абстракция набора отношений, которые возникают между объектами в реальном мире.

Задаем каждую связь из перспективы участвующих объектов.

Каждой связи присваивается уникальный идентификатор, состоящий из буквы и номера
учится



Типы:

а) один к одному

один ко многим

многие ко многим

б) безусловная (всегда держится)

условная (один из объектов может не участвовать в связи)

биусловная (возможно неучастие с обеих сторон)

Формализация связей:

1) Безусловная один к одному: добавляем атрибут связи в любой из объектов (тот, который более осведомлен о системе)



2) Безусловная один ко многим: формализуется со стороны многих (добавляем им атрибут связи)

3) Многие ко многим: формализуется через ассоциативный объект:



4) Если связь условна, биусловна или имеет динамическое поведение, то она формализуется ассоциативным объектом

Некоторые связи – композиция других связей:



Подтипы и супертипы

Возникают ситуации, когда какие-то объекты имеют данные атрибуты, а какие-то не имеют. Объекты похожи, но отличаются небольшим набором атрибутов. Реализуется выделение супертипа, в который выносятся атрибуты, являющиеся общими для всех объектов. В подтипы добавляем атрибуты, свойственные конкретной группе объектов.



[^] смеситель – состояние, калорифер – произвольная связь.

В ОО анализе есть ограничение – мы не можем создавать объекты супертипа, только объекты подтипов.

7. Модель поведения объектов. Жизненный цикл и ДПС. Виды состояний. События, данные событий. Действия состояний. ТПС. Правила переходов.

Можно чётко выделить сущности, которые появляются, проходят через отчетливые стадии и прекращают существование. Соответственно, можно представить мир через изменение состояния объектов.

1 Многие предметы на протяжение своего существования проходят через отчетливые стадии.

2 Порядок перехода из одной стадии в другую (эволюционирование) формирует характерную черту поведения объекта

3 Реальный объект находится в единственной стадии модели поведения в любой момент времени

4 Предметы эволюционируют от одной стадии к другой скачкообразно

5 В схеме поведения разрешены не все эволюции между стадиями (самолёт стоящий должен вначале взлететь а потом убрать шасси)

6 В реальном мире существуют инциденты, которые заставляют объекты переходить из одной

стадии в другую – эволюционировать

На основе выделяемых состояний объекта строится так называемая модель Мура. Она состоит из множества состояний – для объекта формируется «жизненный цикл» состоящий из состояний и событий, переводящих его из одного состояния в другое. Каждое событие представляет инцидент, указание на то, что должен произойти переход из состояния в состояние. Правила перехода определяют, в какое новое состояние перейдет объект, при возникновении с ним в данном состоянии данного события.

Для правил перехода используется либо **ДиаграммаПереходовСостояний**, либо **Таблица Переходов Состояний**.

Состояние – это положение объектов, в котором определяются определённый набор правил, линий поведений, предписаний, определённых законов. Ставится в соответствие уникальные имя или номер, в соответствии с отношением

Виды состояний:

- Создание
- Заключительное
 - Экземпляр становится неподвижным. Нет перехода в другие состояния.
 - Или объект прекращает своё существование
- Текущее (Состояние в котором объект может находиться, которое не является заключительным или начальным.)

Событие – это абстракция инцидента или сигнала в реальном мире, которое сообщает нам о перемещении чего-либо в новое состояние.

- Значение события - короткая фраза, которая сообщает нам, что происходит с объектом в реальном мире.
- Предназначение - это модель состояний, которое принимает событие, может быть один единственный приёмник, для данного события.
- Метка – уникальная метка должна обеспечиваться для каждого события. Внешние события помечаются буквой «Е»
- Данные события – события переносят данные. Все события, которые переносят объект из одного состояния в другое должны нести его идентификатор.

Каждому событию ставим в соответствие действие. Все события, которые вызывают переход в одно и тоже состояние должны нести одни и те же данные. Идентификатор события, к которому применяется событие, должен переноситься как данные.

Действие – это деятельность или операция, которая выполняется при достижении объектом состояния. Каждому состоянию ставится в соответствие одно действие. Действие должно выполняться любым объектом одинаково.

Действие может:

1. Выполнять любые вычисления
2. Порождать события для любого класса
3. Порождать события для чего-либо вне области анализа
4. Выполнять все действия над таймером
5. Читать, записывать атрибуты собственного класса и других классов
6. Гарантировать, что выполняя любые действия

Действие должно оставлять данные, описывающие собственный объект, непротиворечивыми

9. Диаграмма потоков данных действий (ДПДД). Типы процессов: аксессоры, генераторы событий, преобразования, проверки. Таблица процессов состояний. Модель доступа к объектам.

ДПДД (Диаграмма потоков данных действий) – обеспечивает графическое представление модулей процесса в пределах действия и взаимодействия между ними. Строится для каждого состояния каждого объекта класса.

Разбиваем действия на процессы, которые могут происходить:

Процесс проверки

Процесс преобразования

Аксессоры (процесс, чья единственная цель состоит в том, чтобы получить доступ к данным одного архива данных)

Создание

Чтение

Записи

Уничтожение

Генераторы событий (создает лишь одно событие как вывод)

Процесс может выполняться когда все входы доступны. Выводы доступны когда процесс завершит управления.

Данные событий, архива данных и терминаторов всегда доступны.

На основе выделенных аксессорных процессов строится модель доступа к объектам. На модели доступа, модели состояний (объектов) рисуются вытянутыми овалами. Если А использует аксессор модели состояний В, то рисуется стрелка, А будет аксессором. Аксессоры реализуются добавлением в объект действий по записи и чтению атрибутов. Данная диаграмма представляет синхронное взаимодействие. Может использоваться совместно с асинхронной – не обязательно данные переносят события. Автобусная остановка: событие «пришел автобус», говорит лишь о том что «пришел транспорт», подходящий по функции; а может и нести информацию «пришел автобус №». Если данное не переносится, то с объектом надо вступить в аксессорное взаимодействия.

На основе выделенных аксессорных процессов строится модель доступа к объектам. На модели доступа, модели состояний (объектов) рисуются вытянутыми овалами.

Если А использует аксессор модели состояний В,
то рисуется стрелка, А будет аксессором.

Аксессоры реализуются добавлением в объект действий по записи и чтению атрибутов.

Данная диаграмма представляет синхронное взаимодействие.
Может использоваться совместно с асинхронной – не обязательно
данные переносят события. Автобусная остановка: событие «пришел автобус», говорит лишь
о том что «пришел транспорт», подходящий по функции; а может и нести информацию «пришел
автобус №». Если данное не переносится, то с объектом надо вступить в аксессорное
взаимодействия.



8. Модель взаимодействия объектов. Диаграмма взаимодействия объектов в подсистеме. Типы событий. Схемы управления. Имитирование. Каналы управления.

МВО обеспечивает краткое графическое изложение событий взаимодействия между моделями состояний и внешними сущностями, происходящими в подсистеме

МВО (модель взаимодействия объектов) – графическое представление взаимодействия. Каждая модель состояний – овал. Стрелочки – события.

События, которые приходят в систему – приходят извне, эта внешняя сущность – терминатор.

Типы событий

1. Внешние события (приходят от терминатора)
 - a. Не запрашиваемые события (не являются результатом действия предыдущей действенности подсистемы)
 - b. Запрашиваемые события
2. Внутренние (порождаются какой-либо моделью состояний нашей подсистемы)
 - a. Схема верхнего управления (Те события приходят от терминаторов которые наверху)
 - b. Схема нижнего управления (Объекты которые являются программной реализацией чего-либо существующего)

МВО формируется иерархически–объекты, наиболее осведомленные о всей системе(активные) располагаются вверху диаграммы. Если событие приходит извне к МВО, находящимся вверху, терминаторы рисуются вверху–терминаторы верхнего уровня. Если события уходят или приходят к МВО нижнего уровня, терминаторы рисуются снизу. Может быть схема верхнего и нижнего управления – система ограничена терминаторами сверху или снизу. Надо стремиться к тому, чтобы на верхнем уровне взаимодействие терминатора сводилось к одной модели. С нижним этого ограничения нет – может быть сколько угодно МВО, взаимодействующих с терминаторами нижнего уровня. Как правило на нижнем уровне терминатор (внешняя сущность) может быть физическим объектом, с которым мы работаем. С точки зрения организации МВО, есть более и менее осведомленные объекты; терминаторы в этой иерархии взаимодействуют чётко: верхние терминаторы взаимодействуют с осведомлёнными объектами, нижние с неосведомлёнными. Противных схем лучше избегать – нужно продолжать формализацию, искусственно выделяя объекты меньшей осведомленности.

Канал управления – последовательность событий и действий, происходящих в ответ на поступление некоторого незапрашиваемого события. Если возникло событие к терминатору, влекущее за собой новые события от терминатора, то они тоже включаются в канал управления.

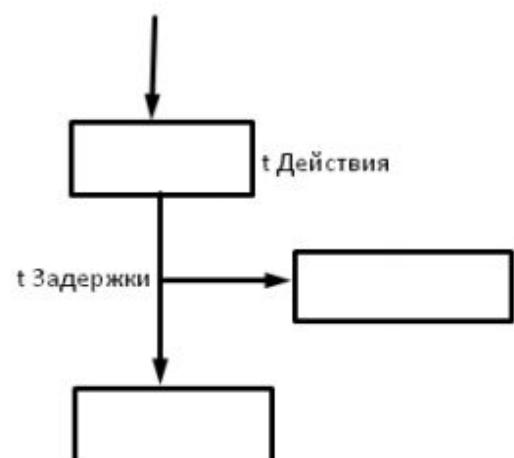
Процесс имитирования: задается некоторое начальное состояние. Генерируем некоторое внешнее событие и смотрим за изменением состояния всех объектов в системе.

Время имитирования:

1. Время выполнения действия;
2. Время задержки – время, в течении которого объект должен находиться в определенном состоянии (резкий переход из состояния в состояние невозможен).

Этапы имитирования:

1. Установка начального состояния;
2. Прием незапрашиваемого события и выполнение канала управления;
3. Оценка конечного результата.



10. Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.

Домен – отдельный реальный или гипотетический населенный отчетливым набором объектов, которые ведут себя в соответствии с присущими домену правилам и линиями поведения.

Типы доменов:

- a) Прикладные (предметная область системы с точки зрения пользователя)
- b) Сервисные (функционал для поддержки прикладного домена)
- c) Архитектурные (обеспечивают единые механизмы управления данными и всей программой как единым целым)
- d) Реализационные (библиотечный класс – взаимодействие по сети, протоколы взаимодействия по сети)

Сервис и домен – разные понятия!

Мост – связь между доменами, когда один домен использует механизмы и возможности другого.

Клиент – использующий возможности домен.

Сервер – домен, предоставляющий свои возможности для использования.

Клиент рассматривает мост как набор предложений, которые будут представлены другим доменом. Сервер подходит к мосту как к набору требований для выполнения.

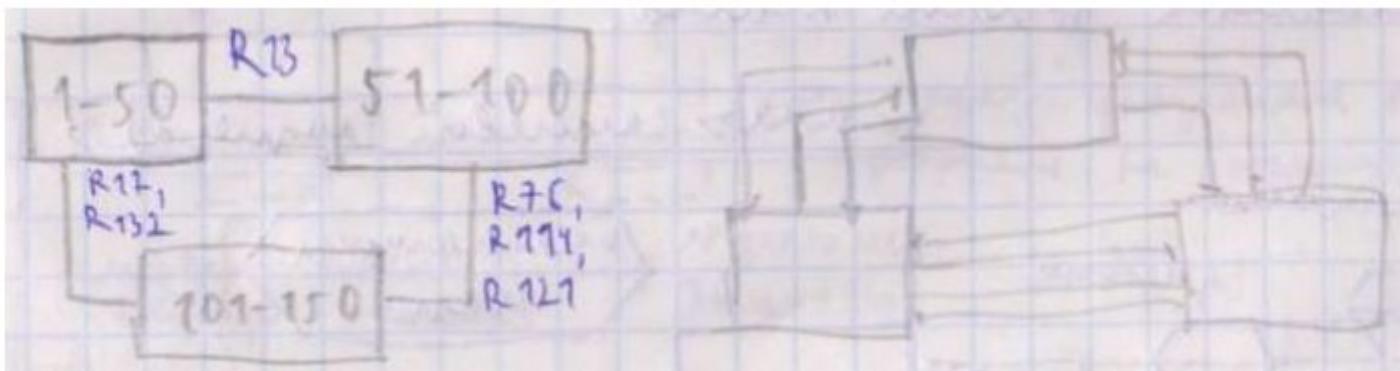
Акти

Схема доменов.

Домены и мости между ними. В овалах – домены. Стрелка – мост. Домен к задаче – внизу, сервисные – внизу.

Строим 3 диаграммы для взаимодействия

- модель связей подсистем (по информационной модели)
- модель взаимодействия подсистем (по МВО)
- модель доступа подсистем (по МДО)
 - Модель доступа к объекту. Стрелкой помечается идентификатор процесса. Модель взаимодействия – асинхронная, событийная модель. Модель доступа – синхронное взаимодействие (один объект может получить данные другого объекта) В итоге получаем модель доступа к объектам, таблица процессов состояний, диаграмма потомков данных действий.



11. Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.

Объектно Ориентированный Дизайн (проектирование)

На основе полученных в ООА документов мы приходим к проектированию – из документов ООА получаем документы ООД.

Одна из нотаций ООД: нотация Буча и Бухра. Четыре основных диаграммы.

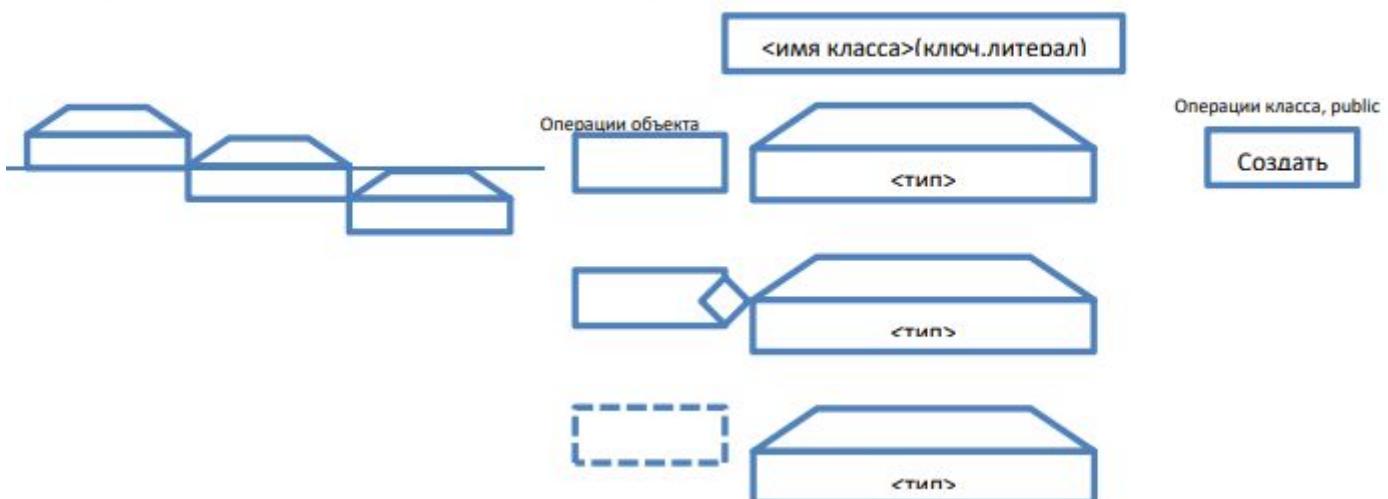
Диаграмма класса – описывает внешнее представление одного класса.

Схема класса – внутреннее представление, структура класса.

Диаграмма зависимостей – схема использования.

Диаграмма наследований – схема наследования классов.

Диаграмма класса: (v – бока, «гробы»)



Слева: принимаемые, возвращаемые, обрабатываемые данные

Ромб – обрабатывает исключительную ситуацию.

Пунктир – отсрочиваемая операция, callback выполняемый в результате события

Схема структуры класса:



Класс реализуется по слоям. Есть публичные методы, есть приватные; доступ к нижним слоям напрямую снаружи недопустим, как и вызов пабликом паблика.

Диаграмма зависимостей: схема использования. Двойная стрелка – дружественная связь.

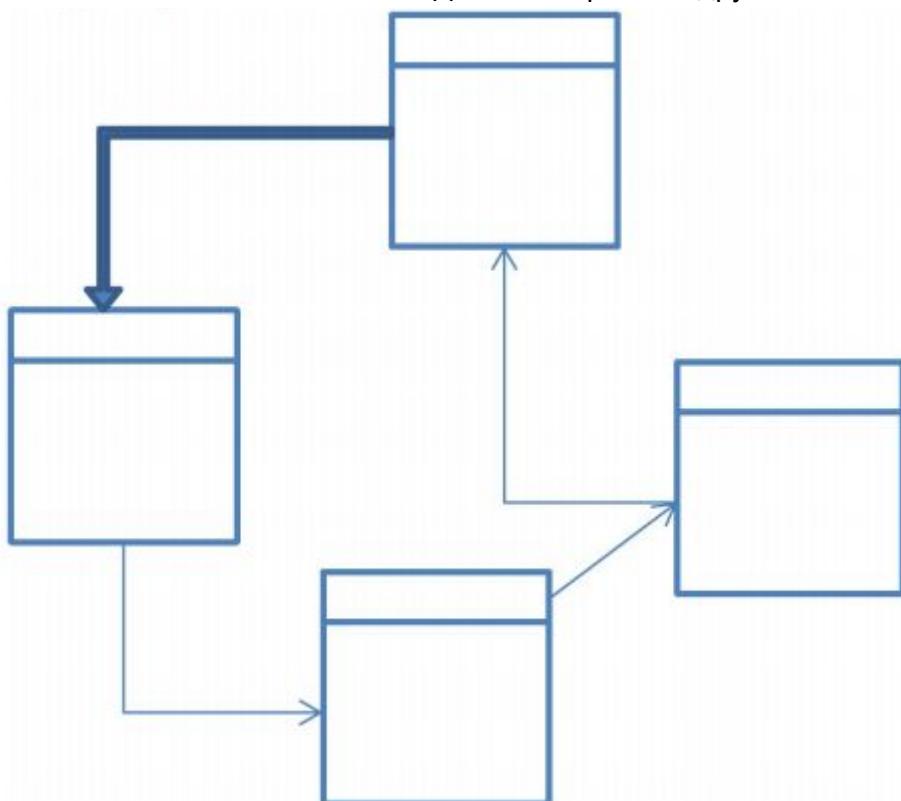
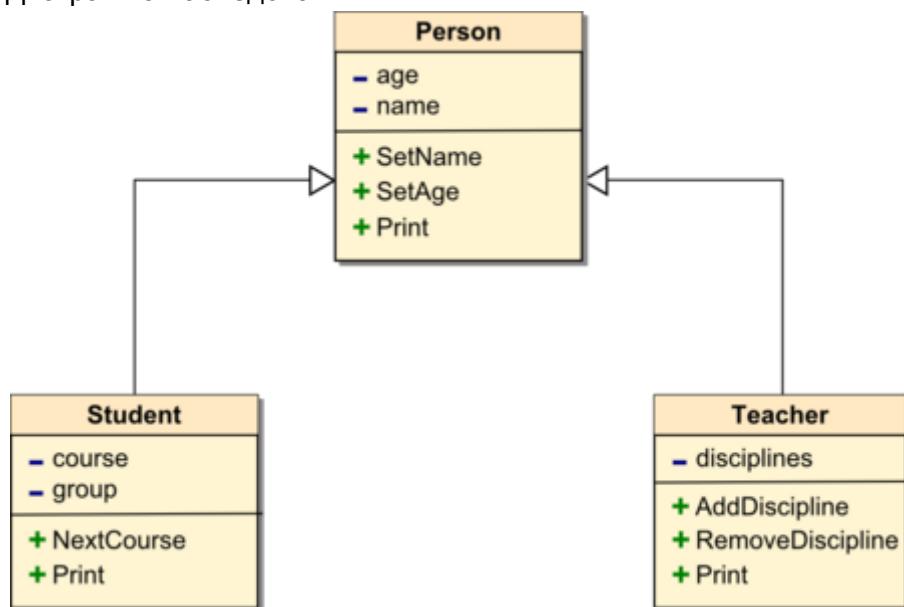


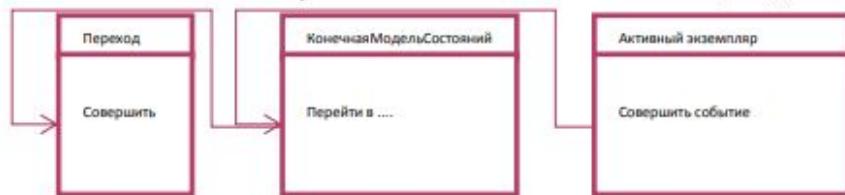
Диаграмма наследования



12. Архитектурный домен. Паттерн КМС. Шаблоны для создания прикладных классов.

Архитектурный домен обеспечивает единые механизмы управления данными и всей программой как единым целым. Архитектурный домен можно реализовать как паттерн КМС.

Создаются несколько классов для архитектурного домена, задача которых – задать правила перехода из состояния в состояние при возникновении событий. Классы, выделяемые для архитектурного домена:



Паттерн КМС берет на себя функцию контроля. То есть уже нет необходимости проверять возможность перехода в данное состояние, как в четвертой лабе.

КМС делается шаблонным классом – позволяет сделать систему гибкой.

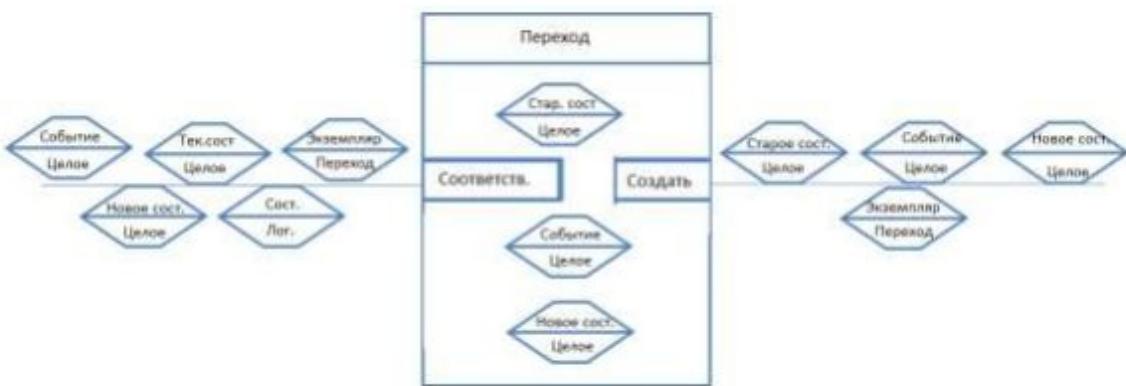
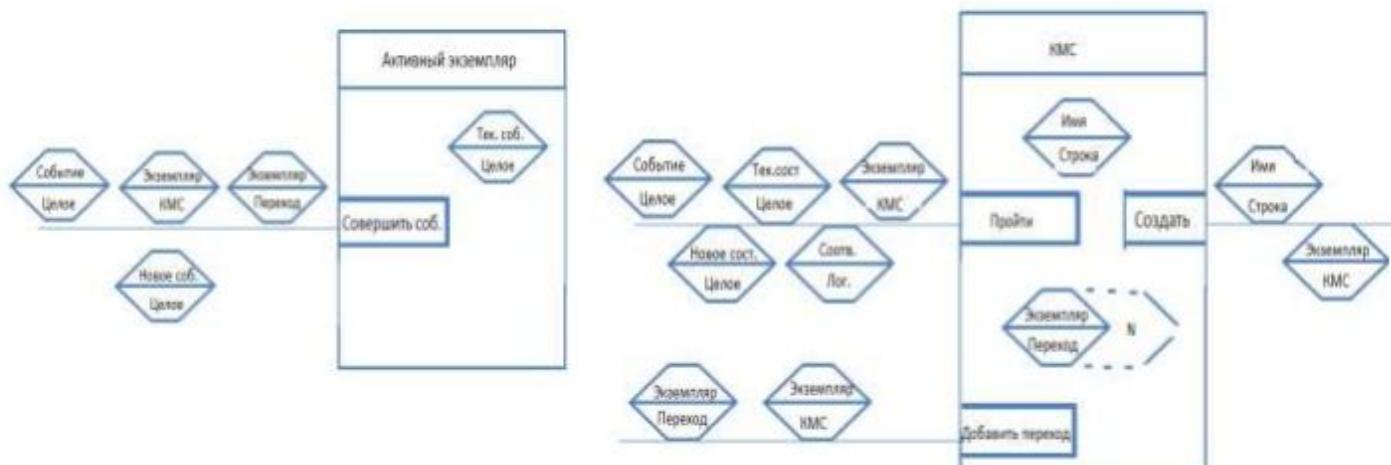
Все активные классы будут производными от активного экземпляра.

Выделяют два типа объектов:

1. Пассивные: конструкторы, аксессоры (объектов, класса);
2. Активные: обработчики событий, конструкторы, аксессоры, инициализатор кмс. (Для активных выделяют жизненные циклы).

Выделяются также объекты, осуществляющие только связь между другими – определители: конструкторы, обработчики, инициализатор КМС.(Определяют модель состояний связей объектов)

▲



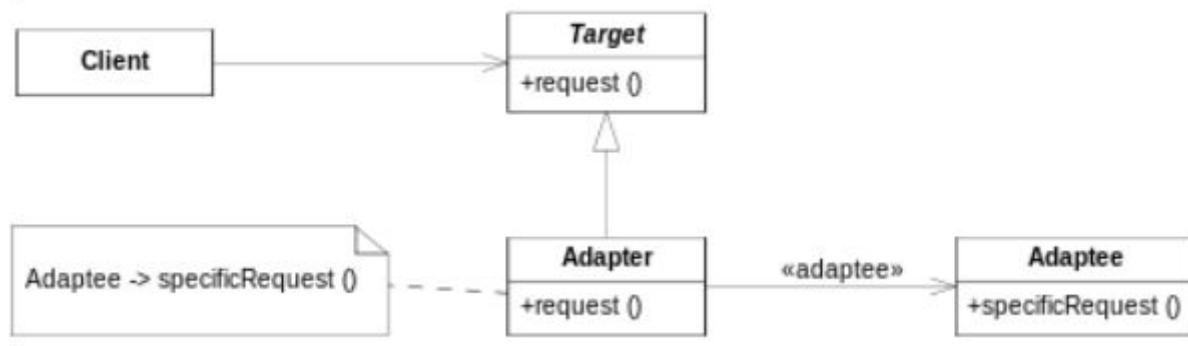
Ак
Что

13. Структурные паттерны: адаптер, компоновщик, декоратор, заместитель, мост, фасад.

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Назначение: для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс (приводит интерфейс класса (или нескольких классов) к интерфейсу требуемого вида)

Применяется: система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс. Чаще всего шаблон Адаптер применяется если необходимо создать класс, производный от вновь определяемого или уже существующего абстрактного класса.



```
class FahrenheitSensor {
public:
    float getFahrenheitTemp() {float t = 32.0; return t;}
};

class Sensor {
public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor {
public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {}
    ~Adapter() {delete p_fsensor;}
    float getTemperature() {
        return (p_fsensor->getFahrenheitTemp() -32.0)*5.0/9.0;
    }
private:
    FahrenheitSensor* p_fsensor;
};
```

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

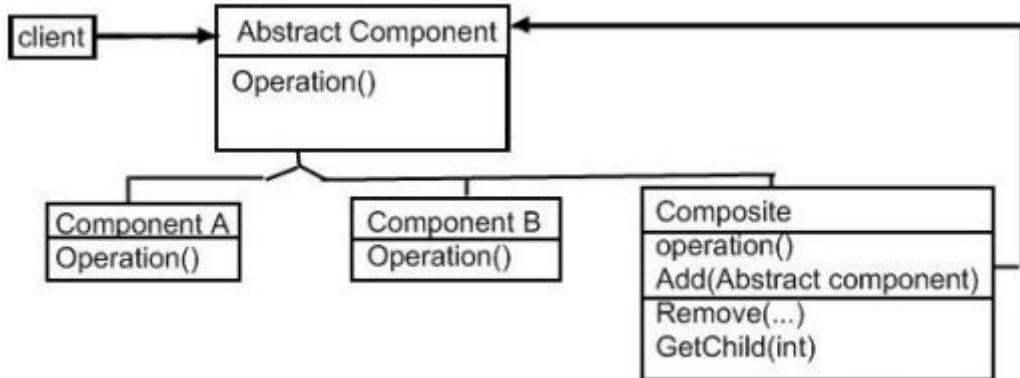
Применимость:

Когда вам нужно представить древовидную структуру объектов.

Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего две основные разновидности объектов.

Когда клиенты должны единообразно трактовать простые и составные объекты.

Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично, с каким именно объектом ему предстоит работать.



```

class Unit {
public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {}
    virtual ~Unit() {}
};

class Archer: public Unit {
public:
    virtual int getStrength() {return 1;}
};

```

```

class Infantryman: public Unit {
public:
    virtual int getStrength(){return 2;}
};

class CompositeUnit: public Unit {
public:
    int getStrength() {
        int total = 0;
        for(int i=0; i<c.size(); ++i)
            total += c[i]->getStrength();
        return total;
    }
    void addUnit(Unit* p){c.push_back(p);}
    ~CompositeUnit() {
        for(int i=0; i<c.size(); ++i)
            delete c[i];
    }
private:
    std::vector<Unit*> c;
};

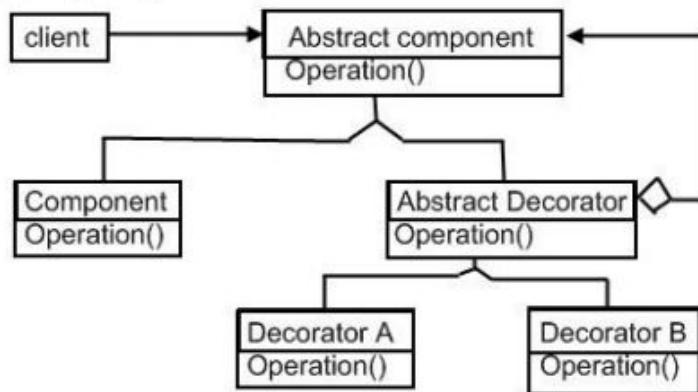
CompositeUnit* createLegion() {
    CompositeUnit* legion = new CompositeUnit;
    for (int i=0; i<3000; ++i)
        legion->addUnit(new Infantryman);
    for (int i=0; i<1200; ++i)
        legion->addUnit(new Archer);
    return legion;
}

```

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Проблема: Можно было бы юзать наследование, но вы не сможете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс. Одним из способов обойти эти проблемы является механизм *композиции*. Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому *наследовать* его поведение.

Декоратор



```
class Widget {
public:
    virtual void draw() = 0;
};

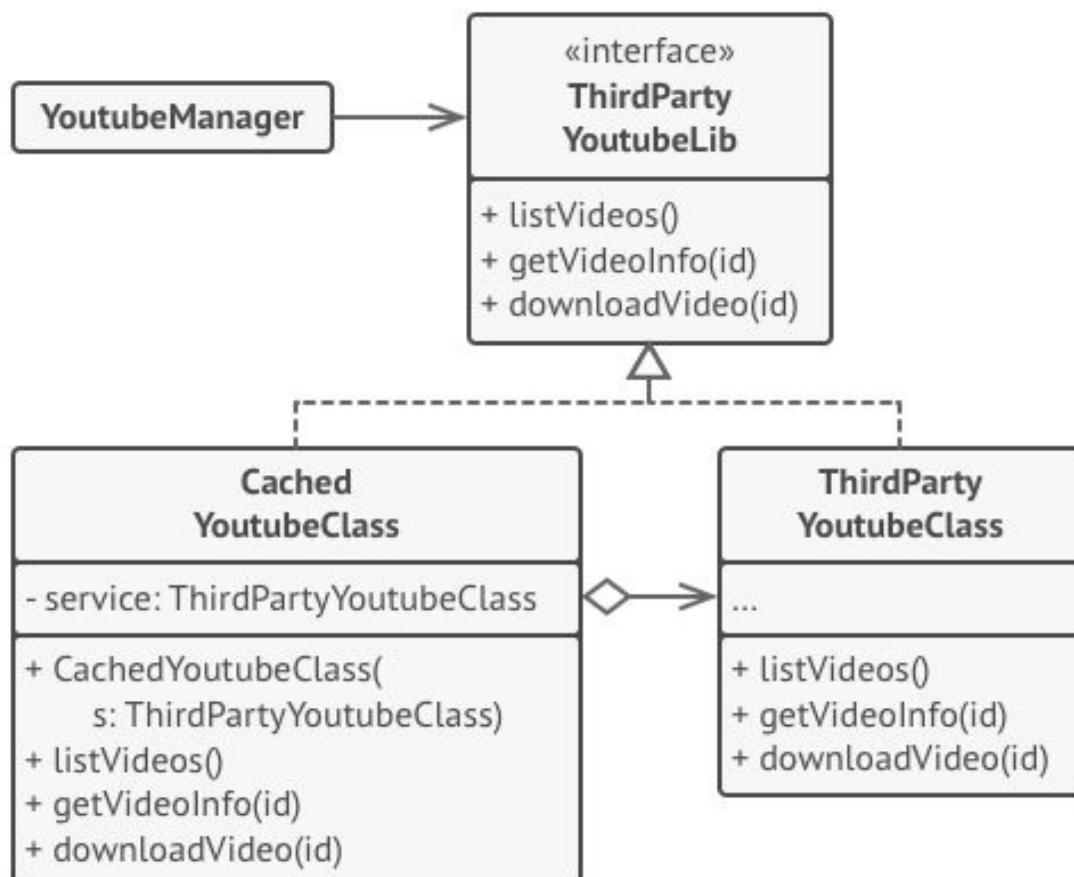
class TextField: public Widget {
    int width, height;
public:
    TextField(int w, int h) {
        width = w;
        height = h;
    }
    void draw() { cout << "TextField: " << width << ", " << height << '\n'; }
};

class Decorator: public Widget {
    Widget *wid;
public:
    Decorator(Widget *w) { wid = w; }
    void draw() { wid->draw(); }
};

class BorderDecorator: public Decorator {
public:
    BorderDecorator(Widget *w): Decorator(w) {}
    void draw() { Decorator::draw(); cout << "BorderDecorator" << '\n'; }
};

Widget *aWidget = new BorderDecorator( new BorderDecorator( (new TextField(80, 24)) ) );
aWidget->draw();
```

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.



```

// Интерфейс удалённого сервиса.
interface ThirdPartyYoutubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)

// Конкр. реализация. Методы этого класса запрашивают различную информацию.
class ThirdPartyYoutubeClass is
    method listVideos() // Получить список видеороликов с помощью API Youtube.

    method getVideoInfo(id) // Получить информацию о каком-то видеоролике.

    method downloadVideo(id) // Скачать видео с Youtube.

// Но, можно кешировать запросы к Youtube и не повторять их какое-то время, пока кеш не устареет. Внести этот код
// напрямую в сервисный класс нельзя: он находится в сторонней библиотеке. Поместим логику кеширования в
// класс-обёртку. Он будет делегировать запросы к сервисному объекту, только если нужно непосредственно выслать запрос.
class CachedYoutubeClass implements ThirdPartyYoutubeLib is
    private field service: ThirdPartyYoutubeClass
    private field listCache, videoCache
    field needReset

    constructor CachedYoutubeClass(service: ThirdPartyYoutubeLib) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache

    method downloadVideo(id) is
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)

// Класс GUI, который использует сервисный объект. Вместо
// реального сервиса, мы подсунем ему объект-заместитель. Клиент
// ничего не заметит, так как заместитель имеет тот же
// интерфейс, что и сервис.
class YoutubeManager is
    protected field service: ThirdPartyYoutubeLib

    constructor YoutubeManager(service: ThirdPartyYoutubeLib) is
        this.service = service

    method renderVideoPage() is
        info = service.getVideoInfo()
        // Отобразить страницу видеоролика.

    method renderListPanel() is
        list = service.listVideos()
        // Отобразить список превьюшек видеороликов.

    method reactOnUserInput() is
        renderVideoPage()
        renderListPanel()

// Конфигурационная часть приложения создаёт и передаёт клиентам
// объект заместителя.
class Application is
    method init() is
        youtubeService = new ThirdPartyYoutubeClass()
        youtubeProxy = new CachedYoutubeClass(youtubeService)
        manager = new YoutubeManager(youtubeProxy)
        manager.reactOnUserInput()

```

Мост — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Отделяется сущность от реализации.

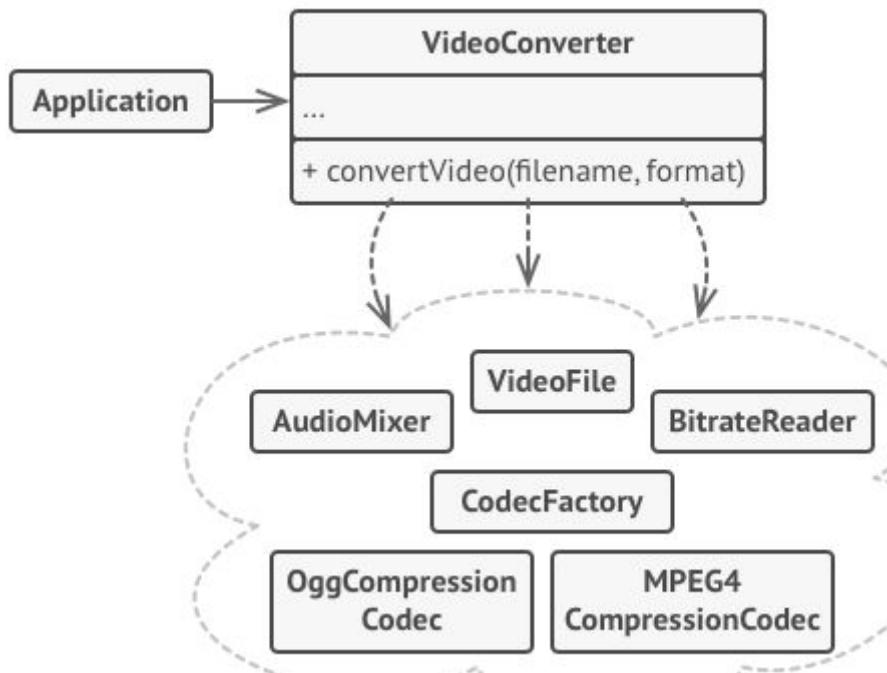


```
class DrawingAPI {
public:
    virtual void drawCircle(double x, double y, double radius) = 0;
    virtual ~DrawingAPI() {}
};

class DrawingAPI1: public DrawingAPI {
public:
    DrawingAPI1() {}
    virtual ~DrawingAPI1() {}
    void drawCircle(double x, double y, double radius) {
        printf("nAPI1 at %f:%f %fn", x, y, radius);
    }
};
class Shape {
public:
    virtual void draw()= 0;
    virtual void resizeByPercentage(double pct) = 0;
    virtual ~Shape() {
    }
};
class CircleShape: public Shape {
public:
    CircleShape(double x, double y, double radius, DrawingAPI& drawingAPI) :
        x(x), y(y), radius(radius), drawingAPI(drawingAPI) {}
    virtual ~CircleShape() {}
    void draw() { drawingAPI.drawCircle(x, y, radius); }
    void resizeByPercentage(double pct) { radius *= pct; }
private:
    double x, y, radius;
    DrawingAPI& drawingAPI;
};
DrawingAPI1 apil;
CircleShape c1(1, 2, 3, apil);
Shape* shapes[1];
shapes[0] = &c1;
shapes[0]->resizeByPercentage(2.5);
shapes[0]->draw();
```

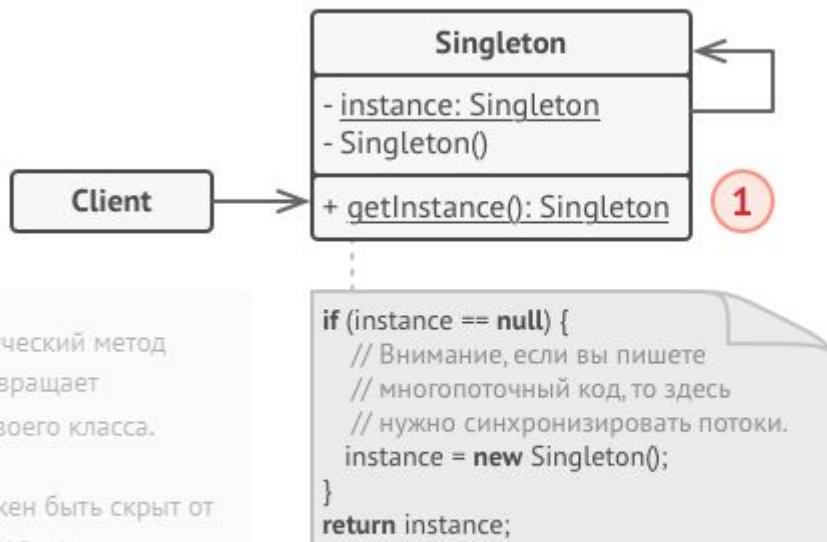
Паттерн фасад.

Много классов. Иерархия разваливается на куски со связями между собой. Чтобы не работать с большим числом объектов, создают классы, являющиеся фасадами системы. Фасад отвечает за взаимодействие одного «мирка» с другим. По аналогии со структурным программированием, когда функция получала информацию из другого домена. Всю задачу можно рассматривать как единый объект. Фасадом можно скрыть большое число разных объектов и классов. Однако возникают задачи, когда нужно работать со множеством мелких объектов одного класса.



14. Порождающие паттерны: одиночка, фабричный метод, абстрактная фабрика, строитель, прототип, пул объектов.

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



1

Одиночка определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

Одиночка

```
template <typename T>
class Singleton
{
public:
    static T *ptr;
protected:
    Singleton();
public:
    static T& instance()
    {
        return ptr?*ptr:*ptr = new T();
    }
private:
    Singleton(Singleton<T> const&);
    Singleton<T>& operator=(Singleton<T> const&);
};
template<T>
T* Singleton<T>::ptr=0;
```

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Фабричный метод

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

```
class Product;
class Creator
{
public:
    Creator():ptr(0) {}
    ~Creator()
    {
        delete ptr;
    }
    Product* getProduct();
protected:
    virtual Product* createProduct() = 0;
private:
    Product* ptr;
};

Product *Creator::getProduct()
{
    return ptr ? ptr : (ptr = createProduct());
}

template <typename T>
class ConCreator: public Creator
{
protected:
    Product* createProduct()
    {
        return new T;
    }
};
```

Активе
Чтобы
раздел

3 Создатель объявляет

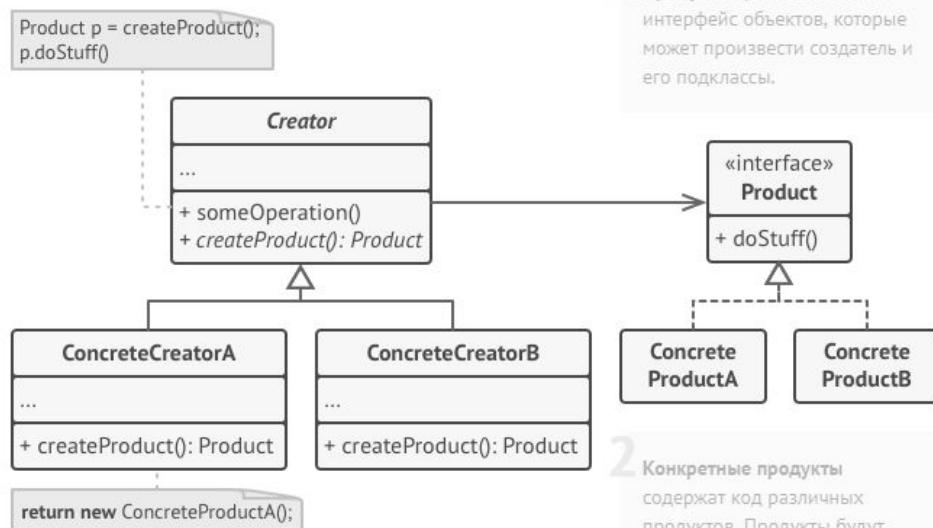
фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов не является единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

1

Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.



4

Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

2

Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

Активация Windows

Чтобы активировать Windows, введите

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

```
class Infantryman{
public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};

class Archer{
public:
    virtual void info() = 0;
    virtual ~Archer() {}
};

class RomanInfantryman: public Infantryman{
public:
    void info() { cout << "RomanInfantryman" << endl; }
};

class RomanArcher: public Archer{
public:
    void info() { cout << "RomanArcher" << endl; }
};

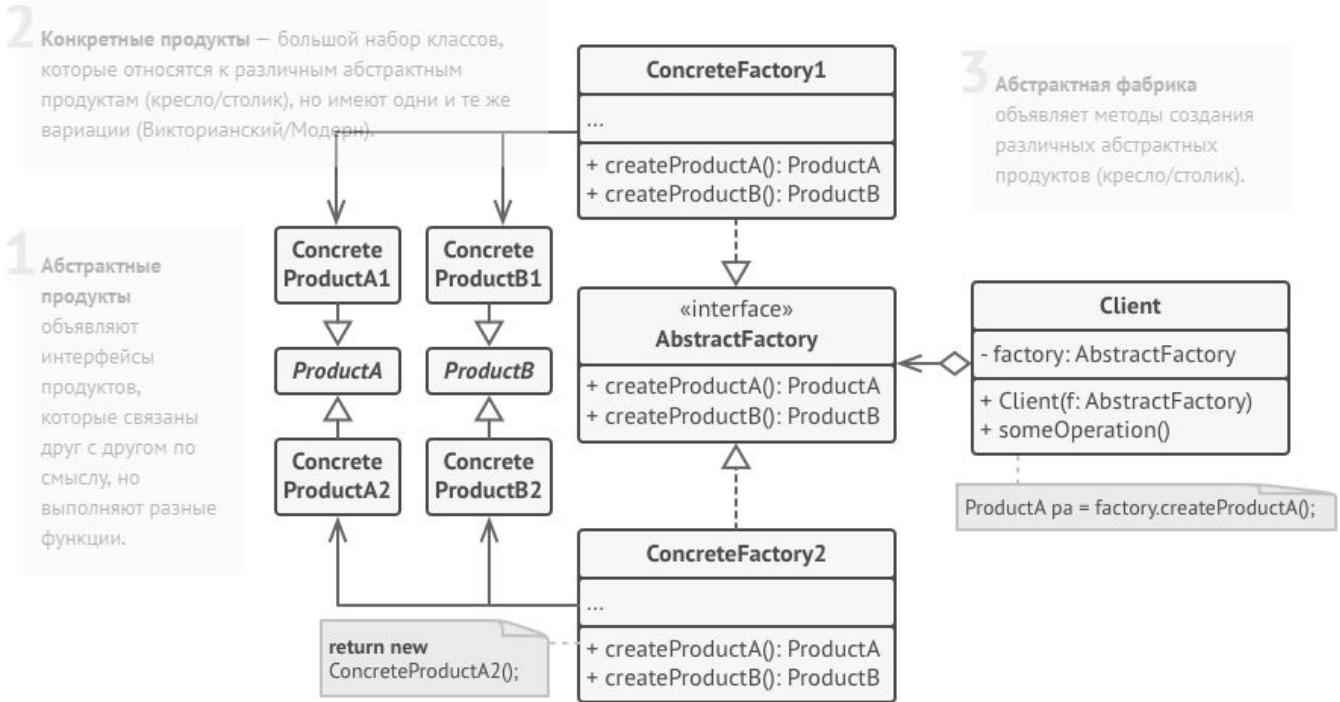
class ArmyFactory {
public:
    virtual Infantryman* createInfantryman() = 0;
    virtual Archer* createArcher() = 0;
    virtual ~ArmyFactory() {}
};

class RomanArmyFactory: public ArmyFactory {
public:
    Infantryman* createInfantryman() { return new RomanInfantryman; }
    Archer* createArcher() { return new RomanArcher; }
};

class Army {
public:
    ~Army() {
        int i;
        for(i=0; i<vi.size(); ++i) delete vi[i];
        for(i=0; i<va.size(); ++i) delete va[i];
    }
    void info() {
        int i;
        for(i=0; i<vi.size(); ++i) vi[i]->info();
        for(i=0; i<va.size(); ++i) va[i]->info();
    }
    vector<Infantryman*> vi;
    vector<Archer*> va;
};

class Game {
public:
    Army* createArmy( ArmyFactory& factory ) {
        Army* p = new Army;
        p->vi.push_back( factory.createInfantryman() );
        p->va.push_back( factory.createArcher() );
        return p;
    }
};

int main(){
    Game game;
    RomanArmyFactory ra_factory;
    Army * ra = game.createArmy( ra_factory );
    cout << "Roman army:" << endl;
    ra->info();
}
```

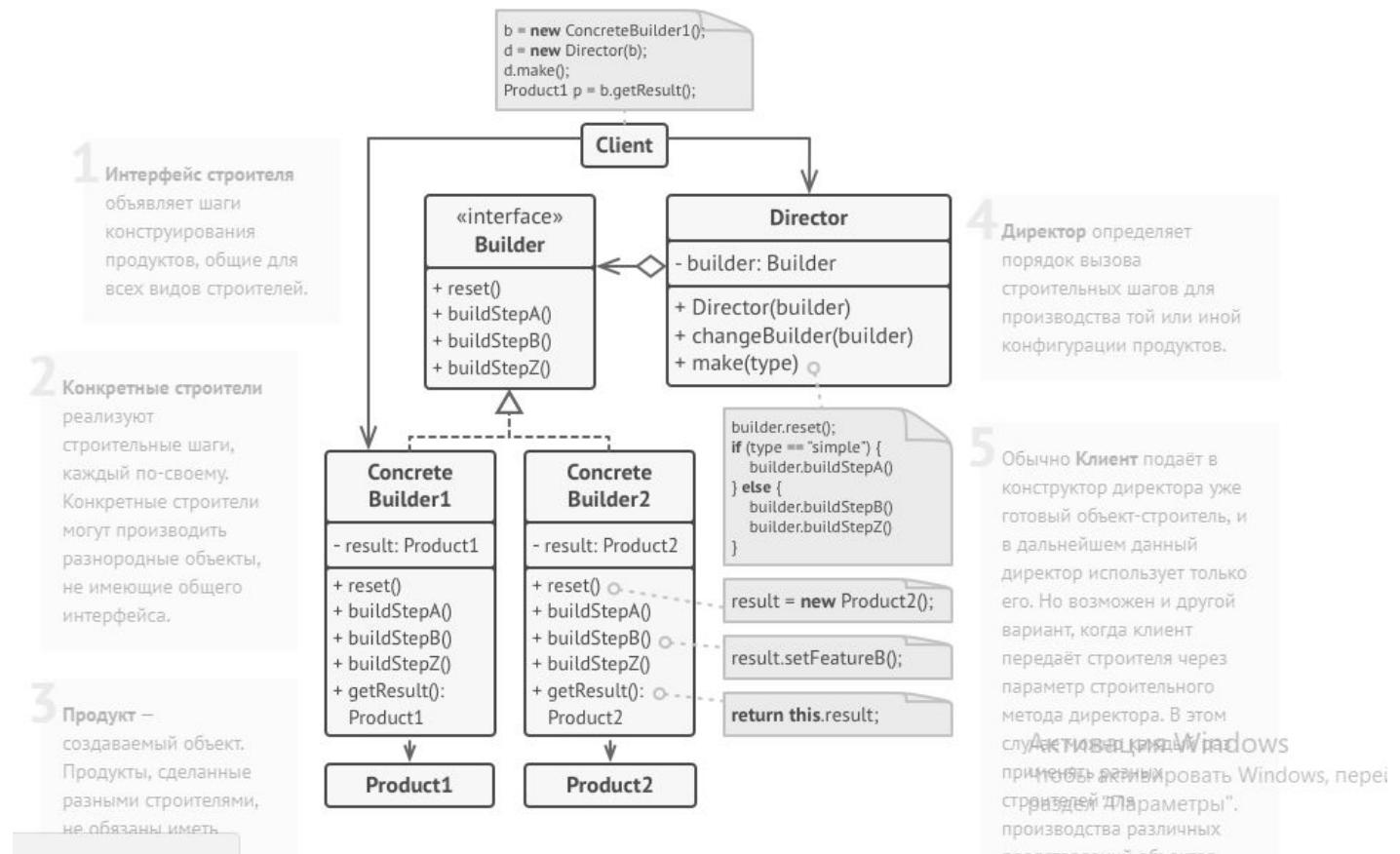


4 Конкретные фабрики относятся к каждой своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.

5 Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

Активация Windows
Чтобы активировать Windc
раздел "Параметры".

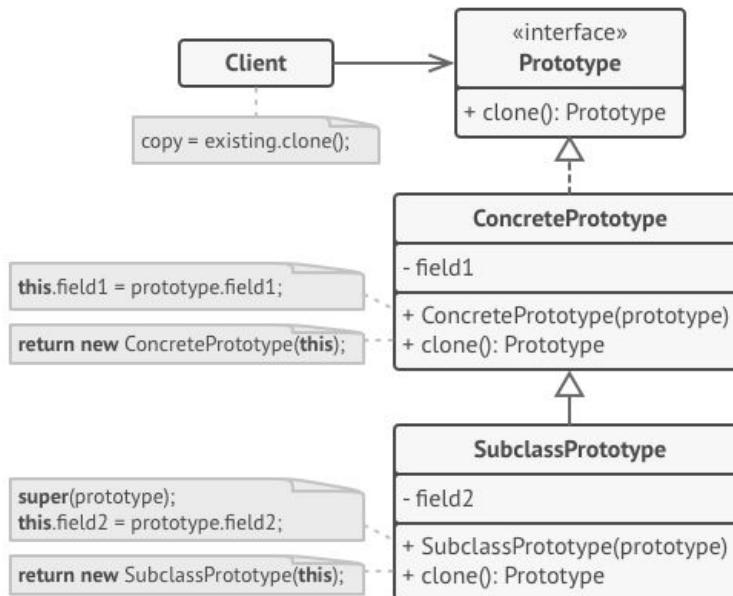
Строитель – это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

3 Клиент создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

1 Интерфейс прототипов описывает операции клонирования. В большинстве случаев — это единственный метод `clone()`.



2 Конкретный прототип реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

Активация Windows
Чтобы активировать Wind

```

class Client
{
    void Operation()
    {
        Prototype prototype = new ConcretePrototype1(1);
        Prototype clone = prototype.Clone();
        prototype = new ConcretePrototype2(2);
        clone = prototype.Clone();
    }
}
  
```

```

abstract class Prototype
{
    public int Id { get; private set; }
    public Prototype(int id)
    {
        this.Id = id;
    }
    public abstract Prototype Clone();
}
  
```

```

class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(int id)
        : base(id)
    {}
    public override Prototype Clone()
    {
    }
}
  
```

```
{  
    return new ConcretePrototype1(Id);  
}  
}  
  
class ConcretePrototype2 : Prototype  
{  
    public ConcretePrototype2(int id)  
        : base(id)  
    {}  
    public override Prototype Clone()  
    {  
        return new ConcretePrototype2(Id);  
    }  
}
```

Объектный пул (англ. object pool) — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности, когда объекты часто создаются-уничтожаются, но одновременно существует лишь небольшое их число.

```
class Object
{
    // ...
};

class ObjectPool
{
private:
    struct PoolRecord
    {
        Object* instance;
        bool    in_use;
    };

    std::vector<PoolRecord> m_pool;

public:
    Object* createNewObject()
    {
        for (size_t i = 0; i < m_pool.size(); ++i)
        {
            if (!m_pool[i].in_use)
            {
                m_pool[i].in_use = true; // переводим объект в список используемых
                return m_pool[i].instance;
            }
        }
        // если не нашли свободный объект, то расширяем пул
        PoolRecord record;
        record.instance = new Object;
        record.in_use   = true;

        m_pool.push_back(record);

        return record.instance;
    }
}
```

```
void deleteObject(Object* object)
{
    // В реальности не удаляем, а лишь помечаем, что объект свободен
    for (size_t i = 0; i < m_pool.size(); ++i)
    {
        if (m_pool[i].instance == object)
        {
            m_pool[i].in_use = false;
            break;
        }
    }
}

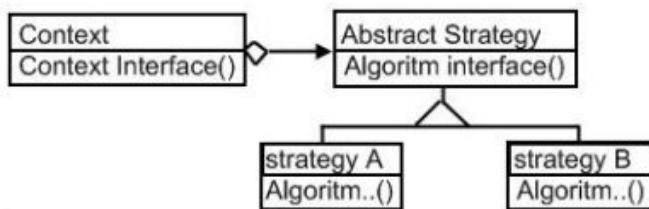
virtual ~ObjectPool()
{
    // Теперь уже "по-настоящему" удаляем объекты
    for (size_t i = 0; i < m_pool.size(); ++i)
        delete m_pool[i].instance;
}
};

int main()
{
    ObjectPool pool;
    for (size_t i = 0; i < 1000; ++i)
    {
        Object* object = pool.createNewObject();
        // ...
        pool.deleteObject(object);
    }
    return 0;
}
```

15. Паттерны поведения: стратегия, шаблонный метод, посетитель, посредник, хранитель, команда.

Стратегия

Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.



```
class Compression{
public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class ZIP_Compression : public Compression{
public:
    void compress( const string & file ) { cout << "ZIP" << endl; }
};

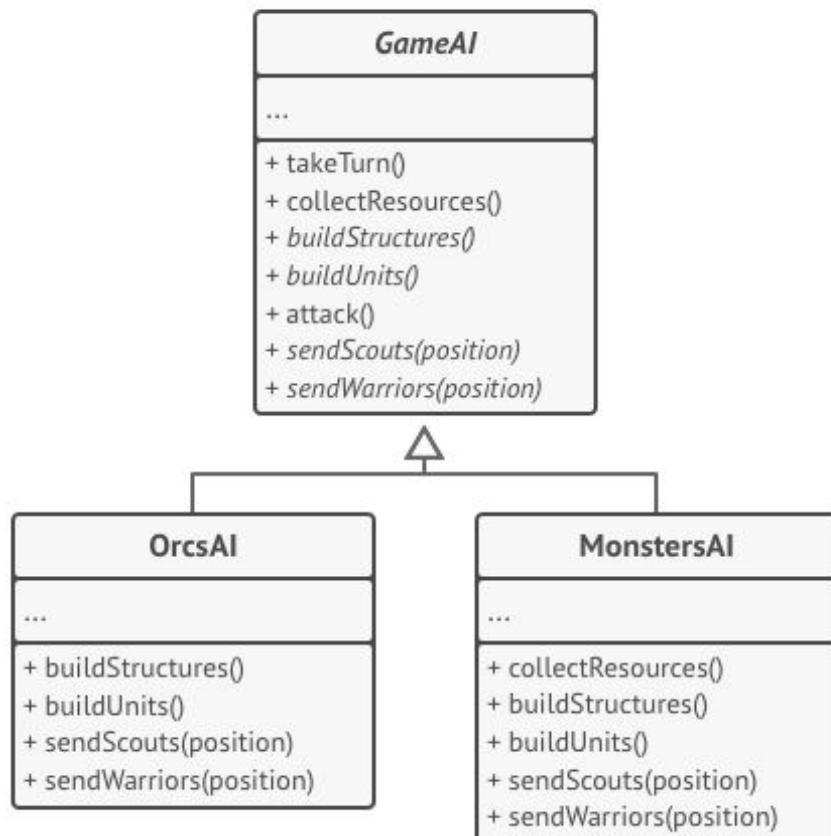
class RAR_Compression : public Compression{
public:
    void compress( const string & file ) { cout << "RAR" << endl; }
};

class Compressor {
public:
    Compressor( Compression* comp): p(comp) {}
    ~Compressor() { delete p; }
    void compress( const string & file ) { p->compress( file); }
private:
    Compression* p;
};

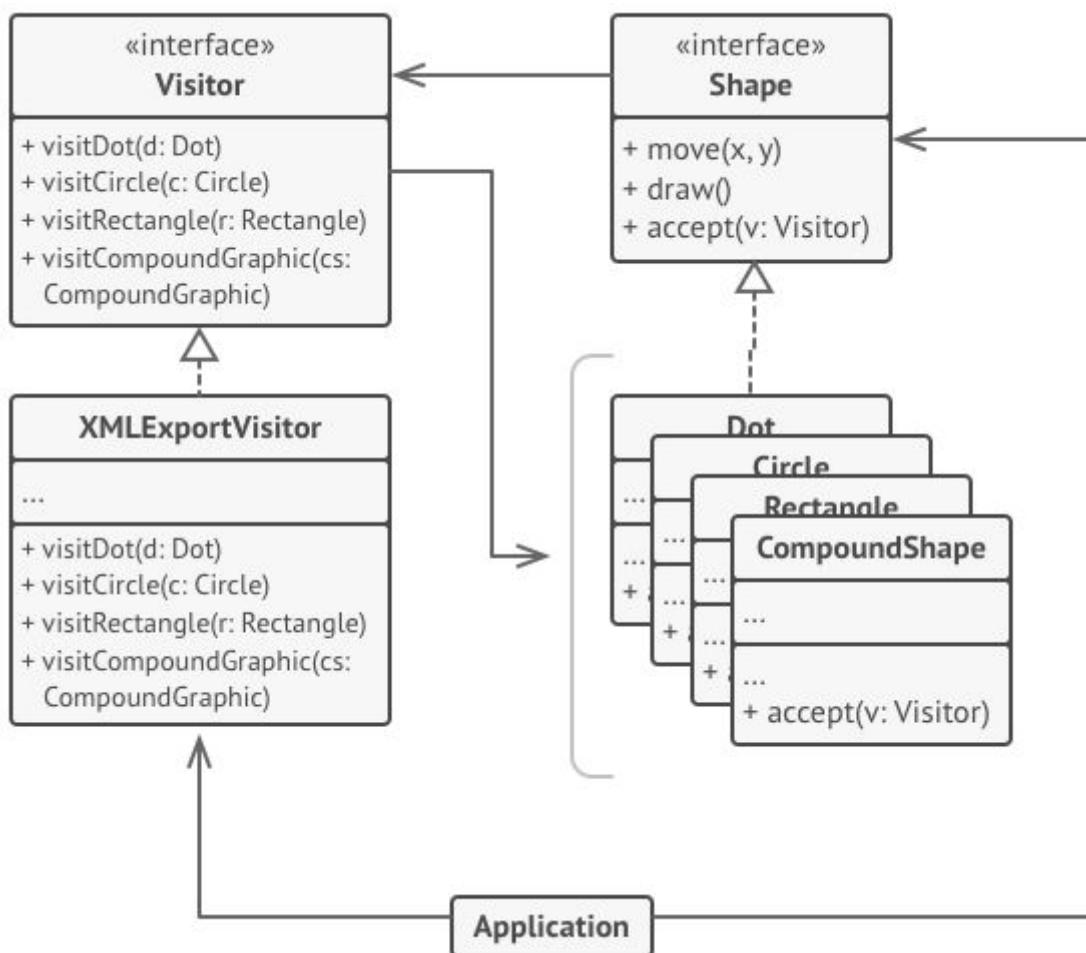
Compressor* p = new Compressor( new ZIP_Compression);
```

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

В этом примере **Шаблонный метод** используется как заготовка для стандартного искусственного интеллекта в простой игре-стратегии. Для введения в игру новой расы достаточно создать подкласс и реализовать в нём недостающие методы.

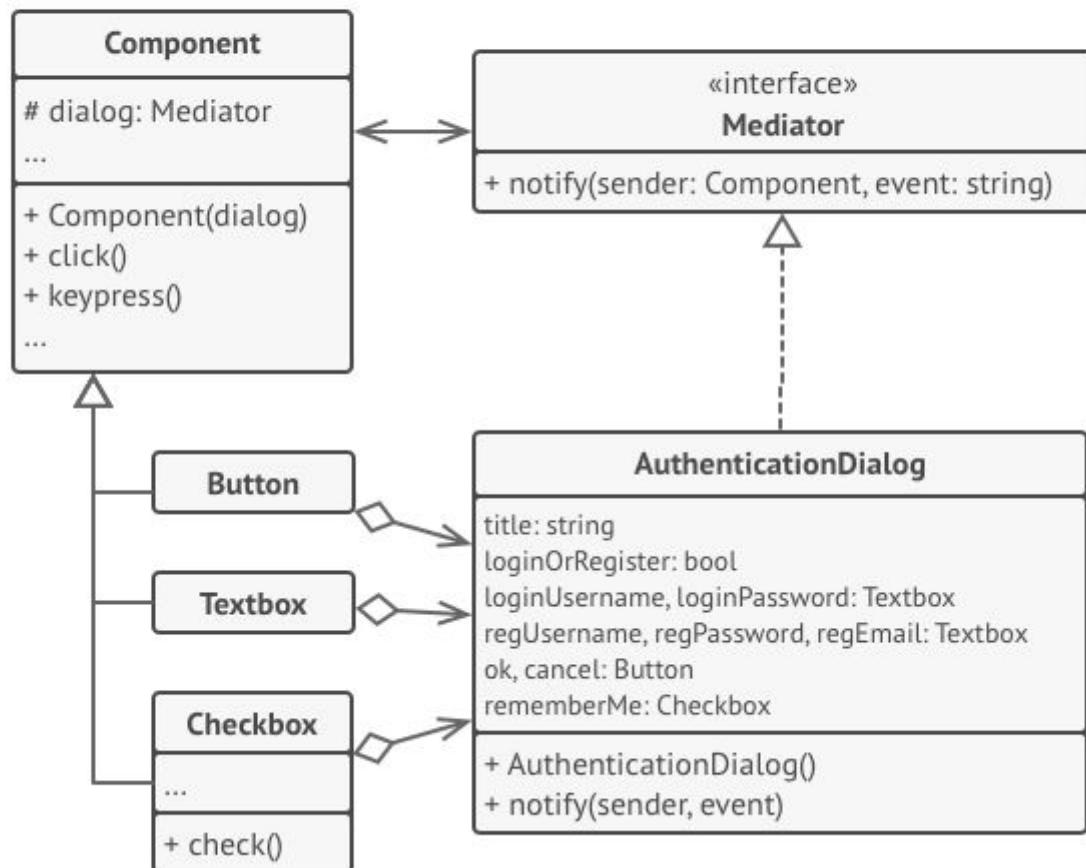


Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



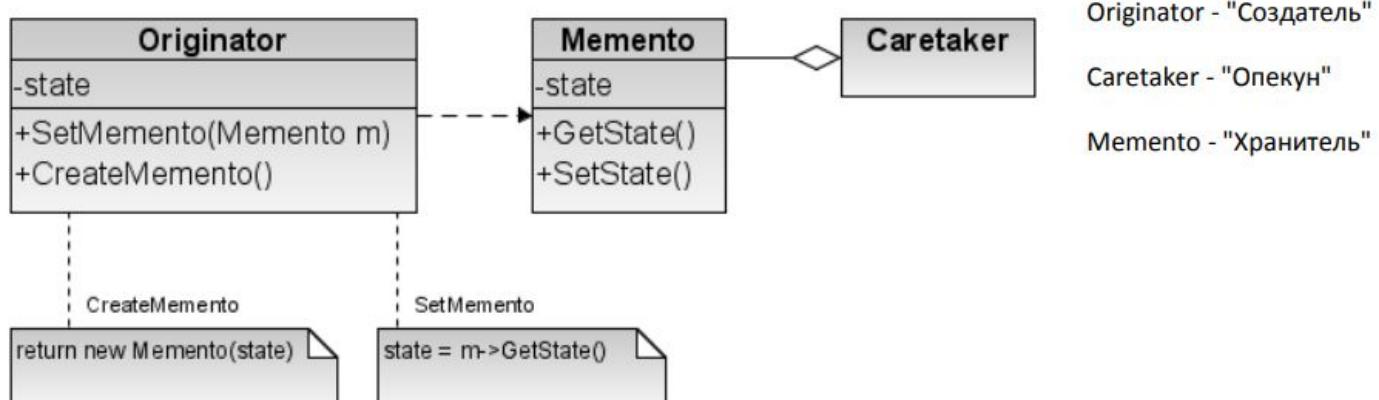
Посредник — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

В этом примере **Посредник** помогает избавиться от зависимостей между классами различных элементов пользовательского интерфейса: кнопками, чекбоксами и надписями.



Хранитель

- о необходимо сохранить снимок состояния объекта (или его части) для последующего восстановления
- о прямой интерфейс получения состояния объекта раскрывает детали реализации и нарушает инкапсуляцию объекта



```

template <typename T>
class Holder;
template <typename T>
class Trule {
private:
    T* ptr;
public:
    Trule(Holder<T>& h) {ptr = h.release();}
    ~Trule() {delete ptr;}
private:
    Trule(Trule<T>& );
    Trule<T>& operator =(Trule<T>& );
    friend class Holder<T>;
};

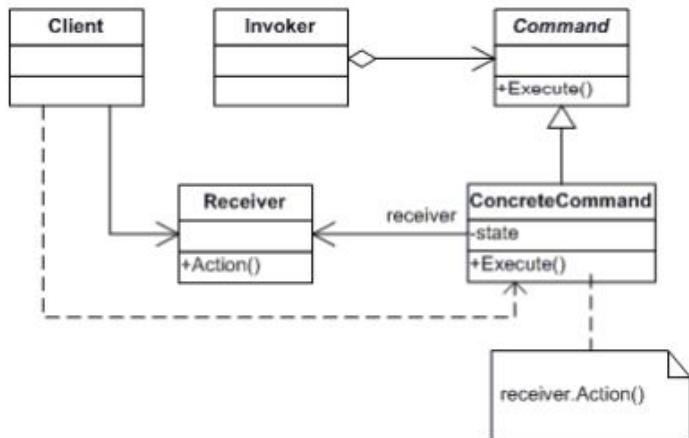
template <typename T>
class Holder {
private:
    T* ptr;
public:
    Holder() : ptr(0) {}
    explicit Holder(T* p) : ptr(p) {}
    ~Holder() {delete ptr;}
    T& operator *() const {return *ptr;}
    T& get() const {return *ptr;}
    T* operator ->() const {return ptr;}
    void exchange(Holder<T>& h);
    Holder(Trule<T> const& t) {
        ptr = t.ptr;
        const_cast<Trule<T>&>(t).ptr = 0;
    }
    Holder<T>& operator =(Trule<T> const& t)
        delete ptr;
        ptr = t.ptr;
        const_cast<Trule<T>&>(t).ptr = 0;
        return *this;
    }
    T* release() {
        T* p = ptr;
        ptr = 0;
        return p;
    }
private:
    Holder(Holder<T> const& );
    Holder<T>& operator =(Holder<T> const& );
};
  
```

Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Команда

Паттерн поведения объектов, известен так же под именем Action(действие).

Обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.



2. Классы и объекты в C++. Ограничение доступа к членам класса в C++. Члены класса и объекта. Методы. Схемы наследования.

Класс – описывается в заголовочных файлах.

Механизм описания класса.

- struct
- union – не может быть базовым классом, и не может быть производным.
- По умолчанию в структуре и объединении члены классов открыты, в классе закрыты, чем достигается принцип инкапсуляции (нет доступа к данным извне)
- Class

Пример:

```
class <имя класса> [::<список базовых классов>]
{
private: //доступ к данным есть только внутри самого класса
    int a;
protected: //доступ есть внутри класса и во всех его наследниках
    int b;
public: //доступны для внешнего кода
    int f();
}; //класс заканчивается ; как и структура
//Классы описываются в заголовочных файлах .h. Методы же определяются в .cpp
```

Уровни доступа:

1. private
2. protected
3. public

Функции класса будем называть методами.

Protected – члены к которым имеют доступ только методы класса и ПРОИЗВОДНЫЕ от него классы

Располагать члены в порядке private, protected, public.

Если создаём библиотеки, то описываем в обратном порядке.

Схем наследования тоже 3:

1. private
2. protected
3. public

Пример наследования:

```
class A
{
private:
    int a;
protected:
    int b;
public:
    int f();
};

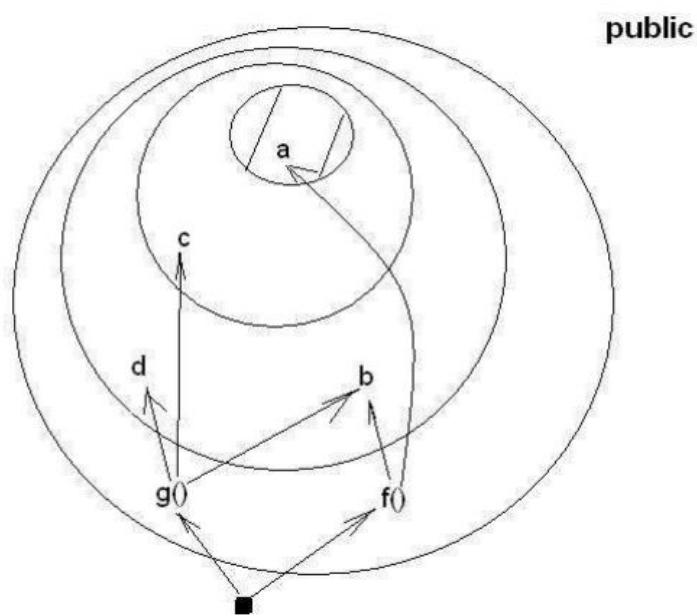
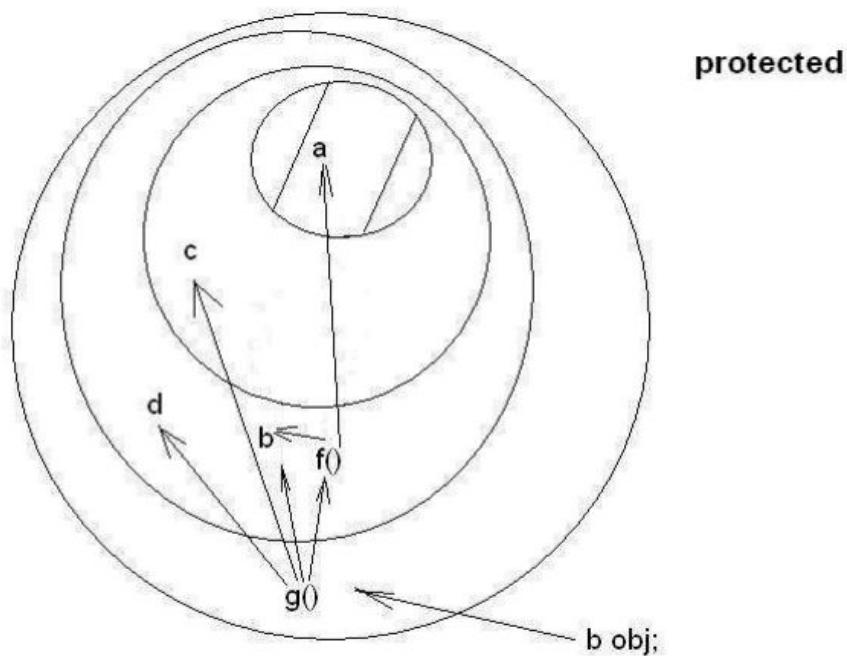
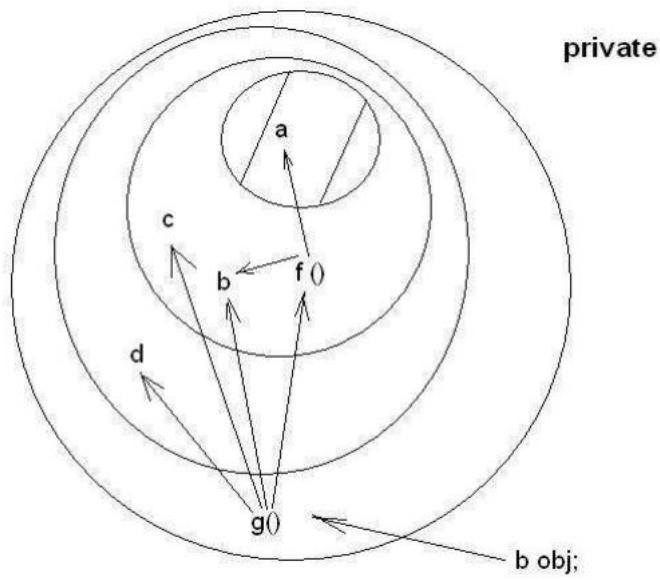
class B: private(или protected, или public) A
{
private:
    int c;
protected:
    int d
public:
    int g ();
};
```

Рассмотрим каждое наследование по отдельности:

private – полное сохранение интерфейса базового класса. (рисунок 1)

protected – (рисунок 2)

public – (рисунок 2)



Статистические члены класса:

- Как данные так и методы
- Особенности статистических данных

Class A

{

Private:

```
    Int a;  
    Static int b;
```

Public:

```
    Int f();  
    Static int g(A* obj)
```

}

- Не член объекта, а член класса (общий для всех экземпляров)
- Int A::b = 0 инициализация статического члена
- Не применим указатель this
- Вызывается для класса
- Можно вызвать без создания объекта (A::g(pobj))

Константные члены

Class B

{

Private:

```
    Const int a;  
    Const static int b = 2;
```

Public:

```
    Int f(); //1  
    Int f() const; // 2
```

}

- Не меняются во время жизни объекта
- Можно инициализировать только в конструкторе
- Можно создать константный объект с константными методами
- Для константного объекта возможен вызов только константных методов

B obj;

Const B cobj;

Obj.f() // 1

Cobj.f() //2

В классах и структурах есть члены, представляющие их данные и поведение. Члены класса включают все члены, объявленные в этом классе, а также все члены (кроме конструкторов и методов завершения), объявленные во всех классах в иерархии наследования данного класса. Закрытые члены в базовых классах наследуются, но недоступны из производных классов.

Поля	Поля являются переменными, объявленными в области класса. Поле может иметь встроенный числовой тип или быть экземпляром другого класса. Например, в классе календаря может быть поле, содержащее текущую дату.
Константы	Константы — это поля или свойства, значения которых устанавливаются во время компиляции и не изменяются.
Свойства	Свойства — это методы класса. Доступ к ним осуществляется так же, как если бы они были полями этого класса. Свойство может защитить поле класса от изменений (независимо от объекта).
Методы	Методы определяют действия, которые может выполнить класс. Методы могут принимать параметры, предоставляющие входные данные, и возвращать выходные данные посредством параметров. Методы могут также возвращать значения напрямую, без использования параметров.
События	События предоставляют другим объектам уведомления о различных случаях, таких как нажатие кнопки или успешное выполнение метода. События определяются и переключаются с помощью делегатов.
Операторы	Перегруженные операторы считаются членами класса. При перегрузке оператора его следует определять как открытый статический метод в классе. Предопределенные операторы (+, *, < и т. д.) не считаются членами. Дополнительные сведения см. в разделе Перегружаемые операторы .
Индексаторы	Индексаторы позволяют индексировать объекты аналогично массивам.
Конструкторы	Конструкторы — это методы, которые вызываются при создании объекта. Зачастую они используются для инициализации данных объекта.
Методы завершения	Методы завершения очень редко используются в C#. Они являются методами, вызываемыми средой выполнения, когда объект нужно удалить из памяти. Они обычно применяются для правильной обработки ресурсов, которые должны быть высвобождены .
Вложенные типы	Вложенными типами являются типы, объявленные в другом типе. Вложенные типы часто применяются для описания объектов, использующихся только тиปами , в которых эти объекты находятся.

Ат
Чт
ра

4. Наследование в C++. Построение иерархии классов. Множественное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Неоднозначности при множественном наследовании.

Выделили метод, который называли конструктором. Это метод вызываемый при инициализации объекта. У него отсутствует тип возврата. Конструктор можно перегружать. Конструктор не наследуется. Если конструктор `private`, то невозможно создание производных классов.

Когда вызывается конструктор:

1. При определении для статических и внутренних объектов. Выполняется до функции `main()`.
2. При определении локальных объектов.
3. При выполнении оператора `new`
4. Для временных объектов.

Конструктор должен быть всегда. Если нет конструктора, то всегда создаётся 2 конструктора

1 – по умолчанию, 2 – конструктор копирования.

Если мы указали хотя бы один конструктор, то конструктор по умолчанию не создаётся. Конструктор копирования создаётся всегда.

Конструктор копирования вызывается:

1. При инициализации одного объекта другим.
2. При передаче по значению параметров
3. При возврате по значению.

```
class Complex
{
private:
    double re, im;
public:
    Complex ();                                //1
    Complex (double r);                        //2
    Complex (double r, double i);              //3
    Complex (Complex &c);                     //4
};

Complex a();
a() – это функция без параметров возвращающая тип Complex.
Complex b;                                  //1
Complex c (1.);   //2
Complex d = 2.;    //2
Complex e (3., 4.); //3
f = Complex (5., 6.); //3
g (f), h = g;
```

Конструктор не может быть `volatile`, `static`, `const`

В C++ реализуется неявный вызов деструктора. Этот метод не принимает параметров. Нет типа возврата. Деструктор имеет такое же имя что и конструктор, но начинается со знака ~. Деструкторы вызываются в обратном порядке. Для локальных статических объектов вызывается деструктор до уничтожении глобальных статических объектов, но после выполнения программы. Временные объекты уничтожаются, когда в них отпадает надобность. Деструктор не перегружается.

Деструктор не может быть `const`, `volatile`, `static`, но может быть `virtual`.

Актив

Конструктор выполняет свою работу в следующем порядке.

1. Вызывает конструкторы базовых классов и членов в порядке объявления.
2. Если класс является производным от виртуальных базовых классов, конструктор инициализирует указатели виртуальных базовых классов объекта.
3. Если класс имеет или наследует виртуальные функции, конструктор инициализирует указатели виртуальных функций объекта. Указатели виртуальных функций указывают на таблицу виртуальных функций класса, чтобы обеспечить правильную привязку вызовов виртуальных функций к коду.
4. Выполняет весь код в теле функции.

Наследование, построение иерархии, множественное наследование и неоднозначности в нём.

Расширение и выделение общей части из разных классов.

Причины выделения общей части:

1. Общая схема использования.
2. Сходство между наборами операций.
3. Сходство реализации.

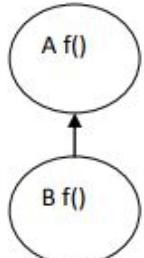
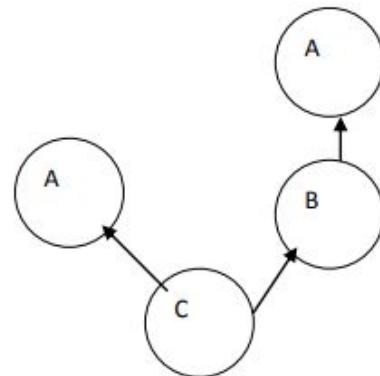
Расщепление классов

1. Два подмножества операций в классе используются в разной манере.
2. Методы класса имеют не связную реализацию.
3. Класс оперирует очевидным образом в 2-х несвязных обсуждениях проекта.

Прямая база – непосредственная база класса. Прямая база может входить только один раз

Косвенная база – А для С. Может быть несколько раз.

Тут А для С и косвенная и прямая база.



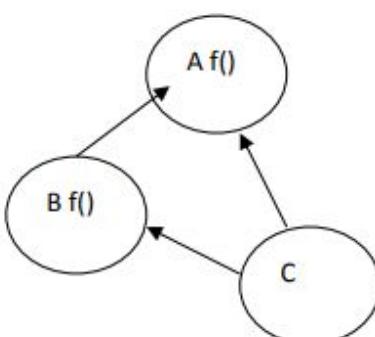
Метод f () в производном классе В доминирует над методом в доминантном классе А.

Активация
Чтобы активировать метод в производном классе, необходимо использовать ключевое слово "override".

Доминирование:

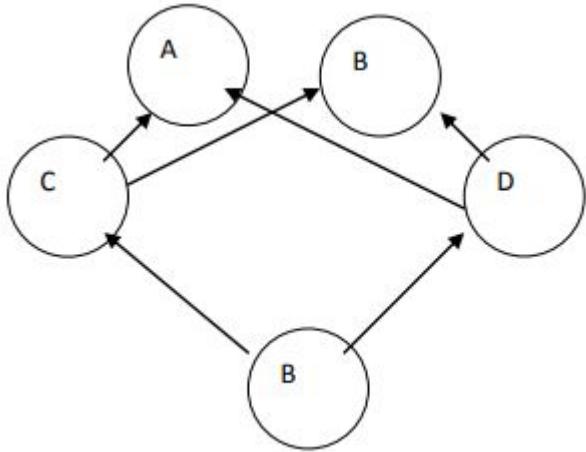
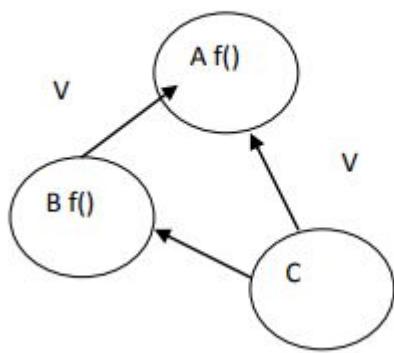
Метод f () в производном классе В доминирует над методом в доминантном классе А.

Не путать доминирование с virtual, потому что при наличии virtual используется таблица виртуальных функций.



ОШИБКА!
(Неоднозначность)

Виртуальное наследование



ООП использует рекурсивный дизайн – постепенное разворачивание программы от базовых классов к более специализированным. С++ один из немногих языков с множественным наследованием. Оно может упростить граф наследования, но также создает пучок проблем для программиста: возникает неоднозначность, которую бывает тяжело контролировать.

Патовая ситуация – если не мы писали А и В, но пытаемся объединить их в один свой. В С можно было бы доминантно определить новые методы взамен старых, но члены классов всё равно будут пересекаться.

Метод в производном классе будет доминировать над методом базового класса.

6. Перегрузка операторов в C++.

Тип данных = множество значений + множество операций.

Перегружать нельзя: "::", ".", ".*", тернарный оператор (: ?), sizeof().

.* - выбор члена класса через указатель на классовые члены

Не меняется арность, приоритет и порядок выполнения (при перегрузке).

```
class Integer
{
    //унарный +
    friend const Integer& operator+(const Integer& i);
    bool operator==(const Integer& left, const Integer& right) {
        return left.value == right.value;
}
```

8. Обработка исключительных ситуаций в C++. Пространства имен.

Обработка исключительных ситуаций.

Идея: передавать управление как обработчик некой ошибки. Проблема в том, что нужно возвращаться в точку возникновения ситуации.

Каждый класс будет отвечать за свою исключительную ситуацию.

Классы, которые отвечают за обработку исключительных ситуаций, должны быть родственными. И тогда если мы создаём новый класс, то передаём указатель в класс-обработчик.

```
try
{
    throw <выражение>; создание объекта класса
}
catch (<параметр>)
{
    ...
}
```

Пространство имён (англ. namespace) — некоторое множество, под которым подразумевается модель, абстрактное хранилище или окружение, созданное для логической группировки уникальных идентификаторов (то есть имён).

Идентификатор, определённый в пространстве имён, ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах.

7. Шаблоны функций и классов в C++. Специализация шаблонов частичная и полная.

Шаблонные классы позволяют получать классы, отличающиеся только типом в отдельных местах

```
template <typename T>
class A
{
    T* u;
public:
    void f();
};
```

У класса могут быть шаблонные методы

```
template <typename T>
void A(t) ::f()
{
}
```

Так же шаблоны могут иметь специализацию для отдельного типа

```
template<>
class A<float>
{
}
```

Шаблоны класса можно частично специализировать, при этом получившийся класс по-прежнему будет шаблоном. Частичная специализация позволяет частично настроить код шаблона для определенных типов, например:

- Шаблон имеет несколько типов, и только некоторые из них требуют специализации. Результат для остальных типов параметризован шаблоном.
- Шаблон имеет только один тип, но специализация необходима для типов указателя, ссылки, указателя на член или указателя на функцию. Специализация сама по себе по-прежнему является шаблоном с типом, на который задан указатель или ссылка.

Частичная специализация:

```
template<typename T1, typename T2>
class A{...}
```

```
template<typename T>
class A<T,T>{...}
```

```
template<typename T>
class A<T,int>{...}
```

```
template<typename T1, typename T2>
class A<T1*,T2*>{...}
```

```
A<int,float>a1;
```

```
A<float,float>a2;
```

```
A<float,int>a3;
```

```
A<int*,float*>a4;
```

```
A<int,int>a5; // ошибка
```

```
A<int*,float*>a6; // ошибка
```

Так же можно указывать значения шаблонных параметров по умолчанию(в конце списка параметров)

```
// первичный класс
template<class T>
struct RankOfArray
{ static const int value = 0; };
// рекуррентный частично специализированный класс
template<class T, int N>
struct RankOfArray < T[N] >
{ static const int value = 1+RankOfArray<T>::value; };
```

```
// первичный класс
template <typename Window, typename Controller>
class Widget
{ /* ... реализация параметризованного класса Widget для любого вида окна Window и любого обраб
// явная специализация первичного класса
template <>
class Widget <DialogWindow, DialogController>
{ /* ... реализация класса Widget под работу с диалоговым окном DialogWindow и обработчиком соб
```

5. Полиморфизм в C++. Понятие абстрактного класса. Дружественные связи.

Полиморфизм – использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.(“Один интерфейс, множество реализаций.”)

Виртуальные методы – это методы, которые предполагаемо будут переопределены в производных классах. Ключевое слово –*virtual*. Гарантируется, что будет выбрана верная функция.

Виртуальный метод может быть дружественным (*friend*) и *inline*.

Пример:

```
class A {  
public:  
    virtual void f();  
};  
class B : public A {  
public:  
    B() { f(); }  
    void g() { f(); }  
};  
class C: public B{  
public:  
    C(){ }  
    void f();  
};  
  
C c; //A::f()  
c.g(); //C::f()
```

Чисто виртуальная функция не имеет тела. Класс с хотя бы одной чисто виртуальной функцией называется **абстрактным**.

```
class A {  
public:  
    virtual void f() = 0;  
};
```

Класс, который не реализует число виртуальных методов, так же называется **абстрактным**.

АКТИ
Чтоб

Дружественный класс получает доступ ко всем объектам данного класса(*объекты плохо звучат*), но не наоборот. Дружественным может быть и метод.

Дружба не наследуется и не транзитивна.

```
class B;  
class A  
{  
    friend class B;  
    friend void B::f(A&); // только метод имеет доступ  
};
```

Абстрактный класс в C++ - это класс, в котором объявлена хотя бы одна чисто виртуальная функция. Абстрактный класс используют, когда необходимо создать семейство классов (много разновидностей монстров в игре) при этом было бы лучше вынести общую реализацию и поведение в отдельный класс. При такой тактике переопределить/дописать придется только специфичные к каждому классу методы (у каждого монстра своя анимация удара/перемещения) и/или расширить функционал класса.

3. Создание и уничтожение объектов в C++. Конструкторы и деструкторы. Виды конструкторов. Способы создания объектов.

Выделили метод, который называли конструктором. Это метод вызываемый при инициализации объекта. У него отсутствует тип возврата. Конструктор можно перегружать. Конструктор не наследуется. Если конструктор `private`, то невозможно создание производных классов.

Когда вызывается конструктор:

1. При определении для статических и внутренних объектов. Выполняется до функции `main()`.
2. При определении локальных объектов.
3. При выполнении оператора `new`
4. Для временных объектов.

Конструктор должен быть всегда. Если нет конструктора, то всегда создаётся 2 конструктора

1 – по умолчанию, 2 – конструктор копирования.

Если мы указали хотя бы один конструктор, то конструктор по умолчанию не создаётся. Конструктор копирования создаётся всегда.

Конструктор копирования вызывается:

1. При инициализации одного объекта другим.
2. При передаче по значению параметров
3. При возврате по значению.

```
class Complex
{
private:
    double re, im;
public:
    Complex ();                                //1
    Complex (double r);                        //2
    Complex (double r, double i);              //3
    Complex (Complex &c);                     //4
};

Complex a();
a() – это функция без параметров возвращающая тип Complex.
Complex b;                                  //1
Complex c (1.);   //2
Complex d = 2.;    //2
Complex e (3., 4.); //3
f = Complex (5., 6.); //3
g (f), h = g;
```

Конструктор не может быть `volatile`, `static`, `const`

В C++ реализуется неявный вызов деструктора. Этот метод не принимает параметров. Нет типа возврата. Деструктор имеет такое же имя что и конструктор, но начинается со знака ~. Деструкторы вызываются в обратном порядке. Для локальных статических объектов вызывается деструктор до уничтожении глобальных статических объектов, но после выполнения программы. Временные объекты уничтожаются, когда в них отпадает надобность. Деструктор не перегружается.

Деструктор не может быть `const`, `volatile`, `static`, но может быть `virtual`.

Актив

Конструктор выполняет свою работу в следующем порядке.

1. Вызывает конструкторы базовых классов и членов в порядке объявления.
2. Если класс является производным от виртуальных базовых классов, конструктор инициализирует указатели виртуальных базовых классов объекта.
3. Если класс имеет или наследует виртуальные функции, конструктор инициализирует указатели виртуальных функций объекта. Указатели виртуальных функций указывают на таблицу виртуальных функций класса, чтобы обеспечить правильную привязку вызовов виртуальных функций к коду.
4. Выполняет весь код в теле функции.

Конструкторы отрабатываются в порядке появления в коде объектов.

У каждого объекта существует жизненный цикл. В начале этого цикла (при создании объекта через `classname object;` или `object = new classname;`) вызывается конструктор. Деструктор всегда вызывается неявно, при удалении объекта через `delete object;` также деструкция происходит при завершении выполнения функции (или при выходе из области видимости), в порядке, обратном созданию объектов – но только «явно» заданных объектов, не через `*a = new`.

Локальные статические объекты – конструктор вызывается при первом определении объекта, деструктор вызывается после выполнения функции `main` но до вызова деструкторов внешних и внешних статических переменных.

Возврат объекта – объект уничтожается когда «необходимость в объекте отпадает». Зависит полностью от компилятора.

Метод класса – `static`, не передаётся указатель `this` на объект. Конструктор специальный метод – создаёт объект, не возвращает значения; имя совпадает с именем класса, могут принимать параметры. В **ЛЮБОМ** классе автоматически создаются два конструктора – конструктор по-умолчанию (не принимающий параметра) и конструктор копирования (вызывается при передаче и возврате по значению). Однако если в классе описан хотя бы один не стандартный конструктор, то конструктор-по-умолчанию не создаётся. Аналогично, можно явно задавать конструктор копирования, в случае если под объект так или иначе выделяются ресурсы. Деструктор также необходимо объявлять явно.

Чтобы акти

Конструктор не может быть константным, `volatile`, `virtual`, статичным. Не наследуется.

Проблема преобразования типов – не даёт нормально обработать объект при преобразовании его из одного типа в другой. Может возникнуть ситуация, что при создании объекта возникнет ошибка, и будет неясно создан объект или нет. Чтобы не гадать о существовании объекта, используют оператор `explicit` – он запрещает преобразование типов и неявный вызов конструктора.

Деструкторы – работают непосредственно с экземпляром класса, `this` передаётся. НЕ МОГУТ быть константными или статически, могут быть виртуальными. Если выделялись ресурсы под объект, то обязан быть описан деструктор, его задача – очищение ресурсов. Также на деструкторы возлагается функция разрыва связей.

1. Структура программы на языках C и C++.

Структура программы на языках C, C++.

Набор файлов, включающих директивы препроцессора, объявления.

Main обязательно должна присутствовать, не может вызывать саму себя.

Особенность: раздельная компиляция.

Чтобы собрать программу: создать заголовочные файлы. Они содержат: const, объявления переменных, типы объявленных функций (определять нельзя).

Определен если макрос, то включение заголовочного файла. Pragma once – не желательно использовать. Группу файлов с единым заголовочным файлом, объединяют в библиотеки.

Ссылка (кличка) – механизм передачи параметров в функцию и возврата из функции.

Перегрузка функций – множество функций с одним и тем же именем (список параметров разный)

```
#include "f.h"

int main(int argc, char* argv[])
{
    return 0;
}
```

Для избегания многократного объявления:

```
#pragma once или

#ifndef G_H
#define G_H

...
#endif
```

Отличия C++ от C:

- Классы и шаблоны
- Перегрузка функций
- Операторы new и delete
- Обработка исключений через throw/catch

9. «Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr.

Использование weak_ptr на примере паттерна итератор.

UNIQUE_PTR (единственный, кот. может держать список элементов)

unique_ptr запрещает копирование.

```
std::unique_ptr<int> x_ptr(new int(42));  
std::unique_ptr<int> y_ptr;
```

```
// ошибка при компиляции
```

```
y_ptr = x_ptr;
```

```
// ошибка при компиляции
```

```
std::unique_ptr<int> z_ptr(x_ptr);
```

Изменение прав владения ресурсом осуществляется с помощью вспомогательной функции std::move (которая является частью механизма перемещения).

```
std::unique_ptr<int> x_ptr(new int(42));  
std::unique_ptr<int> y_ptr;
```

unique_ptr обладают методами reset(), который сбрасывает права владения, и get(), который возвращает сырой (классический) указатель.

```
std::unique_ptr<Foo> ptr = std::unique_ptr<Foo>(new Foo);
```

```
// получаем классический указатель
```

```
Foo *foo = ptr.get();  
foo->bar();
```

```
// сбрасываем права владения
```

```
ptr.reset();
```

Как видно, unique_ptr обезопасил от неявных смен прав владений ресурсом.

shared_ptr

shared_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0. Как видно, система реализует одно из основных правил сборщика мусора.

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));

// после выполнения данной строчки, ресурс
// на который указывал ранее y_ptr (int(13)) освободится,
// а на int(42) будут ссылаться оба указателя
y_ptr = x_ptr;

std::cout << *x_ptr << "\t" << *y_ptr << std::endl;

// int(42) освободится лишь при уничтожении последнего ссылающегося
// на него указателя
```

Также как и unique_ptr, и auto_ptr, данный класс предоставляет методы get() и reset().

WEAK_PTR

weak_ptr не позволяет работать с ресурсом напрямую, но зато обладает методом lock(), который генерирует shared_ptr().

Использование `weak_ptr` на примере паттерна итератор

`std::weak_ptr` - очень хороший способ решить проблему [обвисшего указателя](#). Просто используя необработанные указатели, невозможно узнать, были ли освобождены ссылочные данные или нет. Вместо этого, позволяя `std::shared_ptr` управлять данными и поставляя `std::weak_ptr` пользователям данных, пользователи могут проверять достоверность данных, вызывая `expired()` или `lock()`.

Вы не могли бы сделать это только с помощью `std::shared_ptr`, потому что все экземпляры `std::shared_ptr` делят права собственности на данные, которые не удаляются до удаления всех экземпляров `std::shared_ptr`. Ниже приведен пример того, как проверять висячий указатель, используя `lock()`: