Оглавление

Предисловие редактора перевода 5
Предисловие 6

ЧАСТЬ 1

Общие понятия

Глава 1. Определение надежности программного обеспечения 9
Является ли Луна вражеской ракетой? 10
Что такое ошибка? 10
Что такое надежность? 13
Разве инженеры лучше программистов? 14
Надежность аппаратуры 15
Высокая стоимость программного обеспечения 17
Эксперименты в области программного обеспечения 18
Литература 20

Глава 2. Ошибки — их причины и последствия 21
Макромодель перевода 22
Микромодель перевода 27
Последствия ошибок 29
Литература 31

ЧАСТЬ 2

Проектирование надежного программного обеспечения

Глава 3. Основные принципы проектирования 32
Четыре подхода к надежности 33
Процессы проектирования 37
Сложность 40
Отношения с пользователем 41
Решение задачи 43
Правильность проектирования 46
Литература 48

Глава 4. Требования, цели и спецификации 49
Определение требований 49
Цели программного обеспечения 53
Понимание компромиссов 54
Постановка целей для программного обеспечения 59
Внешнее проектирование 62
Проектирование взаимодействия с пользователем 64
Подготовка внешних спецификаций 68

Проверка правильности внешних спецификаций 72
Планирование изменений 75
Литература 76

Глава 5. Архитектура системы 78
Уровни абстракции 79
Подсистемы, управляемые методом портов 83
Последовательные программы 89
Документация 89
Проверка правильности 90
Литература 90

Глава 6. Проектирование структуры программы 92
Независимость модулей 93
Прочность модулей 93
Сцепление модулей 96
Другие характеристики 99
Композиционный анализ 100
Пример композиционного анализа 102
Проверка правильности 112
Литература 113

Глава 7. Методы проектирования 114
Пассивное обнаружение ошибок 114
Активное обнаружение ошибок 118
Исправление ошибок и устойчивость к ошибкам 120
Изоляция ошибок 122
Обработка сбоев аппаратуры 124
Литература 126

Глава 8. Проектирование и программирование модуля 127
Внешнее проектирование модуля 127
Проектирование логики модуля 129
Структурное программирование и пошаговая детализация 134
Защитное программирование 141
Языки высокого уровня 146
Позиция программиста 147
Документация 149
Стандарты 151
Проверка правильности 151
Литература 154

Глава 9. Стиль программирования 155
Ясность программы 155
Использование языка 159
Микроэффективность 160
Комментарии 164
Определение данных 165
Структура программы 168
Литература 170

ЧАСТЬ 3

Тестирование программного обеспечения

Глава 10. Принципы тестирования 171
Основные определения 174
Философия тестирования 177
Интеграция модулей 180
Восходящее тестирование 180
Нисходящее тестирование 183
Модифицированный нисходящий метод 186
Метод большого скачка 188
Метод сандвича 188
Модифицированный метод сандвича 189
Что лучше? 189
Аксиомы тестирования 193
Литература 197

Глава 11. Тестирование модуля 199
Проектирование теста 199
Выполнение теста 210
Инструменты для тестирования модулей 211
Статический анализ управления 215
Средства периода выполнения 215
Тесты для квадратного уравнения 217
Литература 218

Глава 12. Тестирование внешних функций и комплексное тестирование . 219
Тестирование внешних функций 219
Метод функциональных диаграмм 221
Интеграция системы 229
Комплексное тестирование 233
Инструменты для тестирования внешних функций и комплексного тестирования 241
Планирование и управление при тестировании 245
Тестирование приемлемости 248
Тестирование настройки 248
Литература 249

Глава 13. Отладка 250
Как искать ошибку 250
Как исправлять ошибки 254
Инструменты отладки 255
Изучение процесса отладки 260
Литература 261

ЧАСТЬ 4

Дополнительные вопросы надежности программного обеспечения

Глава 14. Методы руководства и надежность 263
Организация и подбор кадров 264
Программист-библиотекарь 267
Бригады программистов 269
Принципы хорошего руководства 272
Литература 274

Глава 15. Языки программирования и надежность 275
Единообразие 278
Простота 280
Объявление данных 282
Типы и преобразования данных 284
Процедуры и области доступности данных 287
Управляющие структуры 290
Действия с данными 292
Обнаружение ошибок при компиляции 293
Обнаружение ошибок при выполнении программ 295
Литература 296

Глава 16. Архитектура ЭВМ и надежность 298
Структура памяти 300
Структура программы 304
Средства отладки 306
Машины с языком высокого уровня 307
Литература 308

Глава 17. Доказательство правильности программ 310
Метод индуктивных утверждений 311
Что можно и чего нельзя доказать с помощью доказательств . . . 319
Формальные и автоматические доказательства 322
Другие применения методов доказательства правильности 324
Литература 327

Глава 18. Модели надежности 329
Модель роста надежности 330
Другие вероятностные модели 335
Статистическая модель Миллса 336
Простые интуитивные модели 338
Модели сложности 340
Литература 342

Глава 19. Инструментальные системы 344
Библиотеки обеспечения разработки 344
Средства проектирования 351
Литература 353
Предметный указатель 354

Предисловие редактора перевода

Предлагаемая вниманию читателя книга Гленфорда Дж. Майерса выгодно отличается от других книг по технологии программирования тем, что в ней из всего разнообразия характеристик программы выделяется одна, но такая, которая позволяет с единых позиций изложить весь сложный комплекс проблем, связанных с радикальным снижением затрат на производство, внедрение и использование программного обеспечения.

Это первая в мировой литературе книга, целиком посвященная надежности программного обеспечения и не без оснований претендующая на освещение практически всей совокупности факторов, существенно влияющих на этот важнейший показатель. Автор настаивает на том, что повышение надежности — самый выгодный путь к снижению общих затрат, связанных с программным обеспечением. Выбрав надежность основным критерием качества окончательного программного продукта, автор получает точный ориентир для отбора технологических рекомендаций, составляющих сущность книги.

Роль автора как крупного специалиста, осознавшего и пропагандирующего такой взгляд на проблемы рационализации производства программного обеспечения, достаточно заметна. К настоящему времени им написано уже пять монографий по этим вопросам, среди которых вышедшая в 1979 году книга, специально посвященная искусству тестирования.

Как это всегда бывает в работе такого объема и новизны, автору не везде удалось одинаково убедительно и точно обосновать свои утверждения и оценки. Однако в целом книга получилась достаточно строгой, очень информативной и вполне доступной для первого чтения. Она будет полезна всем причастным к производству программ— от начинающих до руководителей программистских коллективов. Много ценного найдут в ней и пользователи, и заказчики программных продуктов.

В. Ш. Кауфман

Посвящается Дженнифер, Энди и Джефу

Предисловие

В конце 60-х годов пристальное внимание прессы привлек тот факт, что вычислительные машины могут делать ошибки, которые способны влиять на нашу жизнь. В сообщении под характерным заголовком «Тупица компьютер!» описывался случай, когда покупатель, получив из универмага счет на 0.00 долларов, пытался сообщить об этой ошибке в магазин, но продолжал без конца получать напоминания от ЭВМ с предупреждением закрыть его счет, если задолженность не будет погашена. В случаях такого рода проще всего послать по почте счет на 0.00 долларов или, как это сделал покупатель, сообщить о недоразумении в финансовый раздел «Нью-Йорк таймс».

Такие недоразумения, конечно, нежелательны, но ущерб от них обычно не выходит за рамки неприятностей для отдельных лиц. Я бы скорее предпочел вступить в дискуссию с ЭВМ универмага, чем очутиться в поезде, который управляющая им ЭВМ пытается разогнать до 1000 км в час вместо 100 из-за ошибки в программе. Мы живем сегодня в мире, где подобная ситуация уже возможна. Поскольку обработка данных затрагивает нашу жизнь во все большей степени, ошибки ЭВМ могут теперь иметь такие последствия, как нанесение материального ущерба, нарушение секретности, оскорбление личности и даже смерть.

Эта книга о том, как решить проблему ненадежности программного обеспечения. Исследуются все аспекты производства программных систем. Предлагаемые решения представлены в виде *принципов* и *методов*. Принципы — это основные стратегические направления в изготовлении надежного программного обеспечения, а методы — это более мелкие тактические решения различных аспектов проблемы ненадежности.

Книга состоит из четырех больших частей. Задача первой части — определить понятие надежности программного обеспечения, рассмотреть основные причины ошибок в программных системах и вызвать у читателя интерес к другим частям книги. В части 2 излагаются принципы и методы, используемые при проектировании надежного программного обеспечения. Слово «проектирование» понимается в широком смысле и обозначает все процессы от определения требований к программной системе до написания отдельных предложений программы.

Часть 3 посвящена обширной области тестирования программ-

ного обеспечения, которое поглощает значительную долю затрат на обработку данных и о котором большинство знает все еще слишком мало. Хотя ключом к надежности служит четкость проектирования, тестирование играет важную роль в обеспечении надежности программного обеспечения. В книге рассматривается целый ряд проверенных принципов и методов тестирования.

Профессионалы знают, что на надежность влияют и другие факторы; многие из них рассматриваются в части 4. На надежности программирования, например, существенно сказываются организационная структура, подбор кадров и взаимоотношения сотрудников, планы руководства и возможности их выполнения, средства программирования, обстановка, в которой проводится работа, и т. п. Рассматриваются также проблемы, связанные с современными языками программирования и архитектурой ЭВМ, и предлагаются решения этих проблем. Уделено внимание также математическому доказательству правильности программ и методам расчета (моделям) надежности.

О надежности программного обеспечения говорится только с точки зрения ошибок в нем самом. Совершенно не затронута важная тема использования программ для исправления ошибок аппаратуры или их последствий, таких, как отказы устройств ввода-вывода. Хотя эта тема, очевидно, должна рассматриваться при анализе надежности системы в целом, она не вошла сюда потому, что заслуживает отдельной книги и, кроме того, довольно хорошо разработана.

Эта книга должна быть полезна всем, кого интересует производство надежного программного обеспечения. Наибольшую пользу из нее смогут извлечь те, кто непосредственно связан с этим процессом: программисты, аналитики, персонал групп тестирования, руководители отделов программирования и обработки данных. Разработчиков языков программирования могут заинтересовать разделы о языках, стиле программирования и архитектуре ЭВМ. Пользователи программного обеспечения, в особенности те, кто отвечает за покупку программных систем или подготовку контрактов на разработку новых систем, познакомятся с идеями надежности и их влиянием на вычислительные системы. Весь материал книги поможет исследователям лучше понять проблемы надежности, оценить пользу прежних исследований и увидеть обещающие области новых изысканий.

Книга будет полезна и как справочник, и как учебник. Она познакомит студентов университетов со многими реальными проблемами разработки программного обеспечения. Ее можно использовать в качестве учебника по разработке программного обеспечения для старшекурсников или аспирантов по специальностям прикладного и теоретического программирования, в особенности если обучение будет сопровождаться решением практических задач, например учебным проектом. Я использовал весь этот материал

в курсах по надежности программного обеспечения для аспирантов в Институте системных исследований IBM и Политехническом институте Нью-Йорка.

Я благодарен многим коллегам из Института системных исследований IBM за ценные советы по материалу книги. В частности, очень полезны были конструктивная критика книги Р. Голдбергом и С. Х. Хаспелом, а также содержательные рекомендации Б. Г. Вейзенхоффера, К. Дж. Бонтемпо и Дж. Е. Флэнагана по материалу части 4. Я обязан отметить, что некоторые положения книги можно считать спорными; взгляды и мнения, высказанные по этим вопросам, принадлежат лично мне и не обязательно выражают мнения перечисленных выше людей или корпорации IBM.

Нью-Йорк, Н.-Й.
Апрель 1976

Гленфорд Дж. Майерс

ЧАСТЬ 1

Общие понятия

ГЛАВА 1

Определение надежности программного обеспечения

На сегодня самая серьезная проблема в области обработки данных — это проблема программного обеспечения. Внешне она проявляется в постоянных жалобах на то, что программное обеспечение, во-первых, дорого и, во-вторых, ненадежно. Большинство специалистов считает первый из недостатков во многом проявлением второго. Поскольку современное программное обеспечение по самой своей природе ненадежно, его тестирование и обслуживание требуют существенных расходов. Хотя эта книга посвящена в основном проблеме надежности программного обеспечения, вопросы его высокой стоимости также будут затрагиваться в той мере, в какой они связаны с надежностью.

Интересно отметить, что проблема надежности программного обеспечения (с позиций, весьма близких к современным) рассматривалась еще на заре применения вычислительных машин:

> «Те, кто регулярно программирует для быстродействующих электронных машин, знают на собственном опыте, что солидная доля подготовительного этапа работы на ЭВМ уходит на устранение ошибок, сделанных при составлении программы. С помощью здравого смысла и отладочных подпрограмм большинство ошибок удается найти и исправить достаточно быстро. Однако некоторые из них настолько неуловимы, что не поддаются обнаружению удивительно долгое время» [1].

Это наблюдение было опубликовано тремя английскими математиками в 1952 г. Хотя ошибки в программном обеспечении встречались и до 1952 г., это, кажется, первое признание проблемы надежности, т. е. того факта, что тестирование требует значительного времени, и даже после его завершения некоторые ошибки в программном обеспечении остаются необнаруженными.

ЯВЛЯЕТСЯ ЛИ ЛУНА ВРАЖЕСКОЙ РАКЕТОЙ?

Первое с чем мы сталкиваемся при анализе надежности программного обеспечения,— это проблема определения: что такое ошибка в программном обеспечении и что такое надежность программного обеспечения? Важно договориться о стандартном определении, чтобы избежать таких ситуаций, когда пользователь утверждает, что обнаружил в системе ошибку, а разработчик отвечает: «Нет, система так и была задумана».

Система раннего обнаружения баллистических снарядов Ballistic Missile Early Warning System должна наблюдать за объектами, движущимися по направлению к Соединенным Штатам, и, если объект не опознан, начать последовательность защитных мероприятий— от попыток установить с объектом связь до перехвата и, возможно, ответного удара. Одна из ранних версий системы ошибочно приняла подымающуюся Луну за снаряд, летящий над северным полушарием. Ошибка ли это? С точки зрения пользователя (Министерства обороны США) — да. С точки зрения разработчика системы — возможно, и нет. Разработчик может настаивать на том, что в соответствии с требованиями или спецификациями защитные действия должны быть начаты по отношению к любому движущемуся объекту, появившемуся над горизонтом и не опознанному как мирный летательный аппарат.

Дело в том, что разные лица по-разному понимают, что такое ошибка в программном обеспечении. Прежде чем приступить к обсуждению методов устранения таких ошибок, следует дать определение ошибки. Однако полезнее будет сначала проанализировать известные определения и указать на их слабые места, а не пытаться без всякой подготовки формулировать свое собственное определение.

ЧТО ТАКОЕ ОШИБКА?

Согласно одному из известных определений, программное обеспечение содержит ошибку, если его поведение не соответствует спецификациям. Это определение страдает существенным недостатком: неявно предполагается, что спецификации корректны. Такое предположение если и бывает справедливым, то редко; подготовка спецификаций — один из основных источников ошибок. Если поведение программного продукта не соответствует его спецификациям, ошибка, вероятно, имеется. Однако, если система ведет себя в соответствии со спецификациями, мы не можем утверждать, что она не содержит ошибок.

Второе известное определение гласит, что программное обеспечение содержит ошибку, если его поведение не соответствует спецификациям при использовании в установленных при разработке пре-

делах. В действительности это определение еще хуже первого. Если система случайно используется в непредусмотренной ситуации, ее поведение должно оставаться разумным. Если это не так, она содержит ошибку. Рассмотрим авиационную систему диспетчеризации, которая прослеживает и координирует движение самолетов над некоторым географическим районом. Предположим, что, согласно спецификациям, система должна управлять движением до 200 самолетов одновременно. Но однажды по непредвиденным обстоятельствам в районе появился 201 самолет. Если поведение системы неразумно — скажем, она забывает об одном из самолетов или выходит из строя,— система содержит ошибку, хотя она используется вне пределов, установленных при проектировании.

Согласно третьему возможному определению, ошибка имеется тогда, когда программное обеспечение ведет себя не в соответствии с официальной документацией и поставленными пользователю публикациями. К несчастью, это определение также страдает несколькими изъянами. Возможны ситуации, когда программное обеспечение ведет себя в *соответствии* с официальными публикациями, но ошибки все-таки имеются, так как они содержатся и в программе, и в публикациях. Другая проблема возникает вследствие тенденции описывать в руководствах для пользователей только ожидаемую и планируемую работу с системой. Предположим, что мы имеем руководство для пользователей системы разделения времени, в котором говорится: «Чтобы дать новую команду, нажмите один раз клавишу «внимание» и напечатайте эту команду». Предположим, что пользователь случайно нажимает клавишу «внимание» дважды и система программного обеспечения выходит из строя, потому что ее разработчики не предусмотрели такой ситуации. Система, очевидно, содержит ошибку, но мы не можем утверждать, что она ведет себя не в соответствии с публикациями.

Согласно последнему определению, которое также иногда используется, ошибка определяется как неспособность системы действовать в соответствии с исходным контрактом или перечнем требований пользователя. Хотя это определение лучше трех предыдущих, оно также не без недостатков. Если, согласно требованиям пользователя, система должна обеспечивать среднее время между отказами из-за ошибки в программном обеспечении на уровне 100 часов, а для действующей системы этот показатель равен 150 часам, система все же имеет ошибки (поскольку ее среднее время между отказами конечно), даже несмотря на то, что она превышает требования пользователя. Кроме того, письменно зафиксированные требования пользователя редко детализированы настолько, чтобы описывать желаемое поведение программного обеспечения при всех мыслимых обстоятельствах.

Есть, однако, разумное определение ошибки в программном обеспечении, разрешающее перечисленные выше проблемы:

В программном обеспечении имеется ошибка, если оно не выполняет того, что пользователю разумно от него ожидать. Отказ программного обеспечения — это проявление ошибки в нем.

Я предвижу две разные реакции на это определение. Реакцией пользователя будет: «Точно!» Разработчик программного обеспечения может возразить: «Определение непрактично. Откуда мне знать, что пользователю разумно ожидать?» Дело в том, что если разработчик программного обеспечения хочет спроектировать удачную систему, то он всегда должен понимать, что именно ее пользователям «разумно ожидать».

Слово «разумно» употреблено в определении для того, чтобы исключить ситуации, когда, например, к терминалу информационно-поисковой системы публичной библиотеки подходит человек и просит определить объем своего вклада в местном банке. Под «пользователем» понимается любой человек, вводящий информацию в систему, исследующий выходные данные или взаимодействующий с системой каким-то иным образом. Большая система программного обеспечения (прикладные программы, операционная система, компиляторы, обслуживающие программы и т. д.) будет иметь много различных пользователей, в частности общающихся с системой через удаленные терминалы или по почте (зачастую это лица, не знакомые с вычислительными машинами или программированием). Кроме них, пользователями будут программисты (как системщики, так и прикладники) и операторы системы.

Теперь читатель должен быть уже готов уловить такую тонкую особенность надежности программного обеспечения: *ошибки в программном обеспечении не являются внутренним его свойством.* Это значит, что, как бы долго и пристально мы ни разглядывали (или тестировали, или «доказывали») программу (либо программу и ее спецификации), мы никогда не сможем найти в ней все ошибки. Мы можем обнаружить некоторые ошибки (например, бесконечный цикл), но, по самой природе ошибок в программном обеспечении, мы никогда не можем рассчитывать найти их все. Коротко говоря, наличие ошибок — функция как самого программного обеспечения, так и ожиданий его пользователей.

Хотя мы примем это определение ошибки в программном обеспечении за основу, в нем имеется по крайней мере один недостаток. Рассмотрим ситуацию, когда в ответ на запрос системы резервирования авиабилетов «ВВЕДИТЕ НОМЕР РЕЙСА И ДАТУ» служащий набирает на терминале «239, МАЙ 10». Система отвечает сообщением «ОШИБКА В ДАТЕ», потому что способна воспринимать дату только в форме «10 МАЯ». Является ли это ошибкой в программном обеспечении? Согласно нашему определению, можно считать и так, но я утверждаю, что это не ошибка; это скорее относится к проблеме человеческих факторов. Мы могли бы расширить наше определение,

чтобы оно охватывало и ситуации вроде этой и относило бы их к ошибкам в проектировании человеко-машинного взаимодействия; где-то, однако, надо подвести черту, поэтому я предпочел исключить подобные проблемы из нашего определения. Человеческие факторы, конечно, исключительно важная сторона проектирования системы, но это — отдельная область исследования, она уже рассматривалась в литературе [2]. К этому вопросу мы еще вернемся в гл. 2 и 4.

ЧТО ТАКОЕ НАДЕЖНОСТЬ?

Второй термин, который следует определить,— это *надежность программного обеспечения.* Мы опять начнем с исследования известного определения, согласно которому надежность есть вероятность того, что при функционировании системы в течение некоторого периода времени не будет обнаружено ни одной ошибки. Основной недостаток такого определения — это то, что в нем не учтено различие между ошибками разных типов. Рассмотрим авиационную систему диспетчеризации с двумя ошибками в программном обеспечении: из-за одной теряется след самолета, а другая состоит в том, что в сообщении оператору неправильно печатается одно слово (например, ТРАНСАММЕРИКАНСКИЙ вместо ТРАНСАМЕРИКАНСКИЙ). По своим последствиям эти ошибки далеко не одинаковы, поэтому надежность должна быть определена как функция не только частоты ошибок, но и их *серьезности.* В соответствии с этим дадим следующее определение:

Надежность программного обеспечения есть вероятность его работы без отказов в течение определенного периода времени, рассчитанная с учетом стоимости для пользователя каждого отказа.

Таким образом, надежность программного обеспечения является функцией воздействия ошибок на пользователя системы; она не обязательно прямо связана с оценкой «изнутри» программного обеспечения. Даже крупный просчет в проектировании может оказаться не слишком заметным для пользователя. С другой стороны, как будто бы тривиальная ошибка может иметь катастрофические последствия. Например, первый запуск космического корабля на Венеру потерпел неудачу из-за того, что в операторе DO программы на Фортране была пропущена запятая.

Надежность не является внутренним свойством программы; она во многом связана с тем, как программа используется. Слово «вероятность» в определении, по существу, означает вероятность того, что пользователь не введет в систему некоторый конкретный набор данных, выводящий систему из строя.

В этой книге термин «надежность» используется довольно свободно. Иногда он просто означает некоторую количественную меру отсутствия ошибок в программе. Смысл, вкладываемый в это слово, всегда будет очевиден из контекста.

РАЗВЕ ИНЖЕНЕРЫ ЛУЧШЕ ПРОГРАММИСТОВ?

Типичен вопрос, с которым руководитель отдела обработки данных после установки новой вычислительной системы обращается к изготовителю: «Как только мы установили систему, сама вычислительная машина (ее аппаратура) сразу заработала хорошо, а вот операционная система была полна ошибок. Разве ваши инженеры лучше ваших программистов? Почему вы не проверяете свое программное обеспечение так же тщательно, как, видимо, проверяете аппаратуру?»

Вопрос совершенно закономерен, но ответить на него довольно сложно. Прежде всего, изготовитель ЭВМ, вероятно, затратил значительно больше денег на проверку программного обеспечения, чем на проверку аппаратуры, и, очевидно, ошибки в программном обеспечении нельзя просто отнести за счет недостатка квалификации программистов. Главная причина в том, что программное обеспечение по сути своей значительно сложнее аппаратуры. Входные данные для операционной системы неизмеримо разнообразнее, чем для центрального процессора. Например, комплект руководств для пользователя операционной системы IBM OS/VS2 и связанного с ней программного обеспечения по объему в сотни раз превосходит руководство, описывающее архитектуру центрального процессора Системы 370.

В основном входные данные центрального процессора (ЦП) — это поток машинных команд. При обработке типичной машинной команды ЦП получает саму команду, размеры которой фиксированы (обычно 16—36 бит), и несколько коротких строк битов (операнды) в памяти, причем предполагается, что входные данные имеют жесткий формат. Более того, значения операндов не оказывают существенного влияния на выполнение большинства команд, и во многих ЦП выполнение команды не зависит от выполнения предыдущих. По сравнению с этим входные данные системы программного обеспечения значительно разнообразнее; обычно нужно учитывать все, что пользователь может захотеть ввести в систему. К тому же в большинстве случаев обработка элемента входных данных зависит от предыстории.

Второе существенное различие — это отношение к возможным применениям. Аппаратура в значительной степени независима от применений. Один и тот же ЦП можно использовать и для реализации системы резервирования мест в гостинице, и для медицинской диагностической системы, и для моделирования мировой экономики. Напротив, большинство программных систем весьма чувствительно к конкретным приложениям.

Третье различие касается природы компонент (строительных блоков) сравниваемых продуктов. Аппаратура состоит из хорошо известных строительных блоков: памяти, регистров, сумматоров, шин и

вентилей. Любой толковый учебник по логическому проектированию описывает оптимальные способы конструирования этих устройств. Разрабатывая процессор, удовлетворяющий определенным требованиям к стоимости и производительности, инженер интересуется главным образом размерами, скоростью и конфигурацией этих устройств. Строительные блоки, доступные разработчику программного обеспечения, гораздо примитивнее по отношению к конечному продукту. Для него такими блоками являются операторы программы, наборы алгоритмов и фрагменты других систем программного обеспечения.

Цель нашего сопоставления — не сравнивать квалификацию программиста и инженера, а показать, что обеспечение надежности является для разработчика программного обеспечения гораздо более сложной задачей, чем для инженера. Однако это не может служить оправданием ненадежности программного обеспечения; на самом деле создатель программного обеспечения должен многому научиться у инженера. Методы, используемые для создания надежных программ, превращают разработку программного обеспечения в достаточно строго регламентированную деятельность, аналогичную инженерному проектированию.

НАДЕЖНОСТЬ АППАРАТУРЫ

Для лучшего понимания надежности программного обеспечения стоит сравнить ее с надежностью аппаратуры. Возможны три причины отказа некоторого устройства: ошибка проектирования, производственный дефект и сбой. Ошибка проектирования — это дефект, который с самого начала присутствует в каждом экземпляре изделия. Это ошибка такого рода, когда, скажем, часть памяти ЦП оказывается недоступной из-за логической ошибки в проектировании схемы адресации. Производственный дефект — это ошибка, имеющаяся в одном или нескольких экземплярах изделия из-за того, что эти конкретные экземпляры изготовлены неверно. Например, производственные дефекты могут быть вызваны плохой пайкой или неправильным соединением проводов. Сбои — это дефекты, первоначально не присутствующие в изделии, но возникающие в процессе его функционирования вследствие некоторых физических явлений, например, ухудшения параметров, вызванного молекулярными дефектами, нагревом, влажностью, трением, радиацией и т. д. Примерами сбоев могут быть физический износ переключателя вследствие трения, размагничивание магнитного сердечника при перегреве или выход из строя интегральной схемы вследствие медленного ухудшения ее герметизации.

На рис. 1.1 показано изменение со временем частоты отказов типичного устройства. Можно проиллюстрировать кривую на примере автомобиля. Средний автомобиль вначале ненадежен из-за

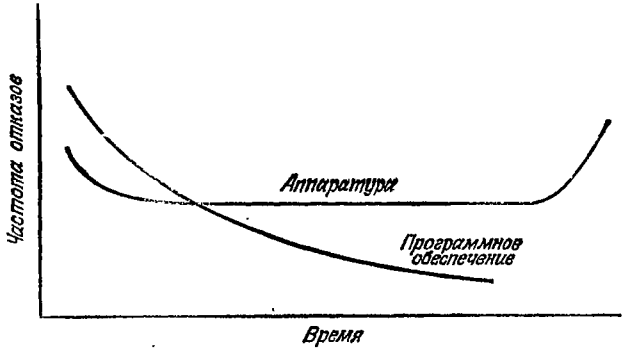


Рис. 1.1. Различие между надежностью аппаратуры и программного обеспечения.

ошибок проектирования и производственных дефектов, таких, как неправильная регулировка, недозатяжка болтов, нехватка деталей. Эти дефекты быстро обнаруживаются, ликвидируются, и машина достигает своего пика надежности. После первоначального этапа идет период относительного постоянства частоты поломок, во время которого детали выходят из строя случайным образом, обычно в соответствии с пуассоновским распределением. По мере того как заканчивается срок службы автомобиля, поломки перестают быть случайными и возникают с возрастающей частотой. Исследования надежности аппаратуры концентрируются в первую очередь на промежуточном периоде случайных сбоев.

Надежность программного обеспечения существенно отличается от надежности аппаратуры. Программы не изнашиваются, поломка программы невозможна. Более того, производственные дефекты (такие, как ошибка при копировании системы на ленту во время ее переноса) не имеют значения, поскольку они относительно редки и быстро обнаруживаются. Таким образом, ненадежность программного обеспечения — следствие исключительно ошибок проектирования, т. е. ошибок, внесенных в процессе разработки. В условиях, когда ошибки исправляются тотчас же, как только они обнаруживаются (причем они никогда не появляются повторно), изменение надежности программного обеспечения обычно соответствует кривой, изображенной на рис. 1.1. Подчеркнем, что эта кривая отражает предположение, что при исправлении обнаруженных ошибок не вносятся новые. Это предположение обычно не справедливо.

Рис. 1.1 отражает тот факт, что поведение надежности программного обеспечения и аппаратуры по сути своей различно. Надежность аппаратуры определяется во многом случайными сбоями, надежность программного обеспечения — скрытыми в нем ошибками.

Чтобы подчеркнуть различия между ошибками в программном обеспечении и аппаратуре, рассмотрим поведение сбоев аппаратуры

и ошибок в программе в зависимости от входных данных и времени. Обычно сбои аппаратуры не зависят от обрабатываемых системой данных. Если схема двоичного сумматора разрушается и вот-вот выйдет из строя, конкретные значения обрабатываемых данных не влияют на сбой. Проявление ошибок в программе, напротив, в высшей степени зависит от входных данных. Такая ошибка обнаруживается в некоторый определенный момент именно потому, что в этот момент обрабатывается ранее не встречавшаяся последовательность входных данных.

Частота сбоев аппаратуры существенно зависит от времени. Все физические устройства имеют некоторый срок службы, по истечении которого эта частота начинает быстро расти. Частота, с которой обнаруживаются ошибки в программном обеспечении, хотя иногда и кажется зависящей от времени, в действительности является функцией входных данных и состояния системы. Бо́льшую часть времени жизни аппаратного устройства его сбои случайны и описываются пуассоновским законом распределения; на этом основаны многие исследования. Ошибки в программе проявляются как систематические, далеко не случайные события.

Природа надежности аппаратуры и программного обеспечения подробнее рассматривается в гл. 18 в связи с попытками применить математический аппарат описания надежности аппаратуры к программному обеспечению.

ВЫСОКАЯ СТОИМОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Как уже говорилось, две основные проблемы в производстве программного обеспечения — его чрезмерно высокая стоимость и низкая надежность. Предлагаемые решения проблемы стоимости обычно сводятся к попытке поднять «производительность программиста», изобрести инструменты и методы, позволяющие ему работать быстрее.

Рис. 1.2 иллюстрирует относительные затраты на программное обеспечение в течение его жизненного цикла для большинства типичных крупных разработок. На долю сопровождения (устранения ошибок и внесения небольших изменений после установки системы) и тестирования приходится почти 75% затрат. Например, среднегодовая стоимость сопровождения программного обеспечения военной системы защиты SAGE после 10 лет ее функционирования составляет примерно 20 млн. долларов при первоначальной стоимости разработки в 250 млн. долларов [3[. Как правило, 60 (иногда до 76) процентов затрат на очередной выпуск операционной системы OS/360 вкладывалось фирмой IBM уже после того, как система становилась доступной пользователям. В обоих случаях речь идет только о затратах на сопровождение. Стоимость сопровождения и тестирования вместе в каждом из этих случаев, вероятно, составляет более 80%

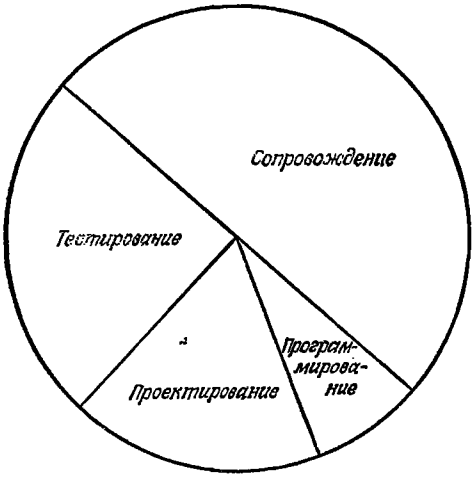


Рис. 1.2. Типичное распределение стоимости программного обеспечения.

всех затрат. Хотя никто не знает ежегодных расходов на тестирование и сопровождение в мире, достоверно известна организация (а именно ВВС США), затратившая в 1972 г. на тестирование программного обеспечения более 750 млн. долларов [4].

Теперь должно быть ясно, что высокая стоимость программного обеспечения — во многом следствие низкой надежности. При увеличении производительности программиста (если измерять ее только скоростью разработки и кодирования программы) стоимость существенно *не* уменьшается. Попытки увеличить производительность программиста могут в некоторых случаях даже повысить стоимость. Наилучший путь решительного сокращения стоимости программного обеспечения — в уменьшении стоимости его тестирования и сопровождения. А это может быть достигнуто не за счет инструментов, призванных увеличить скорость программирования, а лишь в результате разработки средств, повышающих корректность и четкость при создании программного обеспечения.

ЭКСПЕРИМЕНТЫ В ОБЛАСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой книге неоднократно будут высказываться утверждения об «оптимальных» методиках, инструментах и т. п., которые следует применять для достижения надежности. Однако вследствие исключительной сложности проведения экспериментов в области разработки программного обеспечения для большинства заключений такого рода невозможно привести солидные экспериментальные подтверждения. Причины этой сложности перечислены ниже.

1. Разработка программного обеспечения — трудоемкий (точнее, «мыслеемкий») процесс. Эксперименты в этой области во многом связаны с людьми. Мнения большинства исследователей сходятся в том, что самый трудный объект для экспериментирования — человек, поскольку он (или она) — очень сложная и во многом не познанная система. Например, в экспериментах, задуманных с целью сравнить пакетный и диалоговый режимы работы, Сакман неожиданно обнаружил, что отношение показателей разных программистов,

решавших одну и ту же задачу, достигало 28 : 1 [5]. Зачастую индивидуальные различия такого рода существеннее, чем влияние исследуемых факторов.

2. Эксперименты по разработке программного обеспечения недоступно дороги. Для исследования одного фактора, такого, как влияние структурного программирования, мы могли бы начать работу над двумя проектами средней сложности, идентичными во всех отношениях, за исключением того, что в одном случае используется структурное программирование, а в другом — нет. Работа могла бы занять один год и стоила бы вместе с машинным временем несколько миллионов долларов. Затем для чистоты эксперимента следовало бы повторить ту же работу несколько раз. Затем следовало бы повторить ее, изменяя другие факторы. Полная продолжительность эксперимента (исследование только одного фактора разработки программного обеспечения) может составить несколько лет и стоить от 10 до 20 млн. долларов. Даже если бы имелись деньги на такой эксперимент, возникла бы проблема найти несколько сотен профессиональных программистов, которые посвятили бы ему столько лет. А тем временем кто-нибудь придумает кое-что получше структурного программирования.

3. На разработку программного обеспечения влияют сотни факторов, многие из них еще не выявлены. Более того, эти факторы не независимы. Например, нашумевший проект IBM для «Нью-Йорк таймс» часто рассматривается как свидетельство достоинств структурного программирования и бригады ведущего программиста. Невыясненным остается, однако, вопрос: какой процент успеха — результат этих методов, а какой — следствие других факторов, таких, как исключительно высокий класс занятых в проекте программистов?

4. Те немногие тщательно спланированные эксперименты, которые были поставлены, обычно касались индивидуальных работ — курсовых работ и небольших программ. Следует быть весьма осторожным, предполагая, что старшекурсники программируют так же, как профессионалы, и экстраполируя результаты индивидуальной работы над небольшим проектом на сложные коллективные разработки.

Эти трудности перечислены здесь не для того, чтобы оправдывать или расхолаживать экспериментаторов. Наша цель — показать, что, хотя положительное влияние на надежность описанных в книге методов и замечено, экспериментальных данных, подтверждающих это, почти нет. Конечно, можно отказаться применять эти принципы и методы, пока наконец не будут получены убедительные экспериментальные доказательства, но в таком случае есть риск еще долго не увидеть сколько-нибудь значительного повышения надежности программного обеспечения.

ЛИТЕРАТУРА

1. Brooker R. A., Gill S., Wheeler D. J. The Adventures of a Blunder, *Mathematical Tables and Other Aids to Computation*, 6 (38), 112—113 (1952).
2. Martin J. Design of Man-Computer Dialogues. Englewood Cliffs, N. J.: Prentice-Hall, 1973.
3. Thayer R. H. Rome Air Development Center R and D Program in Computer Language Controls and Software Engineering Techniques, RADC-TR-74-80, Griffiss Air Force Base, Rome, N. Y., 1974.
4. Shelly M. Computer Software Reliability, Fact or Myth? TR-MMER/RM-73-125, Hill Air Force Base, Utah, 1973.
5. Сакман Г. Решение задач в системе человек — ЭВМ. Пер. с англ.— М.: Мир, 1973.
6. Baker F. T. Chief Programmer Team Management of Production Programming, *IBM Systems Journal*, 11 (1), 56—73 (1972).

ГЛАВА 2

Ошибки — их причины и последствия

В 1971 г. во Франции проводился крупный метеорологический эксперимент. Было запущено 115 шаров-зондов с измерительными приборами в верхние слои атмосферы, а также спутник для пересылки данных между шарами и наземными станциями. Шары умели реагировать на две команды: команду чтения, по которой данные пересылались от шара спутнику, и команду ликвидации для взрыва помещенного в шаре заряда, если шар собьется с курса. К несчастью, в программном обеспечении системы была ошибка. В результате вместо команды чтения была послана команда ликвидации, уничтожившая 72 шара, находившихся в поле зрения спутника.

Прежде чем разрабатывать методы повышения надежности, позволяющие избежать таких ошибок стоимостью в миллионы долларов, следует понять их причины. Это может помочь в попытках ответить на такие вопросы, как: «Нет ли чего-либо уникального в профессии программиста, такого, что привлекает к ней людей, по своей природе особенно склонных совершать ошибки? Не обладает ли программное обеспечение какими-то внутренними свойствами, объясняющими его особую подверженность ошибкам? Способны ли мы как-то влиять на эти свойства или они нам неподвластны?»

Один из методов, часто применяемых при исследовании причин ошибок, состоит в том, чтобы собрать данные об ошибках и затем разбить их по категориям, подтверждающим, например, что 17% всех ошибок — это ошибки сопряжения, 22% ошибок — следствие неправильного выбора последовательности операторов программы и т. д. В соответствии с другим методом ошибки группируются по типу предложений языка программирования, в которых они совершены: например, оказывается, что 29% всех ошибок приходится на условные предложения, 13% — на объявления данных и т. д.

Хотя результаты таких исследований читаются с интересом, они не особенно полезны при выяснении того, как же избежать ошибок. Например, из первой серии упоминавшихся результатов можно сделать вывод, что необходимо правильно выбирать последовательность предложений программы. Это не очень ценное наблюдение, поскольку ни один программист не станет умышленно располагать предложе-

ния неправильно. Вторая серия результатов, показывающая, что в условных предложениях ошибки встречаются чаще всего, могла бы привести к неправильному заключению, что условные предложения лучше в программах вообще не употреблять.

Польза подобных исследований ограничена, поскольку они занимаются симптомами, а не причинами ошибок. Именно понимание первопричин ошибок и возможность увязать их с процессом создания программного обеспечения необходимы прежде всего.

Единственная важная причина ошибок в программном обеспечении — неправильный перевод информации (из одного представления в другое).

Создание программного обеспечения можно описать просто как ряд процессов перевода, начинающих с задачи и заканчивающих большим набором подробных инструкций, управляющих ЭВМ при решении этой задачи. Другими словами, программирование — это решение задач, а программное обеспечение — это совокупность информационных элементов (но не физических объектов), описывающих решение задачи. Создание программного обеспечения в этом случае — просто совокупность процессов трансляции, т. е. перевода исходной задачи в различные промежуточные решения, пока наконец не будет получен подробный набор машинных команд. Когда не удается полно и точно перевести некоторое представление задачи или решения в другое, более детальное, тогда и возникают ошибки в программном обеспечении.

Прежде чем решать проблему (скажем, проблему надежности), ее следует понять. Понимание того, что именно ошибки перевода являются причиной ошибок в программе, чрезвычайно важно, так как это — ключ к пониманию проблемы надежности.

МАКРОМОДЕЛЬ ПЕРЕВОДА

Чтобы подробнее исследовать проблему ошибок в программном обеспечении, рассмотрим различные типы процессов перевода при его создании. Модель разработки программного обеспечения изображена на рис. 2.1. Прежде чем приступить к ее анализу, следует сделать несколько замечаний. В эту модель не включен процесс тестирования программы. Представлены только два шага проектирования, хотя в действительности их обычно бывает (и должно быть) больше. Задача модели — не в том, чтобы перечислить все рекомендуемые действия (это делается в частях 2 и 3), а показать лишь основные типы возникающих процессов перевода. Кроме того, мы не учитываем пока взаимного перекрытия процессов и циклов обратной связи (например, вследствие ошибок, обнаруженных во время выполнения одного из процессов, часто требуется вернуться назад и скорректировать ранее выполненную работу).

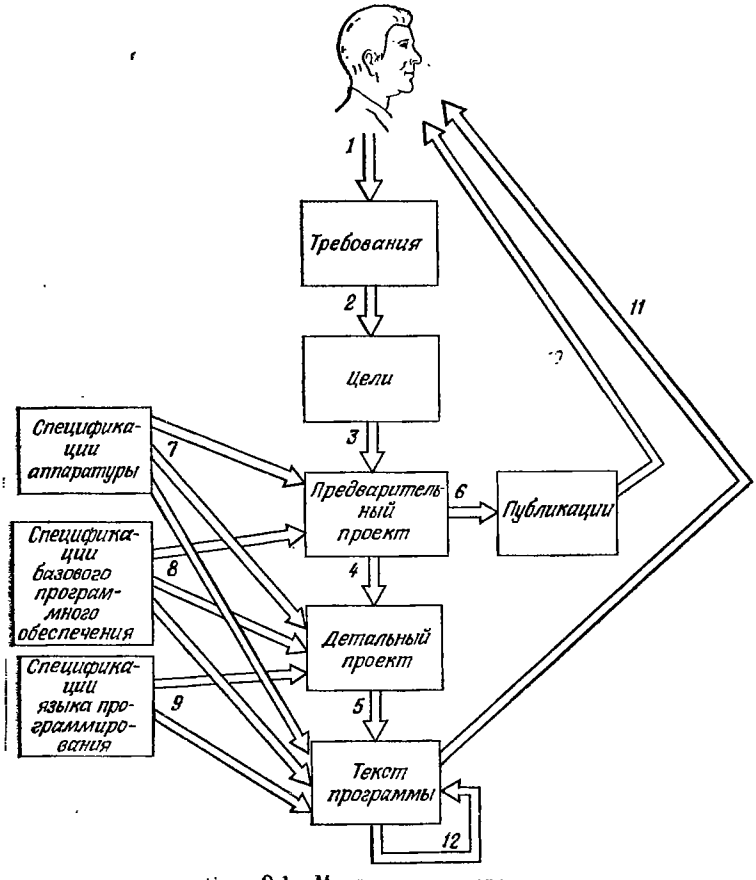


Рис. 2.1. Макромодель перевода.

Чтобы лучше понять проблему перевода, рассмотрим кратко каждый из процессов, изображенных на рис. 2.1.

1. Этот процесс начинается с разработки описания решаемой задачи. Такое описание имеет вид перечня требований пользователя. В некоторых случаях пользователь составляет этот перечень сам. В других случаях это делает разработчик программного обеспечения либо в результате бесед с пользователем, либо исследуя его потребности, либо самостоятельно оценивая эти потребности в будущем, либо комбинируя перечисленные методы.

В любом случае здесь имеются обширные возможности для ошибок; например, пользователь может не суметь адекватно выразить свои потребности, они могут быть неверно поняты либо учтены не в

полном объеме. Ясно, что ошибки, возникающие на этом уровне, обходятся чрезвычайно дорого.

2. Второй процесс состоит в переводе требований пользователя в цели программы. Хотя на этом шаге объем перевода невелик, здесь требуется явно выделить и оценить довольно много компромиссных решений (перечисленных в гл. 4). Ошибки на этом шаге возникают, когда неверно интерпретируются требования, не удается выявить все требующие компромиссных решений проблемы или приняты неправильные решения, а также в случае, когда не сформулированы цели, необходимые, но не поставленные явно в требованиях пользователя.

3. Третий шаг связан с преобразованием целей программы в ее внешние спецификации, т. е. точное описание поведения всей системы с точки зрения пользователя. По объему перевода это самый значительный этап разработки программного обеспечения. Так что он больше всего подвержен ошибкам — они бывают и наиболее серьезными, и наиболее многочисленными.

4. Четвертый шаг представляет собой несколько процессов перевода — от внешнего описания готового продукта до получения детального проекта, описывающего множество составляющих программу предложений, выполнение которых должно обеспечить поведение системы, соответствующее внешним спецификациям. Этот шаг включает такие процессы, как перевод внешнего описания в структуру компонент программы (например, модулей) и перевод каждой из этих компонент в описание процедурных шагов (например, в блок-схемы). Поскольку нам приходится иметь дело со все большими объемами информации, шансы внесения ошибок становятся чрезвычайно высокими.

5. Последний процесс проектирования — перевод описания логики программы в предложения языка программирования. Хотя на этом шаге часто делается много ошибок, они обычно относительно мелкие, легко обнаруживаются и корректируются. (Однако, как вы увидите ниже, это *не всегда* так.)

На этом шаге возникает также и другой процесс трансляции: перевод представления программы на языке программирования в объектный (выполняемый машиной) код. Обычно этот перевод выполняется специальной программой — компилятором. Конечно, иногда и компиляторы содержат ошибки, вследствие чего ошибки могут появиться и в объектной программе. Однако мы считаем эти ошибки неподвластными нам (если только программа, которую мы создаем, сама не является компилятором).

6. В результате работы над программным проектом возникают как само программное обеспечение, так и документы, описывающие его использование. Последние обычно имеют вид печатных руководств, хотя их можно хранить и в самой вычислительной системе и выдавать, например, на терминал пользователя. Эти руководства

обычно получаются переводом внешних спецификаций в материалы, ориентированные на конкретные группы пользователей.

Публикации определенным образом влияют на надежность программного обеспечения. Если ошибка возникает при их подготовке, они не будут точно описывать поведение программы (если только на шагах 4 и 6 не сделаны идентичные ошибки). Прочитав руководство, пользователь начнет работать с программой и обнаружит, что она ведет себя не так, как он ожидал,— это и является, по определению, ошибкой в программе. Конечно, как отмечено в гл. 1, даже если программа и публикации согласуются между собой, в программе тем не менее могут присутствовать ошибки (например, если они возникли при переводе на шагах 1, 2 или 3, а также если одинаковые ошибки совершены на 4-м и 6-м шагах).

7. Еще одним источником информации во время разработки служат спецификации аппаратуры. Например, разработчик операционной системы опирается на описания ЦП (например, набора команд, механизмов прерывания, средств защиты) и всего периферийного оборудования системы. Разработчику прикладной системы часто нужно знать характеристики терминалов и линий связи. Неправильное истолкование этих материалов может привести к ошибкам в программном обеспечении.

8. Обычно прикладные программы взаимодействуют с базовым программным обеспечением вычислительной системы, обращаясь к нему с заказами на ввод-вывод и динамическое распределение ресурсов. Неправильное понимание документации по базовому программному обеспечению служит еще одним источником ошибок.

9. Готовая программа состоит из предложений по крайней мере одного языка программирования. Непонимание синтаксиса и семантики языка также является причиной ошибок в программном обеспечении.

10. Есть две формы связи между пользователем и готовой программой: руководства, описывающие ее использование, и непосредственная работа с ней. Шаг 10 представляет собой изучение пользователем руководств и перевод их содержания в его понимание того, как он желает применять программу.

Чтобы понять, как этот процесс влияет на надежность, представим себе, что у нас есть готовая программа, содержащая несколько неизвестных ошибок. Если пользователь пытается с ее помощью что-то сделать и из публикаций не может понять, как этого добиться (например, из-за того, что они плохо написаны), он может попытаться поэкспериментировать с системой. Эти попытки использовать систему незапланированным способом часто увеличивают вероятность натолкнуться на оставшиеся в программном обеспечении ошибки. Таким образом, качество документации для пользователя, хотя оно и не связано с количеством ошибок в программе, может влять на ее надежность.

11. Этот шаг представляет собой непосредственное взаимодействие пользователя с программным обеспечением, например при вводе данных с терминала и при анализе выдач. Если человеческие факторы учтены слабо (т. е. диалог человек — машина разработан плохо), вероятность ошибки пользователя увеличивается. Ошибки пользователя часто ставят систему в новые, непредвиденные обстоятельства, увеличивая таким образом шансы проявления оставшихся в программе ошибок.

12. Как уже говорилось, мы пока не рассматриваем многочисленные циклы обратной связи, имеющиеся на этой диаграмме. Однако один из них (шаг 12) достаточно важен, чтобы обсудить его уже сейчас.

Значительная доля затрат на программное обеспечение в мире приходится на изменения существующих программ. Хотя модель перевода на рис. 2.1 довольно хорошо представляет и этот процесс, важный дополнительный шаг, возникающий в процессе изменения,— чтение и понимание текста существующей программы для выяснения того, что и как надо изменить. Этот шаг появляется при сопровождении (исправлении ошибок) и модификации (добавлении новых функций существующим программам). Вследствие этого сама по себе готовая программа — это еще не конец процесса перевода; человек должен снова переводить ее при исправлении ошибок и добавлении новых функций. Ошибки, очевидно, могут возникать и на этом этапе, поэтому такие особенности программного обеспечения, как свойства используемого языка программирования и стиль программирования, потребуют нашего внимания.

Макромодель перевода описывает происхождение большинства ошибок в программном обеспечении. (Отметим также, что многие из ошибок перевода возникают внутри каждого из процессов.) Вначале эта модель часто вызывает удивление, так как нередко считается, что ошибки в программе — это те ошибки, которые делает программист, когда пишет программу на языке программирования. Причины некоторых ошибок здесь не отражены, например утеря перфокарты из колоды или перестановка двух букв при вводе предложения с терминала, но доля таких ошибок невелика, и обычно их нетрудно обнаружить.

Эта модель важна, поскольку она описывает причины, лежащие в основе ненадежности. Благодаря ей нам стал известен перечень подлежащих решению задач. Многие из описанных в книге методов основываются на данной модели. Это значит, что рассматриваемые методы сводят к минимуму или позволяют обнаруживать ошибки перевода, возникающие на каждом этапе.

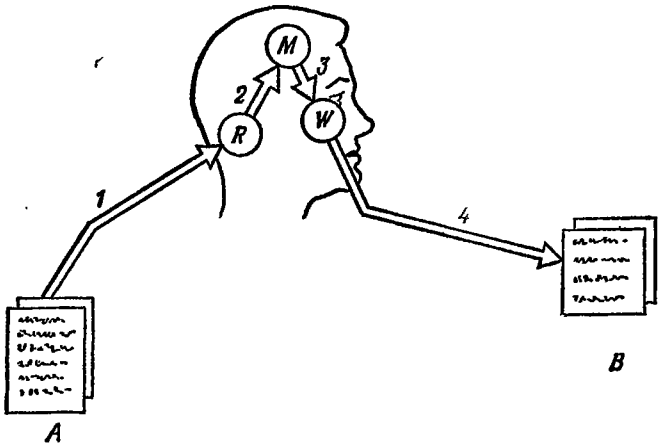


Рис. 2.2. Микромодель перевода.

МИКРОМОДЕЛЬ ПЕРЕВОДА

Чтобы лучше понимать проблемы перевода, рассмотрим модель, изображенную на рис. 2.2. Человек здесь — любое лицо, описываемое макромоделью: пользователь, системный аналитик, проектировщик или программист. Этот человек пытается переводить информацию из формы *A* в форму *B*. Для этого ему следует совершить четыре основных шага.

1. Он (она) получает информацию с помощью своего читающего механизма *R* (областей мозга, управляющих зрением и слухом).
2. Он запоминает информацию в своей памяти *M*.
3. Он выбирает из памяти эту информацию, а также информацию, описывающую процесс перевода, выполняет перевод и посылает результат своему пишущему механизму *W* (областям мозга, управляющим речью или движением рук).
4. Информация физически распространяется с помощью письма, печати на терминале или речи.

Даже такая упрощенная модель перевода позволяет нам исследовать возникающие на каждом из этапов ошибки.

1. Одной из величайших особенностей человеческого интеллекта является способность понимать входную информацию, сопоставляя ее с огромным набором хранимых в памяти образцов, созданных образованием и жизненным опытом. Этот принцип лежит в основе человеческой способности «читать между строк», понимать грамматически неправильные предложения, предсказывать развязку кинофиль-

ма и т. д. К несчастью, эти мощные способности в некоторых случаях вызывают неточности в процессе перевода, являются причиной ошибок в программном обеспечении. Ошибки возникают тогда, когда при чтении документа *A* человек видит то, что он ожидает или хочет увидеть, а не то, что написано, когда он пытается восстановить недостающие факты или просто не замечает существенной информации.

Эти же способности часто оказываются полезными при обнаружении ошибок. Если ошибки присутствуют уже в самом документе *A*, мы способны обнаружить их, осознавая несоответствие с нашим прежним жизненным опытом. Это, однако, полезно лишь тогда, когда мы отправляемся к создателю докумета *A* и выясняем с ним возникшие вопросы. Но чаще делается не так: мы разрешаем конфликт самостоятельно, иногда верно, иногда нет, и продолжаем работу дальше.

2. В большинстве случаев, для того чтобы правильно запомнить информацию, нам надо ее понять. На этом этапе ошибки в программном обеспечении появляются в результате неправильной интерпретации или полного непонимания входной информации. Она может быть слишком сложной или двусмысленной, образовательный уровень человека может оказаться недостаточно высоким.

3. Основной источник ошибок на этом этапе — забывчивость: человек может забыть входную информацию *A* либо точные правила выполнения перевода. Слабость других умственных способностей, таких, как четкость мышления или умение извлекать из памяти относящиеся к делу знания, также способствует появлению ошибок.

4. Последний этап, на котором информация может быть искажена или утрачена,— шаг 4. Многие не умеют ясно писать или выражать свои мысли — это затемняет смысл их сообщений. Если количество выходной информации велико, человека начинает раздражать разница между скоростью мышления и письма. Чтобы справиться с этим, он использует сокращения либо предполагает, что факты будут «интуитивно очевидны» его адресатам. Это увеличивает вероятность того, что следующий участник процесса разработки при переводе совершит ошибки.

Теперь у вас уже, вероятно, сложилось впечатление, что ошибки в программном обеспечении — целиком и полностью следствие несовершенства человеческого разума и единственное возможное решение этой проблемы — разведение породы лучших сегодняшних программистов чисто генетическим путем. Однако для большинства проблем перевода существуют более реалистические решения. Средства представления проекта на различных уровнях (например, языки описаний и программирования) влияют на процесс перевода. Процессы проектирования и свойства самого проекта (например, его сложность) также существенно влияют на ошибки. Подробнее об этом говорится в части 2.

ПОСЛЕДСТВИЯ ОШИБОК

Будучи ещё студентом-старшекурсником, я работал как системный программист в вычислительном центре колледжа. Одним из заданий, которые я получил после установки новой вычислительной системы, была разработка бюджетной системы, т. е. составляющей операционной системы, предназначенной для измерения и регистрации использования ресурсов, и прикладной программы для обработки поступающих данных и периодической выдачи отчетов. Поскольку места в памяти на диске было крайне мало, требовалось оптимально упаковать записи о факультете, проекте и пользователе в наборе данных на дорожке диска.

Несколько лет назад на одной из конференций я случайно встретился с директором вычислительного центра. Он лишил меня оснований гордиться своей системой, сообщив об ошибке, которая недавно была обнаружена. Попытка добавить запись о новом проекте для некоторого факультета вызвала переполнение вместо перехода на вторую дорожку диска, отведенную для этого факультета. Я планировал такую ситуацию, но, к несчастью, совершил ошибку типа «одним меньше»: оператор ЕСЛИ-МЕНЬШЕ-ИЛИ-РАВНО-ТО должен был быть оператором ЕСЛИ-МЕНЬШЕ-ТО.

Уверен, что у многих читателей имеется аналогичный опыт. Система может успешно работать несколько лет, все более и более заслуживая доверие пользователей, пока вдруг не выйдет из строя из-за того, что в новых условиях неожиданно всплывает на поверхность скрытая ошибка.

Готовя рукопись этой книги, я использовал программу обработки и редактирования текстов. Программа работала неплохо, но вдруг довольно поучительно «отказала». Обрабатывая гл. 1, она следующим образом напечатала определение надежности программного обеспечения:

```
Надежность программного обеспечения есть вероятность е
работы без отказов в течение определенного пери
о
времени, рассчитанная с учетом стоимости д
л
пользователя каждого отказа.
```

Впечатление такое, что я задел чувствительный нерв программы: ей не понравилось мое определение надежности! На самом же деле получилось всего лишь так, что мой способ определения форматов строк не был предусмотрен автором программы обработки текстов.

В профессиональных журналах по обработке данных часто публикуются веселые (для всех, кроме их участников) истории о классических ошибках в программном обеспечении. Страховая компания без всяких видимых причин аннулировала полис и отказалась его вос-

становить. Программа регистрации подписчиков направила все экземпляры журнала по одному адресу. Банковская система учета займов представила к оплате месячный счет в 212 958 долларов 72 цента, в то время как баланс по займу составлял всего 2 342 доллара 55 центов. Железнодорожная система потеряла товарные вагоны. Вследствие ошибок в системах обработки результатов голосования голоса избирателей подавались не за тех кандидатов, просто терялись, а в одном случае даже были отданы все одному кандидату (только последняя ошибка была без труда обнаружена). Ошибка в системе управления базой данных большого склада привела к разрушению записей, благодаря чему, помимо прочего, затерялись следы 15 тонн замороженной печени, помещенной в отсек без холодильника. Школьная система учета успеваемости воспринимала только двухзначные числа, так что учащиеся, набравшие более 100 баллов, оказались неуспевающими. Программист, только что закончивший перенос программы начисления зарплаты для своей фирмы с IBM 1401 на IBM 360, открывает конверт с очередной выплатой и обнаруживает, что его оклад сокращен до 0 долларов 00 центов.

К счастью, почти все эти ошибки не имели серьезных последствий. Однако есть и другой классический свод ошибок, закончившихся катастрофами или почти катастрофами. Ошибка в программном обеспечении бортовой ЭВМ космического корабля «Аполлон-8» уничтожила содержимое части памяти машины. За 10 дней полета «Аполлона-14» было обнаружено 18 ошибок. Беспокойство по поводу ошибок в программном обеспечении программы «Аполлон» выражает следующее заявление [1]:

«Самое тщательно спланированное и щедро финансированное программное обеспечение в мире было разработано для серии полетов на Луну по программе «Аполлон». К работе в двух соперничающих группах были привлечены лучшие программисты страны. Проверка программного обеспечения велась со всей полнотой, которую только могли представить себе специалисты. В общей сложности около 660 млн. долларов было затрачено на это программное обеспечение. И все-таки почти все крупные неудачи программы «Аполлон», от ложных тревог до реальных неприятностей, были прямым результатом ошибок в программном обеспечении ЭВМ».

Серьезные ошибки в программном обеспечении не ограничивались только программой «Аполлон». Вследствие ошибки в программе информация с радаров была послана не по назначению, и таким образом были уничтожены все результаты учений ПВО США NORAD в 1963 г. Частота сбоев из-за ошибок в программном обеспечении командной системы 465 L стратегического командования ВВС США, функционирующей уже 12 лет, до сих пор равна в среднем одному сбою в день. Ошибка в единственном операторе программы на Фортране привела к неудаче при первом запуске американского исследовательского корабля на Венеру. Что хуже всего, ошибки в медицин-

ском программном обеспечении явились причиной нескольких смертных случаев [3], а ошибка в программе проектирования самолета вызвала несколько серьезных авиакатастроф, хотя имеющаяся информация об этих ошибках, как и следовало ожидать, весьма неполна.

Две первые главы должны были дать представление о проблеме надежности программного обеспечения. По мере проникновения ЭВМ во все сферы нашей жизни последствия недостаточной надежности становятся все серьезнее. К счастью, решения многих аспектов проблемы надежности существуют; им и посвящен весь остальной материал книги.

ЛИТЕРАТУРА

1. Ulsamer E. Computers — Key to Tomorrow's Air Force, *AIR FORCE Magazine*, 56 (7), 46—52 (1973).
2. Thayer R. H. Rome Air Development Center R and D Program in Computer Language Controls and Software Engineering Techniques, RADC-TR-74-80, Griffiss Air Force Base, Rome, N. Y., 1974.
3. Boehm B. W. Software and its Impact: A Quantitative Assessment, *Datamation*, 19 (5), 48—59 (1973).
4. Naur P., Randell B., Eds. Software Engineering: Report on a Conference Sponsored by the NATO Science Committee. Brussels, Belgium: NATO Scientific Affairs Division, 1968, p. 121.

ЧАСТЬ 2

Проектирование надежного программного обеспечения

ГЛАВА 3

Основные принципы проектирования

Разработка программы включает задачи двух типов: проектирование и тестирование. В части 2 рассматриваются задачи проектирования и те виды тестирования, которые могут возникнуть в процессе проектирования. В части 3 речь идет о тех видах тестирования, которые касаются готового программного обеспечения.

Слово *проектирование* употребляется сейчас в области разработки программного обеспечения в нескольких смыслах. Во многих организациях смысл слова «проектирование» произвольно ограничивается начальным этапом работы над проектом, а для обозначения последующих этапов используются такие термины, как «реализация», «разработка», «программирование». К сожалению, нет общепринятого соглашения об употреблении этих слов; различные организации понимают под этими словами разные группы процессов, что приводит к путанице при попытке сравнить два проекта. Более того, это произвольное деление порождает тенденцию к «кастовости», поскольку программисты часто считают работу «проектировщика» более престижной, чем работу «реализатора».

Чтобы преодолеть эти проблемы, я использую слово «проектирование» так, как оно определено в словаре: «придание формы в соответствии с планом». Это определение охватывает различные виды деятельности по созданию программного обеспечения, начиная с определения требований и целей и кончая написанием текста программы, но подразумевает, конечно, наличие различных стадий проектирования. Фразы «разработка программного обеспечения», «конструирование программного обеспечения» и «производство

программного обеспечения» обозначают весь цикл его создания. В этой главе рассматриваются некоторые принципы, общие для всех стадий проектирования.

ЧЕТЫРЕ ПОДХОДА К НАДЕЖНОСТИ

Все принципы и методы обеспечения надежности в соответствии с их целью можно разбить на четыре группы: *предупреждение ошибок*, *обнаружение ошибок*, *исправление ошибок* и *обеспечение устойчивости к ошибкам*. К первой группе относятся принципы и методы, позволяющие минимизировать или вообще исключить ошибки. Методы второй группы сосредоточивают внимание на функциях самого программного обеспечения, помогающих выявлять ошибки. К третьей группе относятся функции программного обеспечения, предназначенные для исправления ошибок или их последствий. Устойчивость к ошибкам — это мера способности системы программного обеспечения продолжать функционирование при наличии ошибок.

Предупреждение ошибок

К этой группе относятся принципы и методы, цель которых — не допустить появления ошибок в готовой программе. Большинство методов концентрируется на отдельных процессах перевода и направлено на предупреждение ошибок в этих процессах. Их можно разбить на следующие категории:

1. Методы, позволяющие справиться со сложностью, свести ее к минимуму, так как это — главная причина ошибок перевода.
2. Методы достижения большей точности при переводе.
3. Методы улучшения обмена информацией.
4. Методы немедленного обнаружения и устранения ошибок. Эти методы направлены на обнаружение ошибок на каждом шаге перевода, не откладывая до тестирования программы после ее написания.

Должно быть очевидно, что предупреждение ошибок — оптимальный путь к достижению надежности программного обеспечения. Лучший способ обеспечить надежность — прежде всего не допустить возникновения ошибок. Гарантировать отсутствие ошибок, однако, невозможно никогда. Другие три группы методов опираются на предположение, что ошибки все-таки будут.

Обнаружение ошибок

Если предполагать, что в программном обеспечении какие-то ошибки все же будут, то лучшая (после предупреждения ошибок) стратегия — включить средства обнаружения ошибок в само про-

граммное обеспечение. Эта идея нашла отражение в фильме «2001: Космическая Одиссея», правда на примере обнаружения сбоев в аппаратуре.

Компьютер: «У меня затруднения в поддержании контакта с Землей. Поломка в устройстве АЕ-35. Мой центр прогнозирования сбоев сообщает, что оно может выйти из строя в течение 72 часов».

Большинство методов направлено по возможности на незамедлительное обнаружение сбоев. Немедленное обнаружение имеет два преимущества: можно минимизировать как влияние ошибки, так и последующие затруднения для человека, которому придется извлекать информацию об этой ошибке, находить ее место и исправлять.

Исправление ошибок

Следующий шаг — методы исправления ошибок; после того как ошибка обнаружена, либо она сама, либо ее последствия должны быть исправлены программным обеспечением. Исправление ошибок самой системой — плодотворный метод проектирования надежных систем аппаратного обеспечения. Некоторые устройства способны обнаружить неисправные компоненты и перейти к использованию идентичных запасных. Аналогичные методы неприменимы к программному обеспечению вследствие глубоких внутренних различий между сбоями аппаратуры и ошибками в программах. Если некоторый программный модуль содержит ошибку, идентичные «запасные» модули также будут содержать ту же ошибку.

Другой подход к исправлению связан с попытками восстановить разрушения, вызванные ошибками, например искажения записей в базе данных или управляющих таблицах системы. Польза от методов борьбы с искажениями ограниченна, поскольку предполагается, что разработчик заранее предугадает несколько возможных типов искажений и предусмотрит программно реализуемые функции для их устранения. Это похоже на парадокс, поскольку, если знать заранее, какие ошибки возникнут, можно было бы принять дополнительные меры по их предупреждению. Если методы ликвидации последствий сбоев не могут быть обобщены для работы со многими типами искажений, лучше всего направлять силы и средства на предупреждение ошибок. Вместо того чтобы, разрабатывая операционную систему, оснащать ее средствами обнаружения и восстановления цепочки искаженных таблиц или управляющих блоков, следовало бы лучше спроектировать систему так, чтобы только один модуль имел доступ к этой цепочке, а затем настойчиво пытаться убедиться в правильности этого модуля.

Устойчивость к ошибкам

Методы этой группы ставят своей целью обеспечить функционирование программной системы при наличии в ней ошибок. Они разбиваются на три подгруппы: динамическая избыточность, методы отступления и методы изоляции ошибок.

1. Истоки концепции *динамической избыточности* лежат в проектировании аппаратного обеспечения. Один из подходов к динамической избыточности — метод *голосования*. Данные обрабатываются независимо несколькими идентичными устройствами, и результаты сравниваются. Если большинство устройств выработало одинаковый результат, этот результат и считается правильным. И опять, вследствие особой природы ошибок в программном обеспечении ошибка, имеющаяся в копии программного модуля, будет также присутствовать во всех других его копиях, поэтому идея голосования здесь, видимо, неприемлема. Предлагаемый иногда подход к решению этой проблемы состоит в том, чтобы иметь несколько *не*идентичных копий модуля. Это значит, что все копии выполняют одну и ту же функцию, но либо реализуют различные алгоритмы, либо созданы разными разработчиками. Этот подход бесперспективен по следующим причинам. Часто трудно получить существенно разные версии модуля, выполняющие одинаковые функции. Кроме того, возникает необходимость в дополнительном программном обеспечении для организации выполнения этих версий параллельно или последовательно и сравнения результатов. Это дополнительное программное обеспечение повышает уровень сложности системы, что, конечно, противоречит основной идее предупреждения ошибок — стремиться в первую очередь минимизировать сложность.

Второй подход к динамической избыточности — выполнять эти запасные копии только тогда, когда результаты, полученные с помощью основной копии, признаны неправильными. Если это происходит, система автоматически вызывает запасную копию. Если и ее результаты неправильны, вызывается другая запасная копия и т. д. Хотя Ранделл [1] дает набросок хорошего метода реализации такого подхода, этот подход страдает большинством перечисленных ранее недостатков. Кроме того, вполне вероятно, что если ресурсы при работе над проектом фиксированы, то при реализации «запасных» версий проектированию и тестированию будет уделено меньше внимания, чем можно было бы уделить, если бы реализовывалась лишь одна копия и динамическая избыточность не использовалась.

2. Вторая подгруппа методов обеспечения устойчивости к ошибкам называется методами *отступления* или сокращенного обслуживания. Эти методы приемлемы обычно лишь тогда, когда для системы программного обеспечения существенно важно благопристойно закончить работу. Например, если ошибка оказывается в системе,

управляющей технологическими процессами, и в результате эта система выходит из строя, то может быть загружен и выполнен особый фрагмент программы, призванный подстраховать систему и обеспечить безаварийное завершение всех управляемых системой процессов. Аналогичные средства часто необходимы в операционных системах. Если операционная система обнаруживает, что она вот-вот выйдет из строя, она может загрузить аварийный фрагмент, ответственный за оповещение пользователей у терминалов о предстоящем сбое и за сохранение всех критических для системы данных.

3. Последняя подгруппа — методы *изоляции ошибок*. Основная их идея — не дать последствиям ошибки выйти за пределы как можно меньшей части системы программного обеспечения, так чтобы если ошибка возникнет, то не вся система оказалась неработоспособной; отключаются лишь отдельные функции в системе либо некоторые ее пользователи. Например, во многих операционных системах изолируются ошибки отдельных пользователей, так что сбой влияет лишь на некоторое подмножество пользователей, а система в целом продолжает функционировать. В телефонных переключательных системах для восстановления после ошибки, чтобы не рисковать выходом из строя всей системы, просто разрывают телефонную связь. Другие методы изоляции ошибок связаны с защитой каждой из программ в системе от ошибок других программ. Ошибка в прикладной программе, выполняемой под управлением операционной системы, должна оказывать влияние только на эту программу. Она не должна сказываться на операционной системе или других программах, функционирующих в этой системе.

Из этих трех подгрупп методов обеспечения устойчивости к ошибкам только третья, изоляция ошибок, применима для большинства систем программного обеспечения.

Важное обстоятельство, касающееся всех четырех подходов, состоит в том, что обнаружение, исправление ошибок и устойчивость к ошибкам в некотором отношении противоположны методам предупреждения ошибок. В частности, обнаружение, исправление и устойчивость требуют дополнительных функций от самого программного обеспечения. Тем самым не только увеличивается сложность готовой системы, но и появляется возможность внести новые ошибки при реализации этих функций. Как правило, *все* рассматриваемые методы предупреждения и многие методы обнаружения ошибок применимы к любому программному проекту. Методы исправления ошибок и обеспечения устойчивости применяются не очень широко. Это, однако, зависит от области приложения. Если рассматривается, скажем, система реального времени, то ясно, что она должна сохранить работоспособность и при наличии ошибок, а тогда могут оказаться желательными и методы исправления и обеспечения устойчивости. К системам такого типа относятся телефонные переключательные системы, системы управления технологическими процес-

	Предупреждение ошибок	Обнаружение ошибок	Исправление ошибок	Устойчивость к ошибкам
Принципы	Гл. 5,6,8	Нет	Нет	Нет
Методы	Гл. 4,5,6,8,9	Гл. 7,8	Гл. 7	Гл. 7

Рис. 3.1. Соответствие между главами и методами обеспечения надежности.

сами, аэрокосмические и авиационные диспетчерские системы и операционные системы широкого назначения.

На рис. 3.1 указано соответствие между этими методами и главами книги. *Принципы* — это не зависимые от области приложения стратегии обеспечения надежности программных систем. Рассматриваются принципы проектирования системы, программы, модуля, направленные на предупреждение ошибок. Для остальных трех категорий принципы неизвестны. *Методы* — это более мелкие, обычно зависящие от области приложения тактические средства. Главы, в которых обсуждаются методы для всех четырех категорий, указаны на рис. 3.1.

ПРОЦЕССЫ ПРОЕКТИРОВАНИЯ

Проектирование любого программного продукта включает несколько различных процессов. При хорошо поставленном руководстве проектом эти процессы явно выражены, так что могут быть установлены контрольные сроки, выбрана методология и по завершении каждого процесса можно проверить «доброкачественность» его результатов. При плохом руководстве некоторые из этих процессов или все они не выделяются явно: каждый процесс по-прежнему присутствует в каком-то виде, но некоторые из них проходят неявно, вследствие чего контрольные сроки, методология и оценки никогда не устанавливаются.

На рис. 3.2 представлена модель процессов проектирования типичной крупной программной системы. Отметим, что модель не зависит от методологии. Все указанные в ней действия должны выполняться в той или иной форме во всякой разработке, независимо от того, какой язык программирования был принят, писал ли пользователь исходные требования, использовалось ли «структурное программирование» и т. д.

На первом шаге составляется перечень требований, т. е. четкое определение того, что пользователь ожидает от готового продукта. Следующий шаг касается постановки целей — задач, которые ставятся перед окончательным результатом и самим проектом. Затем выполняется внешний проект высокого уровня. На этом

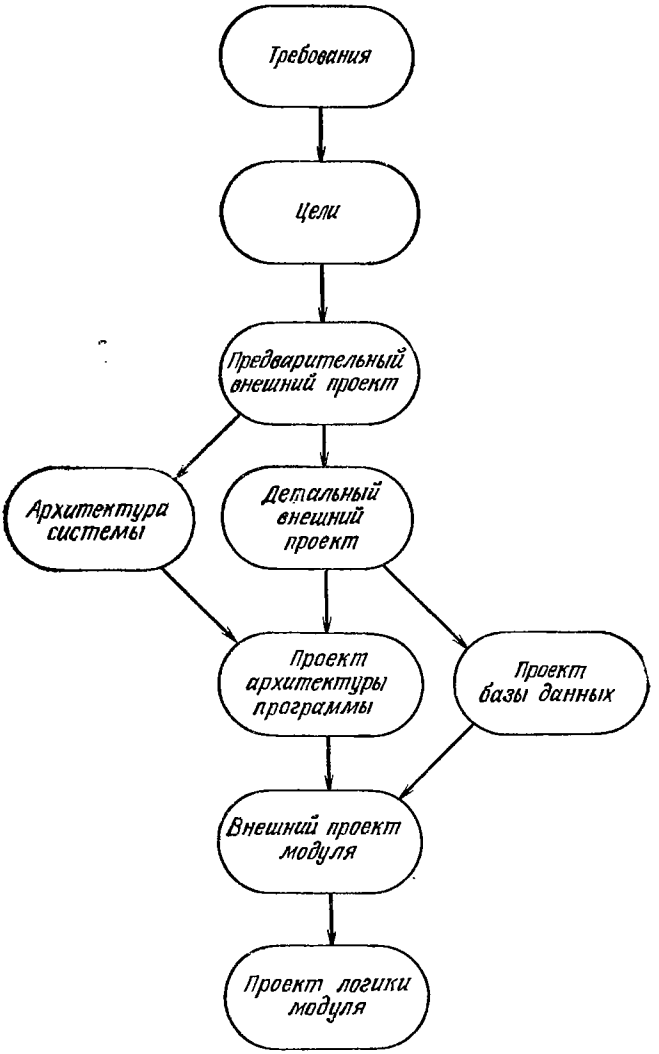


Рис. 3.2. Этапы проектирования крупной системы.

шаге определяется взаимодействие с пользователем, но не рассматриваются многие его детали, такие, как форматы ввода-вывода.

Исходный внешний проект приводит к двум параллельным процессам. В процессе детального внешнего проектирования завершается определение взаимодействия с пользователем, описываются его мельчайшие подробности. В процессе разработки архитектуры системы выполняется разложение ее на множество программ, подсистем

или компонент и определяются сопряжения между ними. Эти два шага ведут к процессу проектирования структуры программы, в котором проектируются модули, их сопряжения и взаимосвязи для каждой программы, компоненты или подсистемы. Следующий процесс — внешнее проектирование модуля — это точное определение всех сопряжений модуля. Последний шаг — проектирование логики модуля — состоит в разработке внутренней логики каждого модуля системы, он включает также выражение этой логики текстом конкретной программы.

Последний изображенный на рис. 3.2 процесс — проектирование базы данных. Это процесс определения всех внешних для программной системы структур данных, например записей в файле или в базе данных.

Рис. 3.2 иллюстрирует основные зависимости между процессами проектирования, но он не претендует на то, чтобы служить планом работы для какого-нибудь конкретного проекта. В частности, возможны и обычно желательны перекрытия между разными процессами. На рис. 3.2 показано, какие процессы проектирования должны быть полностью закончены, прежде чем может быть завершен другой процесс. Например, необходимо закончить внешнее проектирование всех модулей системы до завершения проектирования логики модулей. Однако проектирование логики отдельных модулей, конечно, можно завершить и до окончания внешнего проектирования всех модулей. Такая форма перекрытия между смежными процессами часто бывает полезной. Отметим, однако, что *не должно* быть никакого перекрытия между несмежными процессами. Например, процесс проектирования логики модулей не следует начинать до окончания проектирования структуры программы. И второй пример: не следует приступать к разработке архитектуры системы до того, как полностью будут определены ее цели.

Следует также помнить, что в работе над любым проектом последовательность процессов проектирования отнюдь не так проста. Есть существенная обратная связь между процессами. Например, во время одного из шагов внешнего проектирования могут быть обнаружены погрешности в формулировке целей, тогда нужно немедленно вернуться и исправить их. При проектировании структуры программы можно внезапно обнаружить, что указанная во внешних спецификациях функция неосуществима или обойдется слишком дорого, тогда может понадобиться принять компромиссное решение и изменить внешние спецификации.

Рис. 3.2, с некоторыми изменениями, может служить моделью и для малых проектов. Когда проектируется единственная программа, то часто отсутствуют процессы проектирования архитектуры системы и проектирования базы данных; процессы исходного и детального внешнего проектирования также зачастую сливаются воедино.

СЛОЖНОСТЬ

Сложность — это основная причина ошибок перевода и, следовательно, одна из главных причин ненадежности программного обеспечения. Сложность почти не поддается ни точному определению, ни измерению. Однако можно сказать, что мерой сложности объекта является количество интеллектуальных усилий, необходимых для понимания этого объекта.

В общем случае сложность объекта является функцией взаимодействия между его компонентами. Например, сложность внешнего проекта программной системы в некоторой степени определяется связями между всеми ее внешними сопряжениями, например между командами пользователя и соотношениями между входной и выходной информацией системы. Сложность архитектуры системы определяется связями между подсистемами. Сложность проекта программы — функция связей между модулями. Сложность отдельного модуля — функция связей между его командами.

В борьбе со сложностью программного обеспечения можно привлечь две концепции из общей теории систем [2]. Первая — *независимость*. В соответствии с этой концепцией для минимизации сложности необходимо максимально усилить независимость компонент системы. По существу, это означает такое разбиение системы, чтобы высокочастотная динамика ее была заключена в единых компонентах, а межкомпонентные взаимодействия представляли лишь низкочастотную динамику системы.

Вторая концепция — *иерархическая структура*. Иерархия позволяет стратифицировать систему по уровням понимаиия. Каждый уровень представляет собой совокупность структурных отношений между элементами нижних уровней. Концепция уровня позволяет понять систему, скрывая несущественные уровни детализации. Например, система, которую мы называем «человек», представляется иерархией. Социолог может интересоваться взаимоотношениями людей, не заботясь об их внутреннем устройстве. Психолог работает на более низком уровне иерархии. Он может исследовать различные логические и физические процессы в мозге, не рассматривая внутреннего строения областей мозга. Еще ниже в этой иерархии находится нейролог — он имеет дело со структурой основных компонент мозга. Однако он может изучать мозг на этом уровне, не заботясь о молекулярной структуре отдельных белков в нейроне. Химик-органик интересуется построением сложных аминокислот из таких компонент, как атомы углерода, водорода, кислорода и хлора. Наконец, физик-ядерщик изучает систему на уровне элементарных частиц в атоме и взаимодействия между ними.

Иерархия позволяет проектировать, описывать и понимать сложные системы. Если бы нельзя было принять описанный подход к изучению человека, социологу пришлось бы рассматривать его как

необъятное и сложное множество субатомных частиц. Очевидно, что такое количество деталей подавило бы его, так что невозможны были бы даже те ограниченные знания о человеке, которыми мы располагаем.

К этим двум концепциям сокращения сложности (независимость и иерархическая структура) я добавил бы третью: *проявление связей* всюду, где они возникают. Основная проблема многих больших программных систем — огромное количество независимых побочных эффектов, создаваемых компонентами системы. Из-за этих побочных эффектов систему невозможно понять. И можно быть уверенным, что систему, в которой нельзя разобраться, было очень трудно спроектировать хотя бы с минимальной гарантией надежности.

Проектное решение любого уровня имеет некоторую внутреннюю организацию, или форму. Для минимизации сложности нам нужен метод проявления этой внутренней формы, с тем чтобы в соответствии с нею разбить проект на части. При внешнем проектировании, например, разбиение системы в соответствии с ее внутренней формой называется *концептуальной целостностью* [3]. В главах, посвященных разработке архитектуры системы и проектированию структуры программы, рассматриваются методы определения этой внутренней формы для дальнейшего разбиения системы на множество компонент с высокой степенью независимости. При проектировании логики программы дисциплина, называемая *структурным программированием*, имеет целью, помимо всего прочего, привести эту логику в соответствие с несколькими базовыми стандартными формами.

ОТНОШЕНИЯ С ПОЛЬЗОВАТЕЛЕМ

В гл. 1 было определено, что в программном обеспечении имеется ошибка, если система не выполняет того, что пользователю разумно от нее ожидать. Там же был приведен распространенный вопрос разработчика программного обеспечения: «Как можно знать, что пользователю разумно ожидать от системы?» При правильно организованной работе этот вопрос никогда не должен возникнуть. Если разработчик системы плохо представляет или не представляет вовсе, что пользователю разумно ожидать, программный проект никогда не будет успешным.

Две самые распространенные ошибки при работе над программными проектами — это отказ от вовлечения пользователя системы в процессы принятия решений и неспособность понять его культурный уровень и окружающую его обстановку. При работе над многими проектами имеется тенденция умышленно исключать пользователя из процесса принятия решений. Обычно причина этого в том, что разработчик программного обеспечения чувствует: ес-

ли вовлечь пользователя, тот никогда не придет к окончательному решению, его требования будут постоянно меняться. Для такой тревоги есть некоторые основания, но на практике преимущества от участия пользователя значительно перевешивают эти возможные неудобства. Вторая ошибка в программных проектах — разработчик системы часто слабо знает (или не знает вовсе) обстановку, в которой находится пользователь, т. е. плохо понимает, с какими именно трудностями сталкивается пользователь и как он будет применять программную систему. Бывает, например, так, что в проектировании операционной системы участвуют люди, сами никогда не использовавшие операционных систем. Есть разработчики языков программирования, никогда не пробовавшие реализовать прикладную систему на языке высокого уровня. Есть разработчики систем управления базами данных, которые никогда не пытались использовать базу данных в прикладной программе. Это не может не вести к серьезным ошибкам в программном обеспечении.

Единственный возможный способ избежать этих ошибок — поддерживать прочный контакт с пользователем в течение всего цикла разработки. С коллективом пользователей должны быть установлены такие отношения, чтобы те серьезно участвовали в процессе принятия решений на этапах определения требований, целей и внешнего проектирования. Привлечение пользователей на последующих этапах также желательно, особенно в процессе тестирования, когда пользователь может помочь разработчику системы значительно лучше понять, как следует тестировать систему. Будьте, однако, осмотрительны, привлекая пользователя к обсуждению деталей, судить о которых он некомпетентен. Например, хорошо, чтобы пользователь принимал участие в проектировании внешних характеристик системы, но привлекать его к такой работе, как анализ логики конкретного модуля, неразумно.

Имеется (уже упоминавшаяся) опасность, что пользователь может изменять свои требования к системе. Отметим, однако, что это никак не связано с непосредственным его участием в работе над проектом. Если требования к системе должны измениться, это произойдет независимо от того, привлечен ли пользователь непосредственно к работе или нет. В действительности, если сам пользователь в работе не участвует, разработчик, вероятно, не узнает об изменении требований до тех пор, пока не станет слишком поздно. Если же пользователь непосредственно привлечен к работе, он может значительно лучше представлять себе стоимость каждого изменения. Если правильно предусмотреть условия для изменения требований, участие пользователя может оказаться выгодным и с этой точки зрения.

Участие потенциальных пользователей в создании новых систем, которые разрабатываются не по заказу и сведения о которых составляют коммерческую тайну, также *не* является недопустимым.

Хотя такой продукт предназначен не для конкретного потребителя, разработчик и в этом случае, вероятно, хорошо представляет себе возможных покупателей. С одним или несколькими из них может быть подписано соглашение о сохранении коммерческой тайны, что позволит и возможным покупателям системы участвовать в ее разработке до того, как о ней будет публично объявлено.

Преодоление второй трудности — непонимание запросов пользователя и окружающей его обстановки — требует, чтобы проектировщики программной системы досконально представили себе особенности его работы. Обычно принимается весьма неэффективное решение — командировать основных проектировщиков для изучения положения дел. Проектировщики получают поверхностное представление о существующих системах и совсем не получают сколько-нибудь глубокого представления о том, как система используется и в чем же состоят подлинные проблемы. Если бы я проектировал систему резервирования авиабилетов, то я бы устроился временно работать агентом по резервированию. Через несколько недель я бы по-настоящему представлял себе окружающую пользователя среду и, возможно, имел бы некоторые идеи относительно ее улучшения и увеличения производительности.

РЕШЕНИЕ ЗАДАЧИ

Большинство процессов разработки программного обеспечения — это процессы решения некоторых задач. Внешнее проектирование сводится к решению такой задачи: «Переведите множество целей системы во внешние спецификации», где цели — данные, а внешние спецификации — неизвестные. В задаче проектирования логики модуля даны внешние спецификации модуля, а неизвестное — текст его программы. Отладка — это задача на построение исправления ошибки (неизвестное) по описанию ее симптомов (данные).

Удивительно, что разработчики программного обеспечения не получают никакой специальной подготовки по общим методам решения задач. Преподавая на курсах проектирования программного обеспечения, я заметил, что удачные решения сложных проблем проектирования обычно находят те студенты, которые, сознательно или подсознательно, применяют общие правила решения задач.

Одно из лучших руководств по решению задач — книга Пойа «Как решать задачу» [4]. Хотя речь идет в основном о решении геометрических задач и математических головоломок, большинство изложенных в ней соображений применимы и к проектированию программного обеспечения. Поскольку проблемы программного обеспечения в отличие от большинства математических задач не имеют единственного «правильного» решения, я слегка изменил и реорганизовал метод Пойа. Схема метода представлена на рис. 3.3 и рассматривается ниже.

Решение задачи

1. Поймите задачу
   - Изучите данные.
   - Изучите неизвестные.
   - Достаточно ли данных для решения? Не противоречивы ли они?
2. Составьте план
   - Чего вы должны добиваться?
   - Какие методы проектирования будут использоваться?
   - Встречалась ли вам уже такая задача?
   - Не знаете ли вы близкой задачи? Можете ли вы воспользоваться ее результатом?
   - Можете ли вы решить более специализированную или аналогичную задачу?
   - Можете ли вы решить часть задачи?
3. Выполните план
   - Следуйте своему плану решения задачи.
   - Проверяйте правильность каждого шага.
4. Проанализируйте решение
   - Все ли данные вы использовали?
   - Проверьте правильность решения.
   - Можете ли вы воспользоваться полученным результатом или примененным методом при решении других задач?

Рис. 3.3. Этапы решения проблем проектирования.

Поймите задачу

Худшая из ошибок, которые могут быть сделаны при решении задачи,— не вполне разобраться в ее постановке. Понять задачу — это значит понять два ее компонента: данные и неизвестное. Данные — это все элементарные факты, касающиеся задачи, и связи между фактами и неизвестным. Усвоение всех данных о сложной задаче — большая, но абсолютно неизбежная работа. При этом в первую очередь необходимо хорошо охватить «общую картину» данных без деталей, которые, однако, также запоминаются «в сторонке», чтобы их можно было легко вспомнить позже. Есть много способов добиться этого. Например, кое-кто физически разрезает спецификации на куски, которые затем расклеиваются на стене в определенном порядке. Это позволяет увидеть «общую картину» и при этом определить место для каждой детали.

Вторая часть задачи — неизвестное. Проектировщику следует понимать, какую форму должно иметь решение. Если, например,

задача — детальное внешнее проектирование программы, то проектировщик должен ясно представлять назначение внешних спецификаций, их потенциальных читателей, формат и т. д.

Исследуя задачу, проектировщик должен также исследовать данные, чтобы убедиться, что их достаточно для решения задачи и они не противоречат друг другу.

Составьте план

Прежде чем приступить к решению, следует разработать его план. Отсутствие плана — очень распространенная ошибка. Например, проектировщики программной системы, которые потратили время на то, чтобы понять задачу, но затем немедленно приступили к ее решению, не пожелав тратить время на планирование своих усилий, в конце концов могут прийти к хорошему решению, но не раньше, чем после нескольких ненужных фальстартов.

Прежде всего в плане нужно определить, чего вы хотите добиться. Десять человек могут иметь десять разных мнений относительно «правильного» ответа на задачу проектирования; проектировщик должен предусмотреть те конкретные аспекты решения, которые требуют наибольшего внимания. К сожалению, в большинстве проектов разработчики имеют слишком много свободы в этом отношении: каждый проектировщик принимает компромиссные решения, основываясь исключительно на своем собственном мнении, что приводит к несогласованности многих решений в системе. Идея *целей проекта*, рассматриваемая в гл. 4, является решением этой проблемы. Суть идеи состоит в том, что на уровне всего проекта определяются общие цели, которыми следует руководствоваться во всех решениях при проектировании.

Ключевой компонентой успешного проектирования является методология. Выбор подходящей методологии для каждого конкретного процесса проектирования должен быть зафиксирован в качестве одной из составляющих плана.

Накопленный опыт, образование и имеющиеся решения проблем также существенно влияют на успех дела. Обработка данных в своей эволюции достигла такой точки, когда проектировщик крайне редко сталкивается с задачей, которая уже не была бы решена частично или полностью. Например, разработчик новой операционной системы должен понимать, что уже созданы сотни операционных систем и на эту тему написан не один учебник. Проектировщик, столкнувшись с задачей сортировки, должен знать, что уже придумано и проанализировано множество алгоритмов сортировки. При разумном подходе к решению задач начинать следует с анализа своего опыта и опыта других, с тем чтобы проверить, не была ли задача уже решена.

Даже если готовое решение найти не удается, вероятно, когда-

то была решена близкая задача. Проектировщик системы резервирования авиабилетов может сообразить, что у нее есть много сходства с другими системами резервирования, например с системой резервирования мест в гостинице. Тогда ему, возможно, удастся выделить и применить у себя элементы решения задачи о резервировании мест в гостинице.

Если все эти методы не приносят успеха, может оказаться эффективным решение более специализированной задачи или части задачи. Если количество деталей в постановке задачи слишком велико, разработчику следует посмотреть, нельзя ли упростить ее, отбросив часть деталей. В результате либо станет ясно, как следует изменить упрощенное решение, чтобы учесть и отброшенные детали, либо удастся лучше увидеть возможные решения, так что можно будет прекратить заниматься упрощенным вариантом и начать сначала.

Выполните план

Следующий шаг — действительно решить задачу в соответствии с запланированным подходом. Поскольку решение обычно состоит из целого ряда последовательных шагов, разработчик в процессе решения должен пытаться проверить правильность каждого шага.

Проанализируйте решение

После того как результат получен, нужно еще его проверить. Разработчик должен просмотреть все данные, чтобы убедиться, что учтено все, что имеет отношение к делу. Полезно для этого еще раз перечитать буквально каждое слово постановки задачи, вычеркивая каждый использованный в решении факт, а затем проверить, насколько существенно для задачи то, что осталось незачеркнутым. Разработчик должен также проверить правильность решения задачи.

ПРАВИЛЬНОСТЬ ПРОЕКТИРОВАНИЯ

Явно выделенным этапом всякого процесса проектирования должна быть проверка правильности результатов, т. е. попытка найти ошибки перевода, возникшие в этом процессе. Основания для этого иллюстрирует рис. 3.4. Дело не только в том, что стоимость исправления ошибки тем ниже, чем раньше она будет обнаружена, но и в том, что вероятность правильно исправить ошибку на ранней стадии работы над проектом значительно выше, чем в случае, если ошибка обнаружена на более поздних этапах, например при тестировании системы.

Хотя проверка правильности каждого отдельного процесса проектирования обсуждается в последующих главах, можно сформулировать общую философию проверки в виде правила *n плюс-минус один*.

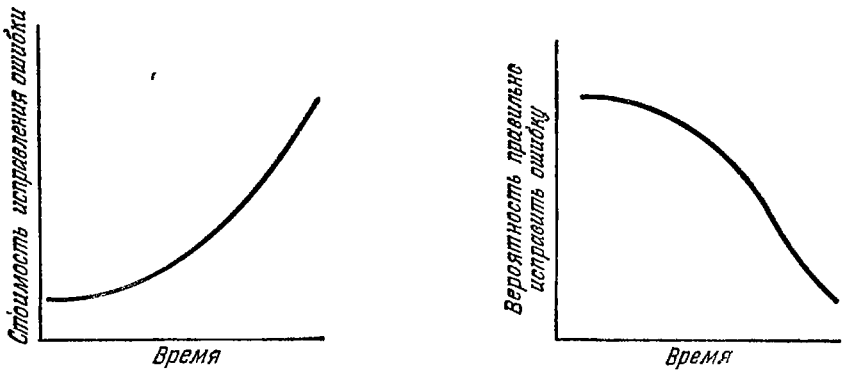


Рис. 3.4. Основания для раннего обнаружения ошибок проектирования.

Вопросом первостепенной важности при проверке правильности проекта является привлечение к этому делу подходящих людей. «Наиболее подходящие» люди — это те, в чьих интересах обнаружить все ошибки. Правило «n плюс-минус один» состоит в следующем:

Проверка правильности фазы n проекта должна осуществляться проектировщиками фаз $n+1$ и $n-1$.

Пусть, например, мы закончили фазу проектирования архитектуры системы. Возвращаясь к рис. 3.2, видим, что проектировщики фазы $n-1$ — это авторы исходных внешних спецификаций, а проектировщики фазы $n+1$ — это разработчики структуры программы. Именно им и следует доверить «тестирование» архитектуры системы.

В основе нашей философии лежит предположение, что всякий процесс проверки правильности (тестирование) должен иметь разрушительный, даже садистский характер. Цель должна состоять в том, чтобы обнаружить любой мыслимый дефект, любую слабость в проекте системы, а не в том, чтобы просмотреть проект и показать, что он правилен. Заинтересованность проектировщика фазы $n-1$ (автора внешних спецификаций) в нахождении ошибок проявляется обычно так: «Ну-ка покажите, каких ошибок эти системные архитекторы насажали в мои великолепные внешние спецификации». Разработчик фазы $n+1$ в такой же мере заинтересован в обнаружении ошибок, хотя и по другой причине. Его ощущения примерно таковы: «Архитектура системы для меня — исходные данные, поэтому мне следует найти в ней все дефекты и убедиться, что я правильно ее понимаю, до того, как я приступлю к своей работе».

Эта философия — еще один пример того, что интуитивно очевидно, но редко применяется на практике. В случае необходимости

должны быть установлены формальные процедуры, обеспечивающие ее применение. Она может быть использована во всех процессах проектирования, изображенных на рис. 3.2. Естественно, ее следует слегка изменить для первого и последнего процессов, а также для случая, когда два последовательных процесса выполняются одними и теми же разработчиками.

ЛИТЕРАТУРА

1. Randell B. System Structuring for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, SE-1 (2), 220—232 (1975).
2. Саймон Г. Науки об искусственном. Пер. с англ.— М.: Мир, 1972.
3. Брукс Ф. П. Как проектируются и создаются программные комплексы. Пер. с англ.— М.: Наука, 1979.
4. Пойа Д. Как решать задачу. Пер. с англ.— М.: Наука, 1961.

ГЛАВА 4

Требования, цели и спецификации

Первые ошибки прокрадываются в программу тогда, когда определяются требования и цели. Причина большинства этих ошибок — неправильное понимание потребностей пользователя. В дальнейшем возникают и другие ошибки, когда требования и цели транслируются во внешние спецификации. К сожалению, слишком редко на эти процессы обращается достаточное внимание. Необходимость особого внимания к ним хорошо выражена Уильямсом [1]:

«Ничто, абсолютно ничто не может более пагубно повлиять на разработку, чем неполное или/и неправильное знание и, как следствие, неадекватная спецификация требований к системе и программному обеспечению».

ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

Программные проекты можно разбить на три группы: управляемые пользователем, контролируемые пользователем и независимые от пользователя. В случае *управляемого пользователем* проекта требования к программному обеспечению разрабатываются непосредственно организацией-пользователем. Разработчик программного обеспечения является ее субподрядчиком, и требования представляют собой контракт или его часть. Например, многие программные проекты для правительства США относятся к этой категории. Правительственное агентство разрабатывает довольно длинный список требований, которые должны выполнять изготовители программного обеспечения.

В проекте, *контролируемом пользователем*, требования к программному обеспечению формулируются либо его разработчиком, либо совместными усилиями разработчика и организации-пользователя. В проектах такого типа организация-пользователь имеет право утверждать требования и, как правило, спецификации следующих уровней, в частности — внешние спецификации.

В *не зависимом от пользователя* проекте вся ответственность за определение требований ложится на разработчика программного обеспечения. Большинство проектов, связанных с разработкой общедоступных, поставляемых на рынок программ, относится к этой группе. Однако, как говорилось в предыдущей главе, и здесь

по-прежнему крайне важно в какой-либо форме привлечь пользователя к работе над проектом.

Требования к программному обеспечению дают возможность пользователю сформулировать свои потребности в отношении конкретного программного продукта. С этой точки зрения наиболее привлекателен проект, контролируемый пользователем. Нет сомнений, что всегда желательно серьезное участие потенциального пользователя окончательного продукта в определении требований. Но, безусловно, желательно и участие в этом процессе организации-разработчика программного обеспечения. Если в управляемом пользователем проекте разработчик программного обеспечения *не* привлечен к определению требований, это увеличивает его шансы неправильно понять или неправильно интерпретировать требования. И управляемые пользователем, и не зависимые от пользователя проекты должны планироваться таким образом, чтобы обеспечить участие обеих сторон.

Требования к средней или крупной системе должны разрабатываться небольшой группой. Один участник этой группы должен быть основным представителем организации-пользователя, облеченным достаточными полномочиями, чтобы принимать решения. Отметим, однако, что он обычно не является настоящим пользователем системы. Поэтому вторым членом группы должен быть человек, который действительно будет пользоваться системой. Например, при разработке требований к системе резервирования авиабилетов им должен быть опытный кассир. Организация-разработчик также должна быть представлена в этой группе. Одним из ее представителей должен быть человек, который в конечном счете будет играть главную роль в процессе внешнего проектирования. Другим членом группы должен быть тот, кто будет играть ключевую роль в одном из процессов внутреннего проектирования. Что касается надежности, здесь ставится цель обеспечить максимально возможную аккуратность и точность в определении требований пользователя, чтобы организация — разработчик программного обеспечения могла транслировать эти требования в проект с минимальным числом ошибок.

Я не собираюсь описывать здесь сам процесс определения требований. В него входят анализ существующих систем, беседы с пользователями, проведение исследований осуществимости и оценки достоинств. Соответствующие методы описаны в работах по системному анализу [2].

Были разработаны автоматизированные системы для описания и обработки требований, но они получили ограниченное распространение [3, 4]. Системы такого рода, например Information Algebra ADS и TAG, ориентируются на небольшие, специализированные и исключительно хорошо определенные задачи.

Примером «ручных» полуформальных методов описания требо-

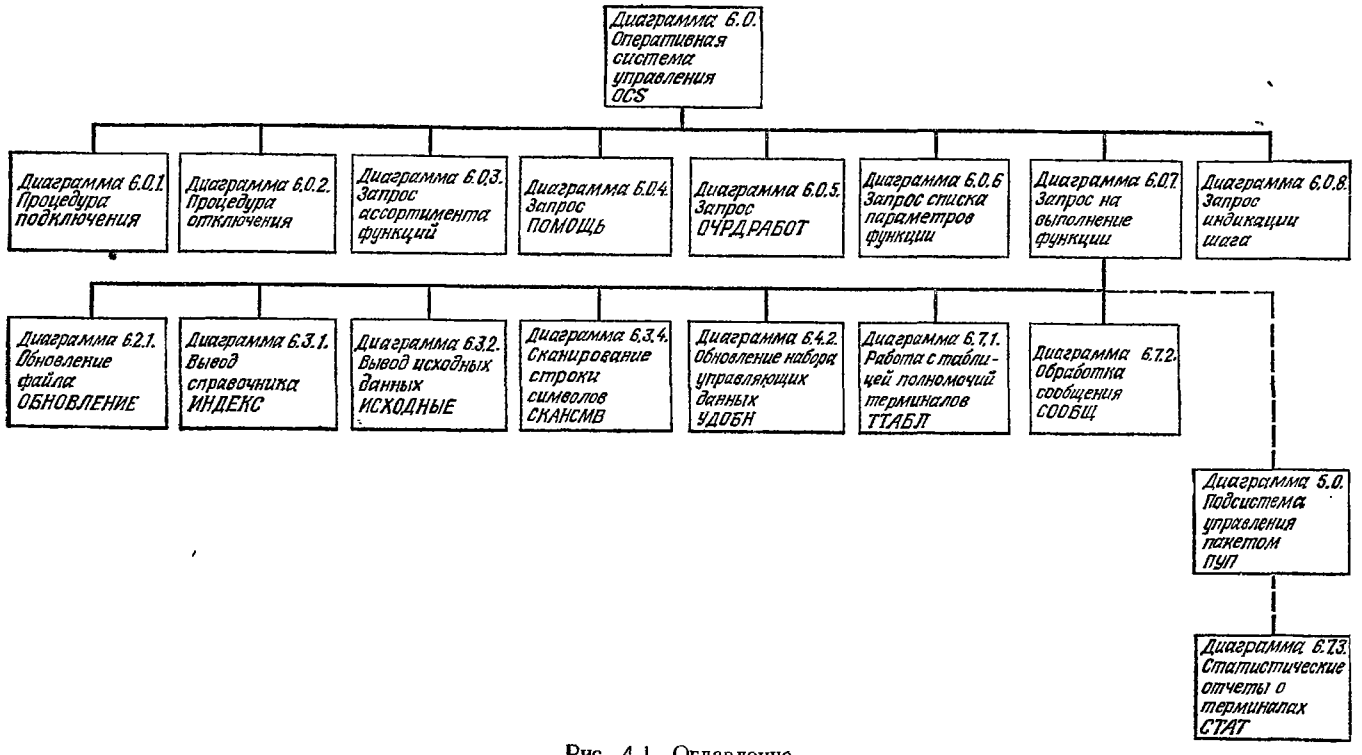


Рис. 4.1. Оглавление.

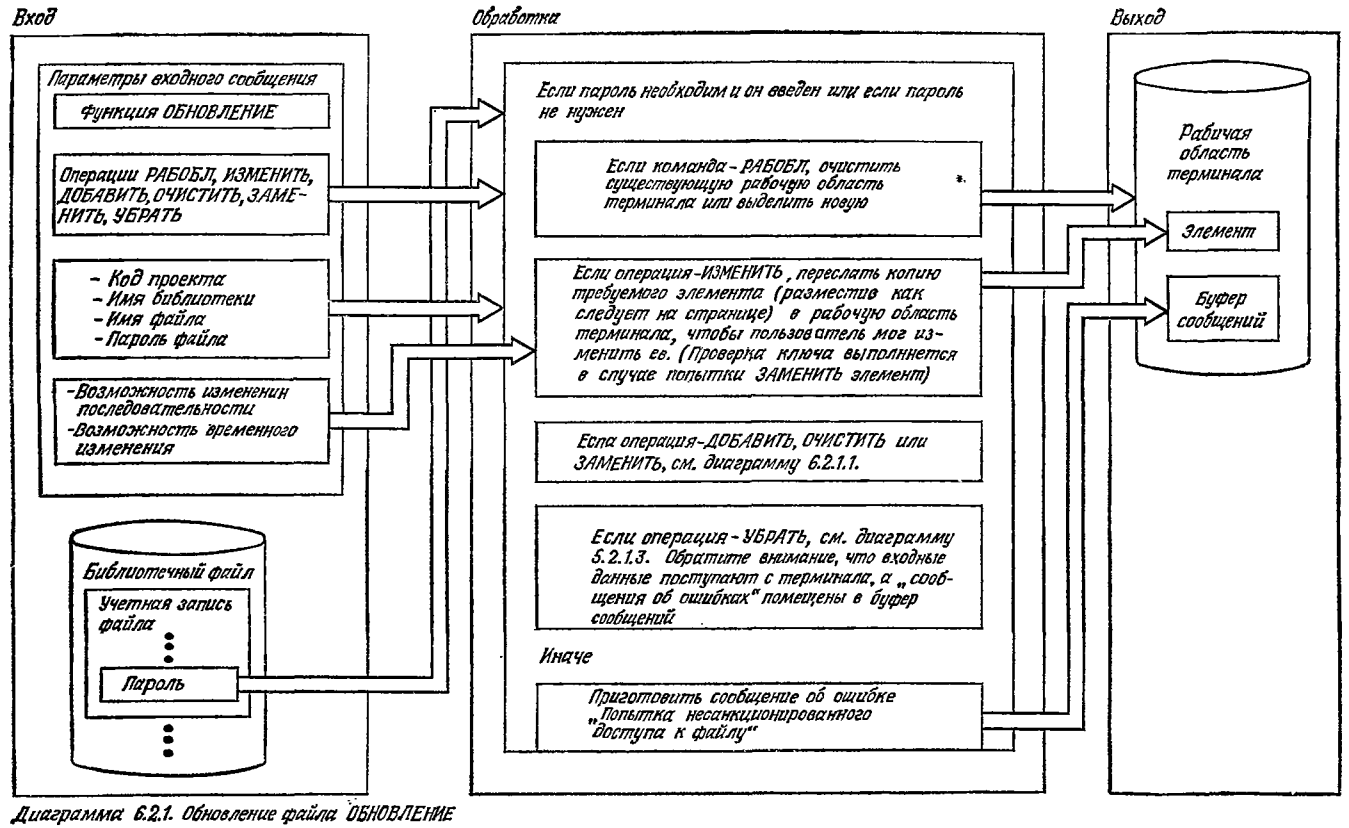


Диаграмма 6.2.1. Обновление файла ОБНОВЛЕНИЕ

Рис. 4.2. HIPO-диаграмма.

ваний может служить метод HIPO-диаграмм (Hierarchy plus Input-Process-Output: иерархия плюс ввод-обработка-вывод) [5]. HIPO-диаграмма строится для каждой основной требуемой функции; в ней дается общая характеристика входных и выходных данных для этой функции и основных шагов обработки. Для отражения иерархической организации таких функций строится *оглавление* (visual table of contents).

На рис. 4.1 и 4.2 изображены оглавление и одна из диаграмм для проекта оперативной библиотеки программ [6]. Каждый прямоугольник на рис. 4.1 представляет крупную функцию, необходимую пользователю. Рис. 4.2 — это HIPO-диаграмма для одного такого прямоугольника.

При использовании этого метода диаграммы обычно готовит организация-разработчик, а затем предъявляет их пользователю для проверки. Диаграммы не содержат никаких предположений о конечной структуре программы и не должны быть написаны на программистском жаргоне. Их задача — дать пользователю наглядное представление о будущем продукте в надежде услышать от него: «Да, я хотел, чтобы программа делала именно это».

При использовании этого метода важно понимать, что HIPO-диаграммы выражают только функциональные требования. Другие требования (например, надежность, производительность, стоимость) должны быть описаны отдельно.

О методах проверки правильности требований можно сказать лишь, что пользователь несет ответственность за проверку требований на полноту и точность, а разработчик — за проверку осуществимости и понятности. В процессе проверки требований желательно установить приоритеты для каждого из них, чтобы помочь разработчику программного обеспечения принимать компромиссные решения на следующих этапах проектирования.

ЦЕЛИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Второй процесс разработки программного обеспечения — постановка целей. Цели — это конкретные ориентиры для программного продукта. Процесс их постановки — прежде всего процесс принятия компромиссных решений. Например, не столь уж невероятно встретить документ, в котором требования формулируются в следующей форме:

> «Обеспечить максимальную надежность, эффективность, адаптируемость, общность и безопасность системы, минимизируя стоимость и время разработки, требуемую память и время реакции терминала».

Такого рода формулировки бессмысленны, поскольку многие из перечисленных факторов противоречат друг другу. Таким образом, назначение постановки целей — не только сами цели, но, если нужно, и компромиссы между этими целями.

При постановке целей распространены следующие ошибки.

1. Цели не формулируются явно.

2. Составляется беглый набросок списка целей, причем жизненно важные цели в него не включаются.

3. Цели формулируются, но таким образом, что они конфликтуют друг с другом. В результате каждый программист, работающий над проектом, разрешает эти конфликты сам, причем все разрешают их по-своему. В проекте с участием ста человек это приведет к тому, что компромиссные решения будут принимать по крайней мере сотней различных путей, что дает совершенно непредсказуемый результат.

4. Необходимость конкретных целей признается, но цели формулируются только для продукта, а не для проекта. При разработке программного обеспечения необходимы два набора целей: *цели продукта*, т. е. определение целей окончательного результата с точки зрения пользователя, и *цели проекта*, такие, как график, стоимость, степень тестированности и т. д.

ПОНИМАНИЕ КОМПРОМИССОВ

Как уже говорилось, постановка целей требует рассмотрения компромиссов между конфликтующими требованиями. Ошибка 3, когда не определяются необходимые компромиссные решения на уровне проекта в целом,— ошибка серьезная и распространенная.

Цели программного обеспечения можно разбить на десять больших групп, одну из которых представляет интересующая нас надежность. Поскольку в этой книге ей уделяется основное внимание, мы рассмотрим, как она соотносится с девятью другими группами.

Общность

Общность характеризуется числом, мощью и областью действия предоставляемых пользователю функций. Хотелось бы надеяться, что документ, в котором определяются требования или цели, никогда не будет содержать формулировок типа «добейтесь максимальной общности»; в нем просто должны быть перечислены необходимые пользователю функции.

Требования надежности и общности конфликтуют между собой, поскольку «обобщенные» системы обычно больше и сложнее. Разрешение этого конфликта, однако, не предполагает, что программные продукты должны быть крайне специализированными. Конфликт можно сгладить, избегая обобщений в тех аспектах, которые не очень или вовсе не важны для пользователя. Например, некоторые компиляторы предлагают пользователю столько дополнительных возможностей, что простое их перечисление занимает несколько стра-

ниц. К тому же не исключено, что некоторые из них так никогда и не используются. Эти отнюдь не обязательные варианты усложняют компилятор, что, несомненно, ведет к ошибкам.

Конфликт общности и надежности разрешить сложнее всего, принятие решений здесь требует особенно серьезной проработки. Необходимо помнить, что всякий элемент взаимодействия с пользователем, какой бы мелочью он ни казался, оказывает какое-то влияние на надежность. Каждая функция программного обеспечения должна быть оценена с точки зрения реальной ее выгоды для пользователя и ее влияния на надежность.

Психологические факторы

Психологические факторы готовой системы — это мера легкости ее понимания и удобства использования, защищенности от неправильного употребления и, как результат, частоты ошибок пользователя. Хотя «гуманизация» взаимодействия с пользователем может увеличить сложность системы и, таким образом, отрицательно влияет на надежность, психологические факторы и надежность, вообще говоря, не находятся в конфликте. Например, когда неожиданно меняются условия функционирования операционных систем, в них проявляется множество «затаившихся» ошибок. Эти новые условия часто являются результатом неожиданных действий пользователя. Хороший учет психологических факторов позволяет свести к минимуму возможность таких неожиданных действий, что уменьшает и возможность проявления «затаившихся» ошибок.

Адаптируемость

Адаптируемость — это мера легкости расширения продукта, например добавления еще одной потребовавшейся пользователю функции. Требования адаптируемости и надежности, как это ни удивительно, согласуются между собой. Дополнительное преимущество многих из рассматривавшихся в предыдущих главах методов проектирования, призванных обеспечить надежность,— их положительное влияние на расширяемость и адаптируемость. Это означает, что рассматриваемые методы проектирования позволяют создавать программные системы, которые не только более надежны, но и легче расширяются.

Удобство сопровождения

Удобство сопровождения — это мера затрат времени и средств на исправление ошибки в работающей системе. Это требование согласуется с требованием надежности, поскольку оно тесно связано с адаптируемостью. Кроме того, такие методы обеспечения надеж-

ности, как обнаружение и изоляция ошибок, положительно влияют на удобство сопровождения системы.

Безопасность

Безопасность — это мера вероятности того, что один пользователь системы может случайно или намеренно обратиться к данным, являющимся собственностью другого пользователя, разрушить их или помешать работе системы. Средства защиты (обеспечения безопасности) включают тщательную изоляцию данных и программ разных пользователей друг от друга и от операционной системы. Поскольку они схожи со средствами изоляции ошибок, они обычно согласуются со стремлением к надежности.

Документация

Этот вопрос касается качества и количества публикаций для пользователя. Цели здесь аналогичны тем, которые касаются психологических факторов,— они тоже связаны с легкостью понимания и использования продукта. Поэтому цели документирования не конфликтуют со стремлением к надежности.

Стоимость продукта

Стоимость программного продукта включает затраты на первоначальную разработку плюс сопровождение продукта. В гл. 1 обсуждалось соотношение между надежностью и возрастающей стоимостью программного обеспечения; было показано, что рост стоимости во многом вызывается возрастанием числа ошибок. Поэтому можно считать, что стремление к высокой надежности и минимальной стоимости разработки и сопровождения не противоречат друг другу.

Календарный план

Еще одна ключевая цель всякого проекта — получить результат к определенному сроку. Выгоды от создания продукта обычно существенно зависят от того, когда он станет доступным. Многие соотношения, связывающие надежность и стоимость программного обеспечения, имеют место также между надежностью и календарным планом. Одна из главных причин срывов графика — ненадежность создаваемого продукта. Например, имеется тенденция чрезвычайно сильно недооценивать время, необходимое для тестирования. Действительно, требующееся на тестирование время тесно связано с числом ошибок, остающихся в программном обеспечении после завершения его проектирования, и это время можно сократить, умень-

шая число ошибок, порождаемых в процессах проектирования. Две тенденции: обеспечить надежность и сократить затрачиваемое на разработку время — вполне согласуются между собой при условии, конечно, что сроки не сокращены до такой крайности, когда на надлежащее проектирование просто не остается времени.

Эффективность

Соотношение между эффективностью, или производительностью, и надежностью крайне сложны. Так, средства обеспечения надежности могут увеличить число команд, которые придется выполнять (например, для обнаружения и регистрации ошибок), увеличить накладные расходы из-за иерархической структуры программы и запретить использование трюков при программировании, что может увеличить время выполнения программы и необходимый объем памяти. С другой стороны, о ненадежной системе нельзя утверждать, что она «эффективна», независимо от того, как быстро она работает. Короче говоря, все общие утверждения о надежности и эффективности (кроме этого) неверны.

Скорость выполнения может быть охарактеризована двумя мерами: мгновенной скоростью продвижения (временем ответа терминала или временем обработки сообщения) и пропускной способностью (полной работой, выполняемой за определенный период времени). Надежность в определенном смысле конфликтует со скоростью продвижения и в то же время согласуется с пропускной способностью (последняя зависит от многих факторов, в их числе — процент времени, когда система работоспособна). Можно провести аналогию с автогонками. Драгстер — это система, для которой целью является мгновенная скорость. Поскольку такого рода соревнования продолжаются лишь несколько секунд, проектировщика не заботит надежность. Он пытается минимизировать вес машины, и его не волнует, что она разваливается на части после каждой гонки. С другой стороны, в таких соревнованиях, как «Индианаполис 500», более важна «пропускная способность». При проектировании автомобиля для продолжительных соревнований выносливость машины часто важнее, чем ее максимальная скорость. Проектировщик заботится о минимизации задержек на ремонт, оптимизации системы смазки и охлаждения мотора, защите водителя при авариях и т. п.

Производство ЭВМ прошло несколько различных этапов решения вопроса о компромиссе между надежностью и эффективностью. В первые годы, когда аппаратное обеспечение было ненадежно, а приложения представляли собой длинные программы для научных расчетов в пакетном режиме, преобладало требование эффективности. Программист заботился о том, чтобы выжать каждую секунду из своей программы и иметь шанс закончить ее до очередного отказа

аппаратуры. Пакетный режим позволял программисту, обнаружив ошибку, просто исправить ее и заново прогнать программу.

В 1960 годах промышленность вступила во вторую фазу, которая продолжается и сейчас и сохранится еще некоторое время. Надежность и скорость аппаратуры увеличилась на несколько порядков. Прикладные системы становятся системами реального времени, управляющими критическими процессами, оказываются неотъемлемой частью повседневной деятельности многих корпораций. Для большинства таких приложений неэффективность терпима, но ошибки — нет. Я цитирую слова одного из моих знакомых из State Farm Automobile Insurance [7]:

«Программное обеспечение системы State Farm обслуживает порядка 15 000 000 владельцев полисов и обрабатывает страховые премии на сумму порядка 2.3 миллиарда долларов ежегодно. В условиях, когда от аккуратности системы зависит так много долларов и людей, надежность должна предшествовать эффективности. Кроме того, State Farm — компания по страхованию автомобилей, и как таковая она зависит от пожеланий около 50 членов государственной комиссии по страхованию и государственных законодательных учреждений. Другими словами, модификации программы — это постоянный процесс, образ жизни. Поэтому удобство сопровождения также более важно, чем эффективность».

Пока промышленность находится во второй фазе, следует считать, что всякий разработчик программного обеспечения, жертвующий надежностью ради эффективности, поступает безответственно. (Как везде, здесь тоже есть исключения, но они должны составлять лишь небольшой процент сегодняшних программных проектов.) Аппаратура ЭВМ постоянно становится дешевле и быстрее. Неэффективность программного обеспечения можно, если это необходимо, устранить позднее; избавиться от недостаточной надежности гораздо сложнее. Результаты неэффективности обычно предсказуемы (более длительное ожидание); результаты ненадежности непредсказуемы и часто разрушительны.

Можно предположить, что промышленность вступит в третью фазу; возможно, это произойдет в конце столетия. Хотя абсолютная надежность останется неосуществимой целью, может быть достигнут такой уровень, когда разработка программного обеспечения станет настолько точной дисциплиной, что высокая надежность будет нормой. В этот момент проектировщики снова смогут сконцентрировать свое внимание на эффективности, решая, например, задачи искусственного интеллекта, неразрешимые сейчас вследствие их вычислительной сложности. До тех пор, однако, основное внимание должно уделяться надежности программного обеспечения даже, где это необходимо, за счет эффективности.

ПОСТАНОВКА ЦЕЛЕЙ ДЛЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

При правильной постановке целей для программного обеспечения не делается никаких предположений о конкретной реализации, но указывается, каким образом на последующих этапах проектирования следует принимать компромиссные решения. Как уже говорилось, должны быть поставлены цели двух типов: цели продукта и цели проекта.

Цели продукта

Это цели с точки зрения пользователя. Должна быть представлена следующая информация.

1. *Резюме.* Вначале нужно коротко сформулировать общее назначение разрабатываемого продукта.
2. *Определение пользователя.* Если разрабатывается большая система, с разными группами пользователей, должны быть определены роли различных пользователей.
3. *Подробное перечисление функций.* Здесь с точки зрения пользователя следует обрисовать функции, которые должны обеспечиваться системой.
4. *Публикации.* Должны быть определены цели для документации, поставляемой пользователям, в том числе типы документации и предполагаемый круг читателей для каждого типа.
5. *Эффективность.* Сюда относятся все цели, касающиеся эффективности, или производительности, такие, как временные характеристики, пропускная способность, использование ресурсов, а также необходимые средства измерения производительности и средства настройки.
6. *Совместимость.* Если конкретный программный продукт должен быть совместим с другими, эти цели указываются здесь. Следует указать также относящиеся к делу международные и государственные стандарты и внутренние стандарты компании.
7. *Конфигурации.* Здесь указываются различные конфигурации аппаратуры и программного обеспечения, в которых система может работать, и другие программные продукты, от которых она зависит, а также дополнительные возможности выбора отдельных частей системы, если это осуществимо.
8. *Безопасность.* Сюда относится описание целей в отношении обеспечения безопасности. Если система связана с финансовой деятельностью (например, игорная система или система управления запасами), должны быть указаны средства надзора.
9. *Обслуживание.* Здесь намечаются стоимость и время исправления ошибок, а также необходимые для достижения намечаемых параметров программные средства, например диагностические программы.

10. *Установка.* Сюда относятся методы и средства настройки системы на конкретные условия эксплуатации.

11. *Надежность.* Цели в области надежности, как и другие цели, существенно зависят от конкретного типа системы. Следующие вопросы, однако, нужно рассматривать обязательно.

a. Для каждого типа отказов должно быть определено среднее время между отказами (отказ системы, ошибка пользователя, отказы конкретных функций) с учетом серьезности такого отказа.
b. Среднее время восстановления системы после отказа.
c. Ориентиры в отношении числа ошибок в программном обеспечении, расклассифицированные по серьезности и времени обнаружения.
d. Последствия отказов системы или отказа наиболее важных функций.
e. Допустимый объем данных, утрачиваемых в случае отказа.
f. Жизненно важная информация, которая должна быть защищена от разрушения.
g. Функции, необходимые для обнаружения ошибок.
h. Функции, необходимые для исправления ошибок.
i. Функции, необходимые для обеспечения устойчивости к ошибкам.
j. Возможности обнаруживать ошибки пользователя и сбои аппаратуры, а также восстанавливать работоспособность.

Цели проекта

Здесь описываются цели, которые касаются только самого процесса разработки и не проявляются в окончательном результате работы. Они должны быть установлены официально, ибо, как показали Вейнберг и Шулмен [8], в случаях, когда программисты не имеют списка целей проекта, они получают противоречивые и неожиданные результаты. Цели проекта должны давать ответы на следующие вопросы:

1. Ориентировочная стоимость каждого процесса.
2. Календарный план проекта.
3. Цели для каждого процесса тестирования.
4. Цели в области адаптируемости, указывающие степень адаптируемости, или расширяемости, которая должна быть достигнута.
5. Вопросы сопровождения создаваемой системы, которые необходимо учитывать при разработке.
6. Уровни надежности, которые должны быть достигнуты на каждом этапе разработки для достижения заданной надежности продукта.
7. Внутренняя документация при работе над проектом.
8. Критерии для оценки готовности продукта к использованию.

Дополнительные подробности по каждому из этих вопросов можно найти в последующих главах, в особенности в части 3.

Общие правила постановки целей

Крайне важно, чтобы цели были *четкими*, *явными*, *разумными* и *измеримыми*. Цель, которую нельзя понять, бесполезна. Ни в коем случает нельзя скрывать цели от разработчиков программного обеспечения; они должны быть известны всем участникам проекта. Цели должны быть достижимы — исследования классических проектов показали, что развертывание работ с недостижимыми целями часто является главной причиной неудачи. Все цели должны быть сформулированы по возможности в количественных терминах, чтобы можно было оценить, в какой мере в окончательном продукте эти цели достигнуты. Один из описанных в части 3 процессов тестирования предполагает проверку соответствия продукта поставленным целям.

Каждая цель должна быть сформулирована достаточно подробно, как того требуют процессы проектирования, но не должна предполагать конкретных проектных решений. Должны быть определены зависимости между целями, чтобы в случае, когда изменяется или не может быть достигнута конкретная цель, проектировщик легко мог определить, как это сказывается на других целях. Для каждой цели должен быть установлен приоритет (например, по шкале от «абсолютно необходимо» до «хорошо, но не обязательно»), чтобы иметь основу для принятия компромиссных решений.

Очень полезный прием — перечислить в документации конкретные «нецели» вместе с целями. Это предотвращает многие случаи ошибочных «предположений» или «чтения между строк», явно показывая, что́ рассматриваемая система *не* предполагает делать. Очевидно, «нецели» любого проекта неисчислимы. Однако, предвидя некоторые возможные недоразумения, следует составить небольшой список «нецелей» для уменьшения этих недоразумений.

Оценка целей

Правило «*n*-плюс-минус-один» предлагает привлечь к оценке целей автора требований и проектировщика исходных внешних спецификаций. Поскольку, однако, цели — самый важный аспект проекта, необходимо участие в их оценке дополнительных сил. К оценке должны быть привлечены полномочные представители пользователей, а также тех, кто будет заниматься проектированием, тестированием, сопровождением и подготовкой публикаций.

Первый шаг — сопоставить цели с требованиями, чтобы убедиться, что все требования правильно переведены на язык целей. Каждая цель должна оцениваться с учетом правил, приведенных в предыду-

щем разделе. Так как этот вопрос очень важен, каждая из целей должна быть лично оценена представителями нескольких уровней руководства как в организации-пользователе, так и в организации-разработчике.

ВНЕШНЕЕ ПРОЕКТИРОВАНИЕ

Внешнее проектирование — это процесс описания ожидаемого поведения разрабатываемого продукта с точки зрения внешнего по отношению к нему наблюдателя. Цель этого процесса — «конструирование» внешних взаимодействий будущего продукта (обычно с пользователем) без конкретизации его внутреннего устройства. Внешний проект выражается в форме внешних спецификаций, предназначенных для широкой аудитории, включающей пользователя (для проверки и одобрения), авторов документации для пользователя, всех участвующих в проекте программистов, а также всех тех, кто будет заниматься тестированием продукта.

Подготовка полных и правильных внешних спецификаций — сегодня самая ответственная задача в разработке программного обеспечения. Как мы видели на рис. 2.1 и 3.2, внешние спецификации участвуют в большем числе процессов перевода, чем любой другой проектный документ.

Хотя методологии внешнего проектирования не существует, важно соблюдать принцип *концептуальной целостности* [9]. Концептуальная целостность — это гармония (или стремление к ней) между внешними функциями системы; в соответствии с этой концепцией лучше иметь относительно небольшой набор хорошо согласованных функций, чем, возможно, больший набор независимых и нескоординированных функций. Особенности, которые кажутся привлекательными, но не согласуются с остальными, вероятно, следует отклонить, чтобы не усложнять взаимодействие с пользователем.

Концептуальная целостность представляет собой меру единообразия способа взаимодействия с пользователем. Система, лишенная концептуальной целостности,— это система, в основе которой нет единообразия; в результате такая система характеризуется слишком сложным взаимодействием с пользователем и излишне сложной структурой. Поскольку концептуальная целостность — всего лишь идея, ее трудно описать в деталях, так как она изменяется в зависимости от применения. Например, система с разделением времени, обладающая концептуальной целостностью, будет иметь по крайней мере следующие характеристики. Все возможности, доступные пользователю переднего плана (за терминалом), доступны также пользователю заднего плана (пакетный режим), и наоборот. Сопряжения с пользователем для переднего и для заднего планов совпадают. Это значит, что, вместо того чтобы вводить приказы с терминала, пользователь может ввести их в виде задачи заднего пла-

на. Все приказы обладают внутренней симметрией по отношению к синтаксису, именам, операндам, соглашениям и правилам умолчания. Семантика всех приказов согласована. Например, если операнд «ФАЙЛ» одного приказа приводит к пересылке сообщений системы в файл вместо выдачи их на терминал, тогда все приказы, для которых возможен подобный вариант, должны использовать слово «ФАЙЛ». Характеристики терминала, запросы и сообщения об ошибках — общие для всех приказов.

Простейший способ добиться *отсутствия* концептуальной целостности — попытаться разрабатывать внешний проект слишком большой группой. Магическое число здесь, кажется, два. В зависимости от масштабов проекта ответственность за внешнее проектирование должны нести один-два человека.

Последнее утверждение, вероятно, шокирует читателя, который в настоящий момент разрабатывает новую операционную систему и с тридцатью сотрудниками пишет внешние спецификации. Опыт, однако, показал, что вероятность успеха резко падает, когда число разработчиков превосходит два, даже для крупного проекта. Это не означает, что в процессе проектирования должны принимать участие только двое; требуется, чтобы ответственность за этот процесс несла лишь маленькая группа людей, принимая все решения и готовя спецификации. В случае крупного проекта этим людям необходима помощь исследователей, ассистентов, чертежников, секретарей и т. д. Помощники занимаются сбором и обработкой информации, но не проектированием, т. е. принятием решений или собственно написанием спецификаций.

Кем же тогда должны быть эти избранные ответственные лица? Начнем с того, что уж во всяком случае они не должны быть программистами или недавними выпускниками-теоретиками. Внешнее проектирование мало чем связано (если связано вообще) с программированием; более непосредственно оно касается понимания обстановки, проблем и нужд пользователя, психологии общения человека с машиной. Более того, эта сторона внешнего проектирования становится все более значительной по мере того, как применение ЭВМ все больше начинает затрагивать пользователей, не знакомых с программированием. Программисты и дипломированные специалисты по теории программирования обычно не имеют подготовки и опыта в проектировании такого типа. Они могут быть специалистами по языкам программирования, алгоритмам, методам программирования, теории компиляторов, тестированию и отладке, но у них мало или вовсе нет опыта в использовании ЭВМ, разработке психологических факторов, психологии взаимоотношений человека и машины. Было бы разумно использовать программистов для внешнего проектирования продукта, предназначенного для программистов, например языков программирования или инструментов отладки, но неразумно ожидать, чтобы программист выполнил внеш-

нее проектирование операционной системы или системы диспетчеризации грузовиков.

Из-за сложности внешнего проектирования и его возрастающей важности для разработки программного обеспечения оно требует специалистов особого рода. Такой специалист должен разбираться в упоминавшихся выше областях, быть знакомым со всеми фазами проектирования и тестирования системы, чтобы понимать влияние на них внешнего проектирования. В качестве возможных кандидатов можно назвать системных аналитиков, психологов, занимающихся вопросами поведения, специалистов по исследованию операций, инженеров, а возможно, и опытных специалистов по теории программирования (если их подготовка включает упомянутые области, что бывает редко). Некоторые организации пытались поручать собственно написание внешних спецификаций профессиональным техническим писателям (занимавшимся у них ранее руководствами для пользователей). Это, однако, не рекомендуется, поскольку добавляется лишний шаг перевода (от разработчика спецификаций к писателю). Все же часто бывает желательно привлечь писателя к этой работе, так как его ориентированность на пользователя оказывается очень полезной.

ПРОЕКТИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ С ПОЛЬЗОВАТЕЛЕМ

При проектировании внешних сопряжений системы разработчик интересуется тремя областями, имеющими отношение к надежности программного обеспечения: минимизацией ошибок пользователя, обнаружением ошибок пользователя, когда они все же возникают, и минимизацией сложности. Книга Мартина [10] является блестящим введением в эту область. Дополнительные примеры проектирования диалоговых систем можно найти в книге Орра [11].

Как уже говорилось, наша книга — об ошибках в программном обеспечении, а не об ошибках пользователей. Однако одно с другим связано. Ошибки пользователя увеличивают вероятность перехода системы в непредвиденное состояние. Минимизация ошибок пользователя не уменьшает числа ошибок в программном обеспечении, но увеличивает его надежность за счет уменьшения вероятности обнаружения оставшихся ошибок. Основные правила минимизации ошибок пользователя в диалоговых системах перечислены ниже. Большинство из них имеет аналогию для случая пакетной обработки.

1. Согласовывайте способ взаимодействия с подготовкой и уровнем пользователя, а также с ограничениями, в условиях которых пользователь работает. Например, можно ожидать, что взаимодействие с пользователем банковской системы должно существенно различаться в зависимости от того, является ли пользователь клиентом банка или опытным кассиром. Я вспоминаю анекдотический пример

того, как не надо делать: больничная информационная система выводила на терминал даты в такой форме:

$$2.5 \times 10^1 \text{ ОСТ } 1.969 \times 10^3$$

2. Проектируйте таким образом, чтобы сообщения, вводимые пользователем, были как можно короче, но не настолько, чтобы исчезла их осмысленность. При этом учитывайте частоту работы с системой для среднего пользователя (часто или изредка), а также возможность стрессовой ситуации для пользователя в момент его работы с системой.

3. Обеспечьте концептуальную целостность для разных типов вводимых и выводимых сообщений. Например, все сообщения, выдачи на экран дисплея, отчеты должны иметь одинаковые форматы, стиль, сокращения.

4. Обеспечьте средства «помощи»: специальный набор функций для оказания пользователю помощи, если тот запутается или забудет какое-то правило взаимодействия. Пример мощных средств помощи демонстрирует Система разделения времени Беркли [12]. Так, пользователь может ввести сообщение "?HELP!HOW CAN I TERMINATE AN EDIT?" («Как мне закончить редактирование?») и получить объяснение в 50 слов с указанием того, как можно вернуться из режима редактирования в режим приказов.

5. Старайтесь, чтобы система не рассердила пользователя, ибо это может привести к некоторым неожиданным ситуациям на входе. Избегайте оскорбительных сообщений системы, общайтесь с пользователем на его языке, а не на тарабарском жаргоне программистов.

6. Всегда на каждое входное сообщение выдавайте какое-нибудь уведомление. Без этого пользователь может засомневаться, правильно ли сообщение было введено, и попытается повторить ввод, вследствие чего может возникнуть ошибочная ситуация.

Помимо минимизации ошибок пользователя, система должна также надлежащим образом обращаться с ошибками, если они все-таки возникают,— а возникать они будут независимо от того, насколько хорошо были спроектированы правила взаимодействия. Например, операторы Нью-Йоркской системы диспетчеризации полицейских машин SPRINT в свободное время развлекались тем, что пытались вывести систему из строя, вводя заведомо неправильные сообщения [10]. Основные правила обнаружения ошибок пользователя перечислены ниже.

1. Спроектируйте систему так, чтобы она принимала *любые* данные. Если введенная информация не является тем, что система считает допустимым, она должна информировать пользователя.

2. Если пользователь вводит сложное сообщение, особенно если для этого нужно несколько обращений к системе, позвольте ему про-

верить это сообщение прежде, чем оно будет обрабатываться, например так, как это показано ниже.

Пользователь: 222, АА, 23МАЙ, 2 F
Машина: ВАМ НУЖНЫ 2 МЕСТА ПЕРВОГО КЛАССА НА РЕЙС 222
AMERICAN AIRLINES ИЗ ORD
(ЧИКАГО) ДО JFK (НЬЮ-ЙОРК) НА 23 МАЯ, ВЫЛЕТ В 8.00 ВЕЧЕРА, ПРИБЫТИЕ В 10.30 ВЕЧЕРА.
ВЕРНО?

3. Проектируйте систему так, чтобы ошибки пользователя обнаруживались немедленно. Если пользователь выдает сообщение «Послать товарные вагоны F23 и F42 из Чикаго в Де-Плейнс поездом 883», система немедленно должна проверить, находятся ли эти вагоны уже в Чикаго и есть ли поезд 883 из Чикаго в Де-Плейнс. Есть две причины, заставляющие проверить это немедленно. Во-первых, пользователь может рассылать много вагонов, и план его дальнейших действий может опираться на то, что F23 и F42 направлены в Де-Плейнс. Если позднее обнаружится, что F23 — в Питтсбурге, а не в Чикаго, весь план рушится, после чего пользователь должен «вернуть» все сообщения о рассылке. Кроме того, если ошибка не будет обнаружена сразу, вычислительная система может запутаться и остановиться, встретив записи о том, что F23 одновременно находится и в Питтсбурге, и в Де-Плейнсе.

4. Там, где особенно важна аккуратность, обеспечьте избыточность входных данных. В банковской системе можно потребовать, чтобы фамилия клиента вводилась вместе с номером счета, так чтобы можно было обнаруживать ошибки при вводе номеров счетов. Другая возможность — сделать номер счета самопроверяемым за счет того, чтобы, например, две последние цифры были некоторой арифметической комбинацией предыдущих цифр [13].

Непосредственное отношение к надежности имеет еще одна задача — минимизация сложности внешнего проекта с целью уменьшения внутренней сложности будущей системы и минимизации ошибок пользователей. Распространено представление, что «гуманизированный» внешний проект должен быть сложным, что он предполагает фантастические по мощности методы, много дополнительных возможностей, автоматическое исправление ошибок (например, орфографических). Это представление ошибочно.

Вопрос, который всегда возникает при внешнем проектировании диалоговой системы,— подсказывать ли пользователю, какие именно части входного сообщения ему следует повторить, когда в них

обнаружена ошибка. Предположим, что пользователь вводит приказ

EDIT GJMPROGSET.1 TYPE(TEXT) DISP(80)

и система обнаруживает ошибку: TEXT — недопустимый операнд для ключевого слова TYPE. Следует ли системе обращаться к пользователю за исправлением, печатая, например, INVALID TYPE. ENTER TYPE FIELD (Неправильный тип. Введите поле типа)? Во-первых, теперь пользователь смущен. Должен ли он вводить ключевое слово и операнд или только операнд? Вместо того чтобы стоять перед такой дилеммой, было бы проще заново ввести весь приказ. Кроме того, просьба ввести только исправления требует, чтобы система делала предположения относительно того, что правильно, а что — нет. Что, если на самом деле ошибка — не в TEXT? Предположим, что сообщение было бы правильным, если бы вместо EDIT было сказано FILE? Что делать пользователю теперь?

Такого рода подсказок следует избегать, поскольку они запутывают пользователя и чрезвычайно усложняют программное обеспечение. Если вы настаиваете на том, что в вашей системе они необходимы, поскольку приказы в ней так длинны, что повторный их ввод займет слишком много времени, то вам могут возразить, что у вас просто плохо спроектированы приказы.

Вторая проблема, связанная со сложностью,— предоставление пользователю слишком большого числа дополнительных возможностей и вариантов. В операционной системе OS/360 имеется процесс настройки, называемый SYSGEN (генерация системы), позволяющий перекраивать систему при ее настройке. Это привело к тому, что почти каждая установка OS/360 имеет свою уникальную версию операционной системы, что усложняет работу IBM по сопровождению. Это в свою очередь наводит на мысль о том, что для IBM было бы дешевле убрать процесс SYSGEN, но обеспечивать каждую установку достаточным количеством «свободной» памяти для хранения тех частей системы, которые для этой установки не нужны.

Всякий раз, когда есть сомнения, давать пользователю некоторую возможность или нет, сейчас слишком часто принимается решение «давать». В системе, с которой я недавно работал, две страницы руководства для пользователей занимает простое перечисление (без описаний) всех вариантов вызова компилятора PL/1. Это не только запутало меня, когда я пытался сообразить, как же скомпилировать мою программу на PL/1, но наверняка усложнило компилятор, что, несомненно, явилось причиной ошибок. Вообще говоря, обилие дополнительных возможностей компилятора неблагоприятно сказывается на работе пользователя. Разработчик должен тщательно рассмотреть каждую из них, сопоставляя ее полезность и степень усложнения компилятора. Когда имеются сомнения, безопаснее отказаться от рассматриваемого варианта. Множество доступных мелких возможностей не повысит конкурентоспособности

продукта. Кроме того, эти возможности могут негативно повлиять на потенциального покупателя, показывая, что разработчик не имеет ясного представления о том, как именно его система будет использоваться.

Последний вопрос, связанный со сложностью,— на каких предположениях основывается система, воспринимая входные данные; например, из каких соображений она пытается исправлять ошибки. Методы такого рода настолько сложны, что часто становятся опасными. Рассмотрим медицинскую информационную систему, у которой пользователь запрашивает этиловый спирт, и предположим, что он случайно нажимает на лишнюю клавишу, вследствие чего входное сообщение приобретает вид «РЭТИЛОВЫЙ СПИРТ». Система не опознает первое слово и передает алгоритму исправления, который решает, что в действительности пользователь имел в виду «МЕТИЛОВЫЙ СПИРТ». Всякий, кто прослушал хотя бы вводный курс химии, должен понимать огромную разницу между тем и другим: этиловый спирт опьяняет, метиловый — убивает.

Таким образом, для обеспечения надежности при разработке взаимодействия с пользователем необходимо обеспечить его единообразие и простоту, ожидать на входе всего и немедленно обнаруживать как можно больше ошибок.

ПОДГОТОВКА ВНЕШНИХ СПЕЦИФИКАЦИЙ

Правильно составленные внешние спецификации — объемистый документ. Чтобы справиться с таким большим документом, лучше всего применить иерархическую организацию. В одном из проектов я недавно применил метод, который состоит в том, чтобы разбить спецификации на *основные компоненты*, затем просто *компоненты*, и, наконец, *функции*. Одной из основных компонент был управляющий библиотекой. В рамках этой основной компоненты примером компоненты может быть администратор секретности. В рамках этой компоненты примером функции был приказ СООБЩИТЬ О НАРУШЕНИЯХ СЕКРЕТНОСТИ.

В гл. 2 внешнее проектирование представлено в виде двух процессов: предварительного и детального внешнего проектирования. В случае если спецификации организованы иерархически, двум этим процессам соответствуют просто контрольные точки в процессе нисходящего проектирования системы в соответствии с этой иерархией. Первый шаг — обрисовать основные компоненты (или то, что находится на первом уровне), затем— компоненты, затем — внешние функции (функции пользователя) и в конце концов детали всех функций пользователя. Предварительное внешнее проектирование включает три первых шага. Система проектируется до такого уровня, когда уже выделены все функции пользователя, но точные их синтаксис, семантика и выходные результаты остаются еще не опреде-

ленными. Этим преследуются две цели: во-первых, внутри продолжительного процесса внешнего проектирования устанавливается контрольная точка для руководства и, во-вторых, становится возможной проверка правильности промежуточного уровня проекта и сопоставление его с поставленными целями.

Детальный внешний проект каждой функции пользователя должен освещать следующие вопросы.

1. *Описание входных данных.* Точное описание синтаксиса (например, формат, допустимые значения, области изменения) и семантики всех данных, вводимых пользователем. Этими «данными» могут быть приказ, управляющая карта, входной документ, ответ на подсказку или аналоговый сигнал.

2. *Описание выходных данных.* Точное описание всех результатов функции (например, реакция терминала, сообщения об ошибках, отчеты, аналоговые управляющие сигналы). Должна быть описана функциональная связь входных данных с выходными; это значит, что читатель спецификаций должен быть в состоянии представить себе выходные данные, порождаемые каждым конкретным вариантом входных. Для каждой функции должны быть указаны также результаты для всех неправильных входных данных.

3. *Преобразования системы.* Многие внешние функции не только порождают выходные данные, но изменяют также состояние системы. Здесь должны быть описаны все такие преобразования системы, но при этом следует помнить, что речь идет о внешних спецификациях, поэтому преобразования должны быть описаны с точки зрения пользователя. Например, складская система могла бы иметь команду «заказ», которая используется продавцом, чтобы начать выполнение заказа. Эта функция имеет два выходных результата: накладная, которая пересылается в отдел доставки, и подтверждение, выдаваемое на терминал продавца. Она вызывает также два преобразования системы: обновляется инвентарный файл и заказ регистрируется в файле контроля.

4. *Характеристики надежности.* Описание воздействия всех возможных отказов функций на саму систему, файлы и пользователя. Я обнаружил, что включение этого раздела оказывает небольшое, но положительное влияние на надежность. При разработке диалоговой системы я попросил, чтобы фраза «никакая ошибка в этом приказе не должна приводить к отказу системы» повторялась в этом разделе для каждого приказа. Хотя разработчики внутренней структуры системы никогда сознательно не допустят, чтобы ошибка в приказе пользователя вызвала общий отказ системы, эта фраза служит постоянным напоминанием об этом.

5. *Эффективность.* Описание всех требований, которые предъявляются к эффективности функции, таких, как затрачиваемое время и используемая память. Эффективность редко удается указать в

абсолютных терминах, поскольку она зависит от конфигурации аппаратуры, скорости линий связи, эффективности всех остальных выполняемых параллельно программ, числа активных пользователей терминалов и т. д. Чтобы справиться с этой проблемой, в спецификациях можно описать несколько стандартных конфигураций и уровней нагрузки, а затем можно указать эффективность отдельных функций по отношению к ним.

6. *Замечания по программированию.* Внешние спецификации должны описывать продукт с точки зрения пользователя и избегать ограничений на внутреннее устройство системы. Однако иногда бывает необходимо подсказать или сообщить какие-то идеи относительно внутреннего проектирования функции. Практиковать это следует как можно реже, но, если это необходимо, соответствующую информацию следует сообщить в этом разделе.

Для всякой достаточно сложной функции пользователя задача описания функциональных связей результатов и преобразований с входными данными нетривиальна. *Таблица решений* — отличный способ справиться с этой проблемой.

На рис. 4.3 приведена таблица решений для команды ОТГРУЗИТЬ ДЕТАЛИ из спецификаций для складской системы. Команда имеет три операнда: номер покупателя, номер детали и количество. Эта таблица решений, сопровождаемая небольшим описанием, является прекрасной внешней спецификацией команды. Таблицы решений представляют собой наглядный и строгий способ отражения отношений между входными и выходными данными. Чтобы оценить их достоинства, попробуйте выразить представленные на рис. 4.3 отношения в другой форме: обычным словесным описанием.

Таблица решений — полезное средство повышения надежности программного обеспечения. Они уникальны по своим достоинствам в качестве средства общения между разработчиком внешнего проекта, пользователем и программистом. С одной стороны, в них легко может разобраться пользователь, просматривающий внешние спецификации. С другой стороны, они служат идеальными исходными данными для программиста, разрабатывающего внутреннюю логику программы. Это двойственное применение таблиц решений сокращает число ошибок перевода. Кроме того, имеются «механические» методы проверки таблиц решений на неполноту, избыточность и неоднозначность [14]. Например, можно было бы применить эти методы к таблице на рис. 4.3, чтобы проверить, все ли условия на входе предусмотрены. Поскольку многие ошибки в программном обеспечении вызываются непредвиденными условиями, такие «механические» средства весьма ценны.

По каким-то необъяснимым причинам таблицы решений не имеют широкого распространения. Их редко включают в программы обучения программистов как в промышленности, так и в университетах.

Команда ОТГРУЗИТЬ ДЕТАЛИ

	Синтаксис команды правилен?	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
	Номер покупателя правилен?	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	
	Номер детали правилен?	Y	Y	Y	Y	Y	N	N	Y	Y	N	N	
	Количество — положительное и целое?	Y	Y	Y	Y	N	Y	N	Y	N	Y	N	
	Кредит покупателя в порядке?	Y	Y	Y	N								
	Достаточно деталей на складе?	Y	Y	N									
	Достаточно ли деталей останется на складе?	Y	N										
Выходные сообщения	Послать уведомление в отдел доставки	X	X										
	Послать на терминал сообщение 1	X	X										
	Послать на терминал сообщение 2			X									
	Послать на терминал сообщение 3				X								
	Послать на терминал сообщение 4					X		X		X		X	
	Послать на терминал сообщение 5						X	X			X	X	
	Послать на терминал сообщение 6								X	X	X	X	
	Послать на терминал сообщение 7												X
	Послать в отдел закупок уведомление 1	X											
	Послать в отдел закупок уведомление 2			X									
Преобразования	Уменьшить количество в инвентарном файле	X	X										
	Зарегистрировать сделку в файле контроля	X	X										

Рис. 4.3. Таблица решений.

Таблицы решений должны стать важной частью инструментария разработчика программного обеспечения, так как они способны существенно повышать надежность.

После подробного обсуждения всех элементов подготовки высококачественных внешних спецификаций иногда можно услышать такую реакцию: «Да, это, может быть, идеально, но я, вероятно, не смогу представить информацию на таком уровне, не сконструировав сначала всю систему». Такое мнение иногда справедливо, иногда нет. Когда оно справедливо, это значит, что речь идет об исследовательском проекте. В *проекте-разработке* результат может быть получен последовательным рядом фаз, сверху вниз и извне внутрь. *Исследовательский проект* — это такой проект, где задачи и окончательный результат не определены (или не могут быть определены) заранее. Я не пытаюсь принизить исследования; они, безусловно, ценны. Однако проектировщик должен понимать, с какого рода проектом он имеет дело, поскольку исследовательский проект по самой своей природе не может иметь фиксированного календарного плана и оценки стоимости. Большинство нынешних программных проектов является разработками; лишь немногие ставят целью внести что-то новое. Эта книга посвящена разработкам, хотя читатель должен понимать, что такие проекты могут потребовать предварительных исследований.

Резюмируя, можно сказать, что внешние спецификации описывают все возможные входные данные системы (как допустимые, так и недопустимые) и соответствующую реакцию системы. Внешние спецификации должны точно и полно описывать внешние сопряжения при минимальных предположениях о внутреннем устройстве системы (или совсем без них).

ПРОВЕРКА ПРАВИЛЬНОСТИ ВНЕШНИХ СПЕЦИФИКАЦИЙ

Завершение внешнего проектирования является поворотным моментом для всего проекта. Процесс внешнего проектирования и предшествовавшие процессы концентрировались вокруг взаимодействия системы и пользователя; остальные процессы проектирования касаются внутренней структуры программного обеспечения. По этой причине, а также вследствие рассматривавшихся выше трудностей в подготовке высококачественных внешних спецификаций проверка их правильности приобретает исключительное значение.

На этом этапе должна быть занята активная позиция в отношении ошибок. Цель всякого процесса проверки правильности или тестирования — найти как можно больше ошибок, а не показать, что спецификации не содержат ошибок. Очень важно понимать это тонкое различие. Важно также проверить правильность спецификаций, пока еще они имеют вид набросков, поскольку часто, как только

документ приобретает «законченную» печатную форму, возникает психологический барьер, препятствующий внесению изменений.

Имеется шесть методов проверки правильности, применимых к внешним спецификациям. Эти методы не исключают друг друга, я рекомендую применять их все.

Контроль по правилу «*n* плюс-минус один»

Правило «*n* плюс-минус один» применимо и здесь. Специалисты уровня $n-1$ ищут ошибки перевода, а специалисты уровня $n+1$ — неясности, которые могут привести к ошибкам в последующих процессах.

Предварительные внешние спецификации оцениваются теми, кто отвечает за подготовку целей, разработку архитектуры и детальное внешнее проектирование. Спецификации следует сопоставить с целями, рассматривая каждую цель и анализируя, насколько адекватно она отражена в спецификациях.

Детальные внешние спецификации должны оцениваться теми, кто отвечает за предварительное проектирование, разработку структуры программы и проектирование базы данных, а также за подготовку документации для пользователей.

Контроль со стороны пользователя

Крайне важно добиться того, чтобы пользователь просмотрел и одобрил предварительные и детальные внешние спецификации. Если по какой-то причине это не может быть сделано (например, в не зависимом от пользователя проекте), тогда должна быть сформирована специальная группа в самой организации-разработчике, которая призвана исполнять роль «адвоката дьявола»; в ее задачу будет входить оценка спецификаций с точки зрения пользователя.

Таблицы решений

Если в спецификациях используются таблицы решений (а должно быть именно так), тогда для их проверки на полноту (поиск всех пропущенных условий) и неоднозначность (идентичные условия приводят к разным выходным данным или преобразованиям) могут быть применены «механические» методы.

Ручная имитация

Эффективный прием оценки детальных внешних спецификаций — подготовить тесты и затем воспользоваться детальными внешними спецификациями для имитации поведения системы. Этот процесс оценки проектного документа методом «выполнения» его на тесто-

вых данных часто называется *сквозным контролем* (или прослеживанием).

Для проверки отдельных внешних функций должны быть выполнены следующие действия. Кто-то (*не* автор спецификаций) должен сначала построить «тесты на бумаге» для этой функции, т. е. список конкретных входных данных (допустимых и недопустимых). Вместе с автором спецификаций он затем имитирует ввод этих данных в систему, используя спецификации как описание поведения системы. Всякий раз, когда оказывается, что спецификации описывают выходные данные или преобразование для какого-то набора входных данных недостаточно полно и правильно, это означает, что обнаружена ошибка. Прежде чем начинать сквозной контроль, обычно бывает необходимо установить исходное состояние системы (например, описать простые начальные состояния всех файлов) и затем обновлять это описание состояния по мере пользования.

Подобным образом можно тестировать и комбинации функций, строя тесты, представляющие собой сложные сценарии.

Важно отметить, что цель всякого такого сеанса сквозного контроля — обнаруживать ошибки, но не исправлять их с лету. Исправление ошибок нужно отложить до тех пор, пока не будут обнаружены по возможности все ошибки. Важно также, чтобы строил тесты и интерпретировал спецификации кто-то другой, а не автор, так как последний будет «читать между строк».

Имитация за терминалом

Более изощренный метод сквозного контроля — игра в «человека, спрятанного в пустой машине». Вместо того чтобы просто читать список тестов, как при ручной имитации, человек садится за терминал и вводит тестовые данные и изучает результаты точно так, как если бы терминал был подключен к настоящей программной системе. Для имитации системы еще один участник проверки, вооружившись спецификациями, садится за другой терминал. Небольшая программа связывает эти терминалы, передавая входные данные с терминала «пользователя» на терминал «имитатора», а выходные данные — обратно, по командам «имитатора». Последний действует так же, как при ручной имитации, т. е. изучает данные, поступающие с терминала «пользователя», ищет по спецификациям соответствующий результат, прослеживает преобразование системы и посылает результат на другой терминал «пользователя».

Эта связывающая программа совсем простая. Если терминалы — обычные устройства типа телетайпа, человеку-имитатору нужны команды обмена сообщениями с терминалом «пользователя» и способ распознавания всех действий, предпринимаемых за этим терминалом, например нажатия клавиши «внимание». Если терминал — устройство типа дисплея, необходимы команды управления форма-

том выходных сообщений и распознавания сигналов всех специальных клавиш, относящихся к функциональным.

Этот метод очень ценен, поскольку он помогает обнаружить в спецификациях все недочеты по части психологических факторов. В процессе внешнего проектирования он особенно полезен для изучения альтернативных вариантов решения проблемы психологических факторов.

Функциональные диаграммы

Построение функциональных диаграмм — это метод, позаимствованный из проектирования переключательных схем и позволяющий проектировщику представить спецификации булевской логической диаграммой. Главное его назначение — служить основой строгого подхода к построению оптимального набора тестов для программного обеспечения. Однако, поскольку метод предполагает детальный анализ спецификаций, он может оказаться полезным и для обнаружения ошибок в них. Построение функциональных диаграмм рассматривается в гл. 12.

ПЛАНИРОВАНИЕ ИЗМЕНЕНИЙ

Изменение требований — объективный фактор разработки программного обеспечения. Эти изменения возникают вследствие технического прогресса, социальных перемен (например, новое законодательство), изменений в людях, предлагаемых улучшений, обнаружения ошибок. Опытный разработчик программного обеспечения помнит об этом; он заранее запланировал изменения, организовал свой проект так, что изменение не становится травмирующим происшествием, он знает, как внести изменения таким образом, чтобы не понизить надежность системы.

Первое правило работы с изменениями — во время всего цикла работы над проектом поддерживать документацию на уровне последних решений. Распространенная ошибка — разработать неплохие цели и внешние спецификации, а затем не обновлять их при появлении изменений. Это плохая практика, поскольку цели и спецификации в таком случае больше не отражают реального состояния системы, что приводит к проблемам в обмене информацией и при использовании спецификаций в процессах тестирования, рассматриваемых в части 3.

Второе правило — в какой-то определенный момент «заморозить» результаты каждого процесса проектирования. «Замораживание» не означает, что больше нельзя вносить изменения; предполагается лишь, что с этого момента изменения должны проходить формальную процедуру утверждения. Требования и цели должны быть заморожены после их утверждения. Внешние спецификации — после успешного завершения проверки их правильности.

Процесс принятия изменений замороженного документа обычно называется *процессом изменений проекта*. Запросы на изменение проекта исходят от пользователя или разработчиков программного обеспечения. Каждый такой запрос оценивается с точки зрения его достоинств и влияния на продукт (на календарный план, стоимость, надежность, безопасность, эффективность). Обычно ответственность за принятие или отказ от всякого изменения (или отсрочку изменения до будущей реализации) несет небольшой комитет. Каждое утвержденное изменение должно быть немедленно отражено в соответствующих проектных документах. Оценивая предложение изменить принятое ранее решение, помните, что на последнем могли быть основаны и другие решения. Ошибки при попытках увидеть все последствия проектных изменений весьма распространены.

Третье правило работы с изменениями — проверять правильность всякого изменения в такой же степени, в какой проверялось исходное решение. Еще одна распространенная ошибка — потратить достаточно времени, тщательно проверяя правильность внешних спецификаций, но затем не выполнить таких же проверок над изменениями. Поскольку изменения проекта делаются в более напряженных условиях, они весьма опасны в отношении ошибок и должны быть особенно тщательно проверены.

Последнее правило — убедиться, что все нужные изменения сделаны на всех уровнях. Программист, проектирующий внутреннюю логику модуля, может сделать изменения, но не заметить при этом, что они влекут изменения внешних характеристик. Программное обеспечение и спецификации теперь рассогласованы, а это означает ошибку в программном обеспечении. Нет простого решения этой проблемы, но нужно позаботиться о том, чтобы все помнили о ней.

ЛИТЕРАТУРА

1. Williams R. D. Managing the Development of Reliable Software, Proceedings of the 1975 International Conference on Reliable Software. New York: *IEEE*, 1975, pp. 3—8.
2. Bingham J. E., Davies G. W. P. A Handbook of Systems Analysis. New York: Halsted, 1972.
3. Teichroew D. A Survey of Languages for Stating Requirements for Computer-Based Information Systems, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, N. J.: AFIPS Press, 1972. pp. 1203—1224.
4. Cougar J. D., Knapp R. W. System Analysis Techniques. New York: Wiley, 1974.
5. HIPO — A Design Aid and Documentation Technique. GC20—1851, IBM Corp., White Plains, N. Y., 1974.
6. Tinanoff N., Luppino F. M. Structured Programming Series, Volume VI, Programming Support Library (PSL) Specifications, RADC-TR-74-300, Volume VI, IBM Federal Systems Div., Gaithersburg, Md., 1974.
7. Fandel P., Milligan L. PL/I Structured Programming Case Study, Proceedings of the 39th Meeting of GUIDE International, New York: GUIDE International Corp., 1974, pp. 585—605.

8. Weinberg G. M., Schulman E. L. Goals and Performance in Computer Programming, *Human Factors*, **16**(1), 70—77 (1974).
9. Брукс Ф. П. Как проектируются и создаются программные комплексы. Пер. с англ.— М.: Наука, 1979.
10. Martin J. Design of Man-Computer Dialogues. Englewood Cliffs, N. J.: Prentice-Hall, 1973.
11. Orr W. D., Ed. Conversational Computers. New York: Wiley, 1968.
12. Pirtle M. W. HELP, in W. D. Orr, Ed. Conversational Computers. New York: Wiley, 1968, pp. 96—101.
13. Gilb T. Reliable EDP Application Design. New York: Petrocelli/Charter, 1974.
14. Hughes M. L., Shank R. M., Stein E. S. Decision Tables, New York: McGraw-Hill, 1968.

ГЛАВА 5

Архитектура системы

Разработка архитектуры — это процесс разбиения большой системы на более мелкие части. Для обозначения этих частей придумано множество названий: программы, компоненты, подсистемы и уровни абстракции.

Процесс разработки архитектуры — шаг, необходимый при проектировании систем, но не обязательный при создании программы. Если внешние спецификации описывают программную *систему*, то следующий шаг проектирования — разработка архитектуры, а за ним — проектирование структуры программы. Если же внешние спецификации описывают *программу*, разработка архитектуры системы не обязательна. Очевидно, немедленно возникает проблема, как различать системы и программы. К сожалению, точных определений этих понятий нет. Словарь здесь бесполезен, потому что все объекты, от операционной системы до подпрограммы и цикла DO, удовлетворяют определению системы.

Мы можем сказать, что программная система представляет собой набор решений множества различных, но связанных между собой задач, и далее положиться на интуицию в случаях, когда надо отличить систему от программы. Так, операционная система, система резервирования авиабилетов или система управления базами данных — все это примеры систем, и поэтому они должны пройти процесс разработки архитектуры. Программа редактирования текстов в системе разделения времени, компилятор, программа учета товаров являются примерами отдельных программ, и в них этап разработки архитектуры отсутствует.

Всякая методология проектирования программного обеспечения состоит из двух частей: набора желательных характеристик результата и руководящих принципов самого процесса проектирования. При разработке методологии проектирования обычно начинают с определения желательных характеристик, а затем конструируют мыслительный процесс, необходимый для получения нужного результата. Поэтому можно предположить, что о желательных свойствах результата известно больше, чем о самом процессе. Мы уже сталкивались с такой ситуацией при анализе внешнего проектирования; некоторые желательные свойства были перечислены в гл. 4,

но сам процесс внешнего проектирования — процедура, все еще во многом зависящая от конкретной ситуации. При проектировании логики модуля желательный результат — структурная программа, и один из мыслительных процессов называется *пошаговой детализацией*.

Если попытаться выделить упоминавшиеся две части методологии проектирования в разработке архитектуры, станет ясно, что это — пока наименее понятный процесс перевода. Здесь не только не определен соответствующий мыслительный процесс, но даже и желательные характеристики структуры системы только начинают осмысливаться. В следующих разделах дается обзор некоторых пока еще плохо определенных подходов к проектированию системы.

УРОВНИ АБСТРАКЦИИ

Концепция уровней абстракции была предложена Дейкстрой [1, 2]. Система разбивается на различные иерархически упорядоченные части, называемые уровнями, удовлетворяющие определенным проектировочным критериям. Каждый уровень является группой тесно связанных модулей. Идея уровней призвана минимизировать сложность системы за счет такого их определения, которое обеспечивает высокую степень независимости уровней друг от друга. Это достигается благодаря тому, что свойства определенных объектов такой системы (ресурсы, представление данных и т. п.) упрятываются внутрь каждого уровня, что позволяет рассматривать каждый уровень как «абстракцию» этих объектов.

Возможны две общие структуры таких уровней, изображенные на рис. 5.1 и 5.2 соответственно. Рис. 5.1 иллюстрирует такой подход, когда задача рассматривается как создание «машины пользователя», начиная с самого низкого уровня — уровня аппаратуры или операционной системы. Последовательность уровней, называемых *абстрактными машинами*, определяется так, что каждая следующая машина строится на предыдущих, расширяя их возможности. Каждый уровень может ссылаться только на один, отличный от него самого уровень (вызывать его), а именно тот, который ему непосредственно предшествует.

В структуре, изображенной на рис. 5.2, уровни не являются полными абстракциями более низких уровней, каждый из них может ссылаться на любой предшествующий уровень.

Хотя современное определение уровней абстракции само довольно абстрактно, некоторые свойства уровней начинают проясняться.

1. На каждом уровне абсолютно ничего не известно о свойствах (и даже о существовании) более высоких уровней [3]. Это — фундаментальное свойство уровней абстракции, существенно сокращающее число связей между частями программной системы и их пред-

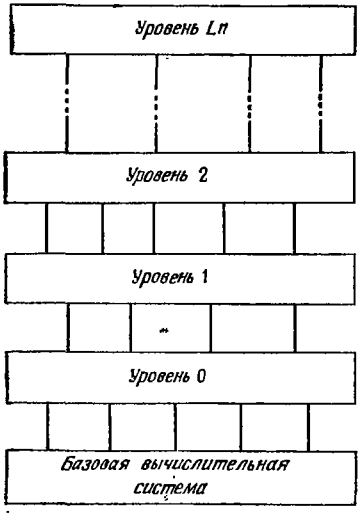


Рис. 5.1. Уровни абстракции.

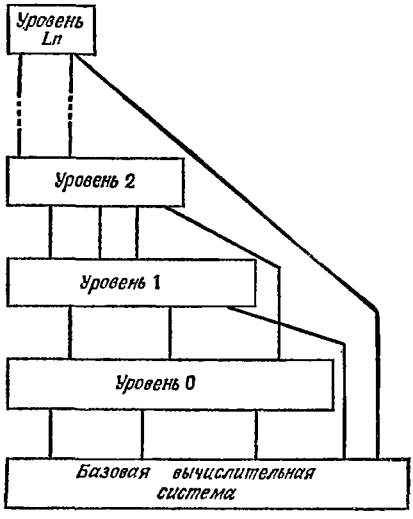


Рис. 5.2. Другой подход к идее уровней абстракции.

положений относительно свойств друг друга, благодаря чему уменьшается сложность системы.

2. На каждом уровне ничего не известно о внутреннем строении других уровней. Связь между уровнями осуществляется только через жесткие, заранее определенные сопряжения.

3. Каждый уровень представляет собой группу модулей (раздельно компилируемых подпрограмм). Некоторые из этих модулей являются внутренними для уровня, т. е. недоступны для других уровней. Имена остальных модулей, вообще говоря, известны на следующем, более высоком уровне. Эти имена представляют собой сопряжение с этим уровнем.

4. Каждый уровень располагает определенными ресурсами и либо скрывает их от других уровней, либо предоставляет другим уровням некоторые их абстракции. Например, в системе управления файлами один из уровней может содержать физические файлы, скрывая их организацию от остальной части системы. Другие уровни могут владеть такими ресурсами, как каталоги, словари данных, механизмы защиты.

5. Каждый уровень может также обеспечивать некоторую абстракцию данных в системе. В системе управления данными один из уровней может представлять файл как древовидную структуру, изолируя тем самым действительную физическую структуру файла от всех других уровней.

6. Предположения, которые на каждом уровне делаются относительно других уровней, должны быть минимальны [4]. Эти предполо-

жения могут принимать вид соотношений, которые должны соблюдаться перед выполнением функции, либо относятся к представлению данных или факторов внешней среды. Например, в системе страхования автомобилей это может означать, что все сведения о государственных законах, касающихся страхования, упрятаны внутри одного уровня, а данные о характеристиках устройств ввода (например, терминалы типа дисплеев или телетайпов, оптическое читающее устройство, телекоммуникационное устройство) — внутри другого.

7. Связи между уровнями ограничены явными аргументами, передаваемыми с одного уровня на другой. Недопустимо совместное использование глобальных данных несколькими уровнями [5]. Более того, желательно полностью исключить использование глобальных данных (даже внутри уровня) в системе [6].

8. Каждый уровень должен иметь высокую прочность и слабое сцепление в смысле, определенном в гл. 6. Это значит, что всякая функция, выполняемая уровнем абстракции, должна быть представима единственным входом. Аргументы, пересылаемые между уровнями, должны быть отдельными элементами данных, а не сложными структурами.

Подход к проектированию системы на основе уровней абстракции до сих пор использовался не очень широко. Наиболее известны два случая его применения при разработке небольших операционных систем: THE [1, 2] и Венера [7]. Система THE — небольшая операционная система, разработанная коллективом из шести человек в Технологическом университете Эйндховена (Голландия). Она является пакетной системой с мультипрограммированием и виртуальной памятью (со страничной организацией). Система включает пять уровней абстракции, структура которых близка к изображенной на рис. 5.2.

Уровень 0 содержит функции диспетчеризации ЦП и синхронизации процессов (задач). Это единственный уровень, осведомленный о мультипрограммных аспектах системы, в том числе и о количестве центральных процессоров (ЦП).

Уровень 1 отвечает за управление памятью, в том числе за управление страничным барабаном. Ни один другой уровень не знает о страничном механизме. Более того, ни один уровень даже не знает, что это система с виртуальной памятью.

Уровень 2 занят обеспечением связи между более высокими уровнями и консолью оператора. Кроме него, никто не знает характеристик и числа консолей оператора.

Уровень 3 отвечает за буферизацию входного и выходного потоков. Остальные уровни не знают об этом процессе ничего, в том числе и характеристик устройств ввода-вывода и того, какие конкретно устройства используются.

Программы пользователей (прикладные программы) находятся на уровне 4.

Система проектировалась и тестировалась методом «снизу вверх» (восходящим методом), начиная с уровня 0. Уровни абстракции минимизировали число внутренних состояний системы, благодаря чему оказалось возможным неформально проверить правильность каждого уровня после его проектирования и провести «исчерпывающее тестирование» окончательного результата. Это позволило основному разработчику системы (Э. Дейкстре) сделать редкостное заявление: «В результате гарантируется, что система не содержит ошибок» [1].

Всякий, кто испытал трагедию внесения существенных изменений в операционную систему (таких, как преобразование системы в мультипрограммную и/или в систему с виртуальной памятью), увидит в такой организации много достоинств. Дело не только в том, что модификации ограничиваются изолированными частями системы; уровни абстракции четко указывают, какие части подлежат изменению.

Хотя в том, что касается надежности, достигнуты выдающиеся результаты, не следует забывать, что система THE является во многих отношениях очень примитивной операционной системой. Например, она не имеет вторичной памяти, кроме страничного барабана, поэтому проблема управления файлами не возникает.

Вторым проектом, использующим идею уровней абстракции, была операционная система Венера [7]. Это был более крупный проект, чем THE, поскольку он включал также разработку аппаратуры. Так, нижние уровни абстракции (диспетчеризация ЦП и управление виртуальной памятью) реализованы микропрограммно.

Еще одним достоинством идеи уровней абстракции является *мобильность*: легкость модификации прикладной системы для работы на другой вычислительной системе. Поскольку все сопряжения с базовой вычислительной системой находятся на уровне 0, а все сопряжения со средствами управления данными изолированы на другом уровне, прикладная программа может быть приспособлена для работы в другой операционной системе или с другой системой управления данными заменой только одного уровня.

Хотя подход на основе уровней абстракции остается пока довольно туманным, эти идеи кажутся полезными при определении общей структуры больших систем (как операционных систем, так и прикладных). Хотя до настоящего времени этот подход использовался при восходящей разработке (нижние уровни проектируются раньше верхних), эта идея может быть использована и при нисходящей разработке. Для тех немногих проектировщиков, которые занимаются полной вычислительной системой — и программным обеспечением, и аппаратурой,— нисходящая разработка оказывается существенно более выгодной; она позволяет отложить решение о том,

следует ли реализовать данную функцию аппаратными или программными средствами, до заключительных этапов цикла проектирования, когда можно будет исследовать соотношение затрат и производительности.

ПОДСИСТЕМЫ, УПРАВЛЯЕМЫЕ МЕТОДОМ ПОРТОВ

Второй метод проектирования систем основан на идее подсистемы, *управляемой портом*, или *очередью* [8—10]. В отличие от метода уровней абстракции, где внимание уделяется в первую очередь правильному распределению функций между уровнями иерархии в системе, здесь основное внимание обращено на методы связи между подсистемами, компонентами или программами.

Механизмы, обеспечивающие взаимосвязь между частями системы, можно разбить на три группы. Механизмы первой группы, такие, как предложение GO TO, прерывание, «обращение к супервизору», передают управление (поток выполнения) от одной части другой, не передавая явно никаких данных. Механизмы типа вызова процедуры или подпрограммы относятся ко второй группе; они передают от одной части системы другой и управление, и данные. К третьей группе относятся механизмы портов — это методы передачи данных без передачи управления.

При использовании механизма портов система делится на несколько подсистем, каждая из которых имеет один или несколько входных портов (одну или несколько входных очередей). Порт — это просто текущий список входных сообщений (списков параметров) для подсистемы. Каждая подсистема рассматривается как асинхронный процесс, т. е. все подсистемы работают параллельно. Если одна из них хочет передать некоторые данные другой, она посылает их во входной порт этой другой подсистемы. Если подсистема готова обрабатывать какие-то данные, она читает их из одного из своих входных портов. Используемые для такой связи две операции называются ПОСЛАТЬ и ПОЛУЧИТЬ.

Механизм портов увеличивает независимость подсистем, в значительной степени освобождая их от временнóй зависимости друг от друга. Более того, подсистемы не должны даже знать о взаимном *расположении*. Только механизм посылки/получения (почта) должен знать расположение каждой подсистемы (например, в той же виртуальной памяти или в другой виртуальной памяти, в другой вычислительной системе). Механизм портов имеет также некоторые преимущества при тестировании, отладке и проверке, которые будут рассматриваться позже.

На рис. 5.3 изображена подсистема (будем называть ее *процессом*, чтобы отличать от не использующих механизм портов подсистем). Такой процесс имеет один или несколько *первичных входных*

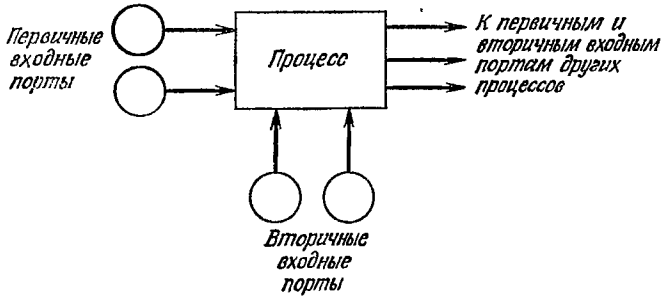


Рис. 5.3. Прямая конфигурация.

портов, по одному для каждой функции процесса. Процесс может выполнять операторы или команды типа

ПОЛУЧИТЬ ПОРТ1 (А, В, С)

по которой из входного порта ПОРТ1 выбирается очередное сообщение и выделенные из него аргументы присваиваются параметрам А, В и С. Процесс может послать данные другим процессам в результате выполнения оператора

ПОСЛАТЬ УПРДАННЫМИ (Х, А)

по которому из аргументов Х и А формируется сообщение, пересылаемое в некоторый порт по имени УПРДАННЫМИ.

У процесса может быть и другой набор входных портов, называемых *вторичными входными портами*. Эти порты используются для связи с подчиненными процессами. Если процессу для выполнения его функции необходимо осуществить чтение из базы данных, он может послать сообщение управлению базой данных и затем ожидать получения результата через вторичный порт. Входное сообщение управлению базой данных обычно должно содержать аргумент, указывающий имя этого вторичного порта.

Процесс на рис. 5.3 — часть *прямой конфигурации*. Каждый такой процесс осуществляет связь посылкой сообщения прямо во входной порт другого процесса. На рис. 5.4 изображен процесс, являющийся частью *непрямой конфигурации*. Процессы этого типа имеют выходные порты. Вместо того чтобы посылать сообщение во входные порты других процессов, они посылают их в свои выходные порты. Выходной порт связывается со входными портами других процессов посредством специальных указаний, например на языке управления заданиями. Типичным примером такого оператора может быть:

СВЯЗАТЬ ВЫХПОРТ2 ПРОЦЕССА АВС С ВХПОРТ1 ПРОЦЕССА XYZ

Всякий раз, когда процесс АВС посылает сообщение в свой выходной порт ВЫХПОРТ2, механизм посылки/получения направит его во ВХПОРТ1 процесса XYZ. Таким образом, выходные порты физи-

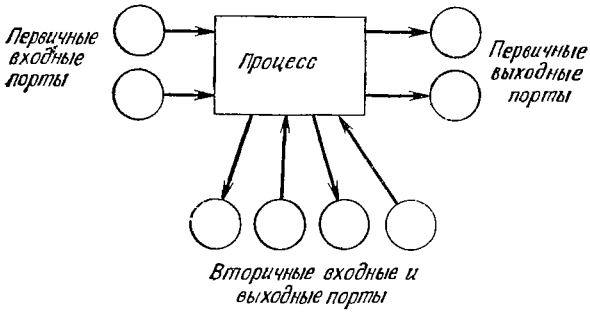


Рис. 5.4. Непрямая конфигурация.

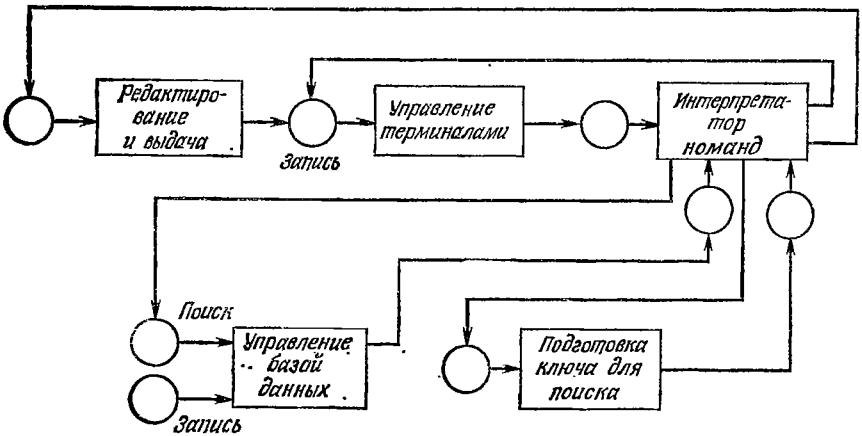


Рис. 5.5. Структура системы, использующей порты.

чески не существуют, это абстрактные наименования для входных портов других процессов.

На рис. 5.5 изображена схема информационно-поисковой системы, использующей прямую конфигурацию. Управление терминалами — это непрерывно выполняемый процесс (точно так же, как и четыре других процесса). Он опрашивает терминалы системы, и всякий раз, когда с терминала получена команда, он посылает подсистеме интерпретации команд сообщение, состоящее из двух аргументов: команды с терминала и идентификатора этого терминала. Управление терминалами выполняет также и другую функцию: выдачу на терминалы выходных строк; для этой цели у терминалов имеются входные порты.

Подсистема интерпретации команд постоянно обращается за данными к своему единственному первичному порту. Получив команду с терминала, она использует две подчиненные подсистемы, чтобы подготовить ключ для поиска и осуществить поиск в базе дан-

ных. Когда поиск закончен, она посылает его результат вместе с идентификатором терминала подсистеме редактирования и выдачи. Если подсистема интерпретации команд обнаруживает ошибку (например, неправильную команду), она посылает сообщение об ошибке (и идентификатор терминала) во входной порт управления терминалами и затем получает новое сообщение из своего входного порта. Действия остальных процессов очевидны. Отметим, что управление базой данных имеет еще один первичный входной порт для записи информации в базу данных, но в этой системе он не используется.

Механизм посылки/получения

Механизм посылки/получения — это отдельная подсистема, вызываемая всякий раз, когда система выполняет операции ПОСЛАТЬ или ПОЛУЧИТЬ. При создании механизма посылки/получения должны быть обеспечены следующие его функции.

1. Функция ПОСЛАТЬ, описанная выше.
2. Функция ПОЛУЧИТЬ, которая приостанавливает процесс, давший эту команду, если нет никаких сообщений. Выполнение приостановленного процесса возобновляется, когда в один из его входных портов поступает сообщение.
3. Функция ПОЛУЧИТЬ-ЕСЛИ-ЕСТЬ, позволяющая процессу продолжать выполнение, если сообщений нет.
4. Функции СОЗДАТЬ-ПОРТ и УНИЧТОЖИТЬ-ПОРТ, позволяющие подсистемам создавать и уничтожать порты динамически.
5. Функции СВЯЗАТЬ и РАЗЪЕДИНИТЬ, чтобы сообщить механизму посылки/получения, между какими подсистемами установлена связь.
6. Расположение любой подсистемы должно быть «скрыто» от всех других подсистем с помощью средств связи между видами памяти и между центральными процессорами (например, совместно используемые файлы, адаптеры канал-канал, телекоммуникационные линии).
7. Возможность представлять порты как очереди (первым пришел — первым ушел), стеки (последним пришел — первым ушел) или некоторые другие структуры данных (например, список, упорядоченный по приоритетам).
8. Контроль сопряжений (проверки соответствия между аргументами и параметрами) или преобразование данных между аргументами «посылаемых» и параметрами «получаемых» сообщений.
9. Сохранение текущего состояния всех портов на диске, позволяющее продолжать с контрольной точки в случае возникновения ошибок.
10. Регистрация всех сообщений (для диагностики ошибок).

11. Возможность по требованию контролировать все сообщения определенного характера (для диагностики ошибок).

Каждая подсистема должна быть отдельной задачей, что позволит выполнять их параллельно в мультипрограммной или мультипроцессорной системе, хотя это не является принципиальным для рассматриваемой идеи.

Простейший способ реализации механизма посылки/получения— воспользоваться, если это возможно, соответствующим механизмом базовой вычислительной системы. Если необходимо построить свой собственный механизм, то в работе Кнотта [11] можно найти примеры сопряжений, а в книге Кнута [12] — соответствующие алгоритмы.

Дополнительные соображения

Механизм посылки/получения сам удовлетворяет определению уровня абстракции, поскольку он обеспечивает абстракцию (понятие порта), обладает ресурсами (физические порты) и скрывает от остальной части системы информацию (физические характеристики портов и физическое расположение всех подсистем). Поэтому, естественно, эти две концепции можно использовать вместе, определяя каждую подсистему, управляемую методом портов, как уровень абстракции. На рис. 5.6 изображена общая структура системы реального времени, в которой применены обе концепции [13].

Понятие порта может использоваться также для представления запоминающих устройств подсистемы. Каждая подсистема может иметь один или несколько «личных портов», то есть портов, в которые она посылает и из которых она же затем получает сообщения.

Понятие порта имеет много достоинств при тестировании и отладке, поскольку благодаря ей в центре всех взаимодействий подсистем оказывается основной механизм связи. В дополнение к упомянутым выше функциям контроля и трассировки механизм посылки/получения мог бы посылать на терминал копию всех сообщений, что позволило бы при отладке наблюдать за работой системы. Благодаря тому что порты можно связывать с устройствами ввода-вывода, такими, как терминалы, имеется возможность имитировать подсистемы. Пусть уже реализована изображенная на рис. 5.5 система, за исключением управления базой данных. Систему можно тестировать и без управления базой данных, если она будет печатать на терминале команду ПОЛУЧИТЬ всякий раз, когда нужно получить сообщение, помещенное во входной порт управления базой данных. Тогда можно имитировать управление базой данных, подбирая вручную для каждого запроса подходящий результат и передавая его с терминала в соответствующий вторичный входной порт подсистемы интерпретации команд с помощью команды ПОСЛАТЬ.

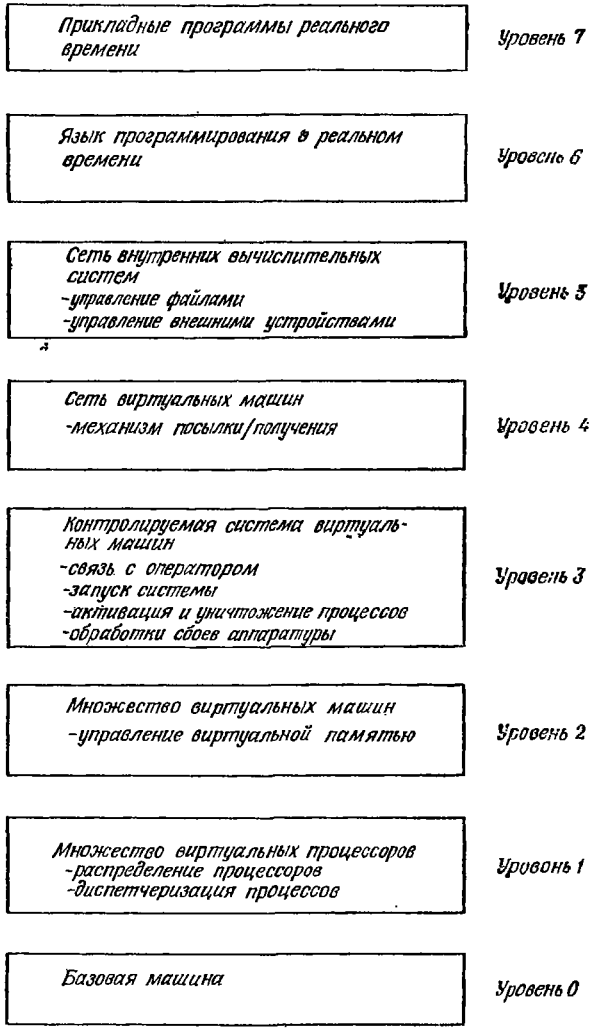


Рис. 5.6. Система реального времени, использующая уровни абстракции и порты.

К сожалению, нынешнее состояние обоих подходов — метода уровней абстракции и метода портов — таково, что оба они описывают желаемые свойства системы, но не сам процесс ее проектирования. Несомненно, в будущем это положение улучшится. Тем не менее оба подхода уже сегодня полезны тем, что дают разработчику некоторые руководящие принципы проектирования программных систем.

ПОСЛЕДОВАТЕЛЬНЫЕ ПРОГРАММЫ

Значительный процент разрабатываемых сегодня систем по-прежнему составляют традиционные пакетные системы, которые могут быть разбиты на ряд последовательных программ, общающихся между собой через файлы во вторичной памяти. Например, система для начисления страховых премий может состоять из программы редактирования и сбора входных данных за неделю, за которой следует программа сортировки, затем программа обновления главного файла и наконец программа генерации отчетов.

Хотя проектирование такого рода систем выполняется уже в течение по крайней мере последних 25 лет, не выработано никакой методологии, кроме не особенно полезного совета «нарисовать блок-схему системы и затем разбить ее на программы». Можно, однако, применить некоторые идеи подхода, основанного на понятии уровня абстракции, и композиционного проектирования [6]. Каждая программа не должна ничего знать о программах, выполнявшихся раньше ее. Должны быть сведены к минимуму предположения о свойствах других программ. Каждая программа должна скрывать от других программ некоторые системные ресурсы. Каждая программа должна обладать функциональной прочностью и минимальной степенью сцепления с другими программами (гл. 6).

ДОКУМЕНТАЦИЯ

Результатом процесса разработки архитектуры системы является документация, описывающая разбиение программной системы на основные части (компоненты, подсистемы, программы, процессы или уровни абстракции). Документация должна описывать функцию каждой компоненты, точные сопряжения между компонентами и структуру системы. Необходимо отразить все перечисленные ниже аспекты структуры системы.

1. Иерархия передач управления с возвратом (т. е. «структура вызовов» компонент).
2. Структура потоков данных в системе.
3. Иерархия задач (параллельных процессов), т. е. структура отношений «мать/дочь/сестра» между задачами.
4. Структура памяти в системе. Это касается только случаев, когда компоненты находятся не в одной адресуемой памяти (например, не в одной виртуальной памяти или не в одном разделе памяти).
5. Отображение структуры программного обеспечения на конфигурацию аппаратуры (например, какие компоненты владеют данными устройствами ввода-вывода или общаются с ними).

Некоторые конкретные идеи относительно описания функций, сопряжения и структуры можно найти в гл. 6 и 8.

ПРОВЕРКА ПРАВИЛЬНОСТИ

Поскольку представления о процессе разработки архитектуры пока еще довольно расплывчаты, это же во многом относится и к проверке его правильности. Можно указать лишь два достижения в этом направлении — метод «*n* плюс-минус один» и метод сквозного контроля.

Цель метода «*n* плюс-минус один» здесь — обнаружить все ошибки перевода, сделанные при разработке архитектуры системы, и минимизировать возможные ошибки в будущем, когда архитектура будет служить исходными данными для последующих процессов проектирования. Из рис. 3.2 следует, что критический анализ архитектуры должны выполнять разработчики исходных внешних спецификаций и предполагаемые разработчики структуры каждой компоненты. Проектировщик уровня $n-1$ ищет примеры неполного или неточного перевода внешнего проекта. Проектировщик уровня $n+1$ проверяет, понятна ли архитектура системы и работоспособна ли она. Поскольку архитектура часто разрабатывается параллельно с детальным внешним проектом, важно также сопоставить результаты обоих процессов, чтобы выявить ошибки.

Процесс сквозного контроля, аналогичный тому, что описан в гл. 4 для внешнего проекта, также может оказаться весьма эффективным. Сначала нужно разработать тесты, затем мысленно проследить за их выполнением, стремясь найти не полностью определенные или несогласованные сопряжения, отсутствующие или недоопределенные функции. Важно включить по крайней мере один тест начала работы системы, еще по крайней мере один тест для завершения, несколько тестов для характерных типов входных сообщений и несколько тестов неправильных входных данных и необычных состояний системы.

ЛИТЕРАТУРА

1. Dijkstra E. W. The Structure of the THE-Multiprogramming System, *Communications of the ACM*, **11**(5), 341—346 (1968).
2. Dijkstra E. W. Complexity Controlled by Hierarchical Ordering of Function and Variability, in P. Naur and B. Randell, Eds. Software Engineering: Report on a Conference Sponsored by the NATO Science Committee. Brussels, Belgium: NATO Scientific Affairs Division, 1968, pp. 181—185.
3. Goos G. Hierarchies, in M. Beckman et al., Eds., Advanced Course in Software Engineering. Berlin: Springer-Verlag, 1970, pp. 29—46.
4. Parnas D. L. Information Distribution Aspects of Design Methodology, Proceedings of the 1971 IFIP Congress, Booklet TA-3. Amsterdam: North-Holland, 1971, pp. 26—30.
5. Liskov B. H. A Design Methodology for Reliable Software Systems, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 191—199.
6. Myers G. J. Reliable Software Through Composite Design, New York: Petrocelli/Charter, 1975.

7. Liskov B. H. The Design of the Venus Operating System, *Communications of the ACM*, **15** (3), 144—149 (1972).
8. Morenoff E., McLean J. B. Inter-Program Communications, Program String Structures and Buffer Files, Proceedings of the 1967 Spring Joint Computer Conference, Montvale, N. J.: AFIPS Press, 1967, pp. 175—183.
9. Balzer R. M. Ports — A Method for Dynamic Interprogram Communication and Job Control, R-605-ARPA, Rand Corp., Santa Monica, Ca., 1971.
10. Balzer R. M. An Overview of the ISPL Computer System Design, *Communications of the ACM*, **16** (2), 117—122 (1973).
11. Knott G. D. A Proposal for Certain Process Management and Intercommunication Primitives, Part I, *Operating Systems Review*, **8** (4), 7—44 (1974).
12. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. Пер. с англ.— М.: Мир, 1976.
13. Sorenson P. G., Hamacher V. C. A Real-Time System Design Methodology, *INFOR*, **13** (1), 1—18 (1975).

ГЛАВА 6

Проектирование структуры программы

Следующий процесс проектирования программного обеспечения — проектирование структуры программы. Он включает определение всех модулей программы, их иерархии и сопряжений между ними. Если разрабатывается отдельная программа, исходными данными для этого процесса будут детальные внешние спецификации, если же система — детальные внешние спецификации и архитектура системы. В этом последнем случае рассматриваемый процесс состоит в проектировании структуры всех компонент или подсистем полной системы.

Традиционный метод борьбы со сложностью — принцип «разделяй и властвуй», часто называемый «модуляризацией». На практике, однако, этот подход часто не приводит к ожидаемому уменьшению сложности. Лисков [1] указала три причины подобной неудачи:

1. Модули выполняют слишком много связанных, но различных функций — это делает их логику запутанной.
2. При проектировании остались невыявленными общие функции, вследствие чего они рассредоточены (и по-разному реализованы) в разных модулях.
3. Модули взаимодействуют посредством совместно используемых или общих данных самым неожиданным образом.

Методология проектирования, называемая *композиционным проектированием* [2],— это принцип проектирования, рассматриваемый здесь на примере проектирования структуры программы. Композиционное проектирование состоит, по существу, из двух компонент: системы явных проектных оценок, позволяющих решить все три перечисленные выше проблемы и еще целый ряд дополнительных проблем, и ряда мыслительных процессов, обеспечивающих разбиение программы на множество модулей, их сопряжений и отношений. В результате композиционного проектирования достигается минимальная сложность структуры программы. Такую программу легче понимать, сопровождать и адаптировать.

НЕЗАВИСИМОСТЬ МОДУЛЕЙ

Чтобы уменьшить сложность программы, нужно разбить ее на множество небольших, в высокой степени независимых модулей. Модуль — это замкнутая программа, которую можно вызвать из любого другого модуля в программе и можно отдельно компилировать (отметим, что тем самым исключаются внутренние процедуры PL/1 и параграфы Кобола). Довольно высокой степени независимости можно достичь с помощью двух методов оптимизации: усилением внутренних связей в каждом модуле и ослаблением взаимосвязи между модулями. Если рассматривать программу как набор предложений, связанных между собой некоторыми отношениями (как по выполняемым функциям, так и по обрабатываемым данным), то основное, что требуется,— это догадаться, как распределить эти предложения по отдельным «ящикам» (модулям) так, чтобы предложения внутри каждого модуля были тесно связаны, а связь между любой парой предложений в разных модулях была минимальной. Нужно стремиться, во-первых, реализировать отдельные функции отдельными модулями (высокая прочность модуля) и ослаблять связь между модулями по данным, применяя формальный механизм передачи параметров (слабое сцепление модулей).

ПРОЧНОСТЬ МОДУЛЕЙ

Прочность модуля — это мера его внутренних связей. Чтобы определить прочность модуля, необходимо проанализировать выполняемую им функцию (или функции), с тем чтобы решить, к какому из семи классов он относится. Классы эти специально определены для того, чтобы ввести количественную характеристику «доброкачественности» конкретных типов модулей.

Прежде чем двигаться дальше, необходимо определить, что понимается под *функцией* модуля. Модуль имеет три основных атрибута: он выполняет одну или несколько *функций*, обладает некоторой *логикой* и используется в одном или нескольких *контекстах*. Функция — это внешнее описание модуля; описывается, *что* делает модуль, когда он вызван, но не *как* это делается. Логика описывает внутренний алгоритм модуля, другими словами, как он выполняет свою функцию. Контекст описывает конкретное применение модуля. Например, модуль с функцией «удалить пробелы из литерной строки» может использоваться в контексте «сжать сообщение для телеобработки». Чтобы увидеть разницу между функцией и логикой, рассмотрим модуль, функция которого — компилировать программу на PL/1. Он может быть головным модулем 83-модульного компилятора либо единственным модулем компилятора. В обоих случаях функция этих двух модулей одинакова, но логика — совершенно разная. Мы видим, что функция модуля может рассмат-

риваться как композиция его логики и функций всех подчиненных (вызываемых им) модулей. Это определение рекурсивно и применимо к любому модулю в иерархии.

Цель проектирования — так определить модули, чтобы каждый из них выполнял одну функцию (говорят, что такие модули обладают *функциональной прочностью*). Чтобы понять важность этой цели, ниже рассмотрим семь классов прочности модулей, начиная с самого слабого типа прочности.

Модуль, *прочный по совпадению*,— модуль, между элементами которого нет осмысленных связей. Трудно привести пример такого модуля, поскольку он не выполняет никаких разумных функций. Описание логики — единственный возможный способ описания модулей этого типа. Одна из причин, по которым такие модули могут возникнуть, — это «модуляризация» программы post factum, когда мы обнаруживаем одинаковые последовательности команд в нескольких модулях и решаем сгруппировать их в отдельный модуль. Если эти последовательности (хотя они и кажутся идентичными) имеют разный смысл в тех модулях, в которые они первоначально входили, то наш новый модуль является прочным по совпадению. Модуль этого типа тесно связан с вызывающими его модулями, поэтому почти любая его модификация в интересах одного из этих модулей приводит к тому, что для всех остальных он станет работать неправильно.

Модуль, *прочный по логике*, при каждом вызове выполняет выбранную функцию из набора связанных с ним. Выбираемая функция обычно запрашивается вызывающим модулем, например с помощью кода функции. Примером может быть модуль, функция которого — читать из файла или писать в файл. Главная проблема с модулями этого типа — это использование одного и того же сопряжения для выполнения многих функций. Это приводит к сложным сопряжениям и неожиданным ошибкам при изменении сопряжения ради одной из функций.

Модуль, *прочный по классу*, последовательно выполняет набор связанных с ним функций. Самые распространенные примеры — «начальный» и «заключительный» модули. Главная проблема с модулями этого типа состоит в том, что обычно они неявно связаны с другими модулями программы, что делает программу трудной для понимания и ведет к ошибкам, когда ее приходится изменять.

Процедурно прочный модуль последовательно выполняет набор тех связанных с ним функций, которые непосредственно относятся к процедуре решения задачи. Вот пример задачи: написать программу регулирования температуры простого парового котла. В определенной степени подобная задача может определять действия программы. Например, в постановке задачи может быть сказано, что при получении сигнала x следует закрыть клапан y и прочитать и зарегистрировать значение температуры. Модуль с функцией «за-

крыть клапан у, прочитать значение температуры парового котла и занести его в журнал» обладает процедурной прочностью. В этом случае единственная проблема, связанная с надежностью, состоит в том, что фрагменты программы, относящиеся к различным функциям, могут быть переплетены. Отметим, что для модулей этого типа, так же как и для большинства других типов, имеются и другие, не связанные с надежностью, проблемы, как показано Майерсом в [2].

Коммуникационно прочный модуль — это процедурно прочный модуль с одним дополнительным ограничением: *все* его функции связаны по данным. Например, модуль «прочитать следующую запись и обновить главный файл» коммуникационно прочен, поскольку обе его функции связаны между собой тем, что обе они работают с одной и той же записью. И здесь обычно возможно переплетение функций, но риск внесения ошибки при модификации несколько меньше, поскольку функции связаны более тесно.

Следующая на нашей шкале — информационная прочность. Однако я ненадолго отложу ее обсуждение.

Функционально прочный модуль — это модуль, выполняющий одну определенную функцию, такую, как «закрыть клапан у», «выполнить команду РЕДАКТИРОВАТЬ» или «подвести итог по сделкам за неделю». Функциональная прочность — это высшая (лучшая) форма прочности модуля.

Отметим, что функционально прочный модуль может быть описан набором более детальных функций. Например, модуль «подвести итог по сделкам» можно описать так: «подготовить начальное состояние итоговой таблицы, открыть файл сделок, читать сделки и обновлять итоговую таблицу». Глядя на это, читатель может подумать, что простой перефразировкой описания модуля понижена его прочность. Однако если эти «функции более низкого уровня» могут быть рационально описаны как одна хорошо определенная функция «более высокого уровня», то следует считать, что модуль обладает функциональной прочностью.

Оставшийся тип прочности — *информационная прочность. Информационно прочный* модуль выполняет несколько функций, причем все они работают с одной и той же структурой данных и каждая представляется собственным входом. Модуль с двумя входами, один из которых соответствует функции «включить элемент в таблицу символов», а другой — функции «искать в таблице символов», обладает информационной прочностью. Модуль этого типа может также рассматриваться как физическое объединение нескольких функционально прочных модулей с целью «упрятывания информации» [3], например для того, чтобы укрыть внутри одного модуля все сведения о конкретной структуре данных, ресурсах или устройстве. В упомянутом выше примере вся информация о структуре и расположении таблицы символов скрыта внутри одного модуля. Это имеет

то преимущество, что всякий раз, когда удается скрыть некоторый аспект программы внутри одного модуля, независимость ее модулей увеличивается. Упоминавшуюся ранее цель проектирования нужно теперь подправить, чтобы наряду с функционально прочными модулями стремиться к информационно прочным.

Хотя выше мы сконцентрировали внимание только на связи между прочностью модуля и защищенностью от ошибок, прочность модуля влияет также на адаптируемость программы, трудность тестирования отдельных модулей и степень применимости модуля в других контекстах и других программах [2]. Шкала прочности упорядочена с учетом всех этих атрибутов.

Отметим, что модуль может соответствовать описанию нескольких типов прочности. Например, коммуникационно прочный модуль удовлетворяет также определению процедурной прочности и прочности по классу. Будем всегда относить модуль к высшему типу прочности, определению которого он удовлетворяет.

СЦЕПЛЕНИЕ МОДУЛЕЙ

Второй важнейший способ увеличить независимость модулей — ослабить связи между ними. Сцепление модулей, т. е. мера взаимозависимости модулей по данным, характеризуется как способом передачи данных, так и свойствами самих этих данных. Проанализировав любую пару модулей, можно определить, к какому из шести видов относится сцепление между ними, либо установить, что между ними прямого сцепления нет.

Цель проектирования состоит в определении таких сопряжений между модулями, чтобы все данные передавались между ними в форме явных и простых параметров. Как и раньше, чтобы понять важность этой цели, мы рассмотрим ниже шесть видов сцепления, начиная с самого жесткого (наихудший случай).

Два модуля *сцеплены по содержимому*, если один *прямо* ссылается на содержимое другого. Например, если модуль *A* каким-либо образом ссылается на данные модуля *B*, используя абсолютное смещение, то эти модули сцеплены по содержимому. Почти всякое изменение *B* или, возможно, просто перекомпиляция *B* с помощью другой версии компилятора внесет ошибку в программу. К счастью, большинство языков програмирования высокого уровня делают сцепление по содержимому трудноосуществимым.

Группа модулей *сцеплена по общей области*, если они ссылаются на одну и ту же глобальную структуру данных. Модули PL/1, ссылающиеся на структуру, объявленную как EXTERNAL, сцеплены друг с другом по общей области. Модули Фортрана, ссылающиеся на данные в блоке COMMON, и группы модулей, ссылающиеся на абсолютные адреса памяти (включая регистры), также служат примерами сцепления по общей области.

Со сцеплением по общей области связан целый ряд проблем. Все такие модули зависят от физического упорядочения элементов общей структуры данных, вследствие чего изменение размеров одного элемента данных влияет на все модули. Использование глобальных данных сводит на нет все попытки управлять доступом каждого модуля к данным. Например, в OS/360 фирмы IBM есть большая глобальная структура данных, называемая таблицей вектора связи. Невозможность управлять доступом к этой таблице (и другим глобальным таблицам) привела к многочисленным проблемам в связи с надежностью и адаптируемостью. Имена глобальных переменных связывают модули еще тогда, когда те только создаются. Это значит, что использование сцепленных по общей области модулей в новых программах затруднено, если не невозможно вообще.

Глобальные данные усложняют и восприятие программы. Рассмотрим следующий фрагмент:

```
DO WHILE (A);
    CALL L (X,Y,Z);
    CALL M (X,Y);
    CALL N (W,Z);
    CALL P (Z,X,Y);
END;
```

Если A не является глобальной переменной и если другие неудачные приемы кодирования (такие, как совмещение A с W, X, Y или *Z*) не используются, то можно утверждать, что этот цикл никогда не закончится. Если A — глобальная переменная, то сразу нельзя определить, закончится ли выполнение цикла. Придется исследовать внутреннее устройство модулей L, M, N и P, а также всех модулей, которые им подчинены, чтобы понять только этот цикл DO!

Борьба против глобальных данных приобретает такую же важность, как борьба против оператора GO TO, и этому начинают уделять внимание в профессиональной литературе [4, 5, 6].

Группа модулей *сцеплена по внешним данным*, если они ссылаются на один и тот же глобальный элемент данных (переменную, имеющую единственное поле). Например, модули PL/1, ссылающиеся на одну переменную (не структурную), объявленную как EXTERNAL, сцеплены друг с другом по внешним данным. Сцепление по внешним данным порождает почти все проблемы, свойственные сцеплению по общей области. Однако проблемы зависимости от физического упорядочения элементов в структуре не возникает.

Два модуля *сцеплены по управлению*, если один явно управляет функционированием другого, например используя код конкретной функции. Сцепление по управлению и прочность по логике обычно сопутствуют друг другу, поэтому основная проблема здесь та же, что и с прочностью по логике: использование одного и того же (сложного) сопряжения для выполнения многих функций. Сцепление по

управлению часто предполагает также, что вызывающий модуль имеет некоторое представление о логике вызываемого модуля, что уменьшает их независимость.

Группа модулей *сцеплена по формату*, если они ссылаются на одну и ту же неглобальную структуру данных. Если модуль А вызывает модуль В и передает ему запись анкетных данных служащего, и при этом как А, так и В чувствительны к изменению структуры или формата этой записи, то А и В сцеплены по формату.

Сцепления по формату следует, где это возможно, избегать, поскольку оно создает ненужные связи между модулями. Предположим, что модулю В нужны только некоторые поля в анкете. Передача ему всей анкеты вынуждает его заниматься ею целиком, и таким образом увеличивается вероятность того, что он неумышленно ее изменит. (По-видимому, можно утверждать, что, чем больше посторонних данных доступно модулю, тем больше возможность ошибки.)

Сцепление по формату часто можно исключить, изолируя все функции, работающие с конкретной структурой данных, в информационно прочном модуле. Другим модулям может понадобиться имя этой структуры, но они и знают только это имя (адрес), а не формат. Этот прием иллюстрируется ниже в этой главе.

Два модуля *сцеплены по данным*, если один вызывает другой и все входные и выходные параметры вызываемого модуля — простые (не структурные) элементы данных. Предположим, что функция модуля В из предыдущего примера — надпечатать конверт для служащего. Вместо того чтобы передавать В всю анкету, мы могли бы передать ему в качестве аргументов фамилию служащего, улицу, дом, город, штат и почтовый индекс. Модуль В теперь не зависит от записи анкетных данных. А и В стали более независимы, и вероятность ошибки в В меньше, поскольку автор модуля В имеет дело с меньшим количеством данных.

Как и в случае с прочностью модуля, сцепление влияет и на другие, не рассматриваемые здесь специально свойства программы (на адаптируемость, сложность тестирования, возможность повторного использования модулей, простоту или сложность мультипрограммирования [2].

Сцепление пары модулей может удовлетворять определениям нескольких типов. Например, два модуля могут быть сцеплены и по образцу, и по общей области. В этом случае мы относим модули к самому жесткому (худшему) из этих типов сцепления (в данном случае — сцепление по общей области).

Степень прочности и сцепления можно использовать для оценки существующего проекта и как руководящий принцип при проектировании новой программы. Это, однако, не означает, что проект, в котором в отдельных случаях прочность и сцепление далеки от идеала, обязательно плох. Исходя из некоторого компромиссного

решения, проектировщик может пойти на включение модуля, прочного всего лишь по логике. Однако поступая так, он должен уметь убедительно объяснить причины и хорошо понимать следствия своего компромиссного решения. Это, во всяком случае, лучше, чем проектировать, основываясь только на интуиции и полагаться на счастливый случай.

Высокая прочность и слабое сцепление способствуют независимости модулей, поскольку они сводят к минимуму их взаимодействие и их предположения друг о друге. Следующие три критерия проектирования, сформулированные Хольтом в [7], хорошо подытоживают сказанное.

1. Сложность взаимодействия модуля с другими модулями должна быть меньше сложности его внутренней структуры.
2. Хороший модуль снаружи проще, чем внутри.
3. Хороший модуль проще использовать, чем построить.

ДРУГИЕ ХАРАКТЕРИСТИКИ

В дополнение к прочности и сцеплению есть и другие характеристики, оказывающие воздействие на независимость модулей. Они кратко охарактеризованы ниже.

Размеры модуля. Размеры модуля влияют на степень независимости элементов программы, легкость ее чтения и тестирования (например, за счет числа ветвей). Можно было бы удовлетворить критериям высокой прочности и минимального сцепления, спроектировав программу как один огромный модуль, но вряд ли таким образом была бы достигнута высокая степень независимости. Желательно разбивать программу на достаточно большое число модулей, поскольку модули представляют собой явные барьеры внутри программы, что сокращает число связей между операторами и данными программы. Как правило, модуль должен содержать от 10 до 100 выполняемых операторов языка высокого уровня.

Предсказуемые модули. Предсказуемый модуль — это модуль, работа которого не зависит от предыстории его использования. Модуль, хранящий следы своих состояний при последовательных вызовах (например, с помощью установки «переключателя для первого раза»), не является предсказуемым. Все модули должны быть предсказуемы, т. е. они не должны сохранять никаких «воспоминаний» о предыдущем вызове. Хитрые, неуловимые, зависящие от времени ошибки возникают в тех программах, которые пытаются многократно вызывать непредсказуемый модуль.

Структура принятия решений. Всюду, где это возможно, желательно организовать модули и принятие решений в них таким образом, чтобы те модули, на которые прямо влияет принятое решение, были подчиненными (вызываемыми) по отношению к принимающему решение модулю. Таким образом обычно удается

исключить передачу специальных параметров-индикаторов, представляющих решения, которые должны быть приняты, а также принимать влияющие на управление программой решения на высоком уровне в иерархии программы.

Минимизация доступа к данным. Объем данных, на которые модуль может ссылаться, должен быть сведен к минимуму. Исключение сцепления по общей области, внешним данным и по формату — крупный шаг в этом направлении. Проектировщик должен попытаться изолировать сведения о любой конкретной структуре данных или записи в базе данных в отдельном модуле (или небольшом подмножестве модулей), возможно, за счет использования информационно прочных модулей. Проблема глобальных данных не должна решаться передачей одного огромного списка параметров всем модулям. Следуя этим правилам, вы уменьшаете область доступа каждого модуля, благодаря чему ошибки легче изолировать, а область их влияния — уменьшить.

Внутренние процедуры. Внутренняя процедура или подпрограмма — это замкнутая подпрограмма, физически расположенная в вызывающем ее модуле. Внутренних процедур следует избегать по нескольким причинам. Внутренние процедуры трудно изолировать для тестирования (автономного тестирования), и они не могут быть вызваны из модулей, отличных от тех, которые их физически содержат. Это не соответствует идее «повторного использования». Конечно, имеется альтернативный вариант: включить копии внутренней процедуры во все модули, которым она нужна. Однако это часто приводит к ошибкам (копии одной и той же процедуры часто становятся «не совсем точными копиями») и усложняет сопровождение программы (когда процедура изменяется, все использующие ее модули должны быть перекомпилированы). Наконец, если только при программировании не установлена строжайшая дисциплина, внутренние процедуры будут иметь плохие сцепления с вызывающими их модулями. Когда возникает потребность во внутренней процедуре, проектировщик должен рассмотреть возможность оформления ее в виде отдельного модуля.

КОМПОЗИЦИОННЫЙ АНАЛИЗ

Прочность модуля, сцепление и другие рассматривавшиеся характеристики полезны при оценке альтернатив, но они не определяют явно ход мысли при проектировании. В рамках композиционного проектирования имеется процесс, называемый *композиционным анализом*,— нисходящий процесс продумывания проекта. Композиционный анализ включает анализ структуры задачи [1] и ана-

[1] Словом «задача» (problem) автор обозначает здесь (и ниже в этом разделе) скорее некоторую деятельность по обработке данных, чем проблему или «задачу», подлежащую решению.— *Прим. ред.*

лиз преобразования данных по мере их прохождения сквозь эту структуру. Эта информация используется для разбиения задачи на «слои» модулей. Каждый модуль рассматривается затем как подзадача, и анализ повторяется для этой подзадачи и т. д.

Имеются три основные стратегии разбиения при применении композиционного анализа. Для разбиения любой подзадачи используется одна из трех следующих стратегий. *Разбиение* STS (*исток-преобразование-сток*) предполагает деление задачи на функции, занимающиеся получением данных, изменением их формы и затем доставкой их в некоторую точку вне задачи. *Операционное разбиение* состоит в делении задачи на функции-«сестры», каждая из которых выполняет операции отдельного типа. *Функциональное разбиение* — это деление задачи на функции, выполняющие преобразования данных. Разбиение STS обычно применяют для выделения первого слоя модулей, а затем к каждой подзадаче применяется одна из трех стратегий, причем выбор зависит от характеристик подзадачи.

Функциональное и операционное разбиения — это в основном интуитивные процессы, и о них немногое можно добавить к тому, что уже сказано. Разбиение STS — более сложный процесс, и его можно кратко описать в виде пяти шагов.

1. Основываясь на потоке данных в задаче, обрисуйте ее структуру в виде 3—10 процессов.

2. Определите главный входной поток данных, поступающих в задачу, и главный выходной поток.

3. Проследите входной поток данных по структуре задачи. При этом вы обнаружите два явления: входной поток будет изменять форму, становясь все абстрактнее по мере того, как вы следуете по структуре задачи, и в конце концов вы попадете в точку, где он как будто исчезает. Точка, где он появляется в последний раз, называется точкой *наивысшей абстракции* входного потока.

Выполните аналогичный анализ выходного потока данных, начиная с «конца» структуры задачи и двигаясь в обратном направлении. Определите точку, где выходной поток впервые появляется в своей самой абстрактной форме.

Эти точки представляют особый интерес, поскольку они делят задачу на наиболее независимые части.

4. Две найденные точки разбивают структуру задачи на части (обычно три). Представьте эти части как функции и определите модули, выполняющие каждую из этих функций. Эти модули становятся подчиненными по отношению к модулю, разбиение которого выполняется.

5. Определите сопряжения этих модулей. В этот момент вы должны интересоваться только определением вида данных для каждого сопряжения. Это значит, что вам следует дать качественное описание входных и выходных аргументов, не заботясь об их точной при-

роде (порядок, атрибуты, представление). Детали каждого сопряжения будут определены в одном из последующих процессов проектирования (внешнее проектирование модуля, описанное в гл. 8).

Процессы разбиения повторяются на следующих, более низких уровнях в иерархии, и так до момента остановки. Этот момент определяется по следующему общему правилу: логика модулей должна стать «интуитивно очевидной» (это означает, что модуль вероятно, будет содержать не более 50 предложений).

Результатом анализа является иерархическая структурная схема, отражающая структурные отношения между всеми модулями (кто кого вызывает), функции каждого модуля и сопряжения между ними. Обозначения для таких схем описаны Майерсом в [2].

ПРИМЕР КОМПОЗИЦИОННОГО АНАЛИЗА

Простейший путь к пониманию композиционного анализа — рассмотреть его применение на примере. Этот же пример будет использован в последующих главах для иллюстрации других процессов проектирования и тестирования.

Выбирая подходящий пример, я сразу же столкнулся с несколькими проблемами. В качестве примера нельзя взять большую программу; она должна по размерам подходить для учебника, но при этом оставаться нетривиальной. Прикладная программа (например, программа начисления зарплаты) может не заинтересовать системных программистов и даже многих прикладных программистов; привлекательность какой-либо компоненты операционной системы была бы в такой же степени ограниченной. В качестве компромисса я выбрал *загрузчик* — программу, находящуюся как бы посередине между прикладными программами и операционной системой. Другая причина, по которой я выбрал в качестве примера загрузчик, такова: я надеялся, что большинство читателей будет по крайней мере в общих чертах знакомо с его назначением.

Как уже говорилось в начале главы, исходной информацией для процесса проектирования структуры программы являются детальные внешние спецификации. Вместо того чтобы дать здесь такие спецификации для загрузчика, я просто опишу его назначение, а также входные и выходные данные настолько подробно, чтобы мы могли спроектировать его структуру.

Назначение загрузчика — поместить программу в основную память в готовом к выполнению состоянии (некоторые загрузчики сразу же и активизируют эту программу, но такие подробности нас здесь не интересуют). Исходными данными для загрузчика является файл ВХФАЙЛ (INFILE), содержащий один или несколько объектных модулей, выработанных компилятором. Загрузчик может иметь в качестве исходных данных также и файл библиотеч-

ных программ БИБПРОГ (PROGLIB), содержащий большую библиотеку объектных модулей, которые могут потребоваться в загружаемой программе. Результатами работы загрузчика являются: загруженная в основную память программа и выходной файл ВЫХФАЙЛ (OUTFILE), который содержит таблицу загрузки, описывающую расположение модулей загруженной программы в основной памяти, и список сообщений об ошибках, если они есть.

ВХФАЙЛ — это последовательный файл, содержащий один или несколько объектных модулей. БИБПРОГ — некоторый файл с индексным или библиотечным методом доступа, в котором объектные модули хранятся раздельно. Все они имеют одинаковый формат: одна или несколько записей СВИМ (ESD — словарь внешних имен), описывающих внешние имена и внешние ссылки в модуле, две или более записи ТКСТ (TXT — текст), которые содержат объектный (машинный) код модуля, за ними может следовать несколько записей СПРМ (RLD — словарь перемещений), описывающих все адресные константы модуля, и (обязательно) запись КОНЕЦ (END). Каждая запись СВИМ содержит некоторое символическое имя, его тип (имя модуля, имя входа или ссылка на внешнее имя) и относительное смещение внутри модуля. Первая запись ТКСТ содержит размер объектного модуля. Каждая следующая запись содержит размер объектного кода и поле, указывающее его длину в этой записи. Записи СПРМ описывают все адресные константы в модуле, т. е. адреса, которые должны быть настроены, когда модуль получает определенное место в основной памяти. Запись СПРМ содержит относительное смещение адресной константы и номер соответствующей записи СВИМ (каждая адресная константа в объектном модуле связана с внешним именем). Адресным константам, указывающим на области внутри модуля, соответствует запись СВИМ для имени входа. Адресным константам, указывающим на другие модули, соответствуют записи СВИМ для ссылок на внешние имена других модулей.

Загрузить программу — это значит сделать следующее:

1. Поместить объектный код каждого модуля из ВХФАЙЛ на предназначенное для него место в основной памяти.

2. Убедиться, что все внешние ссылки замкнуты. Например, если загружается модуль А, который вызывает модуль В, то загрузчик должен убедиться, что модуль В также загружается. Если модуля В нет во ВХФАЙЛ, то он загружается из БИБПРОГ, если он там найден. Отметим, что модуль В также может содержать внешние ссылки на другие модули, которые также должны быть в конечном итоге замкнуты.

3. Настроить все адресные константы. Адресная константа — это часть объектного кода, содержащая некоторый адрес. Мы предполагаем, что все адресные константы имеют одну фиксированную

Исходная программа (на гипотетическом языке)

OFFSET	
0	PROC M
	•
	•
50	CALL C
	•
	•
100	ENTRY B
	•
	•
200	Y DCL ADDR (X)
	•
	•
220	X DCL o o o
	•
	•
300	

0	PROC C
	•
	•
30	CALL B
	•
	•
60	

Содержание ВХФАЙЛа

				Примечания
ESD	M	MD	0000	
ESD	B	EP	0100	
ESD	C	ER	0000	
TXT			0300	
TXT		000000		*Адресная константа со смещением 54 для ссылки на модуль C*
TXT		000220		*Адресная константа со смещением 200 для ссылки на X*
RLD	0054	3		
RLD	0200	1		
END				
ESD	C	MD	0000	
ESD	B	ER	0000	
TXT			0060	
TXT		000000		*Адресная константа со смещением 34 для ссылки на вход B*
RLD	0034	2		
END				

Рис. 6.1. Пример входных данных для загрузчика.

длину. Компилятору ничего не известно о том, в какое место в памяти будет загружена программа, поэтому он помещает в поля адресных констант *относительные* адреса. Вместо адресных констант, указывающих на точки внутри этого же модуля, компилятор помещает смещение этой точки относительно начала модуля. Вместо адресных констант, указывающих на внешние ссылки, компилятор помещает нули. Как только всем необходимым модулям выделено место в основной памяти, загрузчик присваивает соответствующие значения всем адресным константам.

Результаты в ВЫХФАЙЛе

ТАБЛИЦА ЗАГРУЗКИ

МОДУЛЬ/ВХОД	АДРЕС	ТИП
М	100000	МД
В	100100	ВХ
С	100300	МД

Результат в главной памяти

Байты	100000-1002FF	содержат	текст модуля М
»	100054	»	100300
»	100200	»	100220
»	100300-100360	»	текст модуля С
»	100334	»	100100

Рис. 6.2. Соответствующие выходные данные.

Вместо того чтобы объяснить этот процесс более подробно, я привел на рис. 6.1 образец исходных данных загрузчика, а на рис. 6.2— соответствующий результат. Из рис. 6.1 и 6.2 можно извлечь достаточно информации, чтобы понять задачу. Проектируя загрузчик, мы предполагаем, что он будет работать в некоторой операционной системе, имеющей средства распределения основной памяти и средства ввода-вывода, необходимые для выполнения опрераций чтения, записи и поиска над входными и выходными файлами.

Первый шаг в проектировании загрузчика состоит в определении главного модуля, функция которого эквивалентна функции всего загрузчика. Этот модуль назовем ЗАГРУЗИТЬ-ПРОГРАММУ (LOAD-A-PROGRAM).

На следующем шаге проектирования надо рассмотреть этот модуль как задачу, подлежащую решению, и, применяя разбиение STS, разложить его на более мелкие функции. На рис. 6.3 показано, какие пять процессов выделяются в нашей задаче, если проследить за потоком данных. Основной входной поток — это объектные модули; основной результат — загруженная программа (таблица загрузки представляет собой вторичный результат). Осознав, что объектный модуль — это модуль с *перемещаемыми* адресными константами, а загруженная программа — набор модулей с *абсолютными* значениями адресных констант, легко найти точки наивысшей абстракции (они отмечены звездочками). Теперь задача разбита на три функции и начата разработка структуры модуля, как показано на рис. 6.3.

На этом уровне проектирования осталось определить только сопряжения модулей (отмеченные цифрами 1, 2 и 3). Начнем с сопря-

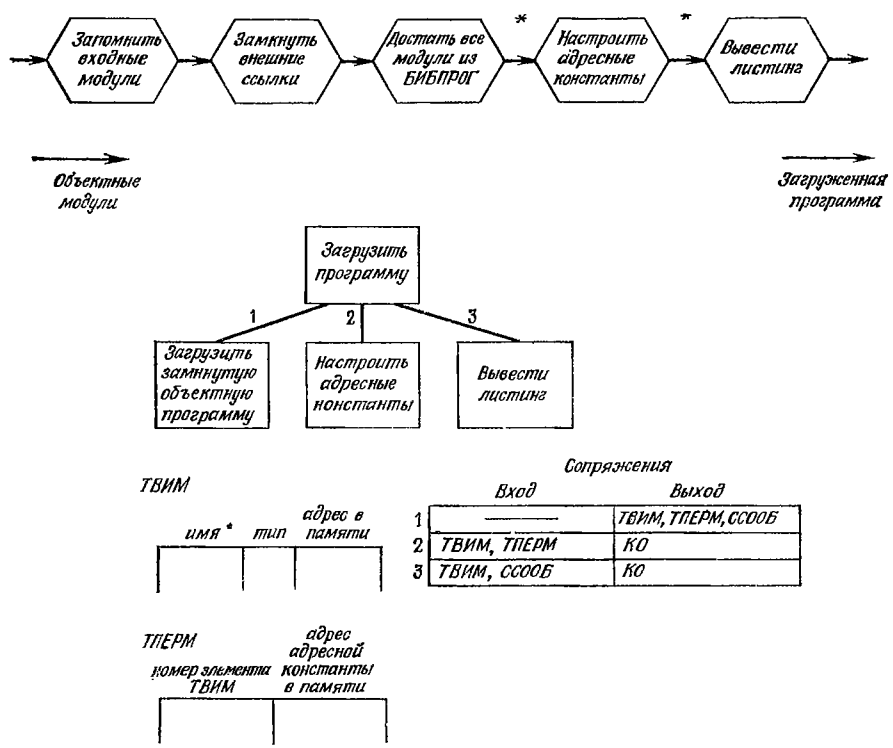
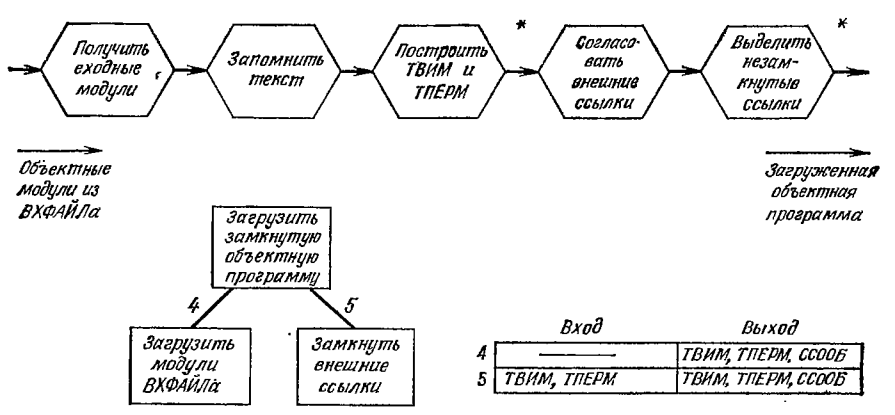


Рис. 6.3. Первоначальное разбиение.

жения 3. Мы знаем, что выходной листинг состоит из таблицы загрузки (списка внешних имен и соответствующих им абсолютных адресов) и сообщений об ошибках. Входные данные сопряжения 3 могут быть определены как список сообщений об ошибках и ТВИМ (ESTAB) — таблица, содержащая внешние имена, их типы и абсолютные адреса. Отметим, что в процессе проектирования структуры программы нас не интересует точное представление данных (например, позднее ТВИМ может быть определена как последовательный или односвязный список). Выходные данные сопряжения 3— еще подлежащий определению код ошибки.

Функции НАСТР-АДР (RELOCATE-ADCONS) нужны входные данные двух типов: описание внешних имен (в частности, их абсолютные адреса) и описание всех адресных констант. Сопряжение 2 содержит два элемента входных данных: ТВИМ и ТПЕРМ (RLTAB), таблицу, содержащую указатель на каждую адресную константу и указатель на соответствующий элемент ТВИМ (или его индекс в таблице). Нет необходимости передавать загруженные объектные модули как параметры, поскольку в ТВИМ есть указатели на них.



	Вход	Выход
4	——	ТВИМ, ТПЕРМ, ССООБ
5	ТВИМ, ТПЕРМ	ТВИМ, ТПЕРМ, ССООБ

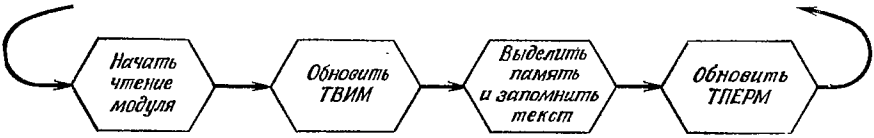
Рис. 6.4. Разбиение ЗАГРУЗИТЬ-ЗАМКН-ПРОГР.

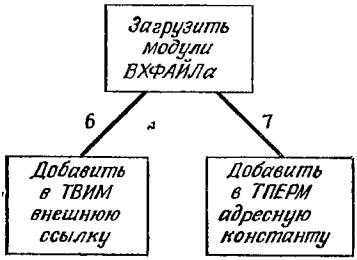
Выход сопряжения — только код ошибки. Теперь определено и сопряжение 1; оно не имеет входных данных, а выходные данные — ТВИМ, ТПЕРМ и список сообщений обо всех ошибках, обнаруженных при выполнении функции.

На следующем шаге нужно взять модули, изображенные на рис. 6.3, и заняться их разбиением. Логику модулей НАСТР-АДР и ВЫДАТЬ-ЛИСТИНГ (PRODUCE-OUTPUT-LISTING) легко себе представить, поэтому их разбиением мы здесь заниматься не будем. Таким образом, остается только модуль ЗАГРУЗИТЬ-ЗАМКН-ПРОГР (LOAD-RESOLVED-OBJECT-PROGRAM). Первым делом рассмотрим его как подлежащую решению задачу и наметим его структуру, как показано на рис. 6.4. Входной поток — множество объектных модулей из ВХФАЙЛ, а выходной поток — объектная программа, в которой замкнуты все внешние ссылки. Точка наивысшей абстракции входного потока — там, где все модули из ВХФАЙЛ размещены в памяти и представлены в ТВИМ и ТПЕРМ. Точка наивысшей абстракции выходного потока находится на самом выходе задачи. Поэтому рассматриваемая задача разбивается только на две функции (определены два соответствующих подчиненных модуля)

Сопряжение 4 не имеет входных данных и возвращает в качестве выходных данных ТВИМ и ТПЕРМ и список сообщений об ошибках. Сопряжению 5, очевидно, необходима в качестве входной информации ТВИМ. Однако, вследствие того что модуль ЗАМКНУТЬ ВНЕШН-ССЫЛКИ (RESOLVE-EXTERNAL-SYMBOLS) может потребовать загрузки модулей из БИБПРОГ, может оказаться необходимым включить дополнительные элементы как в ТВИМ, так и в ТПЕРМ. Поэтому обе таблицы должны быть и входными данными, и выходными.

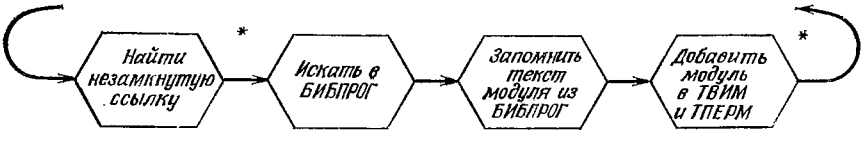
Двигаясь вниз по иерархии, мы теперь можем разбить модуль ЗАГРУЗИТЬ-МОДУЛИ-ВХФАЙЛА (LOAD-INFILE-MODULES).





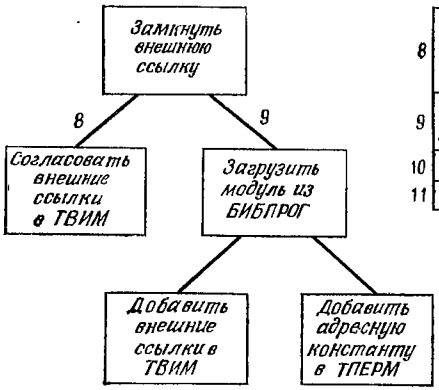
	Вход	Выход
6	ТВИМ, имя, тип, адрес	ТВИМ, имя элемента, КО
7	ТПЕРМ, имя элемента, адрес адресной константы	ТПЕРМ, КО

Рис. 6.5. Разбиение ЗАГРУЗИТЬ-МОДУЛИ-ВХФАЙЛ.



Незамкнутые внешние ссылки

Законченная объектная программа



	Вход	Выход
8	ТВИМ	ТВИМ, признак ГОТОВО, имя незамкнутой внешней ссылки
9	ТВИМ, ТПЕРМ, имя модуля	ТВИМ, ТПЕРМ, ССООБ
10	То же, что для сопряжения 6	
11	То же, что для сопряжения 7	

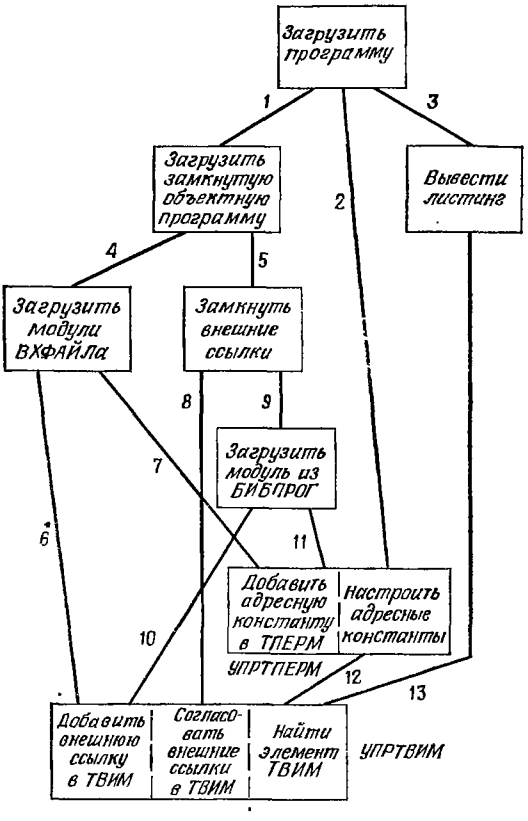
Рис. 6.6. Окончательное разбиение.

Эта задача по своей структуре является циклическим процессом, изображенным на рис. 6.5. Она достаточно проста, так что разбивать ее дальше не обязательно. Поскольку, однако, я преследую определенную цель (минимизировать число модулей, осведомленных о представлении ТВИМ и ТПЕРМ), в ней выделены две функции. Отметим, что модуль ЗАГРУЗИТЬ-МОДУЛИ-ВХФАЙЛ обращается также к операционной системе для чтения из ВХФАЙЛ и выделения блоков основной памяти. ТВИМ является входной для сопряжения 6, а ТПЕРМ — для сопряжения 7, что позволяет обоим модулям быть предсказуемыми и повторно входимыми. Одно из значений кода ошибки (КО) в сопряжении 6 указывает на попытку повторно включить в ТВИМ имя типа МД (модуль) или ВХ (вход).

Теперь мы можем заняться разбиением другого модуля. Выберем ЗАМКНУТЬ-ВНЕШН-ССЫЛКИ, изображенный на рис. 6.4. Входной поток — множество (возможно, пустое) незамкнутых внешних имен, в выходной поток — законченная объектная программа. Точки наивысшей абстракции и полученное в результате разбиение показаны на рис. 6.6.

Стоит обратить внимание на следующее. Модули ЗАГРУЗИТЬ-МОДУЛЬ-БИБПРОГ (LOAD-PROGLIB-MODULE) и ЗАГРУЗИТЬ-МОДУЛИ-ВХФАЙЛА имеют сходные функции, и можно было бы обобщить последний модуль, с тем чтобы он выполнял обе функции. Такое проектное решение не было бы неразумным, но я предпочитаю оставить оба модуля по двум причинам: во-первых, файлы организованы по-разному и, во-вторых, иначе возникло бы сцепление по управлению (одному модулю нужно было бы явно указывать файл, из которого следует читать). Отметим, однако, что я могу использовать модули, подчиненные модулю ЗАГРУЗИТЬ-МОДУЛИ-ВХФАЙЛА, как это показано на рисунке.

Модуль СОГЛАСОВАТЬ-ВНЕШН-ССЫЛКИ (MATCH-ERITEMS) совсем прост. Он просматривает ТВИМ в поисках элементов с еще не замкнутыми именами (отмеченными нулями в адресном поле). Встретив такое имя, он ищет в ТВИМ элемент типа МД или ВХ с этим же именем. Если такой элемент найден, его адресное поле помещается в адресное поле первого элемента и затем модуль продолжает поиски незамкнутых имен. Работа заканчивается, если таковых не осталось (в этом случае устанавливается флажок ГОТОВО) либо если найдено имя, которое не удается замкнуть (оно и возвращается как результат). Модуль ЗАМКНУТЬ-ВНЕШН-ССЫЛКИ также вполне прост; он циклически вызывает модуль СОГЛАСОВАТЬ-ВНЕШН-ССЫЛКИ и ЗАГРУЗИТЬ-МОДУЛЬ-БИБПРОГ до тех пор, пока не будут замкнуты все имена или встретится незамкнутое имя, которое не найдено в БИБПРОГ. Это не самый эффективный способ выполнения функции, но самый простой. Как говорит Кнут в [8], «преждевременная оптимизация — корень всех зол». После того как мы получим работающую программу, можно будет измерить



Примечания

1. «Загрузить модули ВХФАЙЛа» использует функции операционной системы GET (из ВХФАЙЛа) и GETMAIN (для выделения памяти).
2. «Загрузить модуль из БИБПРОГ» использует функции системы FIND и GET (из БИБПРОГ) и GETMAIN.
3. «Вывести листинг» использует функцию системы PUT (в ВЫХФАЙЛ).
4. Элемент ТВИМ содержит имя, тип (модуль, вход или внешн. ссылка) и адрес выделенного поля памяти.
5. Элемент ТПЕРМ содержит номер соответствующего элемента ТВИМ и адрес адресной константы в памяти.

	Вход	Выход
1	———	ТВИМ, ТПЕРМ, ССООБ
2	ТВИМ, ТПЕРМ	КО
3	ТВИМ, ССООБ	КО
4	———	ТВИМ, ТПЕРМ, ССООБ
5	ТВИМ, ТПЕРМ	ТВИМ, ТПЕРМ, ССООБ
6	ТВИМ, имя, тип, адрес	ТВИМ, номер эл-та, КО
7	ТПЕРМ, номер эл-та, адрес адр. константы	ТПЕРМ, КО
8	ТВИМ	ТВИМ, флажок ГОТОВО, имя незамкн. внешн. ссылки
9	ТВИМ, ТПЕРМ, имя модуля	ТВИМ, ТПЕРМ, ССООБ
10	совпадает с	сопряжением 6
11	совпадает с	сопряжением 7
12	ТВИМ, имя элемента	Имя, тип, адрес, КО
13	совпадает с	сопряжением 12

Рис. 6.7. Окончательный результат.

ее производительность и оптимизировать логику одного или нескольких модулей, если для этого будут основания. (Эти подробности упоминаются здесь только для того, чтобы читатель понимал особенности нашего загрузчика; в действительности в этот момент не следует интересоваться логикой модуля.)

Оценивая текущее состояние нашего проектирования, я больше не вижу модулей, подлежащих разбиению. Можно было бы поэтому объявить проектирование законченным. Однако я по-прежнему имею

в виду упоминавшуюся ранее цель — свести к минимуму число модулей, знающих свойства и представление ТВИМ и ТПЕРМ. Я могу осуществить это, создавая из некоторых модулей информационно прочные обобщенные модули, как это показано на рис. 6.7. Чтобы изолировать все сведения о ТВИМ в одном модуле, мне нужно создать новую функцию (вход) с именем НАЙТИ-ЭЛЕМ-ТВИМ (FIND-ITEM-IN-ESTAB), вызываемую из НАСТР-АДР и ВЫДАТЬ-ЛИСТИНГ, как это показано на рисунке. Отметим, что, хотя и другие модули передают ТВИМ и ТПЕРМ как параметры, только два информационно прочных модуля, УПРТВИМ (ESTABMGR) и УПРТПЕРМ (RLTABMGR), знают внутреннюю структуру этих таблиц. Например, только УПРТВИМ знает, каков формат элементов ТВИМ, является ли ТВИМ последовательной таблицей или списком, сортированы ли ее элементы и т. п.

В окончательном проекте, изображенном на рис. 6.7, мы достигли следующих результатов:

1. Шесть модулей функционально прочны; остальные два — информационно прочны.
2. Каждый модуль мал, его логику легко понять.
3. Знания о ТВИМ и ТПЕРМ скрыты в каждом случае в единственном модуле.
4. Только два модуля осведомлены о формате объектных модулей, вырабатываемых компилятором.
5. За исключением этих двух модулей (которые сцеплены по формату), все остальные сцеплены только по данным.
6. Все операции ввода-вывода для каждого файла сосредоточены в единственном модуле.

Напомним только, что в реальном загрузчике будет еще одно дополнение: он либо сразу активизирует загруженную программу, либо вернет адрес ее входа обратившейся к нему программе.

ПРОВЕРКА ПРАВИЛЬНОСТИ

В поисках недостатков и ошибок в структуре программы можно последовательно применить три метода: метод «n плюс-минус один», статическую проверку и сквозной контроль. Метод «n плюс-минус один» — это официальная проверка проектной документации разработчиками этапа $n-1$ (авторами архитектуры системы и внешних спецификаций), которые ищут ошибки перевода, и разработчиками этапа $n+1$ (создателями внешнего проекта модуля), которые проверяют, насколько осуществим и понятен проект, согласуется ли он с языком программирования и операционной системой, которые предполагается использовать.

Статическая проверка состоит в оценке проекта с точки зрения рассмотренных ранее в этой главе характеристик, выполняемой членами второй группы. Участников этой проверки должны интере-

совать такие вопросы: «Все ли модули функционально или информационно прочны? Если нет, то почему? Сильно ли они сцеплены? Все ли модули предсказуемы? Выполнено ли разбиение до конца? (То есть можете ли вы отчетливо представить себе логику каждого модуля?) Минимизирован ли доступ к данным каждого модуля?»

Третий метод проверки — сквозной контроль, аналогичный рассматривавшемуся для предыдущих процессов проектирования. Сначала нужно разработать тесты для «мысленной» проверки (например, рис. 6.1 и 6.2 можно использовать как тесты для загрузчика) и для каждого теста проследить за действиями системы шаг за шагом, двигаясь по структуре модулей. При этом следует предполагать, что логика каждого модуля правильна, т. е. что каждый модуль правильно выполняет свои функции. Ищите в структуре такие дефекты, как недостающие функции, неполные сопряжения, неправильные результаты. Рассмотрите достаточное количество тестов, чтобы каждый модуль был вызван по крайней мере один раз. Рассмотрите в качестве тестов и неправильные входные данные (например, объектный модуль без записей ТВИМ), а также тесты с «граничными условиями» (например, объектный модуль без внешних ссылок или адресных переменных).

ЛИТЕРАТУРА

1. Liskov B. H. A Design Methodology for Reliable Software Systems, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, N. J.: AFIPS Press, 1972, pp. 191—199.
2. Myers G. J. Reliable Software Through Composite Design. New York: Petrocelli/Charter, 1975.
3. Parnas D. L. On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, **15**(2), 1053—1058 (1972).
4. Wulf W., Shaw M. Global Variable Considered Harmful, *SIGPLAN Notices*, **8** (2), 28—34 (1973).
5. Spier M. J. A Critical Look at the State of our Science, *Operating Systems Review*, **8** (2), 9—15 (1974).
6. Goodenough J. B., Ross D. T. The Effect of Software Structure on Reliability, Modifiability, Reusability, Efficiency: A Preliminary Analysis, Report R-2099, SofTech Corp., Waltham, Mass., 1973.
7. Holt R. C. Structure of Computer Programs: A Survey, *Proceedings of the IEEE*, **63** (6), 879—893 (1975).
8. Knuth D. E. Structured Programming with GO TO Statements, *Computing Surveys*, **6** (4), 261—301 (1974).
9. Композиционное проектирование тесно связано с другой методологией, называемой «структурное проектирование»; основные различия касаются лишь терминологии и обозначений. Структурное проектирование описано в E. Yourdon and L. L. Constantine, Structured Design. New York: Yourdon, 1975 [1].

---

[1] См. также Йодан Э. Структурное проектирование и конструирование программ.— М.: Мир, 1979.

ГЛАВА 7

Методы проектирования

В гл. 3 подчеркивалось, что из четырех основных групп методов обеспечения надежности наилучшие результаты дают методы предупреждения ошибок. В большинстве случаев, однако, при разработке программного обеспечения никак нельзя предполагать, что готовая программа не будет содержать ошибок. В этом и состоит исходная предпосылка методов обнаружения ошибок, исправления ошибок и обеспечения устойчивости к ошибкам: готовая программа (система) будет содержать ошибки и поэтому должна быть спроектирована так, чтобы ее поведение было предсказуемо и в случае ошибки. При этом имеются в виду как ошибки в программном обеспечении, так и ошибки пользователя или сбои аппаратуры.

ПАССИВНОЕ ОБНАРУЖЕНИЕ ОШИБОК

Если мы исходим из предположения, что в программном обеспечении будут ошибки, то, очевидно, в первую очередь следует принять меры для их обнаружения. Более того, если необходимо принимать дополнительные меры (например, исправлять ошибки или их последствия), то все равно сначала нужно уметь обнаруживать ошибки.

Меры по обнаружению ошибок можно разбить на две подгруппы: *пассивные* попытки обнаружить симптомы ошибки в процессе «обычной» работы программного обеспечения и *активные* попытки программной системы периодически обследовать свое состояние в поисках признаков ошибок. Пассивное обнаружение рассматривается в этом разделе, активное — в следующем.

Меры по обнаружению ошибок могут быть приняты на нескольких структурных уровнях программной системы. В этой главе мы будем заниматься уровнем подсистем, или компонент, т. е. нас будут интересовать меры по обнаружению симптомов ошибок, предпринимаемые при переходе от одной компоненты к другой, а также внутри компоненты. Все это, конечно, применимо также к отдельным модулям внутри компоненты.

Разрабатывая эти меры, мы будем опираться на следующие положения:

1. *Взаимное недоверие.* Каждая из компонент должна предполагать, что все другие содержат ошибки. Когда она получает какие-нибудь данные от другой компоненты или из источника вне системы, она должна предполагать, что данные могут быть неправильными, и пытаться найти в них ошибки.

2. *Немедленное обнаружение.* Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.

3. *Избыточность.* Все средства обнаружения ошибок основаны на некоторой форме избыточности (явной или неявной).

Конкретные меры обнаружения сильно зависят от специфики прикладной области. Однако некоторые идеи можно почерпнуть из следующего списка:

1. Проверяйте атрибуты любого элемента входных данных. Если входные данные должны быть числовыми или буквенными, проверьте это. Если число на входе должно быть положительным, проверьте его значение. Если известно, какой должна быть длина входных данных, проверьте ее.

2. Применяйте «тэги» [1] в таблицах, записях и управляющих блоках и проверяйте с их помощью допустимость входных данных. Тэг — это поле записи, явно указывающее на ее назначение. Так, в примере с загрузчиком в гл. 6 каждая запись СВИМ снабжена тэгом — она физически содержит символы «СВИМ». Система, которая предполагает получать на входе записи СВИМ, должна проверять наличие в конкретных входных данных этого тэга.

3. Проверяйте, находится ли входное значение в установленных пределах. Например, если входной элемент — адрес в основной памяти, проверяйте его допустимость. Всегда проверяйте поле адреса или указателя на нуль и считайте, что оно неверно, если равно нулю. Если входные данные — таблица вероятностей, проверьте, находятся ли все значения между нулем и единицей.

4. Проверяйте допустимость всех вариантов значений. Если входное поле — код, обозначающий один из десяти районов, никогда не предполагайте, что если это не код ни одного из районов 1, 2, ..., 9, то это обязательно код района 10.

5. Если во входных данных есть какая-либо явная избыточность, воспользуйтесь ею для проверки данных.

6. Там, где во входных данных нет явной избыточности, введите ее. Если ваша система использует крайне важную таблицу, подумайте о включении в нее контрольной суммы. Всякий раз, когда таблица обновляется, следует просуммировать (по некоторому модулю) ее поля и результат поместить в специальное поле контрольной суммы. Подсистема, использующая таблицу, сможет теперь проверить, не была ли таблица случайно испорчена,— для этого только нужно выполнить контрольное суммирование.

7. Сравнивайте, согласуются ли входные данные с какими-либо внутренними данными. Если на входе операционной системы возникает требование освободить некоторый блок памяти, она должна убедиться, что этот блок в данный момент действительно занят.

Когда разрабатываются меры по обнаружению ошибок, важно принять согласованную стратегию для всей системы (т. е. применить идею концептуальной целостности, о которой шла речь в гл. 4, к обнаружению ошибок). Действия, предпринимаемые после обнаружения ошибки в программном обеспечении (например, возврат кода ошибки), должны быть единообразными для всех компонент системы. Это ставит вопрос о том, какие именно действия следует предпринять, когда ошибка обнаружена. Наилучшее решение — немедленно завершить выполнение программы или (в случае операционной системы) перевести ЦП в состояние ожидания. С точки зрения предоставления человеку, отлаживающему программу, например системному программисту, самых благоприятных условий для диагностики ошибок немедленное завершение представляется наилучшей стратегией. Конечно, во многих системах подобная стратегия бывает нецелесообразной (например, может оказаться, что приостанавливать работу системы нельзя). В таком случае используется метод *регистрации ошибок*. Описание симптомов ошибки и «моментальный снимок» состояния системы сохраняется во внешнем файле, после чего система может продолжать работу. Этот файл позднее будет изучен обслуживающим персоналом. Такой метод использован в операционной системе OS/VS2 MVS фирмы IBM. Каждая компонента содержит программу восстановления, которая перехватывает все случаи ненормального завершения и программные прерывания в этой компоненте и регистрирует данные об ошибке во внешнем файле.

Всегда, когда это возможно, лучше приостановить выполнение программы, чем регистрировать ошибки (либо обеспечить как дополнительную возможность работу системы в любом из этих режимов). Различие между этими методами проиллюстрируем на способах выявления причин возникающего иногда скрежета вашего автомобиля. Если автомеханик находится на заднем сиденье, то он может обследовать состояние машины в тот момент, когда скрежет возникает. Если же вы выбираете метод регистрации ошибок (записывая скрежет на магнитофон), задача диагностики будет значительно сложнее.

Пример: система PRIME

Система PRIME — это мультипроцессорная система с виртуальной памятью, разработанная в Калифорнийском университете в Беркли [2]. Ее упрощенная схема показана на рис. 7.1. Один из

процессоров системы выделен в качестве центрального процессора и содержит центральный управляющий монитор (ССМ — central control monitor) — управляющую программу, которая распределяет страницы памяти и пространство на диске, назначает программы другим процессорам (проблемным процессорам) и регулирует пересылки всех межпроцессорных сообщений. Расширенный управляющий монитор (ЕСМ — extended control monitor) — это реализованная микропрограммно управляющая программа, постоянно присутствующая в каждом процессоре и управляющая диспетчеризацией процессов, операциями ввода-вывода и посылки/получения.

Защита данных — одна из основных целей системы. В этом направлении в системе PRIME сделан шаг вперед по сравнению с большинством других систем: в дополнение к обеспечению защиты в нормальных условиях ставится цель гарантировать защиту даже при наличии отдельной ошибки в программном обеспечении или отдельного сбоя аппаратуры. Меры по обнаружению ошибок составляют основу метода достижения этой цели.

Во время разработки системы PRIME были явно выделены все принимаемые операционной системой решения, ошибки в которых могут привести к тому, что данные одного пользователя станут доступными другому пользователю или его программе. Были реализованы средства обнаружения ошибок для каждого из этих решений.

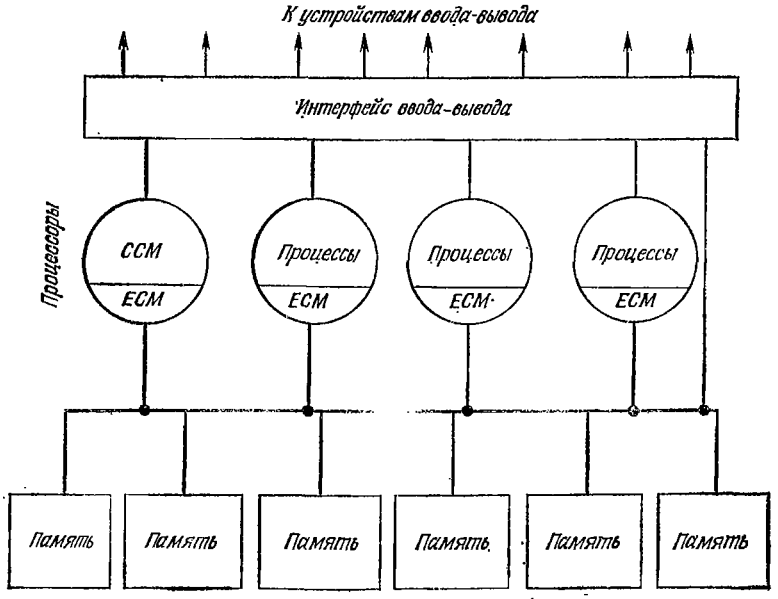


Рис. 7.1. Система PRIME.

Многие из них представляют собой комбинацию аппаратных и программных средств. Например, всякий раз, когда процессу пользователя выделяется терминал, ССМ запоминает идентификатор этого процесса в регистре терминала. Когда ЕСМ посылает данные на терминал, тот всегда сравнивает хранимый идентификатор с идентификатором при посланных данных. Последнее гарантирует, что отдельная ошибка в программном обеспечении или сбой аппаратуры не приведут к печати данных не на том терминале.

Система PRIME содержит механизм посылки/получения, который позволяет процессу одного пользователя послать сообщение процессу другого пользователя. Для этого процесс-отправитель передает своему ЕСМ это сообщение и идентификатор процесса-получателя. ЕСМ добавляет идентификатор отправителя и передает сообщение ССМ. Тот передает сообщение ЕСМ процессора, содержащего процесс-получатель, а ЕСМ наконец передает сообщение указанному процессу пользователя. В этой последовательности выполняются три проверки с целью обнаружения ошибки. ССМ проверяет, правильный ли идентификатор процесса-отправителя добавлен ЕСМ; он может это сделать, поскольку известно, какому пользователю выделен процессор. ЕСМ адресата проверяет, тому ли процессору ССМ направил сообщение; он выполняет это, сравнивая идентификатор процесса в сообщении с идентификатором процесса, которому в данный момент выделен процессор. Третья проверка делается для обеспечения сохранности сообщения при транзите. ЕСМ-отправитель формирует для сообщения контрольную сумму и передает ее вместе с сообщением. ЕСМ-получатель вычисляет контрольную сумму доставленного сообщения и сравнивает с извлеченной из сообщения.

В качестве примера другой проверки отметим, что ССМ распределяет свободные страницы памяти между процессами пользователей и хранит список свободных страниц. Когда страница больше не нужна ЕСМ, он помечает ее и сообщает ССМ, чтобы тот включил ее в список свободных. Получая очередную выделенную страницу, ЕСМ проверяет пометку на ней, чтобы убедиться, что страница действительно свободна. Эта избыточность может быть использована для обнаружения ошибок, потому что список свободных страниц обрабатывается ССМ, а помечают страницы другие процессоры.

АКТИВНОЕ ОБНАРУЖЕНИЕ ОШИБОК

Не все ошибки можно выявить пассивными методами, поскольку эти методы обнаруживают ошибку лишь тогда, когда ее симптомы подвергаются соответствующей проверке. Можно делать и дополнительные проверки, если спроектировать специальные программные средства для активного поиска признаков ошибок в системе. Такие средства называются *средствами активного обнаружения ошибок.*

Активные средства обнаружения ошибок обычно объединяются в *диагностический монитор*: параллельный процесс, который периодически анализирует состояние системы с целью обнаружить ошибку. Большие программные системы, управляющие ресурсами, часто содержат ошибки, приводящие к потере ресурсов на длительное время. Например, управление памятью операционной системы сдает блоки памяти «в аренду» программам пользователей и другим частям операционной системы. Ошибка в этих самых «других частях» системы может иногда вести к неправильной работе блока управления памятью, занимающегося возвратом сданной ранее в аренду памяти, что вызывает медленное вырождение системы.

Диагностический монитор можно реализовать как периодически выполняемую задачу (например, она планируется на каждый час) либо как задачу с низким приоритетом, которая планируется для выполнения в то время, когда система переходит в состояние ожидания. Как и прежде, выполняемые монитором конкретные проверки зависят от специфики системы, но некоторые идеи будут понятны из примеров. Монитор может обследовать основную память, чтобы обнаружить блоки памяти, не выделенные ни одной из выполняемых задач и не включенные в системный список свободной памяти. Он может проверять также необычные ситуации: например, процесс не планировался для выполнения в течение некоторого разумного интервала времени. Монитор может осуществлять поиск «затерявшихся» внутри системы сообщений или операций ввода-вывода, которые необычно долгое время остаются незавершенными, участков памяти на диске, которые не помечены как выделенные и не включены в список свободной памяти, а также различного рода странностей в файлах данных.

Иногда желательно, чтобы в чрезвычайных обстоятельствах монитор выполнял диагностические тесты системы. Он может вызывать определенные системные функции, сравнивая их результат с заранее определенным и проверяя, насколько разумно время выполнения. Монитор может также периодически предъявлять системе «пустые» или «легкие» задания, чтобы убедиться, что система функционирует хотя бы самым примитивным образом.

Пример: программа обнаружения разрушений, разработанная фирмой TRW

Система защиты ресурсов фирмы IBM — это экспериментальная модификация операционной системы OS/360 для изучения проблем, связанных с системами защиты. Используя ее, корпорация TRW разработала монитор, действующий в заранее установленные интервалы времени и пытающийся обнаружить признаки того, что программа пользователя нарушила правила защиты [3].

Этот монитор проверяет много различных условий. Большинство

из них характерно именно для OS/360 и поэтому интересны не для всех. В качестве некоторых примеров, можно, однако, указать, что монитор определяет, вся ли управляющая информация OS/360 о задачах пользователя хранится в защищенной области памяти. Монитор также проверяет, все ли программы пользователя выполняются в режиме задачи и вся ли память пользователя защищена от выборки соответствующим ключом защиты. Монитор контролирует правильность соблюдения очереди ожидающими операциями ввода-вывода и гарантирует, что точки входа для всех прерываний являются соответствующими входами OS/360 и что вся память супервизора надлежащим образом защищена. Как только обнаруживается какое-то несоответствие, немедленно выдается сообщение оператору.

ИСПРАВЛЕНИЕ ОШИБОК И УСТОЙЧИВОСТЬ К ОШИБКАМ

Имея средства обнаружения ошибок в программном обеспечении, естественно предпринять следующий шаг, попробовать создать средства, нацеленные на исправление обнаруженных ошибок. По существу, термин «исправление ошибок» в применении к программному обеспечению означает ликвидацию *ущерба*, нанесенного ошибкой, а не исправление самой ошибки. Как мы уже видели в гл. 3, исправление ошибки в аппаратуре (например, автоматическим переключением на запасное устройство) — вполне жизнеспособный прием, но пытаться исправить настоящую ошибку в программном обеспечении без участия человека бесполезно. Самое большее, что можно сделать в этом случае,— свести на нет ущерб, нанесенный ошибкой. Самое большее, что можно сделать по части устойчивости к ошибкам, — либо сделать нанесенный ущерб незаметным, либо изолировать его лишь в рамках части системы.

Хотя методы исправления/устойчивости и имели ограниченный успех в нескольких системах, в большинстве случаев их лучше избегать. Число возможных ошибок в большой системе так велико, что может считаться практически бесконечным. Разрабатывая методы исправления/устойчивости, мы вынуждены пытаться предугадать лишь несколько типов ошибок, чтобы реализовать средства, предназначенные для борьбы с ущербом от этих ошибок. В лучшем случае наша система будет исправлять ничтожный процент своих потенциальных ошибок. К тому же эти средства сами довольно сложны, так что благодаря им исходное количество ошибок в системе только возрастет. Более того, они сами будут, несомненно, содержать ошибки. Наконец, если некоторые средства исправления/устойчивости все-таки заработают, они тем самым станут маскировать ошибки (делая их менее заметными), и последние, возможно, никогда не будут устранены обслуживающим персоналом, а это — явно нежелательное следствие.

Однако самым сильным доводом против исправления ошибок и обеспечения устойчивости остается следующий аргумент. Поскольку все равно необходимо заранее предвидеть несколько возможных ошибок, обычно лучше при проектировании и тестировании направлять все усилия на их устранение.

Пример: система обслуживания телефонных станций TSPS фирмы «Белл»

Система TSPS фирмы «Белл» демонстрирует некоторые примеры методов исправления ошибок и обеспечения устойчивости [4]. Система предназначена для обслуживания телефонной сети, поэтому на нее накладываются строгие требования в отношении постоянной готовности: суммарное время простоя системы из-за отказов не должно превышать двух часов за 40 лет. В TSPS полный выход системы из строя — серьезное событие, в то время как такие случайные сбои, как разрыв телефонной связи, вполне допустимы. (Отметим, что во многих системах требования прямо противоположны. Например, в системе обработки банковских чеков выход системы из строя менее серьезен, чем ошибка в данных.)

В TSPS применяются и пассивные, и активные методы обнаружения ошибок и средства исправления ошибок для сокращения числа сбоев системы. Основной принцип состоит в обнаружении определенных типов ошибок по возможности раньше, поскольку в противном случае их воздействие быстро распространится по всей системе и система в конце концов выйдет из строя. При исправлении последствий этих ошибок утрата некоторых данных допустима, если такова цена за сохранение работоспособности системы.

Телефонная сеть представлена таблицами в основной памяти, которые хранятся как множество связанных линейных списков. Например, список на рис. 7.2 может представлять все соединения сети, находящиеся в активном состоянии. В этом списке наблюдается существенная избыточность. Например, каждый блок имеет свой тэг, последний блок определяется по двум признакам, и в поле состояния каждого блока в этом списке должно быть установлено значение «активно». Монитор (называемый в TSPS *audit program*) периодически просматривает списки. Если обнаруживается противоречие, например в поле СЛЕДУЮЩИЙ некоторого блока указан неправильный адрес (что обнаруживается по несоответствию тэга блока, находящегося по этому адресу), то список обрывается после последнего правильного блока и в каждом правильном блоке значение бита L (от английского слова lost — потеряно) устанавливается равным 1. Другая компонента монитора периодически просматривает последовательно блоки всех линий. Если бит L блока равен 0 (что значит, что он «потерян»), блок включается в список незанятых линий соответствующего типа. Эти действия могут при-

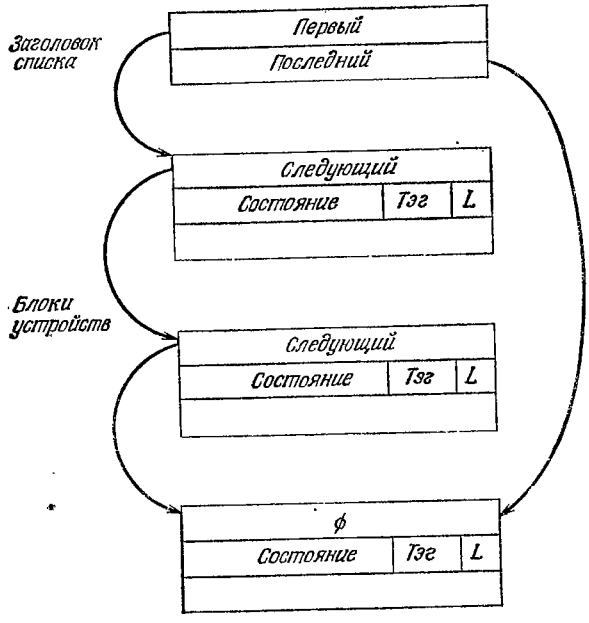


Рис. 7.2. Линейный список таблиц (ресурсов).

вести к разрыву телефонной связи, но сохраняют работоспособность всей системы.

Многие линии имеют также в своих блоках бит «тайм-аут». Монитор использует этот бит, чтобы определить, не была ли линия занята в течение необычно долгого периода времени. Если это так, то после дополнительных проверок линия может быть переведена в свободное состояние. Если монитор обнаруживает, что система находится в хаотическом состоянии, он имеет возможность запустить ее заново. Монитор также регистрирует все обнаруженные ошибочные ситуации для последующей отладки.

Механизм исправления ошибок в TSPS ежедневно исправляет довольно много ошибок. Установка TSPS после трех лет работы выполняет в среднем от 10 до 100 реальных исправлений в день и примерно раз в два месяца — автоматический повторный запуск.

ИЗОЛЯЦИЯ ОШИБОК

В большой вычислительной системе изоляция программ является ключевым фактором, гарантирующим, что отказы в программе одного пользователя не приведут к отказам в программах других пользователей или к полному выводу системы из строя. Основные правила изоляции ошибок перечислены ниже. Хотя в формулировке мно-

гих из них употребляются слова «операционная система», они применимы к любой программе (будь то операционная система, монитор телеобработки или подсистема управления файлами), которая занята обслуживанием других программ.

1. Прикладная программа не должна иметь возможности непосредственно ссылаться на другую прикладную программу или данные в другой программе и изменять их.

2. Прикладная программа не должна иметь возможности непосредственно ссылаться на программы или данные операционной системы и изменять их. Связь между двумя программами (или программой и операционной системой) может быть разрешена только при условии использования четко определенных сопряжений и только в случае, когда *обе* программы дают согласие на эту связь.

3. Прикладные программы и их данные должны быть защищены от операционной системы до такой степени, чтобы ошибки в операционной системе не могли привести к случайному изменению прикладных программ или их данных.

4. Операционная система должна защищать все прикладные программы и данные от случайного их изменения операторами системы или обслуживающим персоналом.

5. Прикладные программы не должны иметь возможности ни остановить систему, ни вынудить ее изменить другую прикладную программу или ее данные.

6. Когда прикладная программа обращается к операционной системе, должна проверяться допустимость всех параметров. Более того, прикладная программа не должна иметь возможности изменить эти параметры между моментами проверки и реального их использования операционной системой.

7. Никакие системные данные, непосредственно доступные прикладным программам, не должны влиять на функционирование операционной системы. Например, OS/360 хранит некоторые блоки, управляющие распределением памяти, в областях основной памяти, доступных прикладным программам. Ошибка в прикладной программе, вследствие которой содержимое этой памяти может быть случайно изменено, приводит в конце концов к сбою системы.

8. Прикладные программы не должны иметь возможности в обход операционной системы прямо использовать управляемые ею аппаратные ресурсы. Прикладные программы не должны прямо вызывать компоненты операционной системы, предназначенные для использования только ее подсистемами.

9. Компоненты операционной системы должны быть изолированы друг от друга так, чтобы ошибка в одной из них не привела к изменению других компонент или их данных.

10. Если операционная система обнаруживает ошибку в себе самой, она должна попытаться ограничить влияние этой ошибки од-

ной прикладной программой и в крайнем случае прекратить выполнение только этой программы.

11. Операционная система должна давать прикладным программам возможность по требованию исправлять обнаруженные в них ошибки, а не безоговорочно прекращать их выполнение.

Реализация многих из этих принципов влияет на архитектуру лежащего в основе системы аппаратного обеспечения. Некоторые аспекты этого влияния рассматриваются в гл. 16.

Читатель может заметить, что многие из перечисленных правил являются также правилами обеспечения защиты в операционных системах. Действительно, в большинстве своем это выдержки из работы Степчика [5], являющейся блестящим анализом требований к защите ресурсов в операционных системах и обзором методов ее обеспечения. Это подтверждает сформулированное в гл. 4 утверждение: цели обеспечения защиты ресурсов и надежности обычно согласуются. Это подтверждается также разработанной IBM экспериментальной Системой защиты ресурсов для OS/360. Благодаря более высокой степени изоляции в этой системе были обнаружены ранее не замеченные ошибки в системных компонентах (например, в компиляторах) и прикладных программах, а также значительно снижена частота случайных отказов системы из-за ошибок в прикладных программах.

ОБРАБОТКА СБОЕВ АППАРАТУРЫ

Улучшая общую надежность системы, следует заботиться не только об ошибках в программном обеспечении (хотя надежность программного обеспечения требует наибольшего внимания). Другая сторона, о которой необходимо подумать,— это ошибки во входных данных системы (ошибки пользователя). Обсуждавшиеся выше средства обнаружения ошибок и правила, рассмотренные в гл. 4 в связи с внешними требованиями, могут быть применены и к ошибкам пользователей.

Наконец, еще один интересующий нас класс ошибок — сбои аппаратуры. Во многих случаях они обрабатываются самой аппаратурой за счет использования кодов, исправляющих ошибки, исправления последствий сбоев (например, переключением на запасные компоненты) и средств, обеспечивающих устойчивость к ошибкам (например, голосование). Некоторые сбои, однако, нельзя обработать только аппаратными средствами, они требуют помощи со стороны программного обеспечения. Такие случаи не рассматриваются в этой книге, но для полноты ниже приводится список возможностей, которые часто бывают необходимы в программных системах для борьбы со сбоями аппаратуры.

1. *Повторное выполнение операций.* Многие сбои аппаратуры не постоянны (например, скачки напряжения, шум в телекоммуникационных линиях, колебания при механическом движении). Всегда имеет смысл попытаться выполнить операцию, искаженную сбоем (например, команду машины или операцию ввода-вывода), несколько раз, прежде чем принимать другие меры.

2. *Восстановление памяти.* Если обнаруженный случайный сбой аппаратуры вызывает искажение области основной памяти и эта область содержит статические данные (например, команды объектной программы), то последствия сбоя можно ликвидировать, повторно загрузив эту область памяти.

3. *Динамическое изменение конфигурации.* Если аппаратная подсистема, такая, как ЦП, канал ввода-вывода, блок основной памяти или устройство ввода-вывода, выходит из строя, работоспособность системы можно сохранить, динамически исключая неисправное устройство из набора ресурсов системы.

4. *Восстановление файлов.* Системы управления базами данных обычно обеспечивают избыточность данных, сохраняя копию текущего состояния базы данных на выделенных устройствах ввода-вывода, регистрируя все изменения базы данных или периодически автономно копируя всю базу данных. Поэтому программы восстановления могут воссоздать базу данных в случае катастрофического сбоя ввода-вывода.

5. *Контрольная точка/рестарт.* Контрольная точка — это периодически обновляемая копия состояния прикладной программы или всей системы. Если происходит отказ аппаратуры, такой, как ошибка ввода-вывода, сбой памяти или питания, программа может быть запущена повторно с последней контрольной точки.

6. *Предупреждение отказов питания.* Некоторые вычислительные системы, в особенности те, в которых используется энергозависимая память, предусматривают прерывание, предупреждающее программу о предстоящем отказе питания. Это дает возможность организовать контрольную точку или перенести жизненно важные данные во вторичную память.

7. *Регистрация ошибок.* Все сбои аппаратуры, с которыми удалось справиться, должны регистрироваться во внешнем файле, чтобы обслуживающий персонал мог получать сведения о постепенном износе устройств.

Тот, кто нуждается в дополнительной информации о программной обработке сбоев аппаратуры, может получить достаточное образование в этой области, изучая методы, применяемые в современных операционных системах, системах управления данными и программах телеобработки. Эти методы рассматриваются также в ряде учебников, часть из которых перечислена в работах [6—10].

ЛИТЕРАТУРА

1. Randell B. Operating Systems: The Problems of Performance and Reliability, Proceedings of the 1971 IFIP Congress. Amsterdam: North-Holland, 1971, pp. 1100—1109.
2. Fabry R. S. Dynamic Verification of Operating System Decisions *Communications of the ACM*, **16** (11), 659—668 (1973).
3. Pinchuk P. L. TRW Evaluation of a Secure Operating System, Data Security and Data Processing, Volume 6, Evaluation and Installation Experiences: Resource Security System, G320—1376, IBM Corp., White Plains, N. Y., 1974, pp. 39—121.
4. Connet J. R., Pasternak E. J., Wagner B. D. Software Defenses in Real-Time Control Systems, Digest of the 1972 International Symposium on Fault-Tolerant Computing. New York: IEEE, 1972, pp. 94—99.
5. Stepczyk F. M. Requirements for Secure Operating Systems, Data Security and Data Processing, Volume 5, Study Results: TRW Systems, Inc., G320—1375, IBM Corp., White Plains, N. Y., 1974, pp. 75—205.
6. Watson R. W. Timesharing System Design Concepts. New York: McGraw-Hill, 1970.
7. Мартин Дж. Системный анализ передачи данных. В 2-х томах, Перев. с англ.— М.: Мир, 1975.
8. Байцер В. Архитектура вычислительных комплексов, т. 2. Перев. с англ.— М.: Мир, 1974.
9. Yourdon E. Design of On-Line Computer Systems, Englewood Cliffs, N. J., Prentice-Hall, 1972.
10. *IBM Systems Journal*, 6(2), 1967 (специальный выпуск, посвященный авиационной системе диспетчеризации IBM 9020).

ГЛАВА 8

Проектирование и программирование модуля

Этапы проектирования и программирования каждого модуля — заключительные в общем цикле проектирования. На этих этапах выполняются процессы *внешнего проектирования модуля* (т. е. разработки сопряжений каждого модуля) и *проектирования логики модуля* (т. е. ряд шагов, включающих определение данных, выбор алгоритма, разработку логики и собственно программирование). Для многих эти процессы олицетворяют сущность программирования; однако после прочтения первых семи и последующих одиннадцати глав этой книги должно быть ясно, что эти два процесса — лишь малая часть полного цикла разработки программного обеспечения.

В этой главе рассматриваются принципы и методы проектирования и программирования модулей. Об отдельных проблемах, которые в соответствии с традицией также могли бы быть включены в нее, говорится в других главах. Например, разговор о стиле программирования, вследствие его важности, составил отдельную главу (гл. 9). Организационные аспекты, такие, как работа коллектива программистов и программиста-библиотекаря, обсуждаются в главе, посвященной методам управления (гл. 14). Такие вопросы, как нисходящая и восходящая разработка программ, рассматриваются в гл. 10 вследствие их тесной связи с проблемами тестирования.

ВНЕШНЕЕ ПРОЕКТИРОВАНИЕ МОДУЛЯ

Первый шаг при проектировании модуля состоит в определении его внешних характеристик. Эта информация выражается в виде *внешних спецификаций модуля*, которые содержат все сведения, необходимые вызывающим его модулям, *и ничего больше*. В частности, внешние спецификации модуля не должны содержать никакой информации о логике модуля или о внутреннем представлении данных. Кроме того, спецификации не должны включать каких бы то ни было ссылок на вызывающие модули или на контексты, в которых этот модуль используется.

Внешние спецификации модуля должны содержать сведения следующих шести типов:

1. *Имя модуля.* Указывается имя, применяемое для вызова модуля. Для модуля с несколькими входами это имя определенного входа (для каждого входа имеются отдельные спецификации).

2. *Функция.* Дается определение функции или функций, выполняемых модулем. Этот раздел *не* должен описывать логику или контексты, в которых модуль применяется (см. определение функции в гл. 6).

3. *Список параметров.* Определяется число и порядок параметров, передаваемых модулю.

4. *Входные параметры.* Дается точное описание всех входных параметров. Сюда включается определение формата, размеров, атрибутов, единиц измерения (например, морские мили) и допустимых диапазонов значений всех входных параметров.

5. *Выходные параметры.* Дается точное описание всех данных, возвращаемых модулем. Сюда должно входить определение формата, размеров, атрибутов, единиц измерения и допустимых диапазонов значений всех выходных данных. Должна быть описана функциональная связь между входными и выходными данными, т. е. следует показать, какие выходные данные какими входными порождаются. Должны быть также определены выходные данные, порождаемые модулями в случае, когда входные данные не годятся. Для того чтобы можно было считать модуль *специфицированным полностью*, должно быть определено его поведение при любых входных условиях.

6. *Внешние эффекты.* Дается описание всех внешних для программы или системы событий, происходящих при работе модуля. Примерами внешних эффектов являются печать сообщения, чтение запроса с терминала, чтение из файла заказов, вывод сообщения об ошибке. Внешние эффекты модуля включают все внешние эффекты подчиненных ему модулей. Например, если модуль А вызывает модуль В и В печатает сообщение, этот внешний эффект должен включаться во внешние спецификации как модуля А, так и модуля В.

Важно отделить внешние спецификации модуля от другой документации (например, описания его логики), потому что изменение логики может никак не повлиять на вызывающие модули, а изменение внешних спецификаций обычно требует изменить вызывающие модули.

Внешние спецификации модуля физически могут принимать разнообразные формы, лишь бы они включали ответы на перечисленные выше шесть вопросов. Лучше всего поместить спецификации в виде комментария в начале текста исходной программы модуля (или, в случае модулей с несколькими входами, у каждого из них). На рис. 8.1 показаны спецификации одного из входов модуля ESTAB-MGR (УПРТВИМ), спроектированного в гл. 6 загрузчика. Отме-

```
/*************************************************************/
/*                                                           */
/* EXTERNAL SPECIFICATION FOR MATCHES                        */
/*                                                           */
/* FUNCTION: MATCH ALL EXTERNAL REFERENCE ITEMS IN ESTAB.    */
/*           IF ONE IS ENCOUNTERED THAT CANNOT BE MATCHED,   */
/*           RETURN ITS NAME.                                */
/*                                                           */
/* PARAMETER LIST: CALL MATCHES (ESTAB,UNRESNAME,MATCHCODE)  */
/*                                                           */
/* INPUT:  ESTAB      POINTER    POINTER TO ESTABLE          */
/*                                                           */
/* OUTPUT: ESTAB      POINTER    POINTER TO UPDATED ESTABLE  */
/*         UNRESNAME CHAR(8)     NAME OF AN UNMATCHED        */
/*                               EXTERNAL REFERENCE          */
/*         MATCHCODE FIXED BIN(15)  RETURN CODE (0, 1, OR 2)*/
/*                                                           */
/*     CAUSE/EFFECT RELATIONSHIP:                            */
/*                                                           */
/*          ANY UNMATCHED (NULL ADDRESS FIELD) EXTERNAL      */
/*          REFERENCES ARE MATCHED WITH A MATCHING MD OR     */
/*          EP ITEM BY TRANSFERRING THE ADDRESS OF THE MD OR*/
/*          EP ENTRY TO THE ER ENTRY UNTIL AN ER CANNOT BE   */
/*          MATCHED.                                         */
/*         -IF ALL ER'S CAN BE MATCHED, MATCHCODE=0 AND      */
/*             UNRESNAME IS UNCHANGED.                       */
/*         -IF AN ER CANNOT BE MATCHED, MATCHCODE=1 AND      */
/*             UNRESNAME = NAME OF ER ITEM                   */
/*         -IF AN ERROR OCCURS (INVALID ESTABLE), MATCH-     */
/*             CODE=2 AND UNRESNAME IS UNCHANGED             */
/*                                                           */
/* EXTERNAL EFFECTS: NONE                                    */
/*************************************************************/
```

Рис. 8.1. Внешние спецификации модуля.

тим, что формат ESTAB (таблицы внешних имен ТВИМ) не описан в этих спецификациях даже несмотря на то, что ESTAB является элементом и входных, и выходных данных модуля. Но такова была исходная цель проектирования этого модуля, поставленная в гл. 6: спрятать в нем все сведения о структуре ТВИМ. Поэтому вызывающий модуль лишь передает ему переменную-указатель, называемую ESTAB, а формат таблицы не считается частью сопряжения этого модуля.

ПРОЕКТИРОВАНИЕ ЛОГИКИ МОДУЛЯ

Последним в длинной цепи процессов проектирования программного обеспечения является процесс проектирования и собственно программирования (кодирования) внутренней логики каждого модуля. Очень часто идея тщательного планирования здесь отбрасывается, и программист разрабатывает модуль более или менее хаотично. Однако процесс разработки модуля может и должен тщатель-

но планироваться. Следующие 11 шагов составляют набросок дисциплинированного подхода к проектированию модуля.

1. *Выберите язык.* Выбор языка обычно диктуется требованиями контракта или принятыми в организации стандартами. Хотя выбор языка и включен сюда, на самом деле язык должен быть выбран в начальный период работы над проектом, поскольку он влияет на планирование работы над проектом (например, обучение программистов, подготовка компиляторов и средств тестирования).

2. *Спроектируйте внешние спецификации модуля.* Это процесс определения внешних характеристик каждого модуля, о котором шла речь в предыдущем разделе.

3. *Проверьте правильность внешних спецификаций.* Правильность спецификаций каждого модуля должна быть проверена сравнением их с информацией о сопряжениях, полученной при проектировании структуры программы, и анализом их всеми программистами, разрабатывающими вызывающие модули.

4. *Выберите алгоритм и структуры данных.* Жизненно важным шагом в процессе проектирования логики является выбор алгоритма и соответствующих структур данных. Сегодня лишь немногие алгоритмы создаются впервые; огромное их число уже было изобретено, и весьма вероятно, что уже имеется один или несколько алгоритмов, вполне устраивающих проектировщика. Вместо того чтобы тратить время, заново изобретая алгоритмы и структуры данных, лучше поискать готовые решения. Если речь идет о нечисленных алгоритмах (т. е. о большинстве видов обработки данных), лучше всего начать с книги Д. Кнута о фундаментальных алгоритмах [1] и последующих томов этой серии. В случае численных алгоритмов начните с издаваемых ACM «Избранных алгоритмов из CACM» (Collected Algorithms from CACM). Другим источником алгоритмов обоих типов являются учебники, технические статьи и существующие программы.

Обычно проектировщик обнаруживает несколько функционально эквивалентных алгоритмов и структур данных и ему приходится выбирать один из них. Поскольку многие современные вычислительные системы имеют многоуровневую память (обычно это основная память, виртуальная память, быстрая буферная память), то основная тенденция у программистов, стремящихся к истинной эффективности,— назад, к простейшим алгоритмам и структурам данных (например, в системе с многоуровневой памятью двоичный поиск может оказаться не намного быстрее, чем более простой последовательный). Это пример того, как эффективность и простота становятся не противоречивыми, а согласованными требованиями!

5. *Напишите первое и последнее предложения.* Следующий шаг — написать предложения PROCEDURE и END будущего модуля (или их эквиваленты, в зависимости от избранного языка программирова-

ния). Если модуль имеет несколько входов, сразу же пишутся и предложения ENTRY. Отметим, что мы здесь опустили традиционный этап вычерчивания блок-схем; причины этого будут рассмотрены ниже.

6. *Объявите все данные из сопряжения.* Следующий шаг состоит в написании тех предложений программы, которые определяют или объявляют все переменные для сопряжения создаваемого модуля.

7. *Объявите остальные данные.* Напишите предложения, которые определяют или объявляют все другие необходимые переменные. Поскольку трудно предсказать все переменные, которые понадобятся, этот шаг часто перекрывается со следующим.

8. *Детализируйте текст программы.* Следующий шаг — итеративный, он предполагает последовательную детализацию логики модуля, начиная с достаточно высокого уровня абстракции и заканчивая готовым текстом программы. На этом шаге используются методы *пошаговой детализации* и *структурного программирования*, о которых говорится в следующем разделе.

9. *Отшлифуйте текст программы.* Теперь модуль нужно отшлифовать для достижения «ясности» и снабдить его дополнительными комментариями, отвечающими на вопросы, которые могут возникнуть при чтении программы. Особенности стиля программирования, которые в максимальной степени облегчают восприятие программы, рассматриваются в гл. 9.

10. *Проверьте правильность программы.* Вручную проверяется правильность модуля. Соответствующие процедуры описаны в последнем разделе этой главы.

11. *Компилируйте модуль.* Последний шаг — компиляция модуля. Этот шаг отмечает переход от проектирования к тестированию; компиляцией, по существу, начинается тестирование программного обеспечения.

Чтобы лучше проиллюстрировать перечисленные пункты, применим их к разработке модуля УПРТВИМ (ESTABMGR). На шаге 1 предполагается, что выбран язык PL/1. Предполагается также, что шаги 2 и 3 выполнены (внешние спецификации одного из входов показаны на рис. 8.1).

Самый важный вопрос при выполнении шага 4 для УПРТВИМ состоит в выяснении структуры ТВИМ (ESTAB) — таблицы, содержащей внешние имена, их типы и абсолютные адреса. Здесь возможны последовательное или связанное размещение, а также организация таблицы перемешивания. Поскольку нет необходимости включать элементы в середину ТВИМ или вычеркивать их, размещение в виде связанного списка не обязательно. Так как выборка будет как последовательной, так и прямой, организация таблицы перемешивания также нежелательна. Таким образом, остается лишь

```
/* MODULE ESTABMGR (EXTERNAL SYMBOL TABLE MANAGER)                    */
/*                                                                    */
/* THIS IS AN INFORMATIONAL STRENGTH MODULE WITH 3 ENTRY              */
/* POINTS:                                                            */
/*                                                                    */
/* MATCHES  (MATCH EXTERNAL REFERENCES IN ESTAB)                      */
/* ADDTOES  (ADD AN EXTERNAL SYMBOL TO ESTAB)                         */
/* FINDES   (FIND AN EXTERNAL SYMBOL IN ESTAB)                        */
/*                                                                    */
/* THE DESIGN OBJECTIVE OF THIS MODULE IS TO HIDE ALL KNOW-           */
/* LEDGE OF ESTAB.  ALTHOUGH ESTAB IS AN INPUT AND OUTPUT             */
/* OF THIS MODULE, THE INTENT IS THAT NO OTHER MODULES HAVE           */
/* ANY KNOWLEDGE OF THE STRUCTURE OF ESTAB.  HENCE THIS               */
/* IS AT WORST DATA COUPLED TO ANY OTHER MODULE.                      */
/*                                                                    */
/* STRUCTURE NOTES: EACH OF THE 3 FUNCTIONS ARE ENCLOSED IN           */
/*                  BEGIN BLOCKS TO ISOLATE NAMES.  MODULE-           */
/*                  WIDE DATA DEFINITIONS (ESTAB) ARE PLACED          */
/*                  OUTSIDE OF THE BEGIN BLOCKS.  ONE OF THE          */
/*                  FUNCTIONS (MATCHES) IS REPRESENTED BY             */
/*                  THE PROCEDURE STATEMENT.  THE OTHER TWO           */
/*                  FUNCTIONS ARE REPRESENTED BY ENTRY                */
/*                  STATEMENTS.                                       */
/**********************************************************************/
MATCHES: PROCEDURE (ESTAB,UNRESNAME,MATCHCODE);
     module-wide data declarations and
     parameter declarations
MATCH_ER_ITEMS_IN_ESTAB: BEGIN;
     local declares for MATCHES function
     code for MATCHES function
END;

ADDTOES: ENTRY (ESTAB,ESNAME,ESTYPE,ESADDR,ENTRYNUM,ADDCODE);
ADD_EXTSYM_TO_ESTAB: BEGIN;
     local declares for ADDTOES function
     code for ADDTOES function
END;

FINDES: ENTRY (ESTAB,ENTRYNUM,ESNAME,ESTYPE,ESADDR,FINDCODE);
FIND_EXTSYM_IN_ESTAB: BEGIN;
     local declares for FINDES function
     code for FINDES function
END;
END;
```

Рис. 8.2. Структура модуля УПРТВИМ.

вариант последовательной организации таблицы. Поэтому ТВИМ будет представлена одномерным массивом (вектором), каждый элемент которого имеет три поля: имя, тип и адрес. Остается вопрос о том, следует ли упорядочивать (сортировать) элементы. Поскольку таблица обновляется часто, решим оставить ее неупорядоченной и использовать простой последовательный поиск. (Отметим, однако, что я сделал несколько предположений, требующих достаточной квалификации. Эти решения позже можно пересмотреть, если понадобится подрегулировать эффективность. Ведь наш

```
/*      MODULE-WIDE DATA DEFINITIONS                                  */
/*                                                                    */
DECLARE ESTAB  POINTER;
%DECLARE TABSIZE FIXED;
%TABSIZE=2000;          /* NO. OF POSSIBLE ESTAB ENTRIES             */
DECLARE 1 TABLE BASED(ESTAB),
          2 HEADER,
            3 TAG  CHAR(4),              /* DOG TAG ESTB */
            3 SIZE BINARY FIXED (15),    /* NO. OF CURRENT ENTRIES */
          2 BODY (TABSIZE),              /* ARRAY OF ENTRIES */
            3 NAME CHAR(8),
            3 TYPE CHAR(2),
            3 ADDRESS POINTER;
DECLARE                                  /* TYPE VALUES */
       MODULE  CHAR(2) STATIC INIT ('MD'),
       ENTRYPT CHAR(2) STATIC INIT ('EP'),
       EXTREF  CHAR(2) STATIC INIT ('ER');
DECLARE NULL BUILTIN;  /*FUNCTION RETURNS VALUE OF EMPTY POINTER*/
DECLARE MATCHCODE FIXED BINARY (15); /*MATCHES RETURN CODE*/
DECLARE UNRESNAME CHAR(8);  /* OUTPUT FROM MATCHES FUNCTION */
```

Рис. 8.3. Объявления данных.

проект загрузчика позволяет изменить структуру ТВИМ, меняя лишь один модуль.)

На шаге 5 требуется определить структуру УПРТВИМ в виде модуля с тремя входами. Пусть у нас предложение PROCEDURE представляет функцию СОГЛАСОВАТЬ-ВНЕШН-ССЫЛКИ (MATCHES), а два предложения ENTRY — функции НАЙТИ-ЭЛЕМ-ТВИМ (FINDES) и ДОБАВИТЬ-ВНИМ (ADDTOES). Используем блоки BEGIN языка PL/1, чтобы таким обрамлением текста каждой из функций изолировать имена переменных. Структура модуля показана на рис. 8.2.

На шаге 6 необходимо объявить все существенные для сопряжения данные. Как упоминалось в предыдущем разделе, параметр ТВИМ (ESTAB) является просто указателем, ссылающимся на таблицу. Поскольку таблица представляет собой структуру, доступную во всем модуле, объявим ее сейчас. Вместо того чтобы строить здесь весь модуль, напишем только его треть: функцию MATCHES. На рис. 8.3 показано определение таблицы внешних имен, параметров MATCHES и встроенной функции NULL, которая позже понадобится нам для проверки пустых (незамкнутых) адресных полей.

Шаг 7 предполагает определение всех локальных переменных функции MATCHES. Понятно, что нам понадобятся два счетчика для поиска по ТВИМ, поэтому добавим в блок MATCH_ER_ITEMS_IN_ESTAB следующие объявления:

```
DECLARE
        I FIXED BINARY (15);/*ИНДЕКС ДЛЯ ПОИСКА НЕ-
                              ЗАМКНУТОГО ИМЕНИ */
        J FIXED BINARY (15);/* ИНДЕКС ДЛЯ ПОИСКА
                              ЗАМЫКАЮЩЕГО ИМЕНИ */
```

Вместо того чтобы выполнить оставшиеся четыре шага сейчас, отложим их до тех пор, пока не обсудим некоторые дополнительные вопросы.

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ И ПОШАГОВАЯ ДЕТАЛИЗАЦИЯ

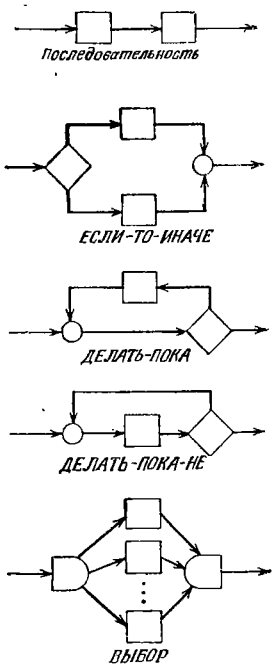
Концепция, называемая структурным програмированием, оказала настолько значительное влияние на разработку программного обеспечения, что она, вероятно, войдет в историю как одно из крупнейших достижений в технологии программирования (наряду с концепциями подпрограммы и языка высокого уровня). Однако, хотя самое общее и довольно смутное представление о структурном программировании имеется почти у всех, общепринятого четкого его определения нет. Вот почему здесь не дается ни строгого определения структурного программирования, ни всестороннего анализа сути дела: на эту тему написаны уже целые книги [2, 3]. Вместо этого рассматриваются основные свойства структурных программ, а также некоторые интересные моменты, которые часто остаются без внимания.

Хотя элементы структурного программирования медленно развивались с начала 60-х годов, как концепция оно впервые привлеко всеобщее внимание в 1968 г. после опубликования знаменитой заметки Э. Дейкстры [4]. Дейкстра указал, что между текстом программы и порядком выполнения ее элементов (потоком управления) должно быть простое соответствие (программа должна быть такой, чтобы ее можно было читать «сверху вниз») и что неограниченное использование операторов перехода (GO TO) нарушает такое соответствие. Вследствие этого структурное программирование часто называют «программированием без GO TO». Однако всегда были примеры программ, которые не содержат GO TO и аккуратно расположены лесенкой в соответствии с уровнем вложенности операторов, но совершенно непонятны, и были другие программы, содержащие GO TO и все же совершенно понятные. Таким образом, наличие или отсутствие GO TO — плохой показатель качества программы, что иллюстрируют многие примеры Д. Кнута [5].

Я предпочитаю определять структурное программирование *как программирование, ориентированное на общение с людьми, а не с машиной*. Чтобы соответствовать этому определению, структурная программа должна удовлетворять следующим основным требованиям:

1. Текст программы представляет собой композицию трех основных элементов: последовательное соединение (следование), условное предложение (развилка) и повторение (цикл).
2. Употребления GO TO избегают всюду, где это возможно.

Рис. 8.4. Пять конструкций структурного программирования.



Наихудшим применением GO TO считается переход на оператор, расположенный выше (раньше) в тексте программы.

3. Программа написана в приемлемом стиле (см. гл. 9).

4. Текст программы напечатан с правильными сдвигами, так что разрывы в последовательности выполнения легко прослеживаются (например, для предложения DO легко найти предложение, заканчивающее группу, без труда устанавливается соответствие между конструкциями THEN и ELSE и т. д.).

5. Каждый модуль имеет ровно один вход и один выход. Отметим, что информационно прочный модуль не нарушает этого требования, поскольку тексты программ для каждого входа физически и логически разделены.

6. Текст программы физически разбит на части, чтобы облегчить чтение. Выполняемые предложения каждого модуля должны умещаться на одной странице печатающего устройства.

7. Программа представляет собой простое и ясное решение задачи.

Эти семь требований четко выражают цели структурного программирования: писать программы минимальной сложности, заставить программиста мыслить ясно, облегчать восприятие программы.

Конструкции структурного программирования

Структурные программы составлены из пяти основных строительных блоков, изображенных на рис. 8.4. Часто говорят о трех строительных блоках, потому что в большинстве языков программирования имеется только один из двух видов циклов и нет конструкции ВЫБОР (CASE).

Относительно этих трех строительных блоков необходимо прежде всего понять, что они определены рекурсивно. Последнее означает, что прямоугольники на рис. 8.4 изображают вложенные стро-

```
      IF (I=2)                GO TO 4
                              GO TO 5
C
  4     THEN
            A=3
            B=2
                              GO TO 10
C
  5     ELSE
            A=2
            B=3
 10   CONTINUE
```

Рис. 8.5. Структурная программа на Фортране.

ительные блоки; каждый из блоков может использоваться вместо прямоугольника. Например, следование может содержать развилку, за которой идет цикл ПОКА, а цикл ПОКА может содержать другой цикл ПОКА. Прямоугольник может представлять также любой отдельный «последовательный» оператор (например, оператор присваивания). Некоторые склонны считать, что использование операторов CALL и RETURN запрещено. Это мнение ошибочно: CALL и RETURN рассматриваются как «последовательные» операторы.

В какой степени может быть достигнута структурность программы, конечно, зависит от используемого языка программирования. Сейчас в стадии разработки находится значительное количество проектов «наилучшего» языка для структурного программирования. Из распространенных языков самыми подходящими считаются PL/1, PL/C, Паскаль и Алгол. Структурное программирование возможно также и при использовании языков Кобол, Фортран и Бэйсик, хотя с бо́льшими трудностями и некоторыми нежелательными последствиями. В Коболе при вложении развилок друг в друга возникает структура, отличная от изображенной на рис. 8.4. Только последний оператор в предложении на Коболе может быть развилкой, откуда заключаем, что время от времени будут необходимы операторы GO TO или PERFORM. Оператор PERFORM в Коболе можно применять в качестве ПОКА. Однако этот оператор действует как вызов подпрограммы, поэтому текст тела цикла должен быть физически отделен от оператора PERFORM. Из-за этих двух трудностей читать структурную программу сверху вниз становится тяжело.

Структурное программирование на Фортране или Бэйсике еще сложнее, поскольку в этих языках отсутствуют последние четыре из показанных на рис. 8.4 конструкций, что вынуждает программиста использовать оператор GO TO. Однако Чармонмен и Ведженер [6] показывают, как можно сделать программу на Фортране похожей на структурную, если моделировать конструкции ВЫБОР

и развилку, сочетая расположение операторов лесенкой с использованием комментариев. На рис. 8.5 показана реализация развилки на Фортране. Текст в рамке выглядит как настоящая развилка, а текст вне ее дополняет картину деталями, необходимыми для того, чтобы программа оставалась правильной программой на Фортране. Структурности программы на Фортране (или Коболе) можно добиться также и по-другому, используя один из многих существующих препроцессоров. Препроцессор расширяет язык, дополняя его стандартными конструкциями структурного программирования.

Структурное программирование на двух других очень распространенных языках — языке ассемблера и АПЛ — почти невозможно. Хотя и были придуманы макросы, чтобы моделировать основные конструкции структурного программирования, большинство программ на ассемблере, использующих эти макросы, никак нельзя назвать структурными. Сам факт использования языка ассемблера указывает на то, что программа написана в основном в терминах машинного языка, а не для человека, что противоречит фундаментальному свойству структурных программ. АПЛ — язык, отличный во многих других отношениях — несовместим со структурным программированием: в нем отсутствуют основные конструкции структурного программирования, запрещено смещение операторов по строке, язык поощряет написание запутанных программ.

Читателю, интересующемуся более тонкими связями между языком и структурным программированием, рекомендуем обратиться к гл. 15.

Чего следует избегать

Поскольку концепции структурного программирования все еще развиваются, некоторые из предложенных ранее идей, как оказалось, отрицательно влияют на надежность, и их следует избегать. Одна из них — так называемый *метод Эшкрофта — Манны*. Он назван так по имени авторов статьи, описывающей метод преобразования произвольной программы в программу без GO TO [7]. Хотя метод представляет теоретический интерес, его никогда не следует использовать на практике [1]. Принципиально важно, чтобы программист, еще только приступая к проектированию логики, мыслил в терминах основных структурных конструкций, а не создавал сначала неструктурные программы, а затем преобразовывал их в структурные.

Позиция в отношении оператора GO TO должна быть следующей: избегать использования GO TO всюду, где это возможно, но

---

[1] Авторы работы [7] не предлагали практического метода получения структурных программ. Они лишь показали, что для произвольной программы существует эквивалентная ей программа без GO TO.— *Прим. ред.*

не ценой ясности программы. Как указывает Кнут [5], в определенных случаях GO TO желательнее других вариантов.

Часто оказывается желательным использование GO TO для выхода из цикла или модуля, для перехода на конец модуля в конструкции ON языка PL/1 или во избежание слишком большой глубины вложенности развилок, тем более что переход осуществляется на *последующие* (расположенные ниже) операторы программы и структурная программа продолжает оставаться легко читаемой сверху вниз.

Правила структурного программирования часто предписывают повторять одинаковые фрагменты программы в разных участках модуля, чтобы избавиться от употребления операторов GO TO. В этом случае лекарство хуже болезни; дублирование резко увеличивает возможность внесения ошибок при изменении модуля в будущем.

Пошаговая детализация

Структурное программирование до сих пор было у нас представлено как свойство или оценка окончательного текста программы. Необходимо добавить еще один ключевой элемент: методологию, или особенности мыслительного процесса, управляющего проектированием модуля для получения структурной программы. Этот мыслительный процесс, который мы будем сейчас рассматривать, называется *пошаговой детализацией* и был первоначально предложен Дейкстрой [8], а затем улучшен Виртом [9, 10].

Пошаговая детализация представляет собой простой процесс, предполагающий первоначальное выражение логики модуля в терминах гипотетического языка «очень высокого уровня» с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор пока наконец не будет достигнут уровень используемого языка программирования. На протяжении всего процесса логика выражается основными конструкциями структурного программирования.

Для иллюстрации этого процесса применим его к разработке логики функции MATCHES. Исходная формулировка такова:

```
DO I=1 TO SIZE WHILE (не встретится элемент, который не
                      удается замкнуть);
   IF   (встретился незамкнутый элемент)
    THEN искать соответствующее внешнее имя;
          IF (найдено)
            THEN пометить незамкнутый элемент;
            ELSE вывести это имя;
   ELSE;
END;
```

Очевидно, эта первоначальная формулировка выражена на языке, уровень которого существенно выше уровня выбранного нами языка программирования (PL/1). Следующий шаг состоит в ее детализации. Мы можем использовать входной параметр MATCHCODE для изображения состояний «в настоящий момент нет незамкнутых элементов» (MATCHCODE = 0) и «в настоящий момент имеем незамкнутый элемент» (MATCHCODE = 1), поскольку это соответствует возвращаемым значениям MATCHCODE. Теперь детализируем текст таким образом:

```
MATCHCODE=0;
DO   I=1 TO SIZE WHILE (MATCHCODE=0);
     IF (встретился незамкнутый элемент)
        THEN DO;
                  MATCHCODE=1;
                  искать соответствующее внешнее имя;
                  IF (найдено)
                    THEN MATCHCODE=0;
                           пометить незамкнутый элемент;
                    ELSE вывести это имя;
             END;
        ELSE;
END;
```

В качестве третьего шага расшифруем предложение «встретился незамкнутый элемент» так: BODY (I). ADDRESS = NULL (в PL/1 встроенная функция NULL вырабатывает неопределенное значение указателя). Следующий шаг состоит в детализации внутреннего цикла поиска соответствующего внешнего имени, а затем в его отметке или возврате незамкнутого имени. Это будет выглядеть так:

```
DO J=1 TO SIZE WHILE(MATCHCODE=1);
  IF (найдено)
    THEN DO;
           MATCHCODE=0;
           BODY(I).ADDRESS=BODY(J).ADDRESS;
         END;
    ELSE;
END;
IF (MATCHCODE=1) THEN UNRESNAME=BODY(I).NAME;
                 ELSE;
```

Пятый, и последний, шаг состоит в детализации условия «найдено». Это условие выполняется, если данное имя и его тип (модуль или вход) совпадают с именем и типом элемента таблицы. Эта окончательная детализация изображена на рис. 8.6.

```
MATCHCODE=0;
DO I=1 TO SIZE WHILE (MATCHCODE=0);
  IF(BODY(I).ADDRESS=NULL)      /*UNMATCHED NAME?*/
    THEN DO;
            MATCHCODE=1;
            DO J=1 TO SIZE WHILE (MATCHCODE=1);
              IF((BODY(I).NAME=BODY(J).NAME)&
                  ((BODY(J).TYPE=MODULE)|
                   (BODY(J).TYPE=ENTRYPT)))
                THEN DO;
                       MATCHCODE=0;
                       BODY(I).ADDRESS=BODY(J).ADDRESS;
                     END;
            ELSE;
            END;
            IF(MATCHCODE=1) THEN UNRESNAME=BODY(I).NAME;
                            ELSE;
          END;
    ELSE;
END;
```

Рис. 8.6. Окончательная детализация.

Текст программы, реализующей функцию MATCHES, не вполне закончен; это будет сделано в следующем разделе. Кроме того, заботящийся об эффективности читатель, вероятно, уже отметил несколько возможных улучшений. Однако для того, чтобы не усложнять пример, мы не будем заниматься повышением эффективности.

Достоинство пошаговой детализации состоит в том, что она позволяет проектировщику упорядочить свои рассуждения. Альтернативный подход — попытка написать окончательный текст (или блок-схему) модуля за один шаг — гораздо более сложен и чреват ошибками. Шаги детализации функции MATCHES — не академические или теоретические рассуждения, а в точности тот метод, которым разрабатывалась данная программа. Другими словами, мною написано пять различных версий MATCHES, первая из которых является абстрактным представлением логики, а последняя — программой на PL/1, изображенной на рис. 8.6.

Одна из связанных с описанным методом проблем состоит в том, что все варианты, кроме заключительного, по-видимому, тотчас же выбрасываются. Это, похоже, расточительство, поскольку предыдущие шаги фиксировали различные уровни абстракции логики и ход мысли разработчика. Чтобы как-то сохранить информацию об этой работе, исходный вариант следует поместить в качестве комментария перед текстом программы на языке программирования.

Стоит ли избегать и ELSE?

Если GO TO не использовать совсем, многие структурные программы становятся похожими на лабиринт глубоко вложенных друг в друга развилок. Такие программы зачастую столь же сложны, как и их неструктурные собратья. Пытаясь понять предложение внутри глубоко вложенных развилок, часто очень трудно определить точные условия, при которых это предложение выполняется. Эти условия могут быть разбросаны по множеству расположенных ранее IF, которые находятся довольно далеко от рассматриваемого оператора.

Блум [11] приводит основательные аргументы в пользу отказа от вложенных развилок. Легко показать, что конструкция «ELSE» обычно не является необходимой, поскольку ELSE эквивалентно IF (¬условие) THEN. Конструкция ELSE необходима только в той редкой ситуации, когда конструкция THEN изменяет одну из переменных в условии. Например, фрагмент программы

```
IF (усл1)  THEN DO;
                 опер1;
                 IF(усл2) THEN опер2;
                          ELSE DO;
                                IF(усл3) THEN опер3;
                                         ELSE опер4;
                                END;
                 END;
           ELSE опер5;
```

можно было переписать так:

```
IF(усл1) THEN опер1;
IF(усл1 & усл2) THEN опер2;
IF(усл1 &¬усл2 & усл3) THEN опер3;
IF(усл1 &¬усл2 &¬усл3) THEN опер4;
IF(¬усл1) THEN опер5);
```

По общему признанию этот подход страдает двумя недостатками из-за многократных проверок условий (например, *усл1* проверяется в шести операторах). Во-первых, если условие необходимо изменить, то должны быть изменены все его вхождения (правда, их в этом случае по крайней мере легче обнаружить). Во-вторых, многократные проверки условия несколько снижают эффективность, но в большинстве случаев это, вероятно, несущественно.

ЗАЩИТНОЕ ПРОГРАММИРОВАНИЕ

Концепции обнаружения ошибок, обсуждавшиеся в гл. 7, можно применять также на уровне отдельных модулей; такое их применение часто называют *защитным программированием*. Защитное прог-

раммирование напоминает поведение осторожного водителя, состоящее в том, чтобы с определенным недоверием относиться к действиям других водителей (или, в программистских терминах, с недоверием относиться к действиям других модулей).

Защитное программирование основано на важной предпосылке: худшее, что может сделать модуль,— это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат. Чтобы разрешить эту проблему, в начале каждого модуля помещаются проверки входных данных на соответствие их свойств атрибутам и диапазонам изменения, на полноту и осмысленность. При выборе надлежащих проверок важно по тексту программы модуля выявить все предположения, которые в нем сделанны относительно входных данных, а затем рассмотреть возможность проверки соответствия входных данных этим предположениям всякий раз, когда модуль вызывается.

Защитное программирование требует разумного подхода, ибо, доведенное до крайности, оно повлечет нежелательные эффекты. Если над входными данными выполнять все мыслимые проверки, защищающая часть программы может стать настолько сложной (и потому чреватой ошибками), что ее влияние на надежность (а также на эффективность) будет не позитивным, а негативным. Чтобы решить, сколько защитных проверок оправдано, сначала изучите по логике модуля все предположения о входных данных, которые в нем сделаны, и составьте список всех проверок, которые можно было бы сделать. Для каждой из них оцените ее сложность, вероятность того, что входные данные могут быть ошибочными, и последствия отсутствия проверки. После этого остается принять трудное компромиссное решение по определению того минимума защитной части программы, который обеспечивает максимально возможный уровень обнаружения ошибок.

Для иллюстрации рассмотрим функцию MATCHES. Единственным входным параметром служит ESTAB — указатель на таблицу внешних имен. Вот список возможных проверок ошибочных ситуаций (я не утверждаю, что обнаружил их все):

1. ESTAB — не является указателем.
2. Значение указателя ESTAB не определено.
3. ESTAB — допустимый указатель, но он ссылается не на таблицу внешних имен.
4. Поле SIZE (число различных элементов) таблицы неправильно.
5. Элемент таблицы имеет неправильный тип.
6. Элемент таблицы (модуль или вход) имеет нулевой адрес.
7. Элемент таблицы (модуль или вход) имеет неправильный адрес.

```
/* MODULE ESTABMGR (EXTERNAL SYMBOL TABLE MANAGER)                    */
/*                                                                    */
/* THIS IS AN INFORMATIONAL STRENGTH MODULE WITH 3 ENTRY              */
/* POINTS:                                                            */
/*                                                                    */
/* MATCHES (MATCH EXTERNAL REFERENCES IN ESTAB)                       */
/* ADDTOES (ADD AN EXTERNAL SYMBOL TO ESTAB)                          */
/* FINDES  (FIND AN EXTERNAL SYMBOL IN ESTAB)                         */
/*                                                                    */
/* THE DESIGN OBJECTIVE OF THIS MODULE IS TO HIDE ALL KNOW-           */
/* LEDGE OF ESTAB.  ALTHOUGH ESTAB IS AN INPUT AND OUTPUT             */
/* OF THIS MODULE, THE INTENT IS THAT NO OTHER MODULES HAVE           */
/* ANY KNOWLEDGE OF THE STRUCTURE OF ESTAB.  HENCE THIS               */
/* IS AT WORST DATA COUPLED TO ANY OTHER MODULE.                      */
/*                                                                    */
/* STRUCTURE NOTES: EACH OF THE 3 FUNCTIONS ARE ENCLOSED IN           */
/*                  BEGIN BLOCKS TO ISOLATE NAMES.  MODULE-           */
/*                  WIDE DATA DEFINITIONS (ESTAB) ARE PLACED*/
/*                  OUTSIDE OF THE BEGIN BLOCKS.  ONE OF THE*/
/*                  FUNCTIONS (MATCHES) IS REPRESENTED BY             */
/*                  THE PROCEDURE STATEMENT.  THE OTHER TWO           */
/*                  FUNCTIONS ARE REPRESENTED BY ENTRY                */
/*                  STATEMENTS.                                       */
/*********************************************************************/
MATCHES: PROCEDURE (ESTAB,UNRESNAME,MATCHCODE);
/*                                                                    */
/*      MODULE-WIDE DATA DEFINITIONS                                  */
/*                                                                    */
DECLARE ESTAB  POINTER;
%DECLARE TABSIZE FIXED;
%TABSIZE=2000;          /* NO. OF POSSIBLE ESTAB ENTRIES             */
DECLARE 1 TABLE BASED(ESTAB),
          2 HEADER,
            3 TAG  CHAR(4),              /* DOG TAG ESTB */
            3 SIZE BINARY FIXED (15),    /* NO. OF CURRENT ENTRIES */
          2 BODY (TABSIZE),              /* ARRAY OF ENTRIES */
            3 NAME CHAR(8),
            3 TYPE CHAR(2),
            3 ADDRESS POINTER;
DECLARE                                  /* TYPE VALUES */
       MODULE  CHAR(2) STATIC INIT ('MD'),
       ENTRYPT CHAR(2) STATIC INIT ('EP'),
       EXTREF  CHAR(2) STATIC INIT ('ER');
DECLARE NULL BUILTIN;  /*FUNCTION RETURNS VALUE OF EMPTY POINTER*/
DECLARE MATCHCODE FIXED BINARY (15); /*MATCHES RETURN CODE*/
DECLARE UNRESNAME CHAR(8);  /* OUTPUT FROM MATCHES FUNCTION */
/*                                                                    */
```

```
MATCH_ER_ITEMS_IN_ESTAB: BEGIN;
/*                                                                    */
/*********************************************************************/
/*                                                                    */
/* EXTERNAL SPECIFICATION FOR MATCHES                                 */
/*                                                                    */
/* FUNCTION: MATCH ALL EXTERNAL REFERENCE ITEMS IN ESTAB.             */
/*           IF ONE IS ENCOUNTERED THAT CANNOT BE MATCHED,            */
/*           RETURN ITS NAME.                                         */
/*                                                                    */
/* PARAMETER LIST: CALL MATCHES (ESTAB,UNRESNAME,MATCHCODE)           */
/*                                                                    */
/* INPUT:  ESTAB      POINTER     POINTER TO ESTABLE                  */
/*                                                                    */
/* OUTPUT: ESTAB      POINTER     POINTER TO UPDATED ESTABLE          */
/*         UNRESNAME CHAR(8)      NAME OF AN UNMATCHED               */
/*                                EXTERNAL REFERENCE                  */
/*         MATCHCODE FIXED BIN(15)  RETURN CODE (0, 1, OR 2)          */
/*                                                                    */
/*     CAUSE/EFFECT RELATIONSHIP:                                     */
/*                                                                    */
/*          ANY UNMATCHED (NULL ADDRESS FIELD) EXTERNAL               */
/*          REFERENCES ARE MATCHED WITH A MATCHING MD OR              */
/*          EP ITEM BY TRANSFERRING THE ADDRESS OF THE MD OR          */
/*          EP ENTRY TO THE ER ENTRY UNTIL AN ER CANNOT BE            */
/*          MATCHED.                                                  */
/*         -IF ALL ER'S CAN BE MATCHED, MATCHCODE=0 AND               */
/*             UNRESNAME IS UNCHANGED.                                */
/*         -IF AN ER CANNOT BE MATCHED, MATCHCODE=1 AND               */
/*             UNRESNAME = NAME OF ER ITEM                            */
/*         -IF AN ERROR OCCURS (INVALID ESTABLE), MATCH-              */
/*             CODE=2 AND UNRESNAME IS UNCHANGED                      */
/*                                                                    */
/* EXTERNAL EFFECTS: NONE                                             */
/*********************************************************************/
/* MATCHES LOGIC INFORMATION                                          */
/*                                                                    */
/* MODULES CALLED: NONE                                               */
/*                                                                    */
/* INITIAL REFINEMENT:                                                */
/*                                                                    */
/*    DO I=1 TO SIZE WHILE(DON'T ENCOUNTER AN ITEM THAT               */
/*                         CAN'T BE MATCHED)                          */
/*       IF (ENCOUNTER AN UNMATCHED ITEM)                             */
/*          THEN SEARCH FOR A MATCHING EXTERNAL SYMBOL                */
/*               IF (FIND A MATCH)                                    */
/*                  THEN MARK UNMATCHED ITEM                          */
/*                  ELSE OUTPUT ITS NAME                              */
/*          ELSE                                                      */
/*    END                                                             */
/*********************************************************************/
```

```
/*  LOCAL DECLARES          */
DECLARE
       I FIXED BINARY (15), /* INDEX FOR SEARCHING FOR AN
                               UNMATCHED NAME */
       J FIXED BINARY (15); /* INDEX FOR SEARCHING FOR A
                               MATCHING NAME  */
/*                                                                  */
MATCHCODE=2;
IF(ESTAB¬=NULL)                          /*NULL IS UNSET POINTER*/
  THEN
    IF((TAG='ESTB')&(SIZE>0)&(SIZE¬>TABSIZE))
    THEN
     DO;
       MATCHCODE=0;
       DO I=1 TO SIZE WHILE (MATCHCODE=0);
         IF(BODY(I).ADDRESS=NULL)     /*UNMATCHED NAME?*/
           THEN DO;
                  MATCHCODE=1;
                  DO J=1 TO SIZE WHILE (MATCHCODE=1);
                    IF((BODY(I).NAME=BODY(J).NAME)&
                        ((BODY(J).TYPE=MODULE)|
                         (BODY(J).TYPE=ENTRYPT)))
                      THEN DO;
                             MATCHCODE=0;
                             BODY(I).ADDRESS=BODY(J).ADDRESS;
                           END;
                      ELSE;
                  END;
                  IF(MATCHCODE=1) THEN UNRESNAME=BODY(I).NAME;
                                  ELSE;
                END;
           ELSE;
       END;
     END;
    ELSE;
  ELSE;
RETURN;
END;
END;
```

Рис. 8.7. Законченный текст программы MATCHES.

Все проверки, кроме 1 и 7, осуществимы (будем предполагать, что компилятор обнаруживает ситуацию 1, а проверить ситуацию 7 для MATCHES невозможно). Вероятность каждой из этих семи ошибок кажется одинаковой и довольно низкой. Последствия 2, 3 и 4 серьезны, поскольку эти ситуации приведут к прерыванию программы или неправильным результатам. Ситуации 5 и 6 в действительности не влияют на функцию MATCHES (по программе видно, что если в элементе-модуле имеется нулевой адрес, он будет сопоставлен себе, так что функция MATCHES проигнорирует его). Поэтому было решено написать защитные проверки для ситуаций 2, 3 и 4. Для случая 2 проверяем, определен ли указатель. Для случая 3 — имеет ли область, на которую ссылается указатель, тэг ESTAB. Для случая 4 проверяем значение SIZE; оно должно быть

больше нуля и не больше максимальноый длины (нет простого способа обнаружить ситуацию, когда таблица содержит элементов больше или меньше, чем указывает SIZE). Дополнительный защитный фрагмент программы изображен на рис. 8.7. Текст программы и необходимые объявления данных для входов ADDTOES и FINDES, а также спецификации этих входов опущены.

ЯЗЫКИ ВЫСОКОГО УРОВНЯ

Используемый язык программирования существенно влияет на надежность; в общем случае, чем «выше» уровень языка, тем меньше ошибок. Возражения против программирования на языке машин (или, что то же самое, против программирования на языке ассемблера) столь сильны, что трудно представить себе серьезную причину, по которой было бы нужно программировать что-нибудь в машинных кодах, за исключением двух нечастых ситуаций: при раскрутке на новой вычислительной системе может понадобиться запрограммировать в кодах ядро компилятора; обычно должно быть также написано на языке машины очень маленькое ядро операционной системы (не из соображений эффективности, а по той причине, что определенные особенности архитектуры, специфические для конкретной машины, например механизм прерываний, не будут отражены в языке программирования). Третье исключение, очевидно, возникает в случае, когда язык машины — единственный язык, доступный вашей вычислительной системе.

Возражения против языка машины настолько убедительны, что этот вопрос относится к числу тех немногих в программировании вопросов, на которые можно дать окончательный ответ. Если программист использует язык машины, он тратит бо́льшую часть своего времени не на решение задачи, а на возню с особенностями машины. Языки высокого уровня устраняют несколько уровней ошибок в программах, скрывая особенности машины и позволяя выразить любую функцию меньшим числом операторов. Программы на языках высокого уровня легче понять и изменять, они в некотором смысле самодокументированы, обладают лучшей совместимостью и мобильностью. Программы на языках высокого уровня дешевле, поскольку, хотя человек обычно программирует с фиксированной скоростью (измеряемой числом операторов в единицу времени), операторы языка высокого уровня по функциональным возможностям превосходят операторы языка машины [12, 13].

Самое большое достоинство языков высокого уровня заключено в их способности работать со сложными структурами данных. Опытный программист согласится, что программирование состоит в работе с данными, а не с тонкостями машины. Вопрос эффективности обычно отодвигается на второй план. Следует понимать, что эффективность достигается разумным выбором алгоритмов и структур

данных, а не благодаря микроэффективности, обеспечиваемой использованием машинного языка. Более того, современные оптимизирующие компиляторы генерируют замечательные по эффективности объектные программы.

Подтверждением сказанного является большая операционная система Multics, реализованная на PL/1 в совместном проекте лабораторий Белл, Дженерал электрик и МТИ [12]. Только 5% системы написано на машинном языке. Это число могло быть и меньше, будь компилятор PL/1 доступен с самого начала работы над проектом.

Сегодня в большинстве программных проектов, по существу, нет оправдания использованию машинного языка. Будь я руководителем отдела обработки данных, я бы отправился в свой вычислительный центр и выбросил все ассемблеры. Изготовители ЭВМ могут помочь в этом отношении, приняв за правило не поставлять ассемблеры для своих систем (некоторые из них уже применяют это правило к отдельным системам). Следует ослабить упор на преподавание языка ассемблера в учебных планах по информатике, за исключением, возможно, специальных курсов по архитектуре ЭВМ и компиляторам.

ПОЗИЦИЯ ПРОГРАММИСТА

Среди множества идей, которые касаются надежности программного обеспечения, не нужно забывать о важном и часто недооцениваемом факторе: позиции (или психологии) программиста. Нас особенно интересуют следующие два аспекта этого вопроса: позиция программиста по отношению к продукту его труда и его точка зрения на роль процесса компиляции.

Вопрос о позиции программиста по отношению к продукту его труда связан, как это показано Вейнбергом [14], с принципами *безличного программирования* и *когнитивного диссонанса*. Когнитивный диссонанс — это психологический принцип, который руководит действиями человека, чьи представления о себе оказались под угрозой. Поведение программиста, который рассматривает свою программу как продолжение себя и подсознательно считает недостатки программы проявлением своих недостатков, может быть изменено вследствие когнитивного диссонанса.

> «Программист, который искренне считает программу продолжением своего «я», не будет пытаться найти все ошибки в ней. Напротив, он постарается показать, что программа правильна, даже если это означает не замечать ошибок, чудовищных для постороннего взгляда... Человеческий глаз имеет почти безграничную способность не видеть то, чего он видеть не желает» [14].

Спасти в такой ситуации может безличное программирование. Вместо того чтобы быть скрытным и защищать свою программу, программист занимает противоположную позицию: его программа в действительности — часть общей работы над проектом, и он открыто

приглашает других программистов читать и конструктивно критиковать ее. Когда кто-то находит ошибку в его программе, программист, конечно, не должен радоваться, что ошибся; его позиция примерно такова: «О! *Мы* нашли ошибку в *нашей* программе! Хорошо, что мы нашли ее сейчас, а не позже! Поучимся на этой ошибке, а заодно посмотрим, не найдем ли еще!» Программист, обнаруживший ошибку в чужой программе, не кричит: «Посмотри на свою идиотскую ошибку!», а реагирует примерно так: «Как любопытно! Интересно, не сделал ли и я такой ошибки в написанном мной модуле?» Заметьте в последнем предложении тонкую деталь, связанную с когнитивным диссонансом: я использовал оборот «написанный мной модуль» вместо «мой модуль».

Вторая важная сторона позиции программиста — это его отношение к точности. Ни одна другая дисциплина не требует такой степени точности, как программирование. Хирург, несомненно, не беспокоится о том, чтобы делать разрез обязательно между клетками тела. Программирование не знает понятия *допуска*, пользуясь которым инженер говорит: «Я буду представлять двоичное значение 1 потенциалом в 5 вольт плюс-минус 1 вольт»; каждый участок программы либо правилен, либо неправилен, и промежуточного положения нет. Поэтому и важно отношение программиста к точности, и сказывается оно во взглядах на процесс компиляции программы. Часто можно услышать такое мнение: «Вместо того чтобы тратить время, добиваясь совершенства моего модуля, я прогоню его через компилятор, и пусть им занимается машина». Заставить работать машину — это, конечно, идея, лежащая в основе всей индустрии обработки данных. Однако как только «работа» отождествляется с программированием, нужно поставить точку, потому что «знания» ЭВМ в области программирования весьма ограниченны.

Экспериментирование с компилятором, компиляция наскоро написанного модуля, чтобы посмотреть, «что получится»,— нездоровая позиция, поскольку противоречит требованию точности. Я считаю, что профессиональный программист должен ориентироваться на успешную компиляцию его программы с первого раза. Он должен сам сделать все возможное, чтобы исключить синтаксические ошибки, а также дать свою программу просмотреть еще кому-нибудь, прежде чем приступать к компиляции. Имеется четыре довольно убедительных довода в пользу этого.

1. Одна из причин синтаксических ошибок — недопонимание синтаксиса языка. Если у вас есть упущения в синтаксисе, то вы, вероятно, недопонимаете и семантику. Компилятор, однако, найдет лишь очень немногие из ваших семантических ошибок.

2. Вторая причина синтаксических ошибок — небрежность. Здесь можно повторить прежние рассуждения. Если вы небрежны в синтаксисе, вы, вероятно, также небрежны в семантике.

3. Ни один из компиляторов, известных мне, не обнаруживает всех синтаксических ошибок. Чем больше их сделано, тем больше вероятность того, что какая-то из них не будет обнаружена и приведет, таким образом, к логической ошибке.

4. Тише едешь — дальше будешь. Я заметил, что первая компиляция модуля является всегда интересным поворотным моментом: внезапно и быстро возрастает давление, оказываемое программистом на самого себя. До компиляции он обычно тратит значительное время на обдумывание каждого изменения или улучшения программы. После первой компиляции модуля возникает определенное психологическое давление, вынуждающее вносить изменения как можно быстрее, чтобы наконец «заставить его работать» или «заставить его скомпилироваться». Изменения, которые делаются под таким давлением, обычно изобилуют ошибками.

ДОКУМЕНТАЦИЯ

Документация всегда была больным вопросом в программировании из-за распространенного представления, что программисты ненавидят документирование. Однако истина состоит в том, что никто не испытывает удовольствия от документирования только ради документации, но там, где документация составляет неотъемлемую часть процесса программирования, она охотно воспринимается и подготавливается. Начиная с гл. 4 мы уже составили основательный список документов, хотя это специально и не подчеркивалось. В гл. 4 — это требования к результату работы, цели проекта и внешние спецификации. В гл. 5 — определение архитектуры системы. После гл. 6 у нас появилась схема структуры программы, а в настоящей главе мы подготовили внешние спецификации модуля и текст его программы. Спрашивается: нужно ли еще что-нибудь (кроме публикаций для пользователя — а их часто готовит специальная группа профессиональных технических писателей)?

Ответ состоит в том, что никакая дополнительная документация о программе не нужна. В частности, блок-схемы или что-либо заменяющее их вовсе не обязательны.

Может показаться, что не хватает документации, описывающей логику модуля (например, блок-схем). Блок-схемы ни разу не упоминались при обсуждении процесса проектирования логики, поэтому они, очевидно, не являются необходимой частью проектирования. На самом деле они не только необязательны, но и нежелательны по нескольким причинам. Блок-схемы не согласуются с идеей пошаговой детализации [1]). Ключевая ее идея — ряд небольших

---

1) Это слишком сильное утверждение. Например, в книге Хьюз Дж., Мичтом Д. Структурный подход к программированию.— М.: Мир, 1980, блок-схемы считаются приемлемым инструментом при пошаговой детализации.— *Прим. ред.*

детализирующих шагов, постепенно приближающих уровень к уровню языка программирования. Это можно сделать и используя блок-схемы, но постоянно перечерчивать их при каждом шаге детализации было бы довольно хлопотно. Кроме того, оставался бы еще один большой шаг перевода: преобразование блок-схемы в программу на исходном языке. Далее, хотя основные конструкции структурного программирования изображены на рис. 8.4 в виде блок-схем, последние не согласуются со структурным программированием, поскольку в значительной степени ориентированы на использование GO TO.

Если не блок-схемы, то что же? В действительности у нас уже есть приемлемая альтернатива: сам текст программы! Коль скоро он написан, почему бы ему и не служить документацией логики? Просмотрев заново рис. 8.7, можно убедиться в том, что текст программы MATCHES вместе с первоначальной его детализацией и внешними спецификациями модуля составляют достаточный и весьма желательный комплект документации для функции MATCHES. Дополнительная документация (например, блок-схемы) нежелательна, так как она была бы избыточной при наличии такого текста программы. Избыточности в документации любого вида следует избегать, поскольку она увеличивает возможность возникновения противоречий. Более того, если не заботиться об обновлении документации (а это особенно трудно, если документация логики физически отделена от текста программы), избыточная документация часто становится совершенно бесполезной после того, как текст программы несколько раз изменен.

Иногда используются также другие конкуренты блок-схем — таблицы решений, словесные описания и HIPO-диаграммы [15].

HIPO-диаграмма состоит из трех блоков: в центре — блок обработки, в котором словесно описана логика модуля, и по сторонам — блоки, описывающие используемые входные и вырабатываемые выходные данные. Хотя HIPO-диаграммы лучше блок-схем, так как изображают и последовательность выполнения, и поток данных, они не нужны по тем же причинам.

Внешние спецификации модуля и текст исходной программы составляют основу документации модуля. Они должны быть дополнены небольшой дозой информации о логике и комментариев к исходной программе. Обратите внимание на блок комментариев, изображенный на рис. 8.7; блок содержит информацию о логике MATCHES, состоящую из исходной детализации программы и списка вызываемых модулей (отметим, что в документации ни в коем случае не должны упоминаться вызывающие модули). Вместе с информацией о логике может также присутствовать и список советов по модификации.

Последний тип документации — комментарии непосредственно в тексте исходной программы. Все объявления данных должны быть

прокомментированы, поскольку понимание данных — ключ к пониманию программы. Выполняемые операторы в модуле если и следует комментировать, то лишь изредка. Если программа соответствует определению структурной, то необходимости в комментариях к выполняемым операторам нет. Дело в том, что избыток комментариев усложняет чтение программы. Разумно руководствоваться таким правилом: читать текст программы и пытаться предугадать вопросы, которые могут возникнуть у другого читателя. Комментарии должны отвечать на эти вопросы. Во всяком случае, они не должны дублировать текст программы; если уж они используются, то их следует рассматривать как указания для читателя.

СТАНДАРТЫ

Проблема стандартов — еще одно больное место у программистов. Имеется подозрение, что стандарты сдерживают творчество. В действительности эффект хорошо выбранных стандартов прямо противоположен; они дают правила для принятия решений по многим рутинным вопросам, неизбежным при работе над проектом, так что творческие способности программиста могут быть направлены на *важные* вопросы (на решение задачи). Стандарты положительно влияют также и на надежность, так как гарантируют единообразие подхода к определенным аспектам программистской деятельности, предупреждая таким образом несогласованность между программистами.

Никакие стандарты в программировании не должны рассматриваться как абсолютные правила, которые ни в коем случае нельзя нарушить. Цель стандартов — установить основные правила для определенных аспектов деятельности, чтобы все предлагаемые отклонения от них можно было публично рассмотреть и определить, оправданы ли они. Было бы бесполезно пытаться привести здесь набор стандартов, поскольку большинство организаций разработало свои наборы, приемлемые в их конкретной обстановке. Однако, для того чтобы дать представление о том, какие аспекты программирования обычно в них отражаются, отметим, что в большинстве стандартов фиксируются руководящие принципы используемой методологии, форматы документации, возможности компилятора, соглашения по размещению текста программы на странице, правила употребления комментариев, а также конструкции языка программирования, которых следует избегать.

ПРОВЕРКА ПРАВИЛЬНОСТИ

Последний шаг в процессе проектирования модуля — проверка правильности его внутренней логики. Здесь рассматривается проверка правильности человеком, т. е. проверка до выполнения програм-

мы вычислительной машиной. Проверке правильности при выполнении программы, т. е. тестированию, посвящена часть 3.

Проверка правильности основывается на различных способах *чтения текста программы*. Хотя разные формы чтения практиковались много лет, значение этого метода для массового производства программ было осознано в достаточной мере только лишь с выходом книги Вейнберга [14].

Проверка может осуществляться как в форме *статического чтения программы*, так и в форме *динамического чтения*. Эти формы дополняют друг друга и должны использоваться совместно. В процессе статического чтения исходный текст модуля просто прочитывается с начала и до конца, как книга. Это обычно делают втроем или вчетвером, причем один играет роль председателя, т. е. следит за тем, чтобы встреча была продуктивной. Цель такой встречи — найти ошибки в модуле, но не исправлять их. Исправлять ошибки нужно потом.

Трудно перечислить, что именно следует проверять в ходе встречи, потому что те, кто пользуется этим методом, обычно вскоре составляют мысленно свой собственный список распространенных ошибок. Фейган идет еще дальше и рекомендует использовать профессиональных «контролеров» и официально фиксировать типы найденных ошибок для лучшей организации последующих встреч [16]. Очень неполный список вопросов может начинаться с проверки полноты документации и легкости чтения программы. Затем следует тщательно проверить условия во всех предложениях IF и DO. Например, очень часто путают условия «больше или равно», «больше чем», «меньше чем», «не меньше или равно», «по крайней мере» и т. д. Необходимо проверить сложные логические условия, включающие связки *или* и *и*, чтобы убедиться, что эти операции указаны в правильной последовательности. Для каждого цикла следует проверить, может ли закончиться его выполнение. Итерационные циклы DO с постоянной (явно указанной) верхней границей следует проверить особенно тщательно. Если вы видите такое предложение, как

```
DO I=1 TO N WHILE (FOUND);
```

следует обратить внимание на две ситуации: цикл никогда не выполняется (FOUND сразу имеет значение «ложь» или N меньше 1) или цикл заканчивается неудачей (I достигает значения N раньше, чем значением FOUND станет «ложь»). Обращайте особое внимание на индексы массивов и циклы, не возникла ли ситуация «плюс-минус один» (цикл выполняется на один раз больше или меньше, чем нужно). Необходимо проверить правильность порядка, форматов, единиц измерения и атрибутов всех аргументов, передаваемых модулю и получаемых от него.

Другой метод проверки правильности — *сквозной контроль*,

сходный по своей идее с обсуждавшимися в предыдущих главах методами сквозного контроля в процессах проектирования. Здесь также организуется встреча нескольких участников (контрольная сессия), один из которых ведет встречу в роли председателя. Но вместо того, чтобы просто читать текст программы, готовится несколько достаточно представительных тестов, и участники встречи мысленно контролируют весь ход их выполнения модулем. Во время этой процедуры контролируется состояние модуля (т. е. значения переменных). Тесты должны быть просты и немногочисленны, чтобы процесс не был утомительным, но они должны охватывать возможные варианты ошибочных входных данных и предельные случаи. На роль человека, занимающегося подготовкой таких тестов, очень хорошо подходит программист-разработчик одного из модулей, вызывающих данный.

Во время сквозного контроля к программисту будут обращаться за разъяснениями тех или иных мест в модуле. Его комментарии часто раскрывают и ошибки такого характера, когда он основывается на сомнительных предположениях. Во время каждого из этих двух процессов составляется список всех обнаруженных упущений (сюда могут входить как ошибки, так и плохие приемы программирования). Позднее в соответствии с этим списком программист делает необходимые исправления. Обычно устанавливается определенная верхняя граница, так что если число обнаруженных упущений превосходит ее, модуль должен снова пройти эту процедуру после того, как будут сделаны все исправления.

Вследствие распространенного среди программистов отношения к роли компилятора контрольные сессии необходимо проводить до первой компиляции модуля. Зато после нее должна использоваться третья форма чтения текста программы: проверка информации, выданной компилятором. Например, большинство компиляторов PL/1 проводит точный анализ атрибутов всех переменных в модуле. Необходимо тщательно проверить распечатку результатов компиляции, чтобы убедиться, что компилятор правильно интерпретирует ваши намерения относительно объявления данных.

Несмотря на достоинства визуальной проверки, она иногда недооценивается из-за распространенного мнения, что «тестирование компьютером» дешевле. Исследование, проведенное в лабораториях «Белл» [17], показало, что это впечатление ошибочно. Средняя стоимость обнаружения ошибки при чтении программы была равна 10 долл. 62 центам, средняя стоимость обнаружения при обычном тестировании — 251 долл. 60 центам (из них 196 долл. 70 центов стоило машинное время). Конечно, могло оказаться, что ошибки, обнаруженные человеком, были простыми, а машина обнаружила более сложные. Анализ, однако, опровергает это предположение. Кроме того, визуальная проверка имеет дополнительное, хотя и неосязаемое преимущество: обучение программистов.

Осталась еще одна (более сложная) форма чтения текста программы, основанная на идее математического доказательства ее правильности, для чего нужно сначала сформулировать, а затем доказать теоремы о свойствах программы. Поскольку эти работы в большинстве своем все еще находятся в стадии исследования, о них мы будем говорить отдельно, в гл. 17.

ЛИТЕРАТУРА

1. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. Пер. с англ.— М.: Мир, 1976.
2. Дал У., Дейкстра Э., Хоор К. Структурное программирование. Пер. с англ.— М.: Мир, 1975.
3. McGowan C. L., Kelly J. R. Top-Down Structured Programming Techniques, New York: Petrocelli/Charter, 1975.
4. Dijkstra E. W. Go To Statement Considered Harmful, *Communications of the ACM*, **11** (3), 147—148 (1968).
5. Knuth D. E. Structured Programming with GO TO Statements, Computing Surveys, **6** (4), 261—301 (1974).
6. Charmonman S., Wagener J. L. On Structured Programming in FORTRAN, *SIGNUM Newsletter*, **10**(1), 21—23 (1975).
7. Ashcroft E., Manna Z. The Translation of GO TO Programs to WHILE Programs, Proceedings of the 1971 IFIP Congress, Booklet TA-2. Amsterdam: North-Holland, 1971, pp. 147—152.
8. Dijkstra E. W. A Constructive Approach to the Problem of Program Correctness, *BIT*, **8** (3), 174—186 (1968).
9. Wirth N. Program Development by Step-Wise Refinement, *Communications of the ACM*, **14** (4), 221—227 (1971).
10. Wirth N. On the Composition of Well-Structured Programs, *Computing Surveys*, **6** (4), 247—259 (1974).
11. Bloom A. M. The ELSE Must Go, Too, *DATAMATION*, **21** (5), 123—128 (1975).
12. Corbato F. J. PL/I as a Tool for System Programming, *DATAMATION*, **15**(5), 68—76 (1969).
13. Myers G. J. Estimating the Costs of a Programming System Development Project, TR 00.2316. IBM System Development Div., Poughkeepsie, N. Y., 1972.
14. Weinberg G. M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.
15. HIPO — A Design Aid and Documentation Technique, GC20—1851. IBM Corp., White Plains, N. Y., 1974.
16. Fagan M. E. Design and Code Inspections and Process Control in the Development of Programs, TR 21.572. IBM System Development Div., Kingston, N. Y., 1974.
17. Shooman M. L., Bolsky M. I. Types, Distribution, and Test and Correction Times for Programming Errors, Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE, 1975, pp. 347—357.

ГЛАВА 9

Стиль программирования

От руководителей программистских коллективов можно иногда услышать такие замечания: «Работая над последним проектом, мы использовали структурное программирование, но программы все равно изобилуют ошибками, и, по мнению группы сопровождения, читать их совершенно невозможно». Иногда подобная реакция — просто проявление консерватизма в отношении новых идей, но чаще она означает неправильную, чрезмерно упрощенную интерпретацию концепции структурного программирования. Как указывалось в гл. 8, структурное программирование включает целый ряд в равной степени важных идей и методов; концентрация внимания на каком-то одном аспекте (например, просто отказ от употребления предложений GO TO) и пренебрежение остальными приводит к неудаче проекта.

Центральное понятие, связанное со структурным программированием,— стиль программирования, т. е. манера, в которой программист (правильно или неправильно) употребляет особенности своего языка программирования — в том же смысле, как писатель (правильно или неправильно) использует естественный язык. Именно плюсы и минусы стиля программирования обычно оказываются основной причиной таких, например, суждений: «Я видел программу, очень красиво размещенную на странице и без единого GO TO, но при этом абсолютно непостижимую, и другую, совершенно понятную программу, в которой было несколько GO TO».

Стилю программирования, ввиду важности предмета, посвящена отдельная глава. Замечания и общие принципы, сформулированные в этой главе, основаны на моем личном опыте, накопленном при чтении программ, опыте коллег и на идеях, почерпнутых из блестящей книги Кернигана и Плогера по стилю программирования [1].

ЯСНОСТЬ ПРОГРАММЫ

Как уже говорилось в гл. 8, задача программиста должна состоять в том, чтобы писать на исходном языке программы, предназначенные для аудитории, состоящей в первую очередь из людей, а не машин. Это требует уделения большего внимания ясности,

простоте и доступности текста за счет игнорирования менее важных критериев, например краткости (число ударов по клавишам, необходимых для того, чтобы напечатать программу) или машинной эффективности. Выполнение следующих правил помогает писать ясные программы.

Используйте осмысленные имена. Это правило выражает самый важный принцип стиля программирования. Ничего нет хуже программы с именами XX, XXX, XXXX, XY, EKK, EKKK, A и AI, заполненной комментариями, разъясняющими смысл и назначение этих имен. Простой прием — использование более длинных содержательных имен — значительно облегчает чтение программы и сводит к минимуму количество необходимых комментариев. Как правило, имена переменных должны содержать от 4 до 12 символов. Слишком длинные имена, типа ЧИСЛО_ТОЧЕК_В_ГРАФЕ_РАССТОЯНИЙ, нежелательны, потому что они отвлекают внимание от собственно программы и сами подвержены ошибкам.

Длина используемых имен зависит от ограничений языка программирования. Фортран существенно ограничивает длину имен (не более 6 символов). В большинстве реализаций PL/1 разрешены идентификаторы длиной до 31 символа, но имена процедур ограничиваются семью символами — серьезное ограничение, если пытаться давать процедурам осмысленные имена.

Даже несмотря на то что использование осмысленных имен диктуется здравым смыслом, иногда встречаются прямо противоположные рекомендации. Например, я был поражен, встретив в тексте, опубликованном в 1974 г., следующий совет: если вы хотите минимизировать время компиляции вашей программы на Фортране, не используйте осмысленных имен; вместо этого применяйте равномерно распределенные однобуквенные, затем двухбуквенные имена и т. д., поскольку это минимизирует время поиска в таблицах в некоторых компиляторах Фортрана!

Избегайте сходных имен. Иногда встречаются программы, в которых используются такие имена переменных, как VALUE и VALUES, или ряд переменных с именами BRACA, BRACB, BRACC и BRACD. Это усложняет чтение программы. Выбирая осмысленные имена, старайтесь, чтобы они были как можно менее похожи.

Если в идентификаторах используются цифры, помещайте их только в конце. Цифры 0, 1, 2 и 5 легко спутать с буквами O, I, Z и S. Если уж цифры в именах переменных необходимы, помещайте их в конце.

Никогда не используйте в качестве идентификаторов ключевые слова. Хотя следующие операторы синтаксически совершенно правильны в Фортране и PL/1, они приводят читателя в замешательство.

```
FORMAT(I6)=1
IF IF=THEN THEN THEN=ELSE;
  ELSE ELSE=THEN;
```

Избегайте промежуточных переменных. Хотя в большинстве программ некоторое количество промежуточных переменных необходимо, никогда не создавайте лишних. Часто можно встретить рекомендации заменять выражения вида

```
X = A(I) + 1/A(I)
```

на

```
AI = A(I)
X  = AI + 1/AI
```

чтобы сэкономить на индексации, или заменять

```
TOTAL = Y + A/B*C
VAR   = X + A/B*C
```

на

```
ABC   = A/B*C
TOTAL = Y + ABC
VAR   = X + ABC
```

чтобы сэкономить несколько арифметических операций. Такие советы обычно следует игнорировать, поскольку они ведут к созданию лишних промежуточных переменных. Эти переменные затрудняют чтение программы, увеличивая количество деталей, с которыми приходится иметь дело. Они также лишают смысла вырабатываемый компилятором словарь перекрестных ссылок (XREF). Последний является ценным подспорьем, поскольку говорит читателю, какие идентификаторы в каких операторах используются. Вводя промежуточные переменные для экономии нескольких микросекунд, можно сделать такой словарь совершенно бесполезным.

Во избежание неоднозначности употребляйте скобки. Порядок выполнения (приоритет) арифметических операций в разных языках программирования всегда является источником путаницы. Например, означает ли — A**2 «возвести в квадрат после изменения знака A» или «изменить знак у результата возведения A в квадрат»? Когда есть хоть какие-то сомнения, следует использовать дополнительные скобки, чтобы указать нужный порядок выполнения операций.

Большинство программистов знакомо с этим правилом, но они чрезмерно обобщают его, считая, что лишние скобки никогда не изменяют смысл оператора. К сожалению, в некоторых языках программирования это правило не выполняется, в частности в операторах ввода-вывода и вызова (CALL). Например, следующие два оператора семантически не эквивалентны:

```
CALL SUB(X,Y);
CALL SUB((X),(Y));
```

В первом из них используется передача параметров «по ссылке», во втором — «по значению».

Будьте внимательны, используя константы как аргументы. Большинство программистов знакомо со следующей ошибкой:

```
CALL SUB(X,2)
I=2
WRITE (6,101) I
...
SUBROUTINE SUB(S,J)
...
J=3
RETURN
```

Напечатанное значение I во многих реализациях Фортрана будет равно 3, а не 2, потому что подпрограмма изменяет значение константы 2.

Располагайте только один оператор на строке. Размещение нескольких операторов на одной физической строке противоречит правилу структурированного программирования, требующему сдвигать оператор по строке в соответствии с уровнем его вложенности. Классическим примером является также программист, который пишет A=B=C в одной строке программы на PL/1, ожидая, что значение C будет присвоено и A, и B. Этот оператор, однако, интерпретируется компилятором как присваивание A логического значения 0 или 1 в зависимости от того, отличается B от C или нет. В действительности это дефект языка PL/1; знак равенства используется в этом языке для двух различных операций: сравнения и присваивания.

Не изменяйте значение параметра цикла в теле цикла. Изменение параметра цикла внутри цикла усложняет и понимание цикла, и доказательство того, что он не является «бесконечным».

Избегайте помеченных операторов. При использовании правил структурного программирования метки при операторах не нужны, если только это не идентификаторы блоков BEGIN, входов или конструкции ON в программе на PL/1. *Никогда* не следует использовать метку, если в программе нет на нее перехода (принятая у некоторых программистов практика помещать номер оператора в качестве метки перед каждым оператором в программе на Фортране совершенно сбивает с толку). Если метки нужны, они должны быть осмысленными и не похожими друг на друга именами.

ИСПОЛЬЗОВАНИЕ ЯЗЫКА

Вторая важная характеристика стиля программирования — способ, которым программист отбирает для употребления (или отбраковывает) возможности языка программирования. Общее правило здесь состоит в том, чтобы понять и использовать все возможности языка, но остерегаться плохо продуманных его особенностей и зависящих от реализации трюков.

Изучите и активно используйте возможности языка. Иногда можно увидеть программу на PL/1, содержащую цикл DO для обнуления всех элементов массива. Обычно это является признаком слабого знания языка, поскольку достаточно было бы одного оператора A=0. В других случаях программист использует циклы DO для организации, например умножения элементов массива на два и сложения с элементами другого массива, в то время как и здесь было бы достаточно одного оператора A=(B*2)+C. В Коболе имеется оператор SEARCH, поэтому обычно не нужно реализовывать поиск в таблице в виде цикла. Суть сказанного состоит просто в том, что в основательное обучение языку программирования выгодно вкладывать средства: в результате программы становятся короче и исключаются определенные типы ошибок.

Изучите и используйте библиотечные и встроенные функции. Многие программисты знакомы с математическими функциями, реализованными в языке (квадратный корень, синус, косинус, абсолютная величина), но меньше знают о других полезных функциях. Чтобы дать некоторое представление о ряде полезных возможностей языка, заметим, что PL/1 содержит более 80 встроенных функций, среди них традиционные MAX и MIN (поиск максимального и минимального значения в группе переменных), функции над одним массивом, такие, как SUM, PROD (суммирование и перемножение всех элементов массива), функции сравнения массивов, такие, как ANY (A=B) (есть ли пара равных между собой соответствующих компонент массивов A и B?) и ALL (A=B) (все ли компоненты A равны соответствующим компонентам B?). В PL/1 имеется также ряд функций над литерными строками, таких, как INDEX

(Z, 'DOG') (искать в строке Z подстроку DOG), VERIFY (проверить, каждая ли литера первой строки входит также и во вторую строку) и TRANSLATE (заменить литеры одной строки соответствующими литерами другой строки). Употребление таких функций там, где это необходимо, способно облегчить чтение программы и уменьшить число возможных ошибок.

Избегайте трюков. Следует избегать тех особенностей языка, которые будут непонятны читателю вашей программы. К ним относятся, например, такие приемы, как использование параметра цикла после окончания цикла или совмещение переменных (выделение им одного участка в памяти), с тем чтобы можно было воспользоваться таким опасным приемом программирования на PL/1, как выполнение арифметических операций над переменными типа указателя.

Не игнорируйте предостерегающих сообщений. Некоторые компиляторы, обнаружив трюкаческое, сомнительное или непредусмотренное использование возможностей языка, выдают предостерегающее сообщение, но доводят компиляцию до конца. Такие предостережения очень важны; они указывают, что либо программист допускает ошибку, либо потенциальный читатель программы этого места не поймет. Текст программы следует изменить так, чтобы не было никаких предостерегающих сообщений.

Внимательно прочтите раздел о «подводных камнях» в руководстве по вашему языку. Многие руководства по языкам содержат главу о распространенных ошибках и недоразумениях при работе с этим языком. Экономьте время, учитесь на ошибках других.

МИКРОЭФФЕКТИВНОСТЬ

К наихудшим нарушениям хорошего стиля программирования относятся многочисленные «улучшения программы для повышения ее эффективности», которые можно найти во многих учебниках, стандартах организаций и головах программистов. Эти предложения не только усложняют чтение программы и снижают ее надежность, но к тому же весьма слабо (если вообще хоть как-то) влияют на эффективность; отсюда и название раздела: «Микроэффективность».

Всякий, кто когда-либо серьезно занимался производительностью систем, знает, что эффективность достигается в результате тщательного анализа структур данных и алгоритмов, а также использования ресурсов, но не за счет таких тривиальностей, как исключение индексации, замена возведения в степень умножением, программирование на машинном языке или поиски самого быстрого способа обнуления регистра (в моей корзине для бумаг много памятных записок по последнему вопросу).

В качестве примера успешного подхода к эффективности расскажу об одном из моих коллег, у которого недавно возникла идея улучшения алгоритма планирования в операционной системе TSS фирмы IBM. Однажды ночью он осуществил свою идею на системе TSS, установленной в Исследовательском центре IBM имени Томаса Дж. Уотсона, после чего измерения показали трехкратное увеличение производительности системы, что во много раз лучше обычных 10—20%, к которым стремятся многие. Здесь особенно примечателен тот факт, что система TSS существовала к тому времени уже примерно шесть лет, и при ее внедрении в Исследовательском центре она тщательно настраивалась. К тому же предложенные изменения не были характерны только для этой организации, они были внедрены и проверены на других использующих TSS установках со сходными результатами.

Второй пример, иллюстрирующий важность макро-, а не микроэффективности, относится к тому времени, когда я работал системным программистом в вычислительном центре, в котором собирались устанавливать новую вычислительную систему третьего поколения. Мне с другими системными программистами поручили улучшить производительность операционной системы, поставляемой изготовителем. Несколько первых месяцев мы посвятили разработке специального монитора для измерения производительности и работе с ним над определением узких мест в системе. После того как анализ системы был завершен, мы предприняли такие шаги, как улучшение алгоритмов буферизации в программном обеспечении ввода-вывода, изменение формата физических блоков системных библиотек для уменьшения времени ожидания из-за вращения диска, изменение взаимного физического расположения некоторых файлов для уменьшения времени поиска и регулирование использования каналов. После этого «узким» местом системы стало системное устройство печати, и мы, обнаружив это, слегка изменили компиляторы и редактор связей, чтобы исключить лишнюю печать (например, компилятор с Фортрана в листинге каждого задания печатал несколько страниц, содержащих по одной строке). Хотя эти меры потребовали изменения менее тысячи операторов исходной программы системы, пропускная способность системы увеличилась примерно в четыре раза!

Эти два примера иллюстрируют центральную идею повышения производительности: сначала измерение и затем оптимизация *макроэффективности*. В обоих случаях в результате удалось значительно улучшить производительность, не занимаясь микроэффективностью.

Игнорируйте все предложения по повышению эффективности, пока программа не будет правильной. Худшее, что может быть сделано,— это начать беспокоиться о скорости программы до того, как она станет работать правильно. Быстрая, но неправильная програм-

ма бесполезна; медленная, но правильная всегда имеет некоторую ценность, а может оказаться и вполне удовлетворительной. Вейнберг [2] рассказывает забавную историю о новой программе, которая из-за слишком большой сложности оказалась совершенно ненадежной. Был вызван новый программист, который нашел лучшее решение и за две недели сделал новую, надежную версию программы. При демонстрации ее работы он отметил, что его программе требуется 10 секунд на каждую карту. Один из разработчиков первоначального варианта, торжествуя, заявил: «А моей программе требуется только одна секунда на карту». Ответ программиста стал классическим: «Но ваша программа не работает. Если программа не должна работать, я могу написать такую, которой хватит одной миллисекунды на карту».

Пусть оптимизирует компилятор. Многие из находящихся в эксплуатации компиляторов выполняют значительную работу по оптимизации: сокращение индексации, выявление выражений типа A**2, которые можно заменить одним умножением, вынесение постоянных выражений из циклов, размещение часто используемых переменных на быстрых регистрах и др. Программисту не нужно состязаться с компилятором; следует писать программы просто и ясно. Пусть об оптимизации заботится компилятор.

Не жертвуйте легкостью чтения ради эффективности. Как правило, предложения по улучшению микроэффективности сводятся к совокупности трюков и мешают достичь легкости восприятия. Я вспоминаю книгу, полную подобных предложений по повышению эффективности, в которой утверждалось: «Большинство приемов, которые увеличивают эффективность программы, не наносит ущерба легкости ее чтения»,— и в то же время почти каждая рекомендация противоречила положениям книги Кернигана и Плогера [1].

На рис. 9.1 приведены рекомендации по программированию цикла умножения матриц, взятые из статьи о программировании для систем с виртуальной памятью. Утверждалось, что, если использовать вторую версию, перелистывание страниц будет сокращено, потому что ссылки на элементы матрицы будут встречаться в том же порядке, как они хранятся в памяти в Фортране. Эта «улучшенная» версия явно сложнее и менее понятна. Честно говоря, просматривая ее в первый раз, я был уверен, что работать она не будет. Более того, автор не отметил, что «улучшенная» версия в действительности во многих случаях будет работать *медленнее* (например, если матрицы не слишком большие или система загружена слабо, так что имеется достаточный объем реальной памяти).

Самая замечательная особенность «улучшенной» версии — это то, что можно написать более простую версию, которая будет быстрее! Напишите цикл DO, в котором вторая матрица транспониру-

Исходная версия:

```
      DO 20 I = 1,M
      DO 20 J = 1,N
      SUM = 0
      DO 10 K = 1,L
   10 SUM = SUM + A(I,K) * B(K,J)
   20 C(I,J) = SUM
```

„Улучшенная“ версия:

```
      M1 = 0
      I3 = -1
      DO 20 J = 1,N
      DO 20 I = 1,M
      SUM = 0
      I3 = -I3
      M1 = M1 + I3
      DO 10 K = 1,L
      SUM = SUM + A(I,M1) * B(M1,J)
   10 M1 = M1 + I3
   20 C(I,J) = SUM
```

Рис. 9.1. Умножение матриц.

ется, вставьте его перед исходной версией, изображенной на рис. 9.1, и замените выражение B (K, J) на B (J, K).

Другой пример — отчет, содержащий советы по программированию для системы IBM/360, в котором в качестве самого быстрого способа удвоения значения регистра предлагается хитроумно использовать команду BXLE (переход по индексу меньше или равно). При этом не только не говорится, что этот прием способен совершенно сбить с толку, но не указывается даже, что он применим только к регистрам с нечетными номерами!

Никогда не оптимизируйте, если в этом нет необходимости. Очевидно, что эффективность не является совсем уж несущественным вопросом. Однако программисту никогда не следует заниматься оптимизацией ради самой оптимизации; он должен оптимизировать только тогда, когда эффективность важна и только когда он знает точно, какая именно часть программы нуждается в оптимизации.

Добивайтесь эффективности за счет макроэффективности. В системе с виртуальной памятью можно значительно увеличить эффективность, располагая модули на странице так, чтобы минимизировать число страничных прерываний и размеры рабочего множества программы [3]. Этот прием не требует изменений в тексте программы. К средствам повышения макроэффективности относятся также разумная организация ввода-вывода, выбор оптимальных алгоритмов и структур памяти. Например, некоторые алгоритмы сортировки и поиска в сотни раз быстрее других формально экви-

валентных алгоритмов. Программисту не следует заботиться о микроэффективности, пока не будут исчерпаны все другие средства.

Добивайтесь эффективности на основе измерений, а не догадок. Доказано, что программисты крайне слабо угадывают причины неудовлетворительной эффективности программ. Это не результат каких-то недостатков самих программистов; из-за сложной природы программ и систем интуиция в вопросе об «узких» местах подводит почти всегда. Лучше всего при первоначальной разработке программы игнорировать большинство соображений, касающихся эффективности. Когда программа работает (и только в том случае, если ее эффективность неудовлетворительна), программист должен выполнить измерения, чтобы обнаружить и исправить те самые знаменитые «5% программы, занимающие 90% времени». Если программа была спроектирована правильно (это значит, что она легко адаптируема), такие изменения post factum должны быть несложными.

В сложных системах самые простые алгоритмы часто и самые быстрые. Многие современные ЭВМ имеют трехуровневую память: небольшой быстрый буфер между ЦП и основной памятью, сама основная память и виртуальная память, отображенная на устройства вторичной памяти. В таких системах локальность ссылок, т. е. отсутствие частых ссылок в широком диапазоне адресов программы и данных,— ключевой фактор эффективности. Это значит, что простые последовательные алгоритмы в таких условиях часто работают быстрее, чем более изощренные и сложные.

КОММЕНТАРИИ

Лучшей документацией внутренней логики программы являются простая и ясная структура текста программы, использование сдвига по строке в соответствии с уровнем вложенности, осмысленные имена и соблюдение других правил, касающихся стиля программирования. Если текст программы обладает этими свойствами, обилие комментариев не является необходимым и часто даже нежелательно. Известны случаи, когда комментарии *мешали* отладке, поскольку человек, отлаживающий программу, склонен верить им и в результате недостаточно тщательно проверяет текст программы. Опытные программисты, отыскивая во время отладки ошибку в модуле, часто закрывают комментарии листом бумаги.

Избегайте обилия комментариев. Изобилие комментариев мешает чтению программы, отвлекая внимание. Всякий раз, когда я вижу модуль с очень большим количеством комментариев, я сразу начинаю подозревать, что программист написал их так много либо потому, что сама программа запутанна или содержит трюки, либо

потому, что он следовал какому-нибудь правилу, вроде «по крайней мере 50% всех операторов должны иметь комментарии»; значит, многие из комментариев бессодержательны. Программист должен быть скуп на комментарии, употребляя их только там, где это абсолютно необходимо.

Комментируйте так, как будто бы вы отвечаете на вопросы читателя. Очень эффективный метод состоит в том, чтобы сначала написать текст программы без комментариев, а затем посмотреть на него глазами читателя. Если окажется, что у читателя в некоторой точке программы может возникнуть вопрос, следует вставить содержательный комментарий с ответом на него.

Физически «выдвигайте» все комментарии из текста собственно программы. При печати комментарии следует смещать вправо от текста программы, так чтобы читатель мог просматривать программу, не прерываемую комментариями. Другими словами, комментарии в программе должны быть аналогичны подстрочным примечаниям в книге.

Прокомментируйте все переменные. Понимание данных — ключ к пониманию программы. Каждое предложение, объявляющее некоторую переменную, должно сопровождаться комментарием, поясняющим смысл этой переменной.

ОПРЕДЕЛЕНИЯ ДАННЫХ

Понимая особую важность данных, приведем несколько правил их определения и использования.

Объявляйте все переменные явно. Во многих языках программирования разрешается определять данные неявно, просто используя их имена в выполняемых операторах. Такие «сокращения» предназначены для дилетантов или тех, кто программирует от случая к случаю, но не для программистов-профессионалов. Профессиональный программист должен явно определять или объявлять все переменные в самом начале модуля [1].

Объявляйте все атрибуты каждой переменной. Не следует также пользоваться имеющимися в языке сокращениями для объявления атрибутов, не указанных явно, по умолчанию. Процесс выбора атрибутов по умолчанию часто сложен и ведет к ошибкам, если программист не вполне его понимает. Более того, некоторые ком-

---

[1] Слишком сильное утверждение; неясно, призывает ли автор, скажем, не пользоваться блоками в Алголе.— *Прим. ред.*

пиляторы позволяют на каждой вычислительной установке изменять правила умолчания, что является крайне опасной практикой.

Избегайте явных констант. При изучении программы на машинном языке, претендовавшей на то, чтобы считаться структурной, поскольку в ней употреблялись макросы для конструкций структурного программирования, прежде всего мне бросился в глаза такой оператор:

```
MVC  DPSREP+32(2),=X'0009'
```

В этом операторе трижды нарушаются требования хорошего стиля (четырежды, если считать и тот факт, что программа написана на языке машины). Здесь применяется адресация по абсолютному смещению в области памяти DPSREP, явно указывается длина поля и используется «встроенная» прямо в текст программы константа X'0009'.

В исполняемом тексте программы не должно быть абсолютных констант, за исключением таких общеупотребительных значений, как единица и нуль. Для констант надо вводить символические имена и использовать их в программе. Это позволит читателю, читая словарь перекрестных ссылок, легко определить, где употреблены конкретные значения.

Пример неправильного подхода к решению этой проблемы содержится в одном из модулей системы OS/360 IBM. Стандарт для модулей на языке машины в OS/360 требует использования символических имен для общих регистров (например, R6 для регистра 6), так что листинг словаря перекрестных ссылок показывает, в каком операторе какой регистр использовался. Один из программистов, вначале использовавший это соглашение, затем решил изменить распределение регистров, уже закончив программирование модуля, после чего в модуле появился фрагмент:

```
R6 EQU 5
R7 EQU 4
R4 EQU 4
```

Нет нужды говорить, что теперь понять этот модуль совершенно невозможно.

Никогда не используйте несколько имен для одной области памяти. Использование операторов EQUIVALENCE в Фортране, REDEFINES в Коболе или атрибута DEFINED в PL/1 относится к числу приемов, наиболее сильно запутывающих программу.

Никогда не используйте переменную более чем для одной цели. Распространенный прием экономии лишнего слова памяти или лиш-

него оператора DECLARE состоит в повторном использовании переменной в различных целях. Программист вполне может решить: «Я закончил работать с TIME для расчетов времени, поэтому теперь буду использовать эту переменную как промежуточную при вычислении даты». Такая практика увеличивает шансы внесения ошибок при модификации программы.

Никогда не используйте особые значения переменной с особым смыслом. В определении параметров подпрограмм часто можно увидеть комментарии вроде такого: «ДЛСТРОКИ — это число символов во входной строке, причем ДЛСТРОКИ=0 означает ошибку при вводе». Это неоднозначное употребление параметра часто приводит к двусмысленным ситуациям и иногда затрудняет изменение программы. Чтобы передавать код ошибки, следует определить отдельный параметр.

Пишите модули, управляемые таблицами. При программировании модулей, реализующих сложный процесс принятия решений, эти решения следует описывать таблицей, а не встраивать в текст программы. Это сокращает и «обобщает» программу, а также значительно облегчает модификации.

Будьте осторожны с двоичной машиной. Двоичные машины обладают тем свойством, что числа с плавающей точкой в них представляются приближенными значениями. В такой машине умножение 10.0 на 0.1 редко дает в результате 1.0. Программист должен остерегаться этого свойства, особенно при попытке сравнивать два числа с плавающей точкой.

Будьте осторожны с действиями над целыми. Следует быть внимательным при умножении или делении целых чисел. Если I — целочисленная переменная, то ответ на вопрос, равны ли между собой I и 2*I/2, зависит от того, четно I или нет и что́ в сгенерированной компилятором программе будет выполняться раньше: умножение или деление.

Избегайте операций со значениями смешанных типов. Программисту не следует употреблять вперемешку переменные различных типов (например, двоичные, десятичные, плавающие) или различной точности в одном выражении; компилятор может выполнить некоторые неожиданные преобразования. Такие трюки, как сложение строки символов с целочисленной переменной, нельзя делать никогда. Необходима осторожность при употреблении констант, поскольку компилятор может приписать им по умолчанию атрибуты, требующие неожиданных преобразований данных. Например, следующий цикл на PL/1

```
DCL I FIXED BIN(15), N FIXED DEC(5);
N=10;
DO I = 1 TO N/2;
...
END;
```

не будет выполнен ни разу вследствие правил преобразования значения выражения N/2 к типу BIN при сравнении с I, которое определено как двоичное. Согласно принятым в некоторых компиляторах правилам, этот оператор семантически эквивалентен оператору

```
DO I=1 TO 0;
```

СТРУКТУРА ПРОГРАММЫ

Остальные правила «хорошего стиля» программирования касаются структуры программы.

Избегайте кратных END. Хорошее правило при программировании на PL/1 — предусматривать отдельный END для каждого оператора DO. Это позволяет компилятору обнаружить некоторые ошибки, а читателю помогает понять подразумеваемую последовательность выполнения. Кратные (с указанием меток) операторы END, такие, как

```
END AMODULE;
```

использовать не следует, поскольку компилятор будет считать, что все недостающие END должны быть вставлены непосредственно перед этим кратным END.

Предусматривайте ELSE для каждого THEN. В условных предложениях должно быть поровну THEN и ELSE. Даже если не нужно ничего делать в случае ELSE, следует предусмотреть пустой оператор. Это подскажет читателю, что случай ELSE также рассматривался, и поможет понять последовательность действий.

Отметим, что этому правилу не обязательно следовать, если будет принято высказанное в гл. 8 предложение полностью отказаться от вложенных условных предложений и конструкции ELSE. Важно помнить о необходимости быть последовательным; если в модуле имеется хоть одно ELSE, значит, во всех условных предложениях должны быть ELSE.

Выполняйте исчерпывающие проверки. При анализе входного параметра, ожидаемое значение которого должно быть 1, 2 или 3,

не следует предполагать, что его значение равно 3, если оно не равно ни 1, ни 2.

Не пишите изменяющих самих себя программ. Языки высокого уровня почти исключили эту практику. Однако такие конструкции, как переменные типа метки в PL/1 и оператор ALTER Кобола, позволяют изменять оператор GO TO, и в таком качестве их нужно избегать [1]).

Будьте осторожны с внутренними процедурами. Если программист решает использовать внутренние процедуры в таком языке, как PL/1, он должен быть внимательным в отношении правил определения областей доступности имен, которые позволяют ссылаться на внешние процедуры. Перечитайте раздел о сцеплении модулей в гл. 6 и примените эти правила к связи с внутренними процедурами.

По возможности используйте рекурсию. Рекурсивные модули — простой путь решения многих сложных вычислительных задач. Чтобы научиться мыслить рекурсивно, требуются определенные усилия, но после более близкого знакомства с этой концепцией окружающая ее тайна исчезает. Рекурсия желательна при обработке структур данных, определенных рекурсивно, например деревьев, представляющих перечень необходимых материалов, или сложных списковых структур, графов или решеток, для которых не известны или изменяются длина и глубина. Источник силы рекурсии состоит в том, что она снимает с программиста бремя забот об управлении памятью и превращает «перебор с возвратом» (т. е. возврат по дереву и затем проход вниз по другой ветви) в совсем простой процесс.

Рекурсию, однако, не следует применять там, где вполне достаточно простой итерации. Например, рекурсивно определенная математическая функция факториал (X!=X*(X—1)!) часто используется для иллюстрации рекурсивных методов программирования. Поскольку, однако, при ее вычислении не требуется перебора с возвратом, то проще всего запрограммировать эту функцию с помощью итерации (цикла DO), а не рекурсивной процедуры.

Между стилем и языком программирования имеется тесная связь, поэтому мы рекомендуем читателю обратить внимание на гл. 15. Кроме того, книги Ледгарда [4, 5] являются дополнительными источниками идей, касающихся стиля программирования, а исследование Вейсмана [6] — один из немногих экспериментов по статистическому исследованию стиля программирования.

---

1) Переменные типа метки позволяют просто моделировать конструкцию CASE (ВЫБОР), и их использование для этой цели вполне оправдано.— *Прим. ред.*

ЛИТЕРАТУРА

1. Kernighan B. W., Plauger P. J. The Elements of Programming Style. New York: McGraw-Hill, 1974.
2. Weinberg G. M. The Psychology of Computer Programming. New York Van Nostrand Reinhold, 1971.
3. Myers G. J. Reliable Software Trough Composite Design. New York: Petrocelli/ Charter, 1975.
4. Ledgard H. F. Programming Proverbs. Rochelle Park, N. J.: Hayden, 1975.
5. Ledgard H. F. Programming Proverbs for FORTRAN Programmers. Rochelle Park, N. J.: Hayden, 1975.
6. Weissman L. M. A Methodology for Studying the Psychological Complexity of Computer Programs, CSRG-37. University of Toronto Computer Systems Research Group, Toronto, Canada, 1974.

ЧАСТЬ 3

Тестирование программного обеспечения

ГЛАВА 10

Принципы тестирования

Многие организации, занимающиеся созданием программного обеспечения, до 50% средств, выделенных на разработку программ, тратят на тестирование, что составляет миллиарды долларов по всему миру в целом. И все же, несмотря на громадные капиталовложения, знаний о сути тестирования явно не хватает и большинство программных продуктов неприемлемо ненадежно даже после «основательного тестирования».

О состоянии дел лучше всего свидетельствует тот факт, что большинство людей, работающих в области обработки данных, даже не может правильно определить слово «тестирование», и это на самом деле главная причина неудач. Если попросить любого профессионала определить слово «тестирование» либо открыть (как правило, слишком краткую) главу о тестировании любого учебника программирования, то, скорее всего, можно встретить такое определение: «Тестирование — процесс, подтверждающий правильность программы и демонстрирующий, что ошибок в программе нет.» Основной недостаток подобного определения заключается в том, что оно совершенно неправильно; фактически это почти определение антонима слова «тестирование». Читатель с некоторым опытом программирования уже, вероятно, понимает, что невозможно продемонстрировать отсутствие ошибок в программе. Поэтому определение описывает невыполнимую задачу, а так как тестирование зачастую все же выполняется с успехом, по крайней мере с некоторым успехом, то такое определение логически некорректно. Правильное определение тестирования таково:

Тестирование — процесс выполнения программы с намерением найти ошибки.

Тестирование оказывается довольно необычным процессом (вот почему оно и считается трудным), так как этот процесс разрушительный. Ведь цель проверяющего (тестовика) — заставить программу сбиться. Он доволен, если это ему удается; если же программа на его тесте не сбивается, он не удовлетворен.

Невозможно гарантировать отсутствие ошибок в нетривиальной программе; в лучшем случае можно попытаться показать наличие ошибок. Если программа правильно ведет себя для солидного набора тестов, нет оснований утверждать, что в ней нет ошибок; со всей определенностью можно лишь утверждать, что не известно, когда эта программа не работает. Конечно, если есть причины считать данный набор тестов способным с большой вероятностью обнаружить все возможные ошибки, то можно говорить о некотором уровне уверенности в правильности программы, устанавливаемом этими тестами.

Не следует относиться к нашему анализу определения тестирования как к простой игре словами. Отмеченное тонкое различие между определениями — самое важное, что нужно усвоить о тестировании. Ведь первое определение не просто описывает невыполнимую задачу, но еще и оказывает нежелательное влияние на отношение программиста к тестированию. Если задача тестовика — продемонстрировать, что в программе нет ошибок, подсознание побуждает его выбирать те тесты, которые с большой вероятностью будут выполняться правильно, и блокирует любые мысли о выполнении «хитрых» тестов, в отношении которых он предчувствует неприятности. Психологические эксперименты показывают, что большинство людей, поставив цель (например, показать, что ошибок нет), ориентируется в своей деятельности на достижение этой цели. Тестовик подсознательно не позволит себе действовать против цели, т. е. подготовить тест, который выявил бы одну из оставшихся в программе ошибок. Поскольку мы все признаем, что совершенство в проектировании и кодировании любой программы недостижимо и поэтому каждая программа содержит некоторое количество ошибок, самым плодотворным применением тестирования будет найти некоторые из них. Если мы хотим добиться этого и избежать психологического барьера, мешающего нам действовать против поставленной цели, наша цель должна состоять в том, чтобы найти как можно больше ошибок. Сформулируем основополагающий вывод:

Если ваша цель — показать отсутствие ошибок, вы их найдете не слишком много. Если же ваша цель — показать наличие ошибок, вы найдете значительную их часть.

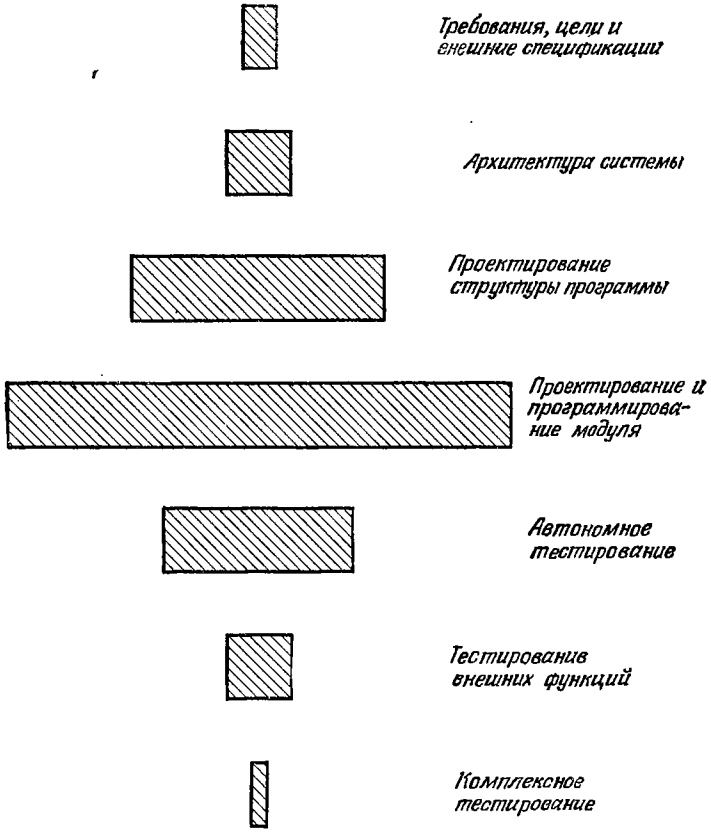


Рис. 10.1. Распределение знаний в 70-е годы.

О тестировании говорить довольно трудно, поскольку, хотя оно и способствует повышению надежности программного обеспечения, его значение ограниченно. Надежность невозможно внести в программу в результате тестирования, она определяется правильностью этапов проектирования. Наилучшее решение проблемы надежности — с самого начала не допускать ошибок в программе. Однако вероятность того, что удастся безупречно спроектировать большую программу, бесконечно мала. Роль тестирования состоит как раз в том, чтобы определить местонахождение немногочисленных ошибок, оставшихся в хорошо спроектированной программе. Попытки с помощью тестирования достичь надежности плохо спроектированной программы совершенно бесплодны.

Еще одна причина, по которой трудно говорить о тестировании,— это тот факт, что о нем известно очень немногое. Если се-

годня мы располагаем 5% тех знаний о проектировании и собственно программировании (кодировании), которые будут у нас к 2000 г., то о тестировании нам известно менее 1%. Кроме того, как показывает рис. 10.1, бóльшая часть сегодняшних знаний концентрируется вокруг «внутренних» процессов разработки программного обеспечения. Знания в области проектирования, кодирования и тестирования отдельных модулей довольно велики, в то время как о процессах начальных (требования, цели, внешнее проектирование) и заключительных (например, комплексное тестирование системы) известно мало.

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Хотя в тестировании можно выделить несколько различных процессов, такие термины, как тестирование, отладка, доказательство, контроль и испытание, часто используются как синонимы и, к сожалению, для разных людей имеют разный смысл. Хотя стандартных, общепринятых определений этих терминов нет, попытка сформулировать их была предпринята на симпозиуме по тестированию программ, организованном ACM [1]. Нашу классификацию различных форм тестирования мы начнем с того, что дадим эти определения, слегка их дополнив и расширив их список.

Тестирование (testing), как мы уже выяснили,— процесс выполнения программы (или части программы) с намерением (или целью) найти ошибки.

Доказательство (proof) — попытка найти ошибки в программе безотносительно к внешней для программы среде. Большинство методов доказательства предполагает формулировку утверждений о поведении программы и затем вывод и доказательство математических теорем о правильности программы. Доказательства могут рассматриваться как форма тестирования, хотя они и не предполагают прямого выполнения программы. Многие исследователи считают доказательство альтернативой тестированию — взгляд во многом ошибочный; более подробно это обсуждается в гл. 17.

Контроль (verification) — попытка найти ошибки, выполняя программу в тестовой, или моделируемой, среде.

Испытание (validation) — попытка найти ошибки, выполняя программу в заданной реальной среде.

Аттестация (certification) — авторитетное подтверждение правительности программы, аналогичное аттестации электротехнического оборудования Underwriters Laboratories. При тестировании с целью аттестации выполняется сравнение с некоторым заранее определенным стандартом. Ко времени написания книги было предпринято несколько попыток установить процедуру аттестации,

например процедуры аттестации компиляторов Кобола и математических программ Национальным бюро стандартов США и Федеральной службой тестирования компиляторов Кобола [2, 3] и аттестации математического программного обеспечения NATS [4].

Отладка (debugging) не является разновидностью тестирования. Хотя слова «отладка» и «тестирование» часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование — деятельность, направленная на обнаружение ошибок; отладка направлена на установление точной природы известной ошибки, а затем — на исправление этой ошибки. Эти два вида деятельности связаны — результаты тестирования являются исходными данными для отладки.

Эти определения представляют один взгляд на тестирование — со стороны среды, на которую оно опирается. Другой ряд определений, приведенный ниже, охватывает вторую сторону тестирования: типы ошибок, которые предполагается обнаружить, и стандарты, с которыми сопоставляются тестируемые программы.

Тестирование модуля, или автономное тестирование (module testing, unit testing) — контроль отдельного программного модуля, обычно в изолированной среде (т. е. изолированно от всех остальных модулей). Тестирование модуля иногда включает также математическое доказательство.

Тестирование сопряжений (integration testing) — контроль сопряжений между частями системы (модулями, компонентами, подсистемами).

Тестирование внешних функций (external function testing) — контроль внешнего поведения системы, определенного внешними спецификациями.

Комплексное тестирование (system testing) — контроль и/или испытание системы по отношению к исходным целям. Комплексное тестирование является процессом контроля, если оно выполняется в моделируемой среде, и процессом испытания, если выполняется в среде реальной, жизненной.

Тестирование приемлемости (acceptance testing) — проверка соответствия программы требованиям пользователя.

Тестирование настройки (installation testing) — проверка соответствия каждого конкретного варианта установки системы с целью выявить любые ошибки, возникшие в процессе настройки системы.

Отношения между этими типами тестов и проектной документацией, на которой основывается тест, показаны на рис. 10.2.

Требования

Цели

Внешние спецификации

Архитектура системы

Структура программы

Внешние спецификации модуля

Логика модуля

Автономный тест

Тест сопряжения

Тест функции

Комплексный тест

Тест приемлемости

Тест настройки

Рис. 10.2. Процессы тестирования и их связь с процессами проектирования.

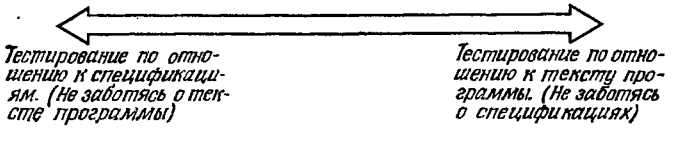


Рис. 10.3. Спектр подходов к проектированию тестов.

ФИЛОСОФИЯ ТЕСТИРОВАНИЯ

Тестирование программного обеспечения охватывает целый ряд видов деятельности, весьма аналогичный последовательности процессов разработки программного обеспечения. Сюда входят постановка задачи для теста, проектирование, написание тестов, тестирование тестов и, наконец, выполнение тестов и изучение результатов тестирования. Решающую роль играет проектирование теста. Возможен целый спектр подходов к выработке философии, или стратегии проектирования тестов, изображенный на рис. 10.3. Чтобы ориентироваться в стратегиях проектирования тестов, стоит рассмотреть два крайних подхода, находящихся на границах спектра. Следует отметить также, что многие из тех, кто работает в этой области, часто бросаются в одну или другую крайность.

Сторонник (или сторонница) подхода, соответствующего левой границе спектра, проектирует свои тесты, исследуя внешние спецификации или спецификации сопряжения программы или модуля, которые он тестирует. Программу он рассматривает как черный ящик. Позиция его такова: «Меня не интересует, как выглядит эта программа и выполнил ли я все команды или все пути. Я буду удовлетворен, если программа будет вести себя так, как указано в спецификациях». Его идеал — проверить все возможные комбинации и значения на входе.

Приверженец подхода, соответствующего другому концу спектра, проектирует свои тесты, изучая логику программы. Он начинает с того, что стремится подготовить достаточное число тестов для того, чтобы каждая команда была выполнена по крайней мере один раз. Если он немного более искушен, то проектирует тесты так, чтобы каждая команда условного перехода выполнялась в каждом направлении хотя бы раз. Его идеал — проверить каждый путь, каждую ветвь алгоритма. При этом его совсем (или почти совсем) не интересуют спецификации.

Ни одна из этих крайностей не является хорошей стратегией. Читатель, однако, уже, вероятно, заметил, что первая из них, а именно та, в соответствии с которой программа рассматривается как черный ящик, предпочтительней. К сожалению, она страдает тем недостатком, что совершенно неосуществима. Рассмотрим попытку тестирования тривиальной программы, получающей на входе три числа и вычисляющей их среднее арифметическое. Тестирование этой программы для всех значений входных данных невозможно. Даже для машины с относительно низкой точностью вычислений количество тестов исчислялось бы миллиардами. Даже имей мы вычислительную мощность, достаточную для выполнения всех тестов в разумное время, мы потратили бы на несколько порядков больше времени для того, чтобы эти тесты подготовить, а затем проверить. Такие программы, как системы реального времени, опе-

рационные системы и программы управления данными, которые сохраняют «память» о предыдущих входных данных, еще хуже. Нам потребовалось бы тестировать программу не только для каждого входного значения, но и для каждой последовательности, каждой комбинации входных данных. Поэтому исчерпывающее тестирование для всех входных данных любой разумной программы неосуществимо.

Эти рассуждения приводят ко второму фундаментальному принципу тестирования: тестирование — *проблема в значительной степени экономическая*. Поскольку исчерпывающее тестирование невозможно, мы должны ограничиться чем-то меньшим. Каждый тест должен давать максимальную отдачу по сравнению с нашими затратами. Эта отдача измеряется вероятностью того, что тест выявит не обнаруженную прежде ошибку. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста. Считая, что затраты ограничены бюджетом и графиком, можно утверждать, что искусство тестирования, по существу, представляет собой искусство отбора тестов с максимальной отдачей. Более того, каждый тест должен быть представителем некоторого класса входных значений, так чтобы его правильное выполнение создавало у нас некоторую убежденность в том, что для определенного класса входных данных программа будет выполняться правильно. Это обычно требует некоторого знания алгоритма и структуры программы, и мы, таким образом, смещаемся к правому концу спектра.

Миф о тестировании путей

Поскольку левый конец спектра недостижим, логично сделать следующий шаг и исследовать правый конец. Эта другая крайность, когда преследуется цель тестировать все выполнимые пути, оказывается, еще хуже. Она не только недостижима (здесь я буду считать «недостижимой» любую цель, для достижения которой требуется более 100 лет), но даже, будь она достигнута, такого тестирования все-таки было бы недостаточно.

На рис. 10.4 изображены возможные пути небольшого программного модуля. Кружками представлены последовательные сегменты, а стрелками — передачи управления (с помощью развилок или циклов). Модуль состоит из цикла, выполняемого от 0 до 10 раз, за которым следует развилка и за ней — второй цикл. В каждом цикле имеется ряд вложенных развилок, как это показано на рисунке. Предполагая, что каждое решение полностью независимо от других (очевидно, это предположение описывает наихудший случай), можно оценить число различных путей в модуле примерно как 10^{18}. Чтобы лучше представить себе это число, укажем, что возраст Вселенной оценивается в 4×10^{17} секунд. В качестве примера реальной ситуации отметим, что выполнение всех возможных путей

для программного обеспечения системы наведения снарядов TITAN потребовало бы 60 000 часов машинного времени [5]. Таким образом, даже при том, что число путей в программе существенно меньше числа комбинаций входных данных, оно все-таки остается астрономическим.

Предположим на минуту, что мы построили вычислительную машину, которая вопреки всем известным законам физики может выполнить все пути, изображенные на рис. 10.4, за время порядка года. Перед нами по-прежнему стоит проблема подготовки и проверки результатов 10^{18} тестов. Предположим, что мы сможем разрешить и эту проблему (я даже не могу вообразить, каким образом). Зададим тогда вопрос: достаточно ли, удовлетворительно ли мы тестировали этот модуль? Ответ: определенно *нет*.

Самое очевидное основание для такого ответа состоит в том, что выполнение всех путей не гарантирует соответствия программы ее спецификациям. Если требовалось, чтобы программист написал программу для вычисления кубического корня, а он сделал программу для квадратного, его программа будет совершенно неправильной, даже если он проверил все пути. Вторая, и более реалистическая, проблема — отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая явно специфицированная функция, как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование путей не даст информации об отсутствующих путях. Третью проблему представляет собой *чувствительность к*

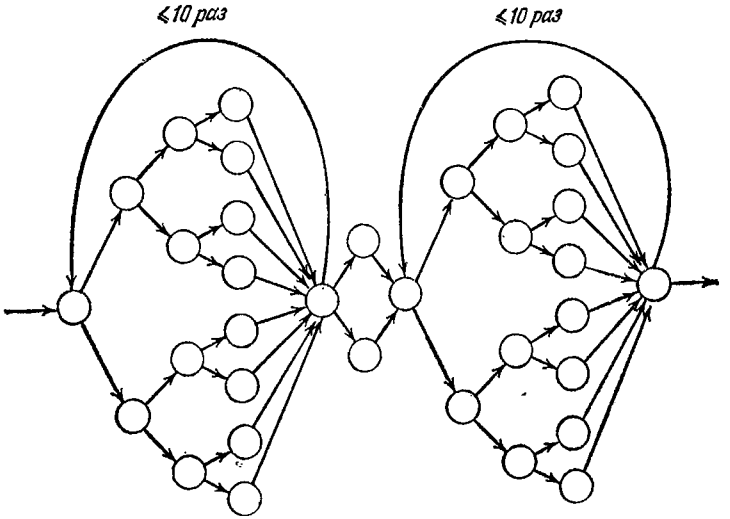


Рис. 10.4. Возможные пути небольшой программы.

данным. Путь может выполняться правильно для одних входных данных и неправильно — для других. Рассмотрим тривиальный пример, когда программисту требуется написать подпрограмму для определения того, равны ли три числа между собой. Он пишет:

```
IF(A+B+C)/3=A
```

и проверка будет правильно выполняться для большинства, но не для всех значений A, B и C (ошибка возникает только в том случае, если из двух значений (B или C) одно больше, а другое настолько же меньше A). Если программист концентрирует свое внимание только на тестировании путей, нет оснований верить, что он обнаружит эту ошибку.

Искусство проектирования тестов выглядит пока, вероятно, бледно; к сожалению, таковым оно в действительности и является. Мы убедились, что обе крайности неудовлетворительны. Чтобы построить разумную и экономичную стратегию, мы используем элементы каждой из них, выбирая точку внутри спектра несколько ближе к левому его концу. Об этой стратегии речь пойдет в гл. 11.

ИНТЕГРАЦИЯ МОДУЛЕЙ

Вторым по важности аспектом тестирования (после проектирования тестов) является последовательность слияния всех модулей в систему или программу. Эта сторона вопроса обычно не получает достаточного внимания и часто рассматривается слишком поздно. Выбор этой последовательности, однако, является одним из самых жизненно важных решений, принимаемых на этапе тестирования, поскольку он определяет форму, в которой записываются тесты, типы необходимых инструментов тестирования, последовательность программирования модулей, а также тщательность и экономичность всего этапа тестирования. По этой причине такое решение должно приниматься на уровне проекта в целом и на достаточно ранней его стадии.

Имеется большой выбор возможных подходов, которые могут быть использованы для слияния модулей в более крупные единицы. В большинстве своем они могут рассматриваться как варианты шести основных подходов, описанных в следующих шести разделах. Сразу же за ними идет раздел, где предложенные подходы сравниваются по их влиянию на надежность программного обеспечения.

ВОСХОДЯЩЕЕ ТЕСТИРОВАНИЕ

При восходящем подходе программа собирается и тестируется снизу вверх. Только модули самого нижнего уровня («терминальные» модули; модули, не вызывающие других модулей) тестируются изолированно, автономно. После того как тестирование этих моду-

лей завершено, вызов их должен быть так же надежен, как вызов встроенной функции языка или оператор присваивания. Затем тестируются модули, непосредственно вызывающие уже проверенные. Эти модули более высокого уровня тестируются не автономно, а вместе с уже проверенными модулями более низкого уровня. Процесс повторяется до тех пор, пока не будет достигнута вершина. Здесь завершаются и тестирование модулей, и тестирование сопряжений программы.

На рис. 10.5 изображена структура модулей загрузчика, спроектированного в части 2. Для краткости мы пометим модули буквами. Загрузчик — программа необычная в том смысле, что у него только один терминальный модуль — модуль H. Первый шаг состоит в автономном тестировании модуля H. Затем можно параллельно тестировать модули C и G, используя уже проверенный модуль

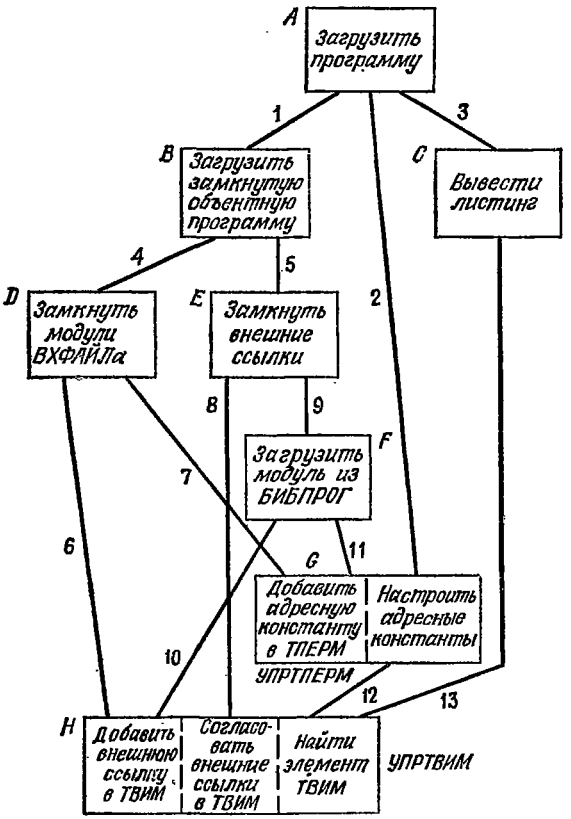


Рис. 10.5. Структура загрузчика,

Примечания

1. «Загрузить модули ВХФАЙЛа» использует функции операционной системы GET (из ВХФАЙЛа) и GETMAIN (для выделения памяти).
2. «Загрузить модуль из БИБПРОГ» использует функции системы FIND и GET (из БИБПРОГ) и GETMAIN.
3. «Вывести листинг» использует функцию системы PUT (в ВЫХФАЙЛ).
4. Элемент ТВИМ содержит имя, тип (модуль, вход или внешн. ссылка) и адрес выделенного поля памяти.
5. Элемент ТПЕРМ содержит номер соответствующего элемента ТВИМ и адрес адресной константы в памяти.

	Вход	Выход
1	———————	ТВИМ, ТПЕРМ, ССООБ
2	ТВИМ, ТПЕРМ	КО
3	ТВИМ, ССООБ	КО
4	———————	ТВИМ, ТПЕРМ, ССООБ
5	ТВИМ, ТПЕРМ	ТВИМ, ТПЕРМ, ССООБ
6	ТВИМ, имя, тип, адрес	ТВИМ, номер эл-та, КО
7	ТПЕРМ, номер эл-та, адрес адр. константы	ТПЕРМ, КО
8	ТВИМ	ТВИМ, флажок ГОТОВО, имя незамкн. внешн. ссылки
9	ТВИМ, ТПЕРМ, имя модуля	ТВИМ, ТПЕРМ, ССООБ
10	совпадает с	сопряжением 6
11	совпадает с	сопряжением 7
12	ТВИМ, имя элемента	Имя, тип, адрес, КО
13	совпадает с	сопряжением 12

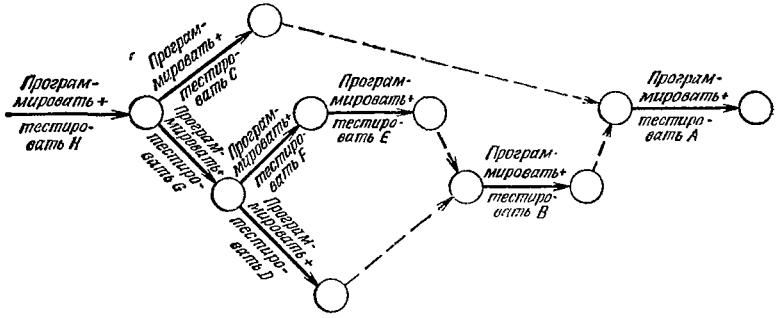


Рис. 10.6. Диаграмма PERT для нисходящего тестирования.

H. Далее тестируются модули D и F, затем E, затем B и наконец A. Так как порядок тестирования тесно связан со структурой программы, для целей планирования легко можно нарисовать PERT-диаграмму (рис. 10.6).

При восходящем тестировании для каждого модуля необходим *драйвер:* нужно подавать тесты в соответствии с сопряжением тестируемого модуля. Одно из возможных решений — написать для каждого модуля небольшую ведущую программу. Тестовые данные представляются как «встроенные» непосредственно в эту программу переменные и структуры данных, и она многократно вызывает тестируемый модуль, с каждым вызовом передавая ему новые тестовые данные. Имеется и лучшее решение: воспользоваться программой тестирования модулей — это инструмент тестирования, позволяющий описывать тесты на специальном языке и избавляющий от необходимости писать драйверы. В гл. 11 упоминается несколько коммерчески доступных программ тестирования модулей.

НИСХОДЯЩЕЕ ТЕСТИРОВАНИЕ

Нисходящее тестирование (называемое также нисходящей разработкой [6]) не является полной противоположностью восходящему, но в первом приближении может рассматриваться как таковое. При нисходящем подходе программа собирается и тестируется сверху вниз. Изолировано тестируется только головной модуль. После того как тестирование этого модуля завершено, с ним соединяются (например, редактором связей) один за другим модули, непосредственно вызываемые им, и тестируется полученная комбинация. Процесс повторяется до тех пор, пока не будут собраны и проверены все модули.

При этом подходе немедленно возникает два вопроса: что делать, когда тестируемый модуль вызывает модуль более низкого уровня (которого в данный момент еще не существует), и как пода-

ются тестовые данные. Ответ на первый вопрос состоит в том, что для имитации функций недостающих модулей программируются модули-*«заглушки»*, которые моделируют функции отсутствующих модулей. Фраза «просто напишите заглушку» часто встречается в описании этого подхода, но она способна ввести в заблуждение, поскольку задача написания «заглушки» может оказаться трудной. Ведь заглушка редко сводится просто к оператору RETURN, поскольку вызывающий модуль обычно ожидает от нее выходных параметров. В таких случаях в заглушку встраивают фиксированные выходные данные, которые она всегда и возвращает. Это иногда оказывается неприемлемым, так как вызывающий модуль может рассчитывать, что результат вызова зависит от входных данных. Поэтому в некоторых случаях заглушка должна быть довольно изощренной, приближаясь по сложности к модулю, который она пытается моделировать.

Интересен и второй вопрос: в какой форме готовятся тестовые данные и как они передаются программе? Если бы головной модуль содержал все нужные операции ввода и вывода, ответ был бы прост: тесты пишутся в виде обычных для пользователей внешних данных и передаются программе через выделенные ей устройства ввода. Так, однако, случается редко. В хорошо спроектированной программе физические операции ввода-вывода выполняются на нижних уровнях структуры, поскольку физический ввод-вывод — абстракция довольно низкого уровня. Поэтому для того, чтобы решить проблему экономически эффективно, модули добавляются не в строго нисходящей последовательности (все модули одного горизонтального уровня, затем модули следующего уровня), а таким образом, чтобы *обеспечить функционирование операций физического ввода-вывода как можно быстрее*. Когда эта цель достигнута, нисходящее тестирование получает значительное преимущество: все дальнейшие тесты готовятся в той же форме, которая рассчитана на пользователя.

Остается еще один вопрос: в какой форме пишутся тесты до тех пор, пока не будет достигнута эта цель? Ответ: они включаются в некоторые из заглушек. Покажем, как это делается, на примере нашего загрузчика.

Прежде всего нужно решить, как обеспечить функционирование операций ввода-вывода,— ключевой вопрос при выборе последовательности сборки модулей. В программе загрузчика, изображенной на рис. 10.5, входные данные читаются модулями D и E, а выходные данные вырабатываются модулем C. Все начинается с программирования и тестирования модуля A. Модуль A на самом деле мал, он просто последовательно вызывает три других модуля и может также добавить одно или несколько сообщений в список сообщений об ошибках, если таковые обнаружит модуль G. Для тестирования модуля A должно быть написано три заглушки. Заглушка вместо модуля B может просто возвращать запасенные заранее

ТВИМ, ТПЕРМ и список сообщений. Заглушка вместо входа модуля G, занимающегося настройкой адресов, может содержать лишь оператор возврата, а вместо модуля С — печатать свои входные параметры. Поскольку следует тестировать и обработку ошибок в модуле А, нужны одна или несколько заглушек для функции настройки адресов — при вызове они должны возвращать код соответствующей ошибки. Это указывает на один из недостатков использования заглушек. При тестировании различных условий в программах часто требуется, чтобы модуль выдавал разные наборы выходных данных. Если одна из заглушек вызывается только один раз за все время выполнения программы, может понадобиться построить несколько разных вариантов заглушек вместо одного модуля и затем заменять их при выполнении каждого теста (имеется и другой вариант, когда заглушка читает данные из внешнего файла).

Следующий шаг при тестировании загрузчика — как можно скорее добраться до модулей ввода-вывода. Заглушку С можно убрать и заменить ее настоящим модулем С. Теперь этому модулю нужна заглушка, имитирующая функцию поиска элемента, выполняемую модулем Н. Такая заглушка могла бы просто возвращать запасенный набор выходных параметров, но это помешало бы нам использовать выходной листинг, вырабатываемый модулем С, для проверки результатов будущих тестов. Вот почему в действительности она должна выполнять ту функцию, которую пытается имитировать. Теперь модули А и С и заглушки для В, G и Н тестируются вместе. Отметим, что тесты — это в действительности заранее запасенные результаты, возвращаемые заглушкой В. Поскольку В вызывается только раз, при тестировании модуля С может понадобиться много вариантов заглушек, представляющих модуль В для разных версий ТВИМ, ТПЕРМ и списков сообщений.

На следующем шаге добавляем модуль В. Заглушки, использовавшиеся для его моделирования, выбрасываются, а для модулей D и Е — добавляются. После того как этот этап тестирования закончится, вместо заглушки D программируется и подключается настоящий модуль. Теперь необходимо добавить заглушки для используемых им входов модулей G и Н. По-видимому, нет простого способа имитировать функции этих модулей, так что про заглушки для них лучше просто забыть, а вместо этого использовать сами модули.

Теперь операции ввода-вывода работают, так что можно готовить тесты в форме записей в ВХФАЙЛе (или, проще, для генерации тестов можно использовать компилятор). Теперь скелет загрузчика функционирует, хотя все еще не настраиваются адресные константы и не замыкаются никакие незамкнутые внешние ссылки. Осталось подключить эти функции к загрузчику.

Нисходящий метод имеет как достоинства, так и недостатки по сравнению с восходящим. Самое значительное достоинство —

в том, что этот метод совмещает тестирование модуля, тестирование сопряжений и частично тестирование внешних функций. С этим же связано другое его достоинство — когда модули ввода-вывода уже подключены, тесты можно готовить в удобном виде. Нисходящий подход выгоден также в том случае, когда есть сомнения относительно *осуществимости* программы в целом или если в проекте программы могут оказаться серьезные дефекты.

Преимуществом нисходящего подхода очень часто считают отсутствие необходимости в драйверах; вместо драйверов вам просто следует написать «заглушки». Как читатель сейчас уже, вероятно, понимает, это преимущество спорно.

Нисходящий метод тестирования имеет, к сожалению, некоторые недостатки. Основным из них является тот, что модуль редко тестируется досконально сразу после его подключения. Дело в том, что основательное тестирование некоторых модулей может потребовать крайне изощренных заглушек. Программист часто решает не тратить массу времени на их программирование, а вместо этого пишет простые заглушки и проверяет лишь часть условий в модуле. Он, конечно, собирается вернуться и закончить тестирование рассматриваемого модуля позже, когда уберет заглушки. Такой план тестирования определенно не лучшее решение, поскольку об отложенных условиях часто забывают.

Второй тонкий недостаток нисходящего подхода состоит в том, что он может породить веру в возможность начать программирование и тестирование верхнего уровня программы до того, как вся программа будет полностью спроектирована. Эта идея на первый взгляд кажется экономичной, но обычно дело обстоит совсем наоборот. Большинство опытных проектировщиков признает, что проектирование программы — процесс итеративный. Редко первый проект оказывается совершенным. Нормальный стиль проектирования структуры программы предполагает по окончании проектирования нижних уровней вернуться назад и подправить верхний уровень, внеся в него некоторые усовершенствования или исправляя ошибки, либо иногда даже выбросить проект и начать все сначала, потому что разработчик внезапно увидел лучший подход. Если же головная часть программы уже запрограммирована и оттестирована, то возникает серьезное сопротивление любым улучшениям ее структуры. В конечном итоге за счет таких улучшений обычно можно сэкономить больше, чем те несколько дней или недель, которые рассчитывает выиграть проектировщик, приступая к программированию слишком рано.

МОДИФИЦИРОВАННЫЙ НИСХОДЯЩИЙ МЕТОД

Нисходящий подход имеет еще один существенный недостаток, касающийся полноты тестирования. Предположим, что есть большая программа и где-то ближе к нижнему ее уровню находится модуль,

предназначенный для вычисления корней квадратного уравнения. Для заданных входных переменных A, B и C он решает уравнение

$$AX^2+BX+C=0.$$

При проектировании и программировании модуля с такой функцией всегда следует понимать, что квадратное уравнение может иметь как действительные, так и комплексные корни. Для полной реализации этой функции необходимо, чтобы результаты могли быть действительными или комплексными числами (или, если дополнительные затраты на нахождение комплексных корней не оправданы, модуль должен по крайней мере возвращать код ошибки в случае, когда входные коэффициенты задают уравнение с комплексными корнями). Предположим, что конкретный контекст, в котором используется модуль, исключает комплексные корни (т. е. вызывающие модули никогда не задают входных параметров, которые привели бы к комплексным корням). При строго нисходящем методе иногда бывает невозможно тестировать модуль для случая комплексных корней (или тестировать ошибочные условия). Можно попытаться оправдывать это тем, что, поскольку такое уравнение никогда не будет дано модулю, никого не должно заботить, работает ли он и в этих случаях. Да, это безразлично сейчас, но окажется важным в будущем, когда кто-то попытается использовать модуль в новой программе или модифицировать старую программу так, что станут возможными и комплексные корни.

Эта проблема проявляется в разнообразных формах. Применяя нисходящее тестирование в точном соответствии с предыдущим разделом, часто невозможно тестировать определенные логические условия, например ошибочные ситуации или защитные проверки. В начале спроектированного в гл. 8 модуля загрузчика были помещены четыре защитные проверки. Проверить этот фрагмент при нисходящем подходе невозможно, поскольку он работает только тогда, когда расположенный выше модуль передает недопустимые входные данные (а если разработчик включил в модуль этот фрагмент, то разумно предположить, что он захочет тестировать также и его). Нисходящий метод, кроме того, делает сложной или вообще невозможной проверку исключительных ситуаций в некотором модуле, если программа работает с ним лишь в ограниченном контексте (это означает, что модуль никогда не получит достаточно полный набор входных значений). Даже если тестирование такой ситуации в принципе осуществимо, часто бывает трудно определить, какие именно нужны тесты, если они вводятся в точке программы, удаленной от места проверки соответствующего условия.

Метод, называемый модифицированным нисходящим подходом, решает эти проблемы: требуется, чтобы каждый модуль прошел

автономное тестирование перед подключением к программе. Хотя это действительно решает все перечисленные проблемы, здесь требуются и драйверы, и заглушки для каждого модуля.

МЕТОД БОЛЬШОГО СКАЧКА

Вероятно, самый распространенный подход к интеграции модулей — метод «большого скачка». В соответствии с этим методом каждый модуль тестируется автономно. По окончании тестирования модулей они интегрируются в систему все сразу.

Метод большого скачка по сравнению с другими подходами имеет много недостатков и мало достоинств. Заглушки и драйверы необходимы для каждого модуля. Модули не интегрируются до самого последнего момента, а это означает, что в течение долгого времени серьезные ошибки в сопряжениях могут остаться необнаруженными.

Чтобы применить метод большого скачка к нашему загрузчику, следует взять все восемь модулей и тестировать их независимо. Затем все восемь модулей следует соединить редактором связей и начать тестирование всей программы. Заметим, что при восходящем и нисходящем подходах каждый раз подключается только один модуль, и, если обнаруживается ошибка, подозрение в первую очередь падает на последний добавленный модуль. Так что метод большого скачка значительно усложняет отладку.

И все же большой скачок не всегда нежелателен. Если программа мала (как, например, программа загрузчика) и хорошо спроектирована, он может оказаться приемлемым. Однако для крупных программ метод большого скачка обычно губителен.

МЕТОД САНДВИЧА

Тестирование методом сандвича представляет собой компромисс между восходящим и нисходящим подходами. Здесь делается попытка воспользоваться достоинствами обоих методов, избежав их недостатков.

При использовании этого метода одновременно начинают восходящее и нисходящее тестирование, собирая программу как снизу, так и сверху и встречаясь в конце концов где-то в середине. Точка встречи зависит от конкретной тестируемой программы и должна быть заранее определена при изучении ее структуры. Например, если разработчик может представить свою систему в виде уровня прикладных модулей, затем уровня модулей обработки запросов, затем уровня примитивных функций, то он может решить применять нисходящий метод на уровне прикладных модулей (программируя заглушки вместо модулей обработки запросов), а на остальных уровнях применить восходящий метод.

Применение метода сандвича трудно проиллюстрировать на ма-

ленькой программе, поэтому пример с загрузчиком здесь не поможет. Это разумный подход к интеграции больших программ, таких, как операционная система или пакет прикладных программ.

Метод сандвича сохраняет такое достоинство нисходящего и восходящего подходов, как начало интеграции системы на самом раннем этапе. Поскольку вершина программы вступает в строй рано, мы, как в нисходящем методе, уже на раннем этапе получаем работающий каркас программы. Поскольку нижние уровни программы создаются восходящим методом, снимаются те проблемы нисходящего метода, которые были связаны с невозможностью тестировать некоторые условия в глубине программы.

МОДИФИЦИРОВАННЫЙ МЕТОД САНДВИЧА

При тестировании методом сандвича возникает та же проблема, что и при нисходящем подходе, хотя здесь она стоит не так остро. Проблема эта в том, что невозможно досконально тестировать отдельные модули. Восходящий этап тестирования по методу сандвича решает эту проблему для модулей нижних уровней, но она может по-прежнему оставаться открытой для нижней половины верхней части программы. В модифицированном методе сандвича нижние уровни также тестируются строго снизу вверх. А модули верхних уровней сначала тестируются изолированно, а затем собираются нисходящим методом. Таким образом, модифицированный метод сандвича также представляет собой компромисс между восходящим и нисходящим подходами.

ЧТО ЛУЧШЕ?

Трудно оценить эти шесть стратегий и найти лучшую, потому что «наилучший» подход зависит от конкретной организации и конкретной программы. С точки зрения надежности программного обеспечения эти стратегии можно оценить по восьми критериям, как показано на рис. 10.7. Первый критерий — время до момента сборки модулей, поскольку это важно для обнаружения ошибок в сопряжениях и предположениях модулей о свойствах друг друга. Второй критерий — время до момента создания первых работающих «скелетных» версий программы, поскольку здесь могут проявиться главные дефекты проектирования. Третий и четвертый критерии касаются вопроса о том, необходимы ли заглушки, драйверы и другие инструменты тестирования. Пятый критерий — мера параллелизма, который возможен в начале или на ранних стадиях тестирования. Это интересный вопрос, поскольку необходимость в ресурсах (т. е. программистах) обычно достигает пика на этапах проектирования и программирования модулей. Поэтому важно, чтобы возможность

	Восходящий	Нисходящий	Модифицированный нисходящий	Метод большого скачка	Метод сандвича	Модифицированный метод сандвича
Сборка	Рано	Рано	Рано	Поздно	Рано	Рано
Время до появления работающего варианта программы	Поздно	Рано	Рано	Поздно	Рано	Рано
Нужны ли драйверы (новые программы или готовые инструменты)?	Да	Нет	Да	Да	Частично	Да
Нужны ли заглушки?	Нет	Да	Да	Да	Частично	Частично
Параллелизм в начале работы	Средний	Слабый	Средний	Высокий	Средний	Высокий
Возможность тестировать отдельные пути	Легко	Трудно	Легко	Трудно	Средне	Легко
Возможность планировать и контролировать последовательность	Легко	Трудно	Трудно	Легко	Трудно	Трудно
Неэффективность			Считается несущественной			

Рис. 10.7. Количественная оценка подходов к сборке.

параллельного тестирования появилась ближе к началу, а не концу цикла тестирования.

Шестой критерий связан с ответом на обсуждавшийся ранее вопрос: возможно ли проверить любой конкретный путь и любое условие в программе? Седьмой критерий характеризует сложность планирования, надзора и управления в процессе тестирования. Это связано с осознанием того факта, что тестирование, которым трудно управлять, часто ведет к недосмотрам и упущениям. Последний критерий оценивает степень перерасхода машинного времени: лишние прогоны ранее проверенных программ. Время от времени раздаются возражения против нисходящего подхода в связи с тем, что тестирование нижних модулей требует многократных лишних прогонов головных модулей. Однако этот критерий отмечен как несущественный. Хотя лишние прогоны действительно бывают необходимы, возможно также, что во многих случаях, которые кажутся лишними, в действительности воссоздаются несколько разные условия. Эти прогоны могут вскрыть новые ошибки, превращая таким образом недостаток в достоинство. Поскольку этот эффект недостаточно осознан, мы им пренебрегаем.

Теперь оценим наши шесть подходов с помощью перечисленных восьми критериев. Как уже говорилось, такая оценка зависит от конкретного проекта. В качестве исходного приближения для выполнения ваших собственных оценок приведен вариант очень грубой оценки. Прежде всего следует взвесить относительное влияние каждого из восьми критериев на надежность программного обеспечения. Ранняя сборка и раннее получение работающего каркаса программы, а также возможность тестировать любые конкретные условия представляются наиболее важными, поэтому им дается коэффициент 3. Сложность подготовки заглушек, а также сложность планирования и управления последовательностью тестов также важны, поэтому они получают вес 2. Третий критерий, необходимость драйверов, имеет вес 1 ввиду доступности общих инструментов тестирования. Критерий, связанный с параллелизмом работы, также имеет вес 1, потому что, хотя он, может быть, и важен по другим причинам, на надежность сильно не влияет. Восьмой критерий получает коэффициент нуль.

На рис. 10.8 показаны результаты этой оценки. В каждой графе таблицы вес берется со знаком плюс или минус либо не учитывается, в зависимости от того, благоприятно, неблагоприятно или безразлично проявляется соответствующий фактор при рассматриваемом подходе. Модифицированный метод сандвича и восходящий метод оказываются наилучшими подходами, а метод большого скачка— наихудшим. Если способ оценки оказывается близким к вашей конкретной ситуации, следует рекомендовать модифицированный метод сандвича для тестирования больших систем или программ и восходящий подход для тестирования программ малых и средних.

Вес		Восходящий	Нисходящий	Модифицированный нисходящий	Метод большого скачка	Метод сандвича	Модифицированный метод сандвича
3	Сборка	Рано +	Рано +	Рано +	Поздно —	Рано +	Рано +
3	Время до появления работающего варианта программы	Поздно —	Рано +	Рано +	Поздно —	Рано +	Рано +
1	Нужны ли драйверы (новые программы или готовые инструменты)?	Да —	Нет +	Да —	Да —	Частично	Да —
2	Нужны ли заглушки?	Нет +	Да —	Да —	Да —	Частично	Частично
1	Параллелизм в начале работы	Средний	Слабый —	Средний	Высокий +	Средний	Высокий +
3	Возможность тестировать отдельные пути	Легко +	Трудно —	Легко +	Легко +	Средне	Легко +
2	Возможность планировать и контролировать последовательность	Легко +	Трудно —	Трудно —	Легко +	Трудно —	Трудно —
0	Неэффективность			Считается несущественной			
Всего		+6	−1	+4	−3	+4	+7

Рис. 10.8. Взвешенная оценка подходов к сборке.

АКСИОМЫ ТЕСТИРОВАНИЯ

Прежде чем перейти к техническим аспектам тестирования программного обеспечения, следует обсудить некоторые из важнейших аксиом тестирования. Они приведены в настоящем разделе и являются фундаментальными принципами тестирования. Этот список неполон, еще о нескольких аксиомах говорится в соответствующих местах оставшихся глав части 3.

Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы. Эта аксиома является фундаментальным принципом тестирования, о нем говорилось в начале главы. Поскольку невозможно показать, что программа не имеет ошибок и, значит, все такие попытки бесплодны, процесс тестирования должен представлять собой попытки обнаружить в программе прежде не найденные ошибки.

Одна из самых сложных проблем при тестировании — решить, когда нужно закончить. Как уже говорилось, исчерпывающее тестирование (т. е. испытание всех входных значений) невозможно. Таким образом, при тестировании мы сталкиваемся с экономической проблемой: как выбрать конечное число тестов, которое дает максимальную отдачу (вероятность обнаружения ошибок) для данных затрат. Известно слишком много случаев, когда написанные тесты имели крайне малую вероятность обнаружения новых ошибок, в то время как довольно очевидные хорошие тесты оставались незамеченными. Эта проблема обсуждается подробнее в нескольких следующих главах.

Невозможно тестировать свою собственную программу. Ни один программист не должен пытаться тестировать свою собственную программу. Это относится ко всем формам тестирования, как к тестированию системы, так и к тестированию внешних функций и даже отдельных модулей. Многие из лежащих в основе этого утверждения причин уже рассматривались в гл. 8 при обсуждении роли чтения текста программы. Тестирование должно быть в высшей степени разрушительным процессом, но имеются глубокие психологические причины, по которым программист не может относиться к своей собственной программе как разрушитель. Дополнительное давление (например, жесткий график) на отдельного программиста или весь коллектив разработчиков проекта часто мешает программисту или всему коллективу выполнить адекватное тестирование. Более того, если модуль содержит дефекты вследствие каких-то ошибок перевода, довольно высока вероятность того, что программист допустит ту же ошибку перевода (например, неправильно интерпретирует спецификации) и при подготовке тестов. Все ошибки

в его понимании других модулей и их сопряжений также отразятся на тестах.

Тестирование всегда должна выполнять внешняя группа, которая в некотором смысле стоит особняком от программиста и проекта. Вместо того чтобы выполнять автономное тестирование модулей самостоятельно, программист должен иметь набор тестов, подготовленных разработчиком одного из модулей, вызывающих тестируемый модуль (а может быть, даже выполненных на машине и проверенных этим разработчиком). Комплексное тестирование всегда должно выполняться независимой группой, например специальным отделом обеспечения качества или группой пользователей-добровольцев.

Необходимая часть всякого теста — описание ожидаемых выходных данных или результатов. Одна из самых распространенных ошибок при тестировании состоит в том, что результаты каждого теста не прогнозируются до его выполнения. Ожидаемые результаты нужно определять заранее, чтобы не возникла ситуация, когда «глаз видит то, что хочет увидеть». Чтобы совсем исключить такую возможность, лучше разрабатывать самопроверяющиеся тесты либо пользоваться инструментами тестирования, способными автоматически сверять ожидаемые и фактические результаты.

Хотя эта аксиома чрезвычайно важна, иногда, например, при тестировании математического программного обеспечения, приходится допускать исключения. Математическое программное обеспечение обладает тем свойством, что выходные данные являются только приближением правильного результата. Это происходит из-за использования конечных вычислительных процессов вместо бесконечных математических процессов, из-за ошибок округления, связанных с конечной точностью машинной арифметики и неточного представления чисел в двоичной машине, а также ошибок из-за конечной точности представления входных данных и констант. Поэтому во многих случаях оказывается важной не абсолютная точность, а корреляция ошибок. Например, когда математическая программа возвращает массив чисел, может оказаться важным, чтобы вычисленное решение было точным решением для набора входных данных, аппроксимирующего реальные входные данные. Поэтому при тестировании математического программного обеспечения прогнозирование точных выходных данных затруднительно. Детальное обсуждение этих проблем выходит за рамки данной книги. Их обзор и библиографию по этому вопросу можно найти в работе Коди [7].

Избегайте невоспроизводимых тестов, не тестируйте «с лету». Использование диалоговых систем иногда мешает хорошему тестированию. Для того чтобы тестировать программу в пакетной системе, программист обычно должен оформить тест в виде специаль-

ной ведущей программы или в виде файла тестовых данных. В условиях диалога программист слишком часто выполняет тестирование «с лету», т. е., сидя за терминалом, задает конкретные значения и выполняет программу, чтобы «посмотреть, что получится». Это — неряшливая и по многим причинам нежелательная форма тестирования. Основной ее недостаток в том, что такие тесты мимолетны; они исчезают по окончании их выполнения. Всякий раз, когда программу понадобится тестировать повторно (например, после исправления ошибок или после модификации), придется придумывать тесты заново.

Тестирование обходится слишком дорого и без этих дополнительных расходов. Никогда не используйте тестов, которые тут же выбрасываются (если только программа не такова, что ее саму тут же надо выбросить). Более того, тесты следует документировать и хранить в такой форме, чтобы каждый мог использовать их повторно.

Готовьте тесты как для правильных, так и для неправильных входных данных. Многие программисты ориентируются в своих тестах на «разумные» условия на входе, забывая о последствиях появления непредусмотренных или ошибочных входных данных. Однако многие ошибки, которые потом неожиданно обнаруживаются в работающих программах, проявляются вследствие никак не предусмотренных действий пользователя программы. Тесты, представляющие неожиданные или неправильные входные данные, часто лучше обнаруживают ошибки, чем «правильные» тесты.

Детально изучите результаты каждого теста. Самые изощренные тесты ничего не стоят, если их результаты удостаиваются лишь беглого взгляда. Тестирование программы означает большее, нежели выполнение достаточного количества тестов; оно также предполагает изучение результатов каждого теста. «Да, я уже проверял такую ситуацию, но *такого* как-то не заметил в выдаче»,— довольно распространенная реакция программиста на обнаруженную новую ошибку.

По мере того как число ошибок, обнаруженных в некоторой компоненте программного обеспечения увеличивается, растет также относительная вероятность существования в ней необнаруженных ошибок. Этот противоречащий интуиции феномен, иллюстрируемый рис. 10.9, означает, что ошибки образуют кластеры, т. е. встречаются группами. С ростом числа ошибок, обнаруженных в компоненте программы (например, в модуле, подсистеме, функции пользователя) увеличивается также вероятность существования в этой компоненте еще не обнаруженных ошибок. Если при тестировании двух модулей в них обнаружены одна и восемь ошибок соответственно, кривая на рис. 10.9 показывает, что для модуля с восьмью ошибками вероятность того, что в нем еще есть ошибки, выше.

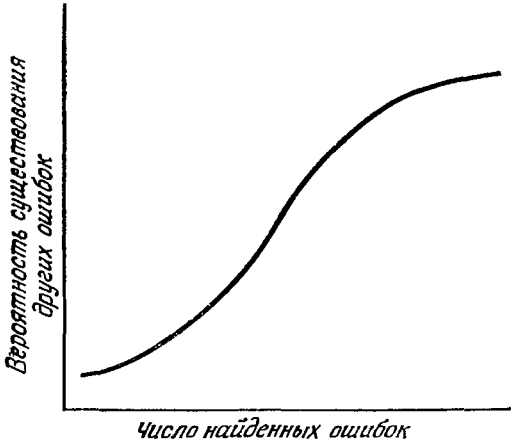


Рис. 10.9. Соотношение между обнаруженными и необнаруженными ошибками.

Этот феномен наблюдался во многих системах, в частности в операционной системе OS/360 IBM. Его понимание способно повысить качество тестирования, обеспечивая обратную связь между результатами прогона тестов и их проектированием. Если конкретная часть системы окажется при тестировании полной ошибок, для нее следует подготовить дополнительные тесты. Другими словами, проверять эту компоненту нужно тщательнее, чем другие.

Поручайте тестирование самым способным программистам. Тестирование, и в особенности проектирование тестов,— этап в разработке программного обеспечения, требующий особенно творческого подхода. К сожалению, во многих организациях на тестирование смотрят совсем не так. Его часто считают рутинной, нетворческой работой, вследствие чего коллективы, занимающиеся тестированием, укомплектованы в основном неопытными или молодыми программистами. Однако практика показывает, что более правильным было бы сделать наоборот. Проектирование тестов требует даже больше творчества, чем разработка архитектуры и проектирование программного обеспечения. Будь у меня для работы над проектом несколько творчески одаренных программистов, я бы по крайней мере половину из них направил на тестирование.

Считайте тестируемость ключевой задачей вашей разработки. Хотя «тестируемость» и не фигурировала явно в части 2, сложность тестирования программы зависит от ее структуры и качества проектирования. Несмотря на то что эта связь осознана еще недостаточно глубоко, можно утверждать, что многие характеристики хорошего проекта, рассматривавшиеся в гл. 5 и 6 (например, небольшие в

значительной степени независимые модули и независимые подсистемы), улучшают и тестируемость программы.

Проект системы должен быть таким, чтобы каждый модуль подключался к системе только один раз. Множество проблем во многих больших программных системах возникает из-за нарушения этой аксиомы. Ситуация, когда во время цикла тестирования большой системы некоторые модули приходится подключать больше десяти раз, не редкость. Каждая версия такого модуля содержит еще одну маленькую дополнительную функцию, необходимую для текущего уровня системы. Если следовать правилам, описанным в гл. 6, и проектировать небольшие модули, каждый из которых выполняет отдельную функцию, бороться с этой проблемой станет легче [1].

Никогда не изменяйте программу, чтобы облегчить ее тестирование. Часто возникает соблазн изменить программу, чтобы было легче ее тестировать. Например, программист, тестируя модуль, содержащий цикл, который должен повторяться 100 раз, меняет его так, чтобы цикл повторялся только 10 раз. Может быть, этот программист и занимается тестированием, но только другой программы.

Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей. Как уже говорилось, цикл тестирования подобен полному циклу разработки программного обеспечения. Тесты должны быть спроектированы, реализованы, проверены и, наконец, выполнены. Поэтому задачи тестирования должны быть сформулированы на каждом его этапе, например для каждого конкретного типа тестирования должны быть определены ориентиры (число пройденных путей, проверенных условных переходов и т. п.) и доля ошибок, которые должны быть обнаружены на этом этапе.

ЛИТЕРАТУРА

1. Hetzel W. C. A Definitional Framework, in W. C. Hetzel, Ed., Program Test Methods, Englewood Cliffs, N. J.: Prentice-Hall, 1973, pp. 2—10.
2. Baird G. N., Cook M. M. Experiences in COBOL Compiler Validation, Proceedings of the 1974 National Computer Conference, Montvale N. J.: AFIPS Press, 1974, pp. 417—421.
3. Sadowski W. L., Lozier D. W. A Unified Standards Approach to Algorithm Testing, in W. C. Hetzel, Ed., Program Test Methods. Englewood Cliffs, N. J.: Prentice-Hall, 1973, pp. 277—290.

---

[1] Это далеко не простая проблема. Один из подходов к ней связан с концепцией «расслоенного программирования» (см. Фуксман А. Л. Технологические аспекты создания программных систем.— М.: Статистика, 1979).— *Прим. ред.*

4. Boyle J. M., Cody W. J., Cowell W. R., Grabow B. S., Ikebe Y., Moler C. B., Smith B. T. NATS: A Collaborative Effort to Certify and Disseminate Mathematical Software, Proceedings of the 1972 ACM Annual Conference. New York: ACM, 1972, pp. 630—635.
5. Thayer R. H. Rome Air Development Center R and D Program in Computer Language Controls and Software Engineering Techniques, RADC-TR-74-80, Griffiss Air Force Base, Rome, N. Y., 1974.
6. McHenry R. C. Management Techniques for Top Down Structured Programming, FSC-73-0001, IBM Federal Systems Div., Gaithersburg, Md., 1973.
7. Cody W. J. The Evaluation of Mathematical Software, in W. C. Hetzel, Ed., Program Test Methods, Englewood Cliffs, N. J.: Prentice-Hall, 1973, pp. 121—133.

ГЛАВА 11

Тестирование модуля

В каждом из шести рассмотренных в гл. 10 подходов к тестированию и сборке внимание при тестировании сначала концентрируется на отдельном программном модуле. В случае нисходящих методов каждый тестируемый модуль подключается к уже тестированным модулям снизу, в случае восходящих — сверху. Побочным продуктом всех этих методов, кроме метода большого скачка, является тестирование сопряжений.

Цель тестирования модуля или программной компоненты — найти несоответствия между логикой и сопряжениями модуля, с одной стороны, и его внешними спецификациями (описанием функций, входных и выходных данных и внешних эффектов), с другой стороны. Компиляция модуля также должна рассматриваться как часть процесса тестирования, поскольку компилятор обнаруживает большинство синтаксических ошибок, а также некоторые семантические и логические ошибки.

ПРОЕКТИРОВАНИЕ ТЕСТА

Для иллюстрации сложности задачи проектирования тестов я часто в начале курсов по надежности программного обеспечения даю в качестве домашнего задания следующую задачу:

«У меня есть небольшая программа, которая читает три целых числа, представляющих собой длины сторон треугольника. Программа исследует входные данные и печатает сообщение о том, является ли треугольник разносторонним, равнобедренным или равносторонним. Напишите тесты для полной проверки программы.

Эта программа тривиальна, а большинство слушателей — профессионалы с опытом работы в области обработки данных, и все же результаты всегда очень показательны, поскольку бóльшая часть аудитории тестирует программу плохо. Это не следует рассматривать как критику в их адрес; просто здесь наглядно иллюстрируются трудности проектирования хороших тестов даже для тривиальной программы. Прежде чем читать дальше, полезно попытаться решить эту задачу самостоятельно.

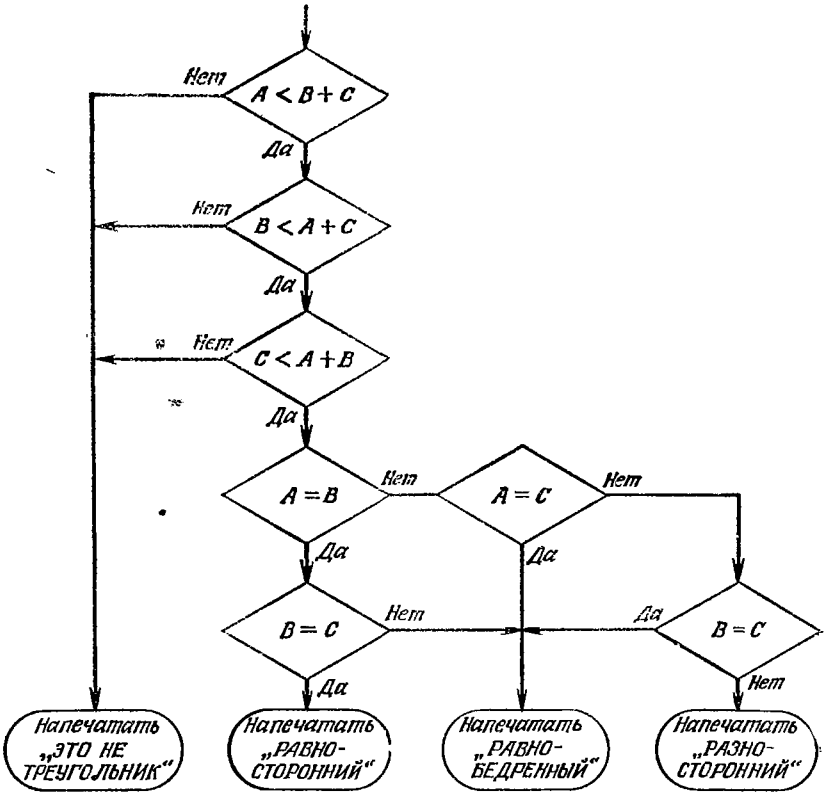


Рис. 11.1. Программа для задачи о треугольнике.

Лучше анализировать полученные ответы, сопоставляя их с алгоритмом этой программы, изображенным на рис. 11.1. Обратите внимание: здесь используется блок-схема, хотя в гл. 8 утверждалось, что блок-схемы не являются важной частью документации при проектировании программы. Блок-схема появилась здесь потому, что она полезна при тестировании; блок-схемы показывают возможные пути в программе более наглядно, чем текст программы на исходном языке сам по себе. В этой программе имеется только 8 различных путей, из чего можно заключить, что желательно иметь по крайней мере восемь тестов.

Следующий шаг при анализе тестов — представить себе возможные ошибки в программе и посмотреть, как много слушателей нашло бы эти ошибки. Гипотетические ошибки можно получить, убирая одну из проверок в блок-схеме на рис. 11.1 или каким-либо образом изменяя ее. На рис. 11.2 показаны некоторые типичные ло-

Гипотетическая ошибка	*Процент слушателей, которые нашли бы эту ошибку*
Не проверяется, составляют ли отрезки треугольник	56
Выполняется только проверка $A<B+C$	28
Не проверяется разносторонний треугольник	84
Не проверяется равнобедренный треугольник	95
Выполняется только одна проверка равнобедренного треугольника: $A=B\neq C$	46
Не проверяется равносторонний треугольник	100

Рис. 11.2. Некоторые результаты тестирования программы для задачи о треугольнике.

гические ошибки и процент слушателей, которые обнаружили бы эти ошибки с помощью своих тестов.

Зная уровень опытности слушателей, тривиальность программы и тот факт, что упражнение выполнялось в курсе по надежности программного обеспечения, можно было бы ожидать, что тесты будут довольно полными, однако, судя по рис. 11.2, дело обстояло иначе. Все включили тесты для случая равнобедренного треугольника, но удивительно высок процент слушателей, которые не проверили случая треугольника разностороннего. (Причина в том, что, вероятно, все намеревались проверить этот случай, но использовали входные данные вроде 1—2—4, которые вообще не подходят для треугольника). Хотя почти все включили по крайней мере по одному тесту для случая равнобедреннего треугольника, более половины слушателей не проверило все три комбинации входных данных, которые могли бы соответствовать этому типу ответа (например, они рассматривали случай 2—2—3, но упустили случаи 2—3—2 и 3—2—2).

Заметьте, что сначала программа должна проверить, описывают ли входные данные вообще какой-нибудь треугольник, потому что,

если бы она печатала «разносторонний» в ответ на входные числа 2—3—6, она допускала бы ошибку (отрезки с длинами 2—3—6 не образуют треугольника). Пятьдесят шесть процентов слушателей проверили это по крайней мере одним тестом, но только 28% слушателей проверили все три перестановки.

Другие результаты, не отраженные на рис. 11.2, также весьма поучительны. Шестьдесят один процент слушателей не выписали ожидаемых результатов тестов, что уменьшает их шансы заметить ошибку, даже если тест обнаружит ее. Один слушатель предложил более тысячи тестов, но все же не обнаружил некоторые из гипотетических ошибок. Семьдесят процентов слушателей проявили достаточно здравого смысла и попробовали подать на входе «мусор» (например, символьные данные), но только 37% слушателей исследовали поведение программы, когда она получает на входе менее трех чисел.

Для проверки своей квалификации в проектировании тестов попробуйте написать тесты для подпрограммы, которая вычисляет корни квадратного уравнения. Получая на входе значения A, B и C, она находит два значения X, удовлетворяющие уравнению $AX^2+BX+C=0$. Сравните свои результаты с приведенными в конце главы.

Сказанное призвано проиллюстрировать трудность высококачественного проектирования тестов. Должно стать ясным, что разработка тестов — творческий процесс, требующий не только особого искусства, но и в некотором смысле разрушительного склада ума. Имеется, однако, несколько простых правил, которыми можно пользоваться, чтобы составить разумный набор тестов. Они рекомендуют сначала рассмотреть модуль как черный ящик (левая граница спектра стратегий), а затем исследовать его внутреннее устройство для подготовки дополнительных тестов. Весь процесс состоит из следующих четырех шагов:

1. Руководствуясь внешними спецификациями модуля, подготовьте тест для каждой ситуации и каждой возможности, для каждой границы областей допустимых значений всех входных данных, областей изменения данных, для всех недопустимых условий.

2. Проверьте текст программы, чтобы убедиться, что все условные переходы будут выполнены в каждом направлении. Если необходимо, добавьте соответствующие тесты.

3. Убедитесь по тексту программы, что тесты охватывают достаточно много возможных путей. Например, для каждого цикла должен быть тест, соответствующий пути без выполнения тела цикла, с однократным его выполнением и максимальным числом повторений.

4. Проверьте по тексту программы ее чувствительность к от-

дельным особым значениям входных данных и, если необходимо, добавьте соответствующие тесты.

Первый шаг проектирования тестов предполагает подход к модулю как к черному ящику и получение тестов за счет манипуляций с входными данными модуля. Именно на этом шаге в основном и требуются творческие способности (остальные три шага, в отличие от первого, довольно методичны). Если число различных входных значений модуля невелико (например, один из пяти запросов), приготовьте тест для каждого из них. Если входных параметров несколько и каждый имеет немного допустимых значений, приготовьте тесты для всех комбинаций.

Больше изобретательности потребуется в том случае, когда входные данные могут принимать значения в широком диапазоне. Разумное решение — написать тесты для проверки модуля на *границах* области допустимых значений его входных данных. Если область значений входного параметра от — 1.0 до +1.0, приготовьте тесты для этих границ. В такой ситуации желателен также третий тест с 0.0 на входе для обнаружения таких ошибок, как деление на нуль. Если в вашей вычислительной системе есть два представления нуля (одно для +0.0, а другое для —0.0), предусмотрите оба теста.

Многие программы имеют также *функциональные границы*. Чтобы обнаружить такие функциональные границы, часто требуется очень напряженно поразмыслить, но это обычно окупается высокой результативностью тестов. Предположим, что тестируется модуль, выполняющий внутреннюю сортировку списка входных записей. Функциональные границы соответствуют случаям, когда список пуст или содержит только одну запись, когда входные записи уже отсортированы или имеют одинаковые значения.

Возвращаемые модулем выходные значения могут покрывать некоторый фиксированный диапазон (например, целые числа от 1 до 65535 или строки литер длиной от 1 до 255 символов). Убедитесь, что у вас есть тесты для проверки модуля на границах диапазонов его выходных значений. Учтите, что не всегда граничные входные значения дают результаты на границе выходных диапазонов.

Остался один вопрос, относящийся к шагу 1,— тесты для недопустимых условий. Если тестируемый модуль специфицирован полностью, должно быть определено его поведение как для правильных, так и для неправильных входных данных. Хороший тест для такого случая — ближайшие значения вне границы области допустимых входных значений.

Проиллюстрируем первый шаг на примере тестов для функции MATCHES загрузчика, спроектированного в части 2. Затем эти тесты будут дополнены на шагах 2, 3 и 4. Вернемся к рис. 8.7 и вспомним внешние спецификации и текст программы MATCHES.

Номер теста	*Входные данные*	*Ожидаемый результат*
1	ESTABLE *заполнено нулями*	MATCHCODE = 2 UNRESNAME *не изменено*
2	ESTB 001 A EP 00200	MATCHCODE = 0 UNRESNAME *не изменено* ESTABLE *не изменено*
3	ESTB 001 X ER *null*	MATCHCODE = 1 UNRESNAME = 'X' ESTABLE *не изменено*
4	ESTB 002 ABCDEFGH ER *null* ABCDEFGH MD 44444	MATCHCODE = 0 UNRESNAME *не изменено* ESTABLE *не изменено, кроме поля адреса первого элемента* = 44444
5	ESTB 006 XX MD 00003 Z ER *null* XX ER *null* XXXXXXXX ER *null* Z ER *null* Z EP 00022	MATCHCODE = 1 UNRESNAME = 'XXXXXXXX' ESTABLE *изменились только поля адреса первого элемента* = 00022 *и третьего элемента* =00003

Рис. 11.3. Исходные тесты для MATCHES.

MATCHES имеет один входной параметр (указатель ESTAB на таблицу внешних имен ESTABLE) и три выходных параметра: обновленную ESTABLE, несогласованное внешнее имя (UNRESNAME) и код возврата (MATCHCODE). Исследуя внешние спецификации этого модуля, можно получить следующий список ситуаций, подлежащих тестированию:

1. ESTABLE, построенная неправильно.
2. ESTABLE, в которой все внешние имена могут быть замкнуты.
3. ESTABLE, которая в исходном состоянии не содержит незамкнутых имен.

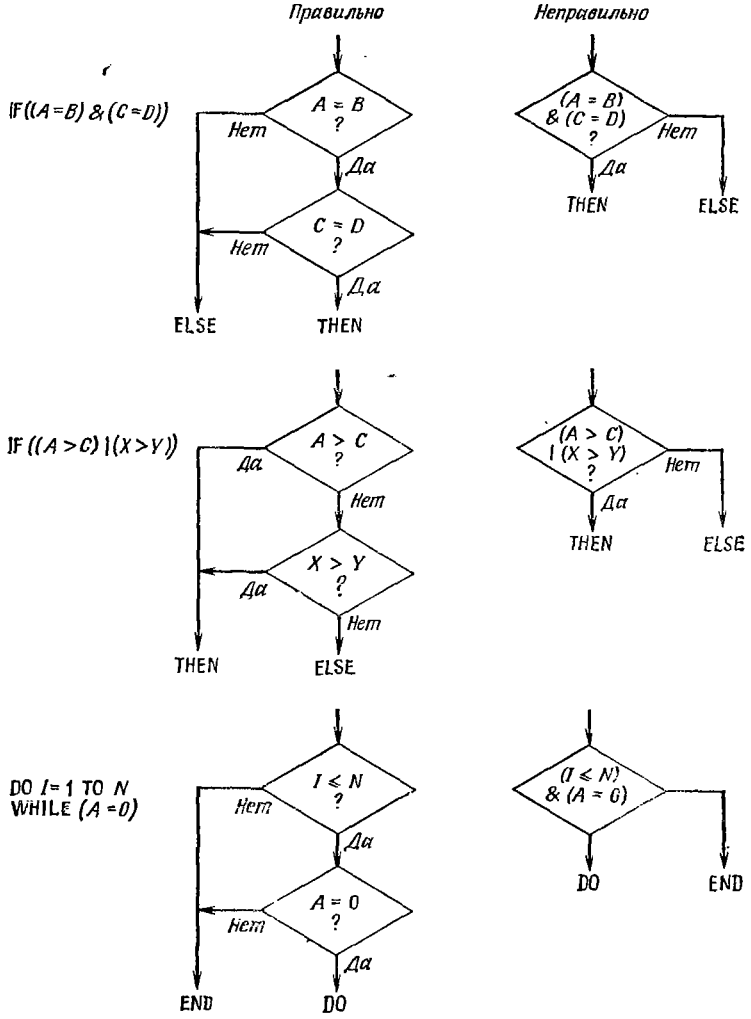


Рис. 11.4. Выявление путей в сложных предложениях.

4. ESTABLE, в которой одно или несколько внешних имен могут быть замкнуты, прежде чем встретится имя, которое не удается замкнуть.

5. ESTABLE, в которой сразу же встречается имя, которое не удается замкнуть (т. е. ни одна попытка не удается).

В качестве граничных условий нам следует проверить случаи, когда ESTABLE содержит единственный элемент и когда имена состоят из одного и восьми символов. Другие граничные условия

включают ситуации, когда незамкнутый и соответствующий замыкающий элементы находятся в первой или последней строке ESTABLE. Тесты, подготовленные для этих условий, изображены на рис. 11.3. Заметьте, что, во-первых, пятый тест охватывает сразу несколько ситуаций и, во-вторых, в соответствии с аксиомами тестирования из гл. 10, выписаны ожидаемые результаты.

Остальные три шага проектирования тестов предполагают исследование логики модуля. Один из основных критериев подбора тестов требует, чтобы каждая команда модуля выполнялась хотя бы раз. Это требование определенно необходимо, но останавливаться на этом нельзя. Гораздо лучше убедить тестовика в том, что нужно подготовить достаточно тестов, чтобы каждое разветвление проходилось в результате принятия решения в каждом направлении по крайней мере один раз. Это приводит нас к еще одной аксиоме автономного тестирования.

Минимальный критерий при автономном тестировании модуля— по крайней мере один раз выполнить все разветвления в каждом из возможных направлений.

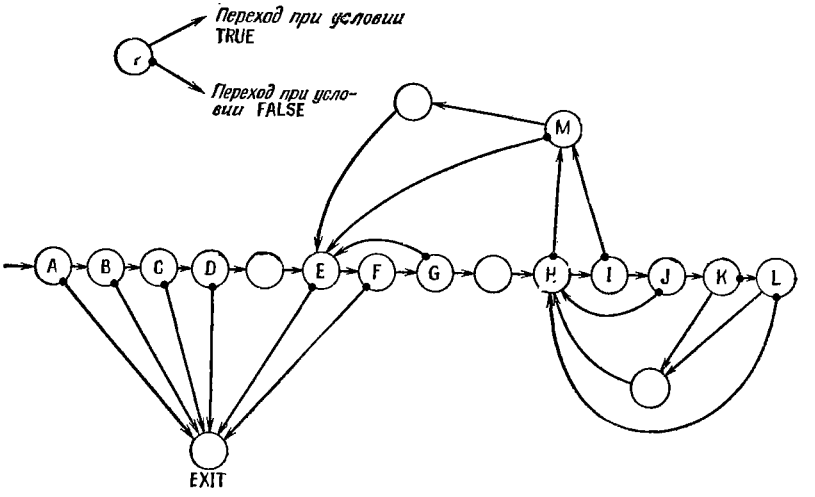
Лучше всего начать с вычерчивания блок-схемы модуля или ее эквивалента. Чтобы выделить все возможные ветви, необходимо учитывать метод, в котором компилятор будет генерировать код для сложных решений [1]. На рис. 11.4 показаны некоторые примеры того, как правильно рассматривать операторы IF и DO в PL/1 при попытке выделить все ветви.

Поскольку нас интересует лишь последовательность выполнения ветвей программы, подробная блок-схема не обязательна. Проще для этой цели использовать *диаграмму управления:* ориентированный граф, изображающий структуру ветвлений в программе. Каждая вершина графа (кружок) представляет линейный участок (последовательность операторов между точками ветвления, или принятия решения, на которую можно попасть только через первый из них). Дуги графа представляют связи между выполнением отдельных линейных участков программы. Такая диаграмма для MATCHES изображена на рис. 11.5.

На диаграмме показано, что имеется 13 разветвлений, каждое на два направления. Чтобы определить, не нужны ли дополнительные тесты для проверки всех 26 направлений, нарисована матрица, и в нее включены наши 5 тестов, чтобы было видно, какие ветви уже будут выполнены. Стоит отметить, что матрицу можно было бы построить в процессе сквозного контроля, рассматривавшегося в

---

[1] Эта рекомендация плохо согласуется с другими идеями книги, в частности с естественным желанием не заставлять человека, работающего на языке программирования, знать представление программы на уровне машинного языка. Лучше было бы анализировать непосредственно сложное условие и соответственно готовить тесты для комбинаций элементарных условий.— *Прим. ред.*



KEY

A	ESTAB¬=NULL
B	TAG='ESTB'
C	SIZE>0
D	SIZE¬>TABSIZE
E	I≤SIZE
F	MATCHCODE=0
G	ADDRESS=NULL
H	J≤SIZE
I	MATCHCODE=1
J	NAME(I)=NAME(J)
K	TYPE=MODULE
L	TYPE=ENTRYPT
M	MATCHCODE=1

Рис. 11.5. Диаграмма управления для MATCHES.

гл. 8, поскольку для этого нужно следить за исполнением каждого теста по алгоритму. Матрица, в которой отмечены ветви для MATCHES, охваченные уже готовыми тестами, изображена на рис. 11.6.

К нашему удивлению (потому что так бывает далеко не всегда), эта матрица показывает, что незатронутыми оказались только три ветви. Чтобы охватить и их, нужно подготовить еще три теста.

Третий шаг — исследовать текст программы, чтобы убедиться, что охвачено достаточно много путей. Число путей MATCHES не фиксировано, оно зависит от числа, расположения и значения эле-

	Условие		Тесты 1	2	3	4	5
A	ESTAB¬=NULL	T	X	X	X	X	X
		F					
B	TAG='ESTB'	T		X	X	X	X
		F	X				
C	SIZE>0	T		X	X	X	X
		F					
D	SIZE¬>TABSIZE	T		X	X	X	X
		F					
E	I≤SIZE	T		X	X	X	X
		F		X	X	X	
F	MATCHCODE=0	T		X	X	X	X
		F					X
G	ADDRESS=NULL	T			X	X	X
		F		X		X	X
H	J≤SIZE	T			X	X	X
		F			X	X	X
I	MATCHCODE=1	T			X	X	X
		F					X
J	NAME(I)=NAME(J)	T			X	X	X
		F					X
K	TYPE=MODULE	T				X	X
		F			X	X	X
L	TYPE=ENTRYPT	T					X
		F			X	X	X
M	MATCHCODE=1	T			X		X
		F				X	X

Рис. 11.6. Матрица учета ветвей.

ментов ESTABLE. Из-за двух циклов MATCHES может иметь астрономическое число путей, поэтому бессмысленно и пытаться тестировать их все. Для каждого цикла, однако, следует проследить за его выполнением ноль, один и максимальное число раз. Просматривая программу MATCHES, можно увидеть, что ее циклы не могут иметь менее одной итерации (SIZE всегда больше нуля, а условие WHILE первоначально всегда истинно). Циклы могут быть

Номер теста	Входные данные	Ожидаемый результат
6	*Входной параметр* ESTAB *установлен в* NULL	MATCHCODE=2 UNRESNAME *не изменено* ESTAB *не изменена*
7	ESTB | 000	MATCHCODE=2 UNRESNAME *не изменено* ESTABLE *не изменено*
8	ESTB | 4000	MATCHCODE=2 UNRESNAME *не изменено* ESTABLE *не изменено*
9	ESTB | 003 ABC | EP | *null* CDEF | MD | *null* P | ER | 22222	MATCHCODE=0 UNRESNAME *не изменено* ESTABLE *не изменена*

Рис. 11.7. Дополнительные тесты для MATCHES.

выполнены только с одной и максимальным числом (SIZE) итераций, но подготовленные тесты уже охватывают эти ситуации.

Далее следует проверить логические выражения (например, в циклах DO с условием WHILE или в сложных условиях IF), все ли возможные логические комбинации тестируются (например, такой оператор, как IF((*A*=0) & (*B*=4)), имеет четыре комбинации элементарных условий). Исследование текста программы MATCHES показывает, что наш набор тестов удовлетворяет и этому требованию.

Последний шаг — проверить чувствительность алгоритма к особым входным значениям. Следует также добавить тесты, пытаясь добиться выхода за верхние и нижние границы массивов или диапазонов числовых значений.

MATCHES имеет несколько чувствительных к особым данным мест, как это было отмечено в гл. 8. Если встречается модуль или вход с нулевым адресным полем (случай, который «никогда не должен возникнуть»), MATCHES просто пропускает его нулевой адрес и переходит к следующему элементу. Кроме того, модуль MATCHES не должен пытаться замкнуть уже замкнутое (т. е. имеющее ненулевое адресное поле) внешнее имя. Для проверки этих условий включаются новые тесты. На рис. 11.7 показаны тесты, которые добавлены в результате исследования алгоритма модуля.

Этапы проектирования тестов могут показаться трудоемкими, но никто никогда не утверждал, что тестирование — легкая задача. Следуя этим правилам, можно получить набор тщательно спроектированных тестов с высокой способностью обнаруживать ошибки. Этот процесс может занять на несколько часов больше, чем выбор нескольких произвольных тестов на основе интуиции, но, если он приведет к обнаружению одной-двух случайных ошибок, значит, дополнительное время было потрачено не зря.

ВЫПОЛНЕНИЕ ТЕСТА

После того как тесты для модуля спроектированы, можно перейти к следующим этапам — написать их, тестировать и выполнить. Как говорилось в гл. 10, форма представления тестов зависит от принятого метода сборки модулей. Для нисходящих методов тесты сначала пишутся в виде конкретных наборов выходных данных, вырабатываемых заглушками, а затем уже в виде внешних входных данных, задаваемых пользователем. Для восходящих методов или при автономном тестировании модулей тесты приобретают вид ведущих программ (драйверов) или операторов языка тестирования (если используются специальные инструменты тестирования модулей). В следующем разделе мы рассмотрим некоторые из таких инструментов. А пока будем предполагать, что они не используются.

Процесс написания тестов (в отличие от их проектирования) носит в основном рутинный характер и представлен здесь поэтому лишь вкратце. Например, если нужен драйвер, пишется небольшая программа, вызывающая тестируемый модуль. Каждый тест представляется вызовом этого модуля и передачей ему конкретного набора входных данных. Чтобы облегчить повторное выполнение теста в будущем и избежать проблем типа «глаз видит то, что хочет видеть», следует попытаться сделать тесты самопроверяемыми. Вместо того чтобы печатать выходные данные для каждого теста, следует, где это возможно, включать непосредственно в драйвер сверку полученных и ожидаемых результатов.

В идеальном случае тесты сами должны быть проверены перед их использованием для тестирования реальной программы. Обычно это неосуществимо; единственная возможность проверить правильность теста (кроме просмотра человеком) — действительно выполнить его. Однако при тестировании следует иметь в виду, что сами тесты могут содержать ошибки.

Критический момент в выполнении тестов — анализ результатов, будь то с помощью программы либо визуально. Распространенная нелепость — потратить часы на проектирование изощренных тестов и затем не заметить ошибку только потому, что результаты теста удостоены лишь беглого взгляда. Чрезвычайно важно тщательно изучить результаты каждого теста, выискивая малейшие

тревожные признаки. Здесь-то уж жизненно важно определить, какие результаты вы ожидаете, прежде чем изучать реальные выходные данные.

Есть много типов ошибок, которые не могут быть обнаружены при простом изучении выходных данных модуля. Их можно определить как ошибочные побочные эффекты. Модуль может возвращать правильные выходные параметры, но одновременно с этим изменять входные параметры, запоминать некоторые данные в произвольном месте памяти либо иметь некоторые другие нежелательные побочные эффекты. Некоторые языки программирования и особенности архитектуры ЭВМ уменьшают вероятность таких ошибок, но часть из них может остаться необнаруженной, и, когда они есть, их бывает очень трудно найти. Имеется несколько типов проверок, которые могут быть выполнены для того, чтобы обнаружить определенные побочные эффекты. Если входной параметр специфицируется только как входной (можно ссылаться, но нельзя изменять), следует проверять после каждого теста, так ли это на самом деле. На рис. 11.3 и 11.7 показано, что после каждого теста MATCHES следует проверять, не изменились ли некоторые области данных. Если это возможно, надо проверить участки памяти, непосредственно прилегающие к каждой области вывода, чтобы обнаружить ошибки типа «одним больше — одним меньше».

Во время тестирования не следует забывать аксиомы гл. 10. Вместо того чтобы самому тестировать модуль, который вы программировали, дайте его тестировать тому, кто разрабатывает один из модулей, вызывающих ваш, либо попробуйте перед автономным тестированием обменяться модулями с другим программистом, работающим над этим проектом. Избегайте тестирования «с лету»; никогда не предполагайте, что тест будет использоваться только один раз. Никогда не упрощайте свою программу, чтобы облегчить тестирование.

ИНСТРУМЕНТЫ ДЛЯ ТЕСТИРОВАНИЯ МОДУЛЕЙ

Зная, что программирование — индустрия с многомиллиардным оборотом, можно было бы ожидать здесь высокого уровня автоматизации, и в частности большого разнообразия изощренных средств разработки программ, предназначенных для минимизации ручной и однообразной работы, являющейся частью деятельности всякого программиста. Дело, однако, обстоит не так; в действительности в отношении программирования часто справедливо замечание, что здесь вычислительные машины применяются меньше, чем в любой другой отрасли промышленности. В этом и следующих разделах описываются некоторые попытки двигаться в правильном направлении — программные инструменты, помогающие в работе по тестированию модулей. Некоторые из описанных инструментов коммер-

чески доступны; другие являются только «домашними» разработками. Я не пытался дать полный обзор всех таких инструментов, поскольку он тотчас же устарел бы. Вместо этого я выбрал несколько показательных примеров, чтобы дать представление о том, что сделано в этой области.

Автоматизированный автономный тест (AUT — Automated Unit Test) — инструмент, разработанный для внутреннего применения в корпорации IBM [1]. Его назначение — облегчить работу, связанную с нисходящим и восходящим методами тестирования за счет устранения необходимости в драйверах и заглушках, а также стандартизировать вид автономных тестов, благодаря чему повторное выполнение таких тестов (например, после модификации программы) становится тривиальной задачей. Кроме того, он должен обеспечивать автоматическую проверку результатов выполнения тестов и должен быть независимым от используемого языка программирования.

Основное средство взаимодействия с пользователем — язык MIL-S, служащий для описания входных данных и ожидаемых результатов каждого теста. Программист начинает с представления своих тестов на языке MIL-S и запоминания их в базе данных. Для тестирования модуля MATCHES он затем напечатает на терминале «AUT MATCHES», после чего AUT выбирает каждый тест на MIL-S для MATCHES, компилирует тест и вызывает модуль. Когда выполнение модуля заканчивается, ожидаемые результаты сравниваются с действительными и пользователь оповещается обо всех расхождениях.

MIL-S — непроцедурный язык для представления тестов. Тест 2 (рис. 11.3) можно было бы представить следующими операторами:

```
IN02      PARMLIST   ESTAB,UNRNAME,MCODE;
ESTAB     PTR        ESTABLE;
UNRNAME   DATA       (8) ('  ');
MCODE     DATA       (2) IGNORE;
ESTABLE   DATA       (4) ('ESTB');
SIZE      DATA       (2) ('1'D);
          DATA       (8) ('A');
          DATA       (2) ('EP');
          PTR        ESTABLE; TO GIVE ADDR FIELD
*                             ANY NON-NULL VALUE
OUT02     COPY;
MCODE     DATA       (2) ('0'D); EXPECT NO UNMATCHED
*                                ITEMS
```

Входной и выходной разделы каждого теста компилируются раздельно, в результате чего генерируется входная структура, пе-

редаваемая тестируемому модулю, и выходная структура, используемая для проверки полученных результатов. Входной раздел в приведенном примере должен быть понятен без пояснений (слово IGNORE означает, что значение соответствующего элемента несущественно). Слово COPY в выходном разделе — сокращение, означающее, что операторы MIL-S из входного раздела должны быть скопированы в выходной раздел перед его компиляцией. Все дополнительные операторы в выходном разделе (в нашем случае MCODE) заменяют соответствующие копируемые операторы либо просто вставляются, если соответствующих операторов во входном разделе нет. Это простой способ включить проверку сохранности входных данных как частный случай проверки результатов.

Для рассматриваемого теста AUT после выполнения модуля проверит следующие условия:

1. Список параметров содержит три параметра.
2. Первый параметр — указатель. Он ссылается на последовательную структуру памяти, в первых четырех байтах которой находятся символы «ESTB», в следующие двух байтах — значение 1, в следующих восьми байтах — символ «А», за которым идет семь пробелов, затем символы «EP» в двух следующих байтах и наконец адрес ESTABLE (любой ненулевой адрес) в последних четырех байтах.
3. Второй параметр — восьмибайтовая строка пробелов (чтобы проверить, что значение UNRNAME не изменено).
4. Третий параметр — двухбайтовое поле со значением, равным нулю.

Если обнаруживаются какие-то расхождения, пользователю сообщается, какой именно элемент данных не соответствует ожиданиям, и выдаются его предполагаемое и реальное значения. Все программные прерывания в отлаживаемом модуле перехватываются программой AUT. Интересно следующее проектное решение, принятое в AUT: система сама может программировать проверки при анализе выходной информации теста (например, когда прослеживается ошибочное значение переменной — указателя). Такие события никогда не видны пользователю. AUT помнит, какая область данных исследуется во время проверки программы, перехватывает программное прерывание и сообщает о том, что в этой области имеется ошибка.

Когда AUT используется для восходящего тестирования, все модули, находящиеся в структуре программы ниже тестируемого, автоматически загружаются по команде AUT. Для нисходящего тестирования AUT предоставляет средства имитации, позволяющие разрешить некоторые проблемы, связанные с использованием заглушек. Функция MATCHES не вызывает никаких других модулей, но мы предположим, для иллюстрации, что она вызывает модуль с

именем INVERT. Для его имитации можно во входной раздел теста MATCHES включить оператор

```
INVERT CALL IN07;
```

Этот оператор вызывает следующие действия AUT. Когда MATCHES вызывает INVERT, AUT перехватывает вызов. В базе данных отыскивается и компилируется (программа AUT рекурсивна) тест 07 для модуля INVERT. AUT определяет, совпадают ли передаваемые INVERT параметры с входными данными, указанными в тесте 07, и, если так, возвращает соответствующие данные из выходного раздела, как если бы INVERT действительно был выполнен, а затем управление возвращается MATCHES. Это не только снимает необходимость заглушек, но позволяет также предусматривать различные результаты для разных тестов при имитации несуществующих модулей.

Весьма похожее по своим концепциям средство — Система тестирования модулей (MTS — Module Testing System), распространяемая на рынке фирмой Management Systems and Programming Ltd. MTS может работать на большинстве систем IBM/360, 370 и ICL и тестирует модули, написанные на языке ассемблера, Коболе, PL/1 или Фортране.

Тесты в MTS описываются на языке, аналогичном MIL-S в AUT. В число операторов языка входят такие, как PARAMETER для описания областей данных, TEST для инициирования выполнения теста, PRINT COMPARE для проверки результатов теста и SIMULATE для имитации результатов несуществующего модуля. Тест для модуля FINDMEAN, вычисляющего среднее арифметическое трех чисел, выглядел бы так:

```
PARAMETER  X       6 CORE
PARAMETER  Y       6 CORE
PARAMETER  Z       6 CORE
PARAMETER  MEAN    6 CORE
X          D(6)    1230.0
Y          D(6)    2722.00
Z          D(6)    1502.00
EXP-MEAN   D(6)    1818.00
TEST       FINDMEAN X Y Z MEAN
PRINT COMPARE MEAN EXP-MEAN
```

Третий инструмент — TESTMASTER, который предоставляет в аренду фирма Hoskyns Inc. Он предназначен для тестирования программ на Коболе, выполняемых в системе IBM 360/370. Тесты

пишутся на языке, похожем на Кобол, и подаются препроцессору, который транслирует их в стандартный Кобол и компилирует. Не выполняется никакой автоматической проверки результатов; выходная информация представляет собой «посмертную выдачу» (дамп), которая должна быть проверена программистом вручную.

СТАТИЧЕСКИЙ АНАЛИЗ УПРАВЛЕНИЯ

Еще одна область тестирования модуля, которая может быть автоматизирована,— статический анализ программы на исходном языке (слово «статический» означает, что этот анализ производится без реального выполнения программы). Например, можно было бы написать программу, которая бы анализировала модуль и строила для него диаграмму управления (например, как диаграмма на рис. 11.5). При таком анализе можно было бы также перечислить все возможные пути в модуле и проанализировать ошибочные ситуации, типа использования данных до присваивания им начальных значений.

Одно из таких средств — Автоматизированная система оценки программного обеспечения ASES (Automated Software Evaluation System) [2]. ASES анализирует исходную программу и получает ее внутреннее представление в виде ориентированного графа. По этому графу ASES получает некоторые структурные характеристики программы и сообщает их пользователю. Такое сообщение включает список всех циклов, всех операторов, которые никогда не могут быть выполнены, и таких чреватых ошибками ситуаций, как переход внутрь цикла.

Для тестирования программ на Фортране можно воспользоваться системой RXVP, которая создана корпорацией General Research [3]. RXVP начинает с построения и печати графа управления программы. Она также вырабатывает список потенциальных ошибок, таких, как использование переменной до присваивания ей начального значения, несоответствие между аргументами и параметрами подпрограммы, а также наличие знаменателей, которые способны обратиться в нуль.

СРЕДСТВА ПЕРИОДА ВЫПОЛНЕНИЯ

Остальные инструменты для тестирования модулей взаимодействуют с программой во время ее выполнения. Их основная цель — измерить качество тестирования: определить, в какой мере тесты проверяют алгоритм программы. Некоторые из них просто контролируют выполнение каждой подпрограммы, подсчитывая, сколько раз она вызывалась. Другие учитывают, сколько раз выполнялся каждый оператор исходной программы. Более изощренные показывают, во всех ли направлениях выполнялся каждый условный пере-

ход. Как правило, эти средства собирают статистику такого рода, вставляя «зонды», или «ловушки», в каждой точке ветвления в контролируемой программе.

Более сложные средства, кроме этого, *накапливают* статистические сведения о конкретной программе в базе данных или в файле. Хотя это и не обязательно при автономном тестировании, поскольку обычно все тесты модуля выполняются почти одновременно, зато важно при тестировании внешних функций или комплексном тестировании системы, которое растягивается на несколько недель или даже месяцев. Накопленная статистика позволяет определить, какие элементы алгоритма еще не проверялись достаточное число раз.

Две системы, обеспечивающие контроль в период выполнения,— это контролирующий (checkout) и оптимизирующий (optimising) компиляторы PL/1 фирмы IBM. Они имеют функцию COUNT, которая в конце выполнения сообщает, сколько раз был выполнен каждый оператор программы. Другая система, МетаКобол фирмы Applied Data Research, сообщает, сколько раз выполнялся каждый параграф. Пакет «Series-J» фирмы National Computing Industries выдает отчет, в котором указаны все предложения в программе на Коболе, которые не были выполнены в данном тесте.

Более сложным средством является система Оценки и Тестирования программ PET (Program Evaluator and Tester) для программ на Фортране [4]. PET указывает, сколько раз выполнялся каждый оператор программы и, что более важно, сколько раз выполнялся каждый условный переход (оператор IF или вычисляемый GO TO). PET имеет одно необычное средство отладки: она контролирует результаты присваивания и значения параметров циклов DO и печатает первое, последнее, минимальное и максимальное значения каждой переменной.

Уже упоминавшаяся ранее система ASES также осуществляет контроль во время выполнения. Имея список переменных с указанными верхними и нижними границами их значений, ASES наблюдает за всеми изменениями переменных и сообщает обо всех попытках выйти за указанные границы. Корпорация TRW разработала другое средство, называемое PACE (Product Assurance Confidence Evaluator — Оценка степени доверия к продукту), которое считает количество выполнений каждого оператора и каждой подпрограммы [5]. Результаты подсчета печатаются, а также сохраняются и обновляются в файле.

Уже упоминавшаяся система RXVP также осуществляет контроль во время выполнения. Она сообщает, сколько раз выполнялся оператор, сколько раз выполнялась каждая ветвь в точках принятия решений, минимальное, максимальное и среднее значения результатов каждого оператора присваивания. RXVP сообщает как сводные данные, так и результаты каждого отдельного прогона.

Несколько слов с целью предостеречь читателя, который задумал разработать свой собственный инструмент тестирования. Многие организации пытаются этим заниматься, но, хотя идеи часто привлекательны, созданные средства не лучшим образом воспринимаются программистами. Основная причина, которая губит большинство инструментов тестирования,— слабый учет психологических факторов. Проектировщик, занимающийся разработкой нового инструмента тестирования, должен уделить им особое внимание. Основной недостаток автоматизированных средств тестирования — слишком большие распечатки, вследствие чего программисту приходится прочитывать страницу за страницей, чтобы найти один-единственный интересующий его результат. Например, большинство программ, осуществляющих контроль во время выполнения, выдает листинг программы на исходном языке, дополненный несколькими столбцами со счетчиками числа выполнения каждого оператора. Это может быть приемлемо при тестировании модуля, но слишком обременительно, когда тестируется вся программа или система и когда более интересно узнать, что не выполнялось. Мониторы, собирающие сводную статистику, должны уметь выдавать краткие отчеты с указанием только тех ветвей, которые не были проверены.

ТЕСТЫ ДЛЯ КВАДРАТНОГО УРАВНЕНИЯ

Для читателя, который пытался в начале главы построить тесты для модуля, решающего квадратное уравнение, ниже перечисляются тесты, разработанные Грюнбергером [6].

1. $A=0$, $B=0$, $C=0$. В этом случае уравнение сводится к виду $0=0$ и не может быть разрешено относительно X. Пробовали ли вы этот тест для проверки поведения модуля при таких входных данных?
2. $A=0$, $B=0$, $C=10$. Это соответствует уравнению $10=0$, которое не имеет решений. Интересный тест на ошибочные входные условия.
3. $A=0$, $B=5$, $C=17$. Соответствующее уравнение ($5X+17=0$) не является квадратным. Справится ли с ним модуль? Этот тест может обнаружить также попытку деления на нуль.
4. $A=6$, $B=1$, $C=2$. Это один из нескольких «нормальных» тестов, которые вам следует выполнить. Не забыли ли вы заранее вычислить результат для каждого теста?
5. $A=3$, $B=7$, $C=0$. Еще один «нормальный» тест. Проверяется ситуация, когда один из корней равен нулю.
6. $A=3$, $B=2$, $C=5$. Помните ли вы, что квадратное уравнение может иметь комплексные корни?
7. $A=7$, $B=0$, $C=0$. Этот тест проверяет, умеет ли модуль извлекать квадратный корень из нуля.

8. Полезны также тесты, проверяющие границы диапазонов арифметических значений и точность модуля.

ЛИТЕРАТУРА

1. Heuermann C. A., Myers G. J., Winterton J. H. Automated Test and Verification, *IBM Technical Disclosure Bulletin*, **17** (7), 2030—2035 (1974).
2. Meeker R. E., Ramamoorthy C. V. A Study in Software Reliability and Evaluation, Tech. Memo., No. 39, Electronics Research Center, University of Texas at Austin, Austin, Texas, 1973.
3. RXVP User's Guide, RM-1942, General Research Corp., Santa Barbara, Cal., 1975.
4. Stucki L. G. Automatic Generation of Self-Metric Software, Record of the 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 94—100.
5. Brown J. R., DeSalvio A. J., Heine D. E., Purdy J. G. Automated Software Quality Assurance, in W. C. Hetzel, Ed., Program Test Methods. Englewood Cliffs, N. J.: Prentice-Hall, 1973, pp. 181—203.
6. Gruenberger F. Program Testing and Validation, *DATAMATION*, **14** (7), 39—47 (1968).

ГЛАВА 12

Тестирование внешних функций и комплексное тестирование

После того как проведено тестирование модулей и закончено тестирование сопряжений, являющееся неотъемлемой компонентой пяти (из шести) подходов к сборке модулей, процесс тестирования еще только начинается. Еще предстоит тестирование соответствия программы или системы внешним спецификациям (тестирование внешних функций), целям (комплексное тестирование), требованиям (тестирование приемлемости), а также необходимо определить, правильно ли система установлена (тестирование настройки).

ТЕСТИРОВАНИЕ ВНЕШНИХ ФУНКЦИЙ

Цель теста внешней функции — найти расхождения между программой и ее внешними спецификациями. Необходимым условием успешного тестирования функций является наличие четких и точных внешних спецификаций. Если внешние спецификации неполны или неоднозначны, результаты тестирования не могут не быть такими же.

С целью проектирования тестов внешних функций внешние спецификации обычно разбиваются на отдельные внешние функции (например, по типу входных сообщений или команд пользователя), и после тщательного изучения каждой функции строятся тесты. При этом большинство правил проектирования тестов для модулей, рассматривавшихся в гл. 11, применимо также и к тестированию функций. Например, тесты должны строиться для всех входных условий и вариантов, а также на границах всех областей допустимых значений на входе и областей изменения на выходе. Тесты должны также проверять поведение программы у функциональных границ и в случаях ввода недопустимых или непредусмотренных данных. В следующем разделе рассматривается методология проектирования тестов, основанная на *функциональных диаграммах* (*cause-effect graphing*).

Существуют средства для автоматической генерации тестов, но ими обычно не следует пользоваться. Например, обслуживающая программа IEBDG [1] может использоваться для генерации потока депозитных сделок с разными учетными номерами и суммами

в долларах для тестирования банковской системы. Однако такой подход с позиции «грубой силы» непродуктивен по нескольким причинам: такие тесты слабо обнаруживают ошибки и при таком тестировании нарушается одна из фундаментальных аксиом тестирования, которая требует рассчитывать результаты для каждого теста заранее. Без этого программист, выполняющий тестирование, скорее всего не заметит проявления ошибки (кроме таких очевидных случаев, как программные прерывания).

Как и при автономном тестировании, тесты внешних функций не должны быть тестами «одноразового использования». Каждый тест должен быть документирован (например, необходимы инструкции по его использованию, должны быть приведены ожидаемые результаты) и, если это возможно, самопроверяемым. Хорошо также оценить относительную важность каждого теста. Это позволяет небольшую часть тестов выделить в качестве *тестов регрессии*, т. е. тестов, которые в будущем должны выполняться после каждой модификации, каждого исправления программы (чтобы проверить, не возникло ли движение вспять, не привнесены ли этим исправлением ошибки).

Тестирование функций — процесс *контроля*, поскольку оно обычно выполняется в моделируемой среде (в противоположность обстановке реальной). Другими словами, тестирование функций обычно выполняется для компонент системы прежде, чем она будет собрана воедино. Например, могут быть недоступны определенные устройства ввода-вывода, вследствие чего потребуется написать специальные программы для имитации их работы, могут отсутствовать или быть неполными отдельные компоненты программного обеспечения, что также потребует имитации или применения вспомогательных программ — «подпорок».

Точка, соответствующая тестированию внешних функций на спектре стратегий (рис. 10.3), находится ближе к левому (черный ящик) концу. В первую очередь следует составить тесты по внешним спецификациям, не обращаясь к внутренней структуре или алгоритму программы. Однако следует контролировать и выполнение программы, чтобы определить, в какой степени охвачены все условные переходы.

Обязательный минимум при тестировании внешних функций — выполнить все разветвления по крайней мере один раз в каждом направлении, за исключением тех ветвей, которые обеспечивают защитное программирование.

Во время тестирования функций следует использовать монитор, который собирает сводную статистику. Если какие-то ветви оказались непроверенными, следует добавить соответствующие тесты. Они должны создавать такие ситуации, как недопустимые входные данные или ошибки ввода-вывода. Единственное исключение со-

ставляют ветви, выполняемые только при наличии ошибок в программном обеспечении (т. е. проверки, связанные с защитным программированием). Здесь разумно просто положиться на качество автономного тестирования.

МЕТОД ФУНКЦИОНАЛЬНЫХ ДИАГРАММ

Метод функциональных диаграмм [2, 3], относящийся к левому краю спектра стратегий тестирования, предлагает способ перевода спецификаций, написанных на естественном языке, на язык формальный. Это способствует проектированию высокорезультативных тестов, не страдающих избыточностью, и, кроме этого, как отмечалось в гл. 4, обнаруживаются случаи неполноты и неоднозначности в исходных спецификациях.

Метод предполагает анализ семантического содержания внешних спецификаций и перевод их на язык логических отношений между входными данными (ситуациями) и выходными данными и преобразованиями (эффектами), представленных в форме логической диаграммы («и-или»-графа), называемой *функциональной диаграммой*. Диаграмма снабжается примечаниями в виде синтаксических правил и ограничений внешней среды и затем преобразуется в таблицу решений с ограниченным входом. Каждый столбец таблицы соответствует будущему тесту.

Рассмотрим пример применения этого метода. Первый шаг — разбить внешние спецификации на отдельные функции, комбинаторные свойства которых и должны тестироваться. В качестве примера займемся фрагментом спецификации, изображенным на рис. 12.1. Отметим, что эти спецификации слегка упрощены, чтобы пример был не слишком большим.

Второй шаг — проанализировать спецификации в поисках всех явных и неявных ситуаций (условия на входе) и эффектов (действия на выходе). Лучше всего делать это, подчеркивая каждую ситуацию и каждый эффект, по мере того как они встречаются при чтении спецификаций. Все ситуации и эффекты нумеруются произвольным образом. Для рассматриваемого примера они перечислены ниже.

Ситуация 1. Первый, отличный от пробела символ, следующий за символом «С» и одним или несколькими пробелами, является символом «/».

Ситуация 2. Команда содержит ровно два символа «/».

Ситуация 3. Длина строки 1 равна единице.

Ситуация 4. Длина строки 1 равна 30.

Ситуация 5. Длина строки 1 — между 2 и 29.

Ситуация 6. Длина строки 2 равна нулю.

Ситуация 7. Длина строки 2 равна 30.

Ситуация 8. Длина строки 2 — между 1 и 29.

3.4. Подкоманда ИЗМЕНИТЬ

Подкоманда ИЗМЕНИТЬ используется для изменения последовательности символов в «текущей» строке редактируемого файла.

3.4.1. Входные данные

Команда имеет следующий синтаксис:

С/строка1/строка2

Строка1 — это строка символов, которую вы хотите заменить. Она может иметь длину от 1 до 30 символов и состоять из любых литер, за исключением «/». Строка2 — строка символов, на которую следует заменить строку1. Она может иметь длину от 0 до 30 символов и содержать любые литеры, кроме «/». Если строка2 опущена (имеет нулевую длину), строка1 просто вычеркивается.

За именем команды «С» должен следовать по крайней мере один пробел

3.4.2. Результаты

Измененная строка в случае удачного завершения команды печатается на терминале. Если изменение не может быть выполнено, поскольку не удается найти в текущей строке строку1, печатается сообщение «НЕ НАЙДЕНА». Если синтаксис команды ошибочен, печатается сообщение «СИНТАКСИЧЕСКАЯ ОШИБКА».

3.4.3. Преобразование системы

Если синтаксис правилен и строка1 найдена в текущей строке, тогда строка1 удаляется, а ее место занимает строка2. При этом текущая строка расширяется или сжимается в зависимости от разницы в длинах строки1 и строки2. Если команда синтаксически неправильна или в текущей строке не удается найти строку1, текущая строка остается неизмененной.

Рис. 12.1. Спецификации.

Ситуация 9. Текущая строка содержит вхождение строки 1.

Эффект 31. Напечатана измененная строка.

Эффект 32. Первое вхождение строки 1 в рассматриваемую строку заменено на строку 2.

Эффект 33. Напечатано «НЕ НАЙДЕНО».

Эффект 34. Напечатано «СИНТАКСИЧЕСКАЯ ОШИБКА».

Рассматриваемая команда может иметь астрономическое число вариантов, поскольку строка 1 и строка 2 вместе допускают 30×31 различных комбинаций длин, а каждый символ строки может быть любой из примерно 80 литер, имеющихся на терминале. Ситуации 3—8 представляют собой некоторый компромисс; здесь предполагается, что мы в первую очередь интересуемся граничными значе-

ниями длин и что (хочется верить) обработка команды не зависит от конкретных символов строк 1 и 2. Эффект 32 показывает, что в спецификациях уже обнаружены неоднозначности: явно не сказано, что происходит, когда имеется несколько вхождений строки 1 в рассматриваемую строку. Следовало бы указать, что изменяется только *первое* вхождение.

Третий шаг — нарисовать функциональную диаграмму. Ситуации изображаются в виде вершин на левом краю листа бумаги, а эффекты — на правом. С помощью базовых логических отношений, изображенных на рис. 12.2, из ситуаций и эффектов строится структура логических отношений. В случае необходимости для изображения более сложных отношений (например, эффект *M* имеет место при условии «ситуация *A* И (ситуация *B* ИЛИ ситуация *C*)») используются промежуточные вершины. В терминах булевой алгебры, ситуации и эффекты могут находиться в двух состояниях: *нуль* (т. е. не имеет места, или отсутствует) и *один* (т. е. имеет место, или присутствует).

На рис. 12.3 функциональная диаграмма приведена в незавершенном виде. Девять ситуаций и четыре эффекта изображены в виде вершин 1—9 и 31—34. Промежуточные вершины 21 и 22 представляют состояния, соответствующие случаю, когда операнды «стро-

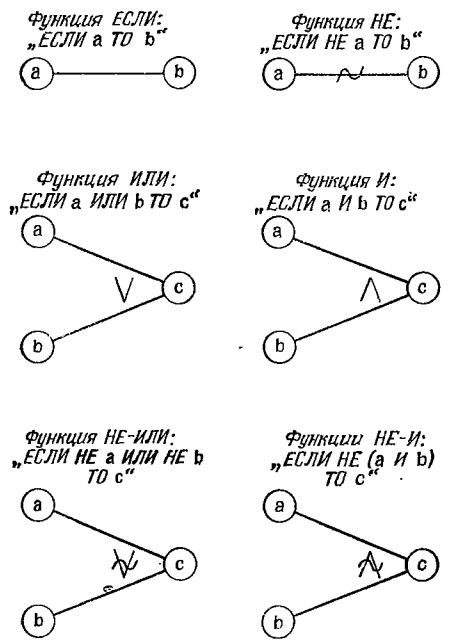


Рис. 12.2. Функциональные связи.

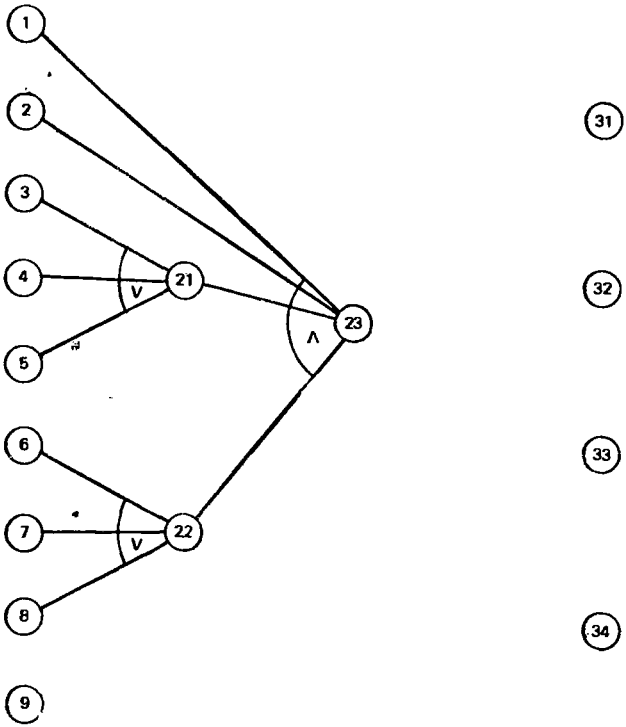


Рис. 12.3. Незавершенная диаграмма.

ка1» и «строка2» синтаксически правильны. Промежуточная вершина 23 представляет состояние, соответствующее случаю, когда синтаксически правильна вся команда.

На рис. 12.4 изображена законченная диаграмма. Она представляет собой формальное выражение приведенных на рис. 12.1 спецификаций на естественном языке. При попытках построить полную диаграмму часто обнаруживаются такие ошибки в спецификациях, как наличие ситуаций или комбинаций ситуаций, не имеющих видимых эффектов, или наличие эффектов, никак не связанных с ситуациями на входе.

Вероятно, во всякой такой диаграмме определенные комбинации ситуаций и эффектов невозможны из-за ограничений синтаксиса или внешней среды. Такие ограничения следует добавлять к диаграмме, чтобы предупредить тестирование невозможных ситуаций. На рис. 12.5 изображены дополнительные базовые отношения, которые в случае необходимости уточняют диаграмму. *Исключая*, *включая*, *одно-и-только-одно*, *требует* — такие ограничения используются для описания отношений между ситуациями. Ограничение *маскирует* используется для описания таких отношений между

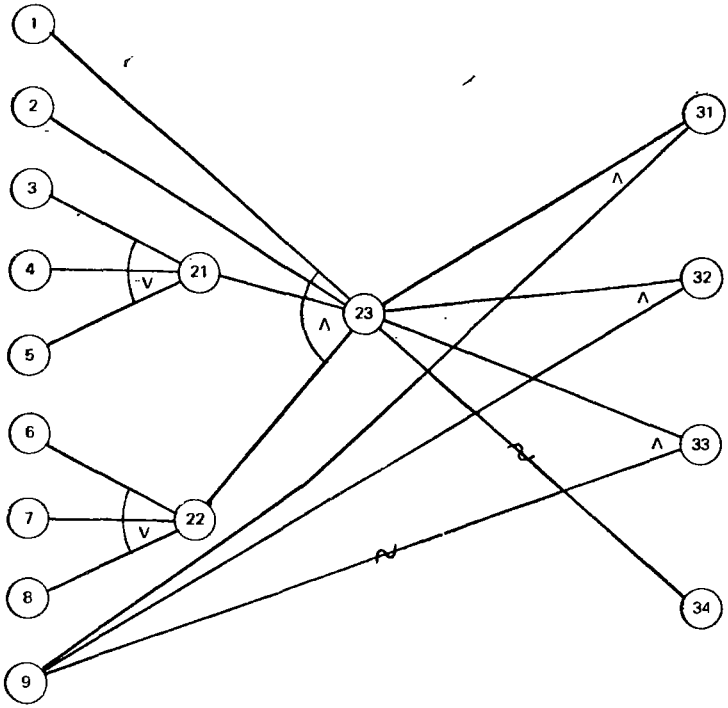


Рис. 12.4. Завершенная функциональная диаграмма.

эффектами, когда наличие одного эффекта маскирует, или подавляет, другой эффект.

На рис. 12.6 показана окончательная диаграмма со всеми дополнительными ограничениями. Ситуации 3, 4 и 5 взаимно исключают друг друга, потому что длина строки 1 в одной команде фиксирована и определена однозначно. Они *не* связаны ограничением «одно-и-только-одно», потому что длина строки 1 может быть меньше единицы или больше 30. Ограничение «исключая» связывает также ситуации 6, 7 и 8.

Последний шаг — преобразовать диаграмму в таблицу решений с ограниченным входом. Для этого нужно выбрать некоторый эффект и записать все комбинации ситуаций, которые его вызывают, затем выписать также состояния всех остальных эффектов при этих комбинациях ситуаций. Первые девять столбцов на рис.12.7 представляют комбинации ситуаций, вызывающие эффект 31.

Следующим необходимо рассмотреть эффект 32. Однако диаграмма показывает, что он возникает в тех же ситуациях, что и эффект 31, поэтому его учесть совсем легко.

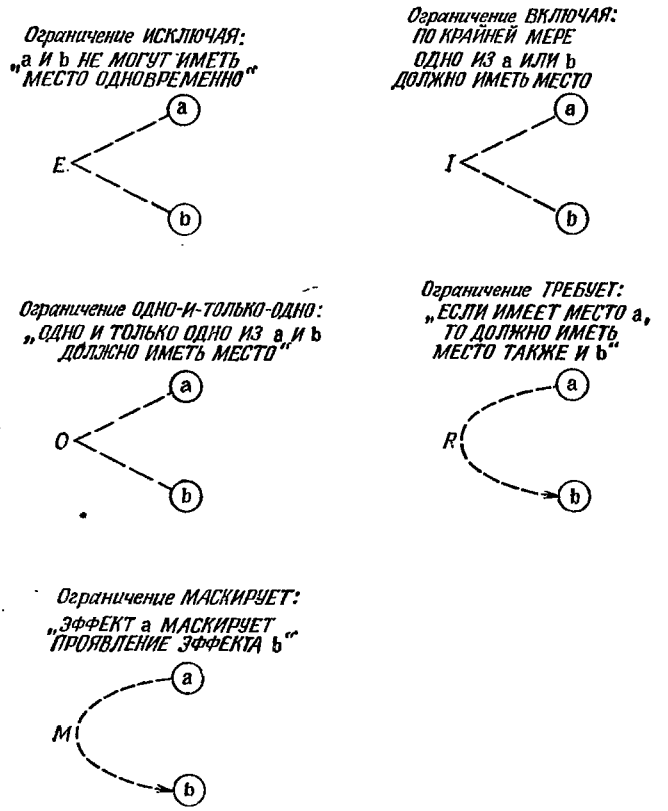


Рис. 12.5. Ограничения на функциональные связи.

Эффект 33 может быть вызван девятью новыми комбинациями, представленными столбцами 10—18. Последний эффект, 34-й, имеет место всегда, когда промежуточная вершина 23 находится в состоянии «нуль». Это бывает, когда в том же состоянии находится хотя бы одна из вершин: 1, 2, 21 и 22. Мы могли бы представить это четырьмя столбцами, в каждом из которых одной из вершин соответствует «нуль», а другим — «безразлично» (т. е. их состояние несущественно). Однако качество тестов можно улучшить *повышением чувствительности к путям*. В соответствии с этим принципом в случае, когда любая из нескольких ситуаций может произвести один и тот же эффект, столбцы следует определить так, чтобы этот эффект был связан только с одной ситуацией. Например, если бы мы устранили ситуации 1 и 2 одновременно, мы могли бы не обнаружить такую ошибку, когда программа проверяет одно условие и не проверяет другого. Поэтому столбцы 19—22 организованы таким обра-

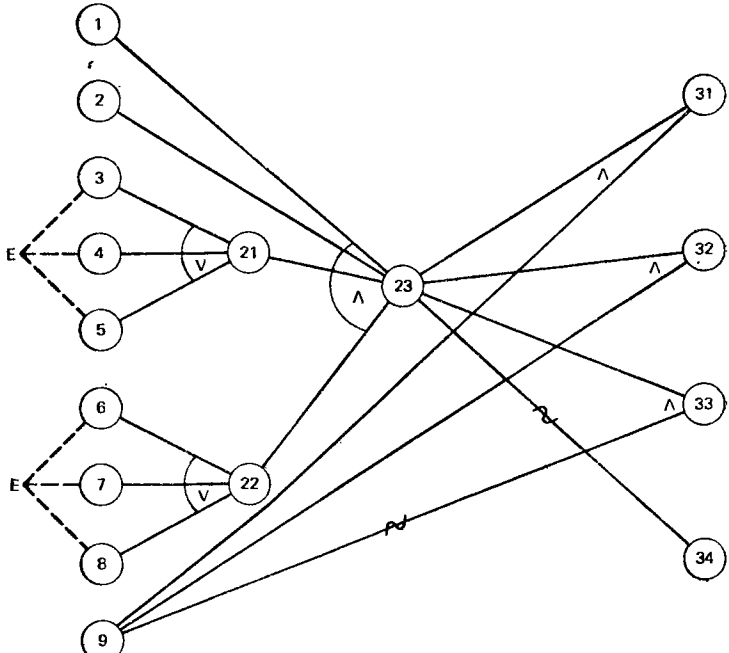


Рис. 12.6. Окончательная функциональная диаграмма.

зом, чтобы только одна вполне определенная ситуация приводила к эффекту 34.

На рис. 12.7 изображена полученная таблица решений. Описаны входные условия и ожидаемые результаты для каждого из 22 тестов. Заметьте, что число построенных тестов значительно меньше, чем в результате примитивного комбинирования всех ситуаций. Общее число таких комбинаций — 512 (девять ситуаций, для каждой — два состояния), но они сведены лишь к 22 существенно различающимся случаям. Если бы тестовик не применял функциональных диаграмм, он мог бы начать с выявления всех 512 комбинаций, а затем сохранил бы только 20—30 из них из-за временных ограничений. Однако, он, вероятно, сделал бы это более или менее произвольно, так что его 20 или 30 тестов были бы далеко не оптимальным подмножеством. Достоинство функциональных диаграмм именно в том, что благодаря им выявляется ограниченное число высокорезультативных тестов.

Привлекая интуитивные доводы, можно еще сильнее сократить число тестов. Тесты 10—18 соответствуют случаям, когда строка 1 не будет найдена в текущей строке файла. Интуитивно ясно, что программа в таком случае не должна быть чувствительной к свойствам строки 2 (ситуации 6—8). Поэтому такие комбинации, как в

Ситуации \ Тесты	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	S
2	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	S	I
3	I	I	I	S	S	S	S	S	S	I	I	I	S	S	S	S	S	S	S	S	S	S
4	S	S	S	I	I	I	S	S	S	S	S	S	I	I	I	S	S	S	S	S	S	S
5	S	S	S	S	S	S	I	I	I	S	S	S	S	S	S	I	I	I	S	I	I	I
6	I	S	S	I	S	S	I	S	S	I	S	S	I	S	S	I	S	S	S	S	S	S
7	S	I	S	S	I	S	S	I	S	S	I	S	S	I	S	S	I	S	S	S	S	S
8	S	S	I	S	S	I	S	S	I	S	S	I	S	S	I	S	S	I	I	S	I	I
9	I	I	I	I	I	I	I	I	I	S	S	S	S	S	S	S	S	S	X	X	X	X
31	P	P	P	P	P	P	P	P	P	A	A	A	A	A	A	A	A	A	A	A	A	A
32	P	P	P	P	P	P	P	P	P	A	A	A	A	A	A	A	A	A	A	A	A	A
33	A	A	A	A	A	A	A	A	A	P	P	P	P	P	P	P	P	P	A	A	A	A
34	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	P	P	P	P

Эффекты

S: *не имеет места*
I: *имеет место*
X: *несущественно*
A: *отсутствует*
P: *присутствует*

Рис. 12.7. Представление диаграммы в виде таблицы решений.

тестах 11, 12, 14, 15, 17 и 18, вероятно, не будут иметь высокую результативность, и их можно отбросить, сократив число тестов до 16. Компромиссы такого рода имеют чисто экономический характер и вполне соответствуют аксиоме, согласно которой один из элементов искусства тестирования в том и состоит, чтобы знать, когда остановиться. Они часто бывают нужны, когда функциональная диаграмма приводит к большому числу возможных тестов. В таких случаях, если экономические соображения превалируют, следует, основы-

ваясь на оценке результативности тестов, затрат на их подготовку и серьезности для пользователя обнаруживаемой ошибки, расставить приоритеты тестов и отбросить тесты с низким приоритетом.

Анализ метода

Метод функциональных диаграмм конкретизирует некоторые из аксиом тестирования, обсуждавшиеся в гл. 10 и 11. Предсказание ожидаемых результатов — существенный элемент метода. Отсутствие ожидаемых эффектов подсказывает тестовику, что следует поискать ошибочные побочные эффекты. Например, согласно тестам 10—18, кроме проверки того, что сообщение «НЕ НАЙДЕНО» выдается, необходимо проверить отсутствие эффекта 32 (т. е. убедиться, что рассматриваемая строка не изменена).

Этот метод обладает и некоторыми недостатками. Он не «нацелен» на тесты большой «разрушительной силы», поэтому получаемые с его помощью тесты должны дополняться тестами, основанными на подозрениях, интуиции, изобретательности тестовика. Рис. 12.7 следует дополнить тестами, включающими такие случаи, как наличие специальных символов (табуляция или возврат каретки) в строке 1 и строке 2, вхождение искомой строки в самое начало или в самый конец рассматриваемой строки. Функциональные диаграммы могут описывать только комбинаторные соотношения между ситуациями. Они не помогают учитывать такие особенности, как временные задержки, итеративные функции, обратную связь эффектов и ситуаций.

В фирме IBM было разработано несколько экспериментальных программ получения функциональных диаграмм, с тем чтобы труд по анализу диаграммы и построению таблицы решений снять с плеч программиста [2]. Эти программы используют алгоритмы проектирования электронных схем и минимизации графов для построения оптимальных наборов тестов.

Подводя итог, можно сказать, что метод функциональных диаграмм является строгим подходом к преобразованию спецификаций на естественном языке к более формальному виду. Он вырабатывает набор тестов, который экономичнее любого метода ad hoc или метода грубой силы и имеет дополнительное достоинство — помогает обнаружить неоднозначности в спецификациях. Те, кто пользовался этим методом, часто находят, что он требует слишком много сил и времени, но, повторяя сказанное в гл. 11, напомним: никто и никогда не утверждал, что тестирование — легкое дело. Построение функциональных диаграмм — трудная работа, но результаты стоят труда.

ИНТЕГРАЦИЯ СИСТЕМЫ

Для большинства проектов шесть подходов, описанных в гл. 10,— это все, что нужно для сборки системы из частей (модулей). Бывают,

однако, уникальные проекты, связанные с разработкой чрезвычайно больших систем или систем, различные части которых разрабатываются в разных географических точках или разными субподрядными организациями. В таких проектах возникают особого рода логические проблемы, а также проблемы планирования и управления.

Планирование сборки

Процесс интеграции таких систем начинается с подготовки *плана сборки*, или *интеграции*, в котором уточняется, как, кем, где и когда будет выполняться сборка системы из отдельных компонент. Основная задача этого плана — описать процесс постепенной сборки системы. Каждая последовательная версия, или уровень системы, называется *спином* (часто используется термин *драйвер*, но его можно спутать с понятием драйвера модулей, рассматривавшимся в гл. 11). Спин 0 обозначает «пустую» версию системы, на базе которой строятся все последующие спины. Для новой прикладной системы спин 0 может содержать просто базовую операционную систему. Если задача проекта состоит в получении нового выпуска операционной системы, спин 0 — это предыдущий ее выпуск. Спин N обозначает последний спин, т. е. законченную систему.

В плане сборки, кроме прочих вопросов, указывается число спинов, календарный план их разработки и их функциональное содержание. Ключевым моментом процесса является принятие решения о числе спинов. Одна крайность — единственный спин (по аналогии с методом «большого скачка»), другая крайность — новый спин ежедневно, причем новый спин отличается от старого добавлением всего лишь одного модуля. Есть основания считать последний случай наиболее предпочтительным. Такая стратегия называется *непрерывной интеграцией*. Она предполагает, что новые спины создаются часто; возможно, ежедневно, максимум — еженедельно. Каждый спин отличается от предыдущего лишь немногим, и это облегчает отладку и позволяет «забраковать» всякую новую функцию, которая ведет себя неправильно.

В качестве примера, демонстрирующего достоинства непрерывной интеграции, можно вспомнить, что каждый выпуск гигантской операционной системы OS/360 IBM строился с помощью лишь нескольких крупных спинов. Некоторые выпуски строились только с одним спином — это означает, что буквально тысячи модулей сколачивались в систему одновременно. Другие выпуски строились в четыре спина. Спины разрабатывались автономно около двух месяцев, и каждый из них содержал сотни и даже тысячи новых модулей. При этом подходе выявилось несколько серьезных проблем. Каждый этап сборки был необъятным, что вело к утере модулей, логическим тупикам, отнимало многие недели на то, чтобы добиться

хотя бы какого-то функционирования спина. Отладка была трудна, потому что впервые стыковались друг с другом большие куски программы. Исправления ошибок задерживались, так как коррекции собирались в группы и выполнялись довольно редко.

Непрерывная интеграция впервые была применена к OS при работе над выпуском OS/VS2. Спины создавались почти ежедневно. Тот факт, что это был первый выпуск OS, законченный с опережением графика, во многом объясняется выгодами непрерывной интеграции.

План сборки зависит от разбиения системы на *сборочные узлы*. Сборочный узел — это набор модулей, который организация-разработчик будет собирать в конкретный спин. Сборочный узел обычно представляет некоторую функцию системы, более крупную, чем модуль, но меньшую, чем подсистема. Основная часть процесса планирования сборки — выделение сборочных узлов и составление календарного плана их разработки. Эта работа требует достаточно высокой квалификации в технических и коммерческих вопросах, а также в искусстве переговоров и определяется следующими факторами:

1. План сборки должен учитывать ограничения, связанные с ресурсами и графиком организаций-разработчиков.
2. Последовательность сборки должна быть определена таким образом, чтобы работающая каркасная версия системы была готова уже в качестве одного из первых спинов.
3. Последовательность сборки должна быть такой, чтобы все важнейшие функции попали в первые спины.
4. Сборочные узлы могут быть *зависимы* от других сборочных узлов (например, функция обработки депозитных сообщений будет зависеть от некоторых функций управления базой данных). Последовательность сборки должна быть определена с учетом этой зависимости.
5. Во всех спинах, кроме последнего, некоторые фрагменты системы будут отсутствовать. Эти отсутствующие фрагменты часто приходится имитировать написанными специально для этого программами, называемыми «*подпорками*» (*scaffold code*) (аналогичными по идее *заглушкам*). План сборки должен по возможности минимизировать потребность в «подпорках».

План сборки должен быть таким, чтобы каждый модуль системы подсоединялся *только один раз* (кроме случаев исправления ошибок). Если это кажется невозможным, значит, либо что-то неправильно в проекте системы, либо неправильно выделены сборочные узлы. Так как план сборки связан с планами и графиками работы организаций-разработчиков, его следует составить заранее, во всяком случае, до начала кодирования.

Другой ключевой элемент плана сборки — некоторый измеримый *критерий приемки*, например требование такого качества каж-

дого сборочного узла, чтобы после его сборки было обнаружено не более двух ошибок на 1000 операторов программы. Последнее часто вызывает удивление, потому что нет способа, которым бы организация-разработчик измеряла эту величину в момент сборки. Однако даже и в таком виде требование это имеет некоторый положительный психологический эффект. Поскольку организации-разработчики знают, что их будут публично оценивать по этому критерию, создается атмосфера здорового соперничества, положительно влияющая на работу по проектированию и тестированию.

Управление сборкой

Оставшийся этап интеграции системы — процесс непосредственно конструирования каждого спина. Ключ к этой деятельности — *управление*. Организация, выполняющая интеграцию, должна точно знать, из чего состоит каждый уровень системы. Единственный способ внести любое изменение в систему — через очередной сборочный узел или формальную процедуру исправления ошибок. Все исправления ошибок должны делаться заменой исходного текста целого модуля. Другие виды изменений, такие, как «сверхзаплаты» (заплаты в объектной программе) или «карты коррекции» (заплаты в исходной программе), допускаться не должны. Даже организация, выполняющая интеграцию, сама не должна изменять систему. Когда обнаружена ошибка, организация должна, если необходимо, изъять из системы весь неправильный сборочный узел и официально известить об этом организацию-разработчика.

Организация, выполняющая интеграцию, отвечает за сопровождение очередного спина в контролируемых ею библиотеках, утверждение всех изменений (что предполагает анализ текста всех изменений на исходном языке), сборку каждого спина, документирование способа использования и ограничений каждого спина, а также рассылку каждого спина организациям-разработчикам. Эта организация сначала рассылает спин 0 и затем ждет получения всех сборочных узлов, запланированных для следующего спина. Она добавляет эти узлы к прежнему спину (создавая таким образом новый спин), добавляет все необходимые программы-подпорки, *тестирует регрессию* нового спина, чтобы убедиться, что система не регрессирует, и затем распространяет новый спин по организациям-разработчикам. После того как создан последний спин, продукт готов к комплексному тестированию.

Каждая группа разработчиков выполняет одновременно с другими группами следующие шаги: она выполняет автономное тестирование и тестирование сопряжений модулей, подлежащих сборке в следующий спин; затем она включает эти модули в свою копию предыдущего спина и выполняет тестирование внешних функций, эти модули отсылаются группе, занимающейся интеграцией, и про-

цесс повторяется со следующим спином. Когда дело доходит до последнего спина, группа разработчиков должна повторно выполнить тестирование внешних функций параллельно с комплексным тестированием.

Наконец, нужно добиться того, чтобы каждая группа разработчиков заменяла свою копию прежнего спина, как только станет доступным следующий спин. Таким образом гарантируется, что все организации используют копии одной и той же, и самой последней, версии системы.

КОМПЛЕКСНОЕ ТЕСТИРОВАНИЕ

Комплексное тестирование, вероятно, самая непонятная форма тестирования. Во всяком случае, комплексное тестирование *не* является тестированием всех функций полностью собранной системы; тестирование такого типа называется тестированием внешних функций. Комплексное тестирование — процесс поисков несоответствия системы ее исходным целям. Элементами, участвующими в комплексном тестировании, служат сама система, описание целей продукта и вся документация, которая будет поставляться вместе с системой. Внешние спецификации, которые были ключевым элементом тестирования внешних функций, играют лишь незначительную роль в комплексном тестировании.

В гл. 4 подчеркивалась важность постановки измеримых целей. Часть аргументов в пользу этого должна быть уже очевидной: измеримые цели необходимы, чтобы определить правила для процессов проектирования. Остальные соображения должны проясниться сейчас. Если цели сформулированы, например, в виде требования, чтобы система была достаточно быстрой, вполне надежной и чтобы в разумных пределах обеспечивалась безопасность, тогда нет способа определить при тестировании, в какой степени система достигает своих целей.

Если вы не сформулировали цели вашего продукта или если эти цели не измеримы, вы не можете выполнить комплексное тестирование.

Комплексное тестирование может быть процессом и контроля, и испытаний. Процессом испытаний оно является тогда, когда выполняется в реальной среде пользователя или в обстановке, которая специально создана так, чтобы напоминать среду пользователя. Однако такая роскошь часто недоступна по ряду причин, и в подобных случаях комплексное тестирование системы является процессом контроля (т. е. выполняется в имитируемой, или тестовой, среде). Например, в случае бортовой вычислительной системы космического корабля или системы противоракетной защиты вопрос о реальной среде (запуск настоящего космического корабля или вы-

стрел настоящей ракетой) обычно не стоит. Кроме того, как мы увидим дальше, некоторые типы комплексных тестов не осуществимы в реальной обстановке по экономическим соображениям, и лучше всего выполнять их в моделируемой среде.

Проектирование комплексного теста

Комплексное тестирование — наиболее творческий из всех обсуждавшихся до сих пор видов тестирования. Разработка хороших комплексных тестов требует часто даже больше изобретательности, чем само проектирование системы. Здесь нет простых рекомендаций типа тестирования всех ветвей или построения функциональных диаграмм. Однако следующие 14 пунктов дают некоторое представление о том, какие виды тестов могут понадобиться:

1. *Тестирование стрессов.* Распространенный недостаток больших систем состоит в том, что они функционируют как будто бы нормально при слабой или умеренной нагрузке, но выходят из строя при большой нагрузке и в стрессовых ситуациях реальной среды. Тестирование стрессов представляет собой попытки подвергнуть систему крайнему «давлению», например попытку одновременно подключить к системе разделения времени 100 терминалов, насытить банковскую систему мощным потоком входных сообщений, или посылку системе управления процессами аварийных сигналов от всех ее процессов.

Одна из причин, по которой тестирование стрессов опускают (кроме очевидных логических проблем, рассматриваемых в следующем разделе), состоит в том, что персонал, занимающийся тестированием, хотя и признает потенциальную пользу таких тестов, считает, что столь жесткие стрессовые ситуации никогда не возникнут в реальной среде. Это предположение редко оправдывается. Например, бывают случаи, когда все пользователи системы разделения времени пытаются подключиться в одно и то же время (например, когда произошел отказ системы на минуту-две и система только что восстановлена). У банковских систем, обслуживающих терминалы покупателей, бывают пиковые нагрузки в первые часы работы магазинов и в час обеденного перерыва. Более тщательное тестирование стрессов помогло бы избежать перегрузки программного обеспечения программы «Аполлон», которая возникла при приземлении «Аполлона-11».

2. *Тестирование объема.* В то время как при тестировании стрессов делается попытка подвергнуть систему серьезным нагрузкам в короткий интервал времени, тестирование объема представляет собой попытку предъявить системе большие объемы данных в течение более длительного времени. Если система обрабатывает многотомные файлы на ленте или на дисках, то при тестировании объема ей

подается настолько много данных, чтобы заставить ее перейти с одного тома на другой. На вход компилятора следует подать до нелепости громадную программу. Программе обработки текстов — огромный документ. Очередь заданий операционной системы следует заполнить до предела. Цель тестирования объема — показать, что система или программа не могут обрабатывать данные в количествах, указанных в их спецификациях.

3. *Тестирование конфигурации.* Многие системы, например операционные системы или системы управления файлами, обеспечивают работу различных конфигураций аппаратуры и программного обеспечения. Число таких конфигураций часто слишком велико, чтобы можно было проверить все варианты. Однако следует тестировать по крайней мере максимальную и минимальную конфигурации. Система должна быть проверена со всяким аппаратным устройством, которое она обслуживает, или со всякой программой, с которой она должна взаимодействовать. Если сама программная система допускает несколько конфигураций (т. е. покупатель может выбрать только определенные части или варианты системы), должна быть тестирована каждая из них.

4. *Тестирование совместимости.* В большинстве своем разрабатываемые системы не являются совершенно новыми; они представляют собой улучшение прежних версий или замену устаревших систем. В таких случаях на систему, вероятно, накладывается дополнительное требование совместимости, в соответствии с которым взаимодействие пользователя с прежней версией должно полностью сохраниться и в новой системе. Например, возможно, потребуется, чтобы в новом выпуске операционной системы язык управления заданиями, язык общения с терминалом и скомпилированные прикладные программы, использовавшиеся раньше, могли использоваться без изменений. Такие требования совместимости следует тестировать. Как периодически подчеркивалось при разговоре обо всех других формах тестирования, цель при тестировании совместимости должна состоять в том, чтобы показать наличие несовместимости.

5. *Тестирование защиты.* Так как внимание к вопросам сохранения секретности в сегодняшнем автоматизированном обществе возрастает, к большинству систем предъявляются определенные требования по обеспечению защиты от несанкционированного доступа. Например, операционная система должна устранить всякую возможность для программы пользователя увидеть данные или программу другого пользователя. Административная информационная система не должна позволять подчиненному получить сведения о зарплате тех, кто стоит выше его на служебной лестнице. Цель тестирования защиты — нарушить секретность в системе. Один из методов — нанять профессиональную группу «взломщиков», т. е. людей с опытом разрушения средств обеспечения защиты в системах.

6. *Тестирование требований к памяти.* При проектировании многих систем ставятся цели, определяющие объем основной и вторичной памяти, которую системе разрешено использовать в различных условиях. С помощью специальных тестов нужно попытаться показать, что система этих целей не достигает.

7. *Тестирование производительности.* Требования к производительности и эффективности, например время ответа и пропускная способность для различных нагрузок и различных конфигураций,— важная часть большинства проектов систем. По сравнению с другими типами комплексного тестирования системы о тестировании производительности известно очень много, этому предмету посвящена даже целая книга [4].

8. *Тестирование настройки.* К сожалению, процедуры настройки многих систем сложны. Например, есть фирмы, которые специализируются на настройке за определенную плату OS/360 IBM. Тестирование процесса настройки системы очень важно, поскольку одна из наиболее обескураживающих ситуаций, с которыми сталкивается покупатель, заключается в том, что он оказывается не в состоянии даже настроить новую систему.

9. *Тестирование надежности/готовности.* Ключевой момент в комплексном тестировании заключается в попытке доказать, что система не удовлетворяет исходным требованиям к надежности (среднее время между отказами, количество ошибок, способность к обнаружению, исправлению ошибок и/или устойчивость к ошибкам и т. д.). Тестирование надежности крайне сложно, и все же следует постараться тестировать как можно больше этих требований. Например, в систему можно намеренно внести ошибки (как аппаратные, так и программные), чтобы тестировать средства обнаружения, исправления и обеспечения устойчивости. Другие требования к надежности тестировать почти невозможно. Если требуется, чтобы среднее время между отказами системы было равно 250 часам, система должна работать значительно больше 250 часов, чтобы можно было статистически достоверно установить, что она удовлетворяет этому требованию. Невозможно показать, что в момент поставки в программном обеспечении системы больше ошибок, чем определено требованиями, потому что, если бы вы могли подсчитать оставшиеся ошибки, вы смогли бы и исправить их. В гл. 18 рассматриваются некоторые простые методы, которые можно использовать для проверки этих требований.

10. *Тестирование средств восстановления.* Важная составная часть требований к операционным системам, системам управления базами данных и системам передачи данных — обеспечение способности к восстановлению, например восстановлению утраченных данных в базе данных или восстановлению после отказа в телекоммуникационной линии. Во многих проектах совершенно забывают о тестировании соответствующих средств. Лучше всего попытаться

показать, что эти средства работают неправильно, при комплексном тестировании системы.

11. *Тестирование удобства обслуживания.* Либо в требованиях к продукту, либо в требованиях к проекту должны быть перечислены задачи, определяющие удобство обслуживания (сопровождения) системы. Все сервисные средства системы, например программы выдачи дампов памяти, программы трассировки, диагностических сообщений,— все это нужно проверять при комплексном тестировании. Все документы, описывающие внутреннюю логику, следует проанализировать глазами обслуживающего персонала, чтобы понять, как быстро и точно можно указать причину ошибки, если известны только некоторые ее симптомы. Все средства, обеспечивающие сопровождение и поставляемые вместе с системой, также должны быть проверены.

12. *Тестирование публикаций.* Проверка точности всей документации для пользователя является важной частью комплексного тестирования. Все комплексные тесты следует строить только на основе документации для пользователя. В частности, должна быть проверена правильность всех приводимых в публикациях примеров.

13. *Тестирование психологических факторов.* Хотя во время тестирования системы следует проверить и психологические факторы, эта сторона не так важна, как другие, потому что обычно при тестировании уже слишком поздно исправлять серьезные просчеты в таких вопросах. Вот почему так важны методы моделирования, рассматривавшиеся в главе 4 (чтобы обнаружить психологические несоответствия еще в процессе внешнего проектирования). Однако мелкие недостатки могут быть обнаружены и устранены при тестировании системы. Например, может оказаться, что ответы или сообщения системы плохо сформулированы или ввод команды пользователя с терминала требует постоянных переключений верхнего и нижнего регистров.

14. *Тестирование удобства эксплуатации.* Большинство систем обработки данных либо является компонентами более крупных систем, предполагающих деятельность человека, либо сами регламентируют такую деятельность во время своей работы. Нужно проверить, что вся эта деятельность, например поведение оператора или пользователя за терминалом, удовлетворяет определенным условиям.

Не все из перечисленных 14 пунктов применимы к тестированию всякой системы (например, когда тестируется отдельная прикладная программа), но тем не менее это перечень вопросов, которые разумно иметь в виду.

Основное правило при комплексном тестировании — «все годится». Пишите разрушительные тесты, проверяйте все функцио-

нальные границы системы, пишите тесты, представляющие ошибки пользователя (например, оператора системы). Нужно, чтобы тестовик думал так же, как пользователь или покупатель, а это предполагает доскональное понимание того, для чего система будет применяться. Поэтому возникает вопрос, кто же должен выполнять комплексное тестирование, и в частности кто должен проектировать тесты. Во всяком случае, этого не должны делать программисты или организации, ответственные за разработку системы. Группа тестирования системы должна быть независимой организацией и должна включать профессиональных специалистов по комплексному тестированию систем, несколько пользователей, для которых система разрабатывалась, основных аналитиков и проектировщиков системы и, возможно, одного или двух психологов. Очень важно включить самих проектировщиков системы. Это не противоречит аксиоме, согласно которой невозможно тестировать свою собственную систему, поскольку система прошла через много рук после того, как ее описали архитектор или проектировщик. В действительности проектировщик и не пытается тестировать собственную систему; он ищет расхождения между окончательной версией и тем, что он первоначально — возможно, год или два назад — имел в виду. Для комплексного тестирования желательно также иметь информацию о рынке или пользователях, уточняющую предположения о конфигурации и характере применения системы.

Комплексное тестирование системы — такая особая и такая важная работа, что в будущем возможно появление компаний, специализирующихся в основном на комплексном тестировании систем, разработанных другими. Уже и сейчас в рамках ВВС США действует рекомендация, в соответствии с которой в случае, если разрабатывающий систему подрядчик не имеет опыта, достаточного для самостоятельного комплексного тестирования, то на комплексное тестирование его системы должен быть заключен отдельный контракт с другой компанией [5].

Как уже упоминалось, компонентами комплексного теста являются исходные цели, документация, публикации для пользователей и сама система. Все комплексные тесты должны быть подготовлены на основе публикаций для пользователя (а не внешних спецификаций). Ко внешним спецификациям обращаться следует только для того, чтобы разбираться в противоречиях между системой и публикациями о ней.

По своей природе комплексные тесты никогда не сводятся к проверке отдельных функций системы. Они часто пишутся в форме сценариев, представляющих целый ряд последовательных действий пользователя. Например, один комплексный тест может представлять подключение терминала к системе, выдачу последовательно 10—20 команд и затем отключение от системы. Вследствие их особой сложности тесты системы состоят из нескольких компонент: сцена-

рия, входных данных и ожидаемых выходных данных. В сценарии точно указываются действия, которые должны быть совершены во время выполнения теста.

Выполнение комплексного теста

Комплексное тестирование находится у самой левой границы спектра, описанного в гл. 10. При этом вся система рассматривается как черный ящик; в частности, несущественно, какие участки программы затрагивает комплексный тест.

Один из методов, позволяющих вовлечь в тестирование пользователей,— *опытная эксплуатация*. Для проведения опытной эксплуатации с одной или несколькими организациями пользователей заключаются контракты на установку у них созданной системы. Это часто выгодно обеим сторонам; организация-разработчик оповещается об ошибках в программном обеспечении, которые она не заметила сама, а организация-пользователь получает возможность изучить систему и экспериментировать с ней до того, как она станет доступной официально.

Второй полезный метод — использовать систему в организации-изготовителе для внутренних нужд. Это можно сделать часто, но далеко не всегда (например, компании по разработке программного обеспечения негде использовать систему управления процессом очистки нефти). Так, производитель ЭВМ может установить новую операционную систему для опытной эксплуатации во всех своих внутренних вычислительных центрах, прежде чем начинать поставлять ее пользователям, а компания по разработке программного обеспечения может использовать свои собственные компиляторы, программы обработки текстов, начисления зарплаты и т. д. Одной из причин относительного успеха операционной системы TSO (Time-Sharing Option) фирмы IBM был тот факт, что она использовалась для разработки прикладных программ в самой фирме IBM до того, как началось ее распространение.

Воспользуйтесь собственным продуктом, прежде чем передавать его другим.

При комплексном тестировании часто начинают с простых тестов, приберегая более сложные тесты к концу. Так делать не нужно, потому что комплексное тестирование приходится на самый конец цикла разработки, так что на отладку и исправление найденных ошибок остается мало времени. Поскольку сложные тесты часто обнаруживают более сложные для исправления ошибки, измените последовательность: начните с самых трудных тестов, а затем переходите к более простым.

Хотя процессы тестирования рассматривались последовательно, это не означает, что и реализовывать их нужно также последова-

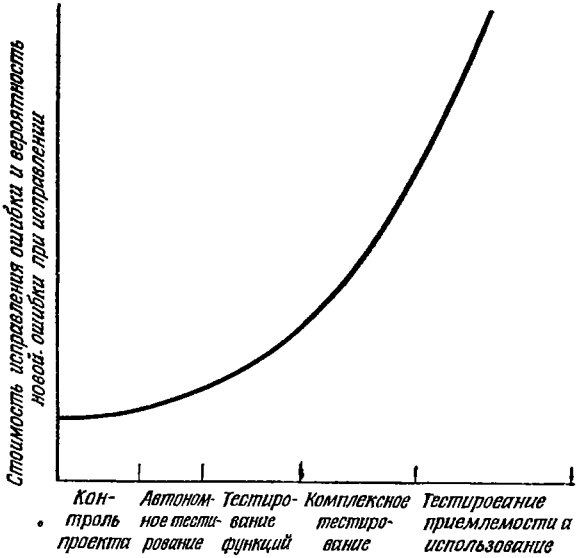


Рис. 12.8. Зависимость стоимости исправления ошибки от времени.

тельно; в случае необходимости разные процессы тестирования больших проектов часто перекрываются. Например, организация, занимающаяся комплексным тестированием системы, может выполнять тесты, относящиеся ко многим из описанных 14 классов, еще до того, как вся система будет собрана.

На рис. 12.8 отражены две закономерности, связывающие исправление ошибок и календарный план. Первая из них, в соответствии с которой стоимость исправления ошибок быстро растет для более поздних этапов цикла разработки, должна быть очевидной для большинства читателей. Эта же кривая описывает, однако, и другое, менее известное соотношение. Вероятность внести новую ошибку при исправлении старой также быстро растет на последних этапах. Это явление представляет особый интерес, потому что ошибка, внесенная при исправлении другой ошибки, может быть хуже первоначальной. Возникает важный вопрос: есть ли какой-то смысл в том, чтобы намеренно не исправлять ошибку в программе?

Прежде чем отвечать на этот вопрос, мы, чтобы лучше понять проблему, рассмотрим следующую гипотетическую ситуацию. Предположим, что серьезность каждой ошибки можно оценить, основываясь на ее воздействии на пользователя, числом от 1 до 10 (напомним, что серьезность ошибки не связана непосредственно с размерами необходимых для ее устранения исправлений). Предположим, что при исправлении ошибки во время тестирования системы вероятность не исправить ошибку равна 0.2, а вероятность вне-

сти новую ошибку — 0.3 и значения оценки серьезности этой новой ошибки распределены равномерно между 1 и 10. Из этого следует, что, может быть, лучше воздержаться и сознательно отказаться от исправления не очень серьезных ошибок, потому что их исправление не улучшит, а только ухудшит систему.

Такую пилюлю не просто проглотить, особенно учитывая тот факт, что книга посвящена надежности. С одной стороны, проектировщик хотел бы утверждать, что он выпустил систему, в которой не известно ни одной ошибки, но, с другой стороны, он должен учитывать только что описанный феномен. Конечно, чем лучше структурирована система, тем меньше этот феномен проявляется, но свидетельств в пользу того, что он совсем исчезает в случае систем с хорошей структурой, нет.

Из этого следует, что для каждой «мелкой» ошибки, обнаруженной при тестировании системы и на других поздних этапах тестирования, приходится отдельно решать, стоит ли исправлять эту ошибку или следует отложить исправление до следующей версии системы. Ключ к такому решению — оценка ошибок по их серьезности. Но мнение программиста о серьезности ошибки часто значительно расходится с мнением пользователя. Всякое решение отложить исправление должно быть основано на полном понимании условий работы пользователя.

ИНСТРУМЕНТЫ ДЛЯ ТЕСТИРОВАНИЯ ВНЕШНИХ ФУНКЦИЙ И КОМПЛЕКСНОГО ТЕСТИРОВАНИЯ

В 1969 г. я имел счастливую возможность принять участие в небольшом проекте по разработке операционной системы реального времени и прикладных программ для системы, автоматизирующей тестирование интегральных схем и выполняющей в режиме реального времени инженерные эксперименты с электронными схемами. Система была, как это показано на рис. 12.9, связана с большой мультипроцессорной системой на базе OS/360, что позволяло передавать OS для более эффективного выполнения большие программы обработки данных и сами данные.

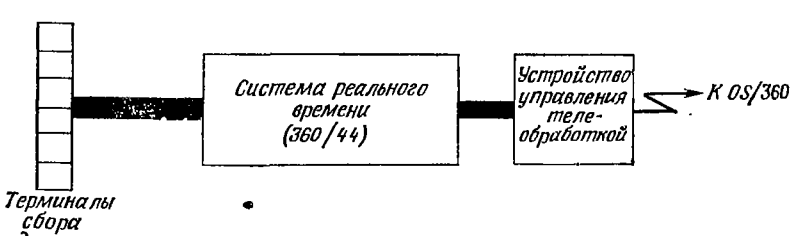


Рис. 12.9. Система обработки технологической информации.

Моя работа в основном состояла в проектировании, кодировании и тестировании всего программного обеспечения телеобработки в операционной системе. Планируя процесс тестирования, я тотчас же столкнулся с тремя вопросами:

1. Как можно тестировать внешние функции программного обеспечения телеобработки без устройства управления телепередачей (которое еще не выпускалось)?

2. Поскольку лица, ответственные за данную установку OS/360, весьма прохладно относились к идее подключения этой «новоиспеченной» системы, как тестировать прикладные программы, использующие функции телеобработки, без подключения к OS/360?

3. Как можно обнаруживать и отлаживать ошибки телепередачи после того, как две эти системы будут связаны?

Для решения этих проблем были разработаны три программы: *имитатор устройства*, *имитатор терминала* или *удаленной системы* и *монитор ввода-вывода*. Имитатор устройства моделировал характеристики устройства управления телепередачей и использовался для тестирования телекоммуникационного метода доступа (программного обеспечения ввода-вывода низкого уровня), включая все его процедуры восстановления после ошибки. Имитатор перехватывал все приказы операционной системы начать операции ввода-вывода и моделировал прерывания от устройства управления. С помощью управляющих записей можно было заставить имитатор сгенерировать конкретную ошибку ввода-вывода в конкретном типе операции ввода-вывода и указать ему, должна ли быть ошибка постоянной или это нерегулярный сбой, какую временную задержку сделать для этой операции и какие данные выдавать в качестве результата операции чтения. Отметим, что многие из этих условий было бы трудно проверить даже в случае, если бы аппаратное устройство имелось.

Второй имитатор был написан для тестирования функции объединения прикладных программ телеобработки и метода доступа. Этот имитатор перехватывал все элементарные операции ввода-вывода, передаваемые устройству управления, и моделировал работу удаленной OS/360, включая все операции управления линией связи и необходимые временные задержки. Когда данные посылались OS/360, имитатор возвращал соответствующее подтверждение для последующего анализа. Периодически имитатор брал инициативу в свои руки и начинал «разговор» с системой реального времени, посылая поток заранее определенных данных.

Третий инструмент — монитор ввода-вывода — был разработан для комплексного тестирования. Оператор системы реального времени мог инициировать работу монитора как параллельной задачи низкого приоритета и указать ему на любое устройство ввода-вывода. Монитор перехватывал все данные, пересылаемые этому

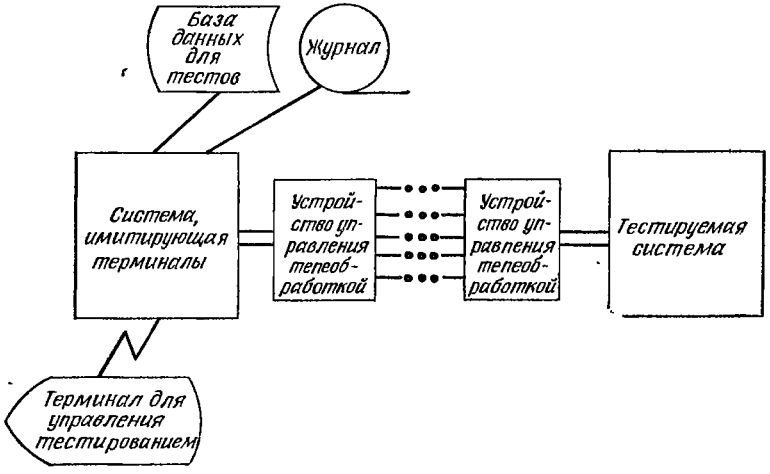


Рис. 12.10. Конфигурация с имитатором терминала.

устройству и от него, все связи с канальными программами, все прерывания и сохранял эти данные в кольце буферов. Когда не было других активных задач, монитор пересылал эти данные в ленточный файл. Была написана также вспомогательная программа, анализирующая ленточные файлы.

Во время комплексного тестирования монитор настраивался на наблюдения за работой устройства управления телепередачей. В первую же неделю тестирования системы была обнаружена ситуация, когда связь между двумя подсистемами внезапно прерывалась. Анализ лент трассировки показал, что обе системы пытались одновременно передать данные друг другу, так как обе неправильно истолковывали соглашения об управлении линией. Чтобы обнаружить такую ошибку без монитора, понадобилось бы несколько недель, а не часов.

Самый распространенный инструмент тестирования диалоговых систем или прикладных программ — *имитатор терминала.* Обычная конфигурация имитатора терминала изображена на рис. 12.10. Вместо того чтобы связывать тестируемую систему с реальными терминальными устройствами, ее соединяют с другой системой, которая моделирует много терминалов, извлекая тесты из базы данных и передавая их (как команды с терминала) проверяемой системе в различных вариантах и по разным линиям связи. Это позволяет изменять конфигурации терминалов, чтобы тестировать тяжелые стрессы, значительную нагрузку, большие объемы данных, не сталкиваясь с логическими проблемами при использовании настоящих терминалов. Обычно во время тестирования имитатор записывает все сообщения в регистрационном файле.

Примером успешного применения имитатора терминалов было использование Имитатора системы коллективного пользования MUSE (Multi-User Environment Simulator) для тестирования системы разделения времени TTY RESPOND для серии CD 6000 [6]. MUSE подает системе RESPOND набор заранее определенных команд, моделируя действия до 64 терминалов. MUSE имеет интересную особенность, позволяющую ему изменять свои сообщения в зависимости от ответов RESPOND. Выдаваемая последовательность сообщений может изменяться с помощью расположенных в тесте «условных переходов», зависящих от ответов, получаемых от системы RESPOND.

MUSE использовалась в первую очередь для обнаружения временных несоответствий и ситуаций насыщения в системе RESPOND. Во время тестирования было обнаружено несколько серьезных ошибок, например таких, которые возникали только в случае, когда кратное девяти число пользователей за терминалами запрашивало функцию сортировки или когда в очередь обмена с диском поступал тринадцатый запрос. Есть подозрение, что эти ошибки не были бы обнаружены при тестировании системы, если бы не использовался имитатор терминалов [6].

Множество имитаторов терминалов было разработано и специально приспособлено для отдельных проектов. Характерными примерами имитаторов широкого назначения могут служить драйверная система DB/DC фирмы IBM [7] и эмулятор удаленного терминала фирмы Mitre [8].

Имитаторы играют важную роль при тестировании внешних функций и комплексном тестировании диалоговых систем и систем реального времени, но, очевидно, значительный объем тестирования должен быть выполнен и в реальной обстановке. Есть, однако, ситуации, когда при комплексном тестировании приходится полагаться только на имитацию. Примером такой ситуации является тестирование системы управления NORAD — большой сложной системы реального времени Командования Североамериканской ПВО [9]. Здесь приходилось полагаться на имитацию, поскольку тестирование в реальной обстановке могло быть выполнено только в условиях настоящей войны. Имитировалась посылка системе NORAD сообщений, описывающих разнообразные инциденты, в том числе массированные атаки самолетов и ракет с использованием всевозможных тактических приемов. Важной стороной комплексного тестирования наряду с тестированием программного обеспечения и аппаратуры системы NORAD была проверка реакции операторов.

Имитация жизненно необходима также и для тестирования аэрокосмических систем, где испытания в реальной обстановке могут стоить миллиарды долларов и даже быть связанными с риском для жизни людей. Имитация использовалась для тестирования программ-

ного обеспечения бортовой ЭВМ ракеты Сатурн — ракеты-носителя космического корабля «Аполлон» [10]. Бортовая машина выполняет предстартовые проверки, управляет запуском и выводом на околоземную орбиту и траекторию к Луне. Она управляет двигателями и соплами ракеты, анализируя большой объем информации от датчиков и сообщения с Земли. Моделирующая система «окружает» бортовую ЭВМ моделируемой средой и позволяет специалистам выполнять тесты, вводя данные о различных помехах в полете, нарушая последовательность действий и наблюдая за реакцией бортовой ЭВМ.

Кроме уже рассматривавшихся имитаторов устройств, терминалов, удаленных систем и мониторов, остается еще одно полезное средство тестирования — *виртуальная машина*. Виртуальная машина, реализованная, например, фирмой IBM в системе VM/370, является гипервизором, позволяющим выполнять на одной машине несколько систем программного обеспечения, причем каждой системе представляется, что она одна имеет доступ к собственной физической машине и полностью контролирует ее. Виртуальные машины полезны при тестировании систем, имеющих прямой доступ к физическим ресурсам машины (например, операционных систем). Концепция виртуальной машины позволяет, например, на одной и той же физической машине одну систему эксплуатировать в рабочем режиме, а другую систему в это же время тестировать на другой виртуальной машине.

ПЛАНИРОВАНИЕ И УПРАВЛЕНИЕ ПРИ ТЕСТИРОВАНИИ

Поскольку эта книга не о методах руководства проектами, то вопросам планирования, контроля и управления, связанным с процессом программирования, было до сих пор посвящено лишь несколько слов. Однако для тестирования эти вопросы имеют решающее значение, и на этот счет уместно привести несколько рекомендаций.

Одна из самых распространенных ошибок при тестировании программного обеспечения — планировать этот процесс, подсознательно предполагая, что не будет обнаружено ни одной ошибки. Напомню, что цель тестирования — найти ошибки. Планирование выделяемых на тестирование ресурсов (людей, машинного и календарного времени), основанное на предположении, что каждый тест будет выполняться с первого же раза, ведет к двум серьезным проблемам. Первая состоит в крупной недооценке необходимых ресурсов. Тот факт, что тестирование — одна из последних фаз разработки, усугубляет проблему, поскольку серьезное перераспределение ресурсов в это время почти невозможно. Вторая проблема состоит в том, что когда персонал, занимающийся тестированием, оказывается в подобной ситуации, то он часто стремится все-таки уложить-

ся даже в такой неразумный график, а это вынуждает пропускать ошибки и употреблять только «легкие» тесты, избегая чересчур разрушительных.

Как и для всякого сложного дела, *план* является жизненно важной компонентой процесса тестирования. План должен быть разработан до начала тестирования, возможно — во время проектирования системы. Хороший план тестирования должен содержать следующие компоненты:

1. *Цели.* Определение задач каждой фазы тестирования.
2. *Ответственность и точный график.* Определяется, кто и в какие сроки будет осуществлять проектирование, разработку и выполнение тестов, а также отладку для каждой фазы тестирования.
3. *Инструменты.* Описание необходимых инструментов тестирования с указанием того, как и когда они будут получены.
4. *Машинное время.* Оценки количества машинного времени, необходимого на каждом этапе, с указанием того, как и когда оно будет получено.
5. *Конфигурации.* Описание всех необходимых особых конфигураций аппаратного обеспечения или устройств, включая графики заказов и установки, с указанием ответственных за исполнение.
6. *Библиотеки тестов и стандарты.* Определение правил хранения тестов, стандарты их подготовки.
7. *Процедуры контроля.* Описание способа контроля за процессом тестирования.
8. *Процесс отладки.* Описание процедур, необходимых для выдачи сообщений об ошибках и их исправлении. Подробнее они рассматриваются в гл. 13.
9. *Критерии успеха.* Критерии, в соответствии с которыми следует принимать решение о том, насколько успешно завершена каждая фаза тестирования.
10. *План сборки.* О нем уже говорилось в этой главе. В нем должны быть описаны все необходимые программы-«подпорки» и график их подготовки с указанием ответственных.

Из всех перечисленных элементов самое сложное — определить критерии успеха. Например, распространенный критерий при тестировании внешних функций — считать тестирование завершенным, если 100% тестов выполняются правильно. Слабость этого критерия в том (хотя я и отказываюсь верить, что люди в основе своей нечестны), что он определенно не настраивает того, кто занимается тестированием, писать в высокой степени разрушительные тесты. Чтобы вдохновить на поиски как можно большего количества ошибок, предпочтительны *негативные* критерии, например такой: считать, что тестирование функций не закончено, пока не будет обнаружено и исправлено по крайней мере по две ошибки на

каждую тысячу строк текста программы. Такого рода критерии соответствуют тому принципу, что цель тестирования — найти ошибки. Конечно, такой критерий ставит некоторые проблемы для тестовика, которому достались исключительно корректно реализованные функции, но такого рода ситуации поддаются урегулированию (все мы желали бы почаще иметь дело с проблемами вроде этой, не так ли?). Прежде всего он заставляет его тестировать строже, а затем, если никак не удается найти ошибки, тестовику нужно переговорить с группой, выполняющей комплексные тесты, и убедить ее участников в том, что больше ошибок действительно не удалось бы найти. Следует применять и другие ранее упоминавшиеся критерии, например критерий тестирования функций, требующий выполнения каждой ветви хотя бы раз (за исключением защитных участков).

Критерий завершения фазы тестирования должен указывать число ошибок, которые должны быть найдены, а не опираться на такие ничего не значащие факторы, как процент тестов, выполняемых правильно.

Такие же принципы могут быть применены и к определению момента, когда комплексное тестирование можно назвать законченным. Можно считать, что оно завершено, если достигнута позднейшая из двух дат: истек срок, намеченный по календарному плану, или при тестировании обнаружено по крайней мере X ошибок. Еще одна полезная мера — вычертить график числа ошибок, обнаруженных за единицу времени (например, за неделю). Вместо того чтобы считать все ошибки на графике одинаковыми, следует взвешивать их в соответствии с их серьезностью. Такой график позволяет принимать руководящие решения на основе анализа взвешенных скоростей обнаружения ошибок.

Обсуждая эту тему, я избегал употреблять обороты типа «тест был успешным», поскольку они заключают в себе интересный парадокс. Руководители любят употреблять слово «успешный», говоря о тесте, при выполнении которого ошибок не обнаружено. Если, однако, рассматривать тестирование как процесс поиска ошибок, слово «успешный» следует употреблять скорее для описания теста, который позволил обнаружить новую ошибку.

Контролируя процесс тестирования, не следует забывать аксиому из гл. 10, в соответствии с которой, чем больше ошибок обнаружено на участке системы, тем выше вероятность, что есть еще другие, необнаруженные ошибки. Анализ данных об ошибках с целью выявить опасные новыми ошибками модули или функции может еще на ранних этапах тестирования помочь установить обратную связь, позволяющую определить, где именно сконцентрировать оставшиеся ресурсы.

При тестировании всегда встречается еще один интересный парадокс: с одной стороны, хочется найти не слишком много ошибок и уложиться в график, а с другой стороны, понятно, что цель тестирования — найти как можно больше ошибок. Это еще одно подтверждение двух обязательных принципов, уже обсуждавшихся выше: критерии завершения должны формулироваться в негативном духе (например, фаза тестирования не закончена, пока не обнаружено заранее установленное число ошибок) и, кроме того, тестирование должно выполняться независимой организацией, которая не испытывает давления календарных планов организации-разработчика.

ТЕСТИРОВАНИЕ ПРИЕМЛЕМОСТИ

Тестирование приемлемости представляет собой процесс испытаний, во время которого проверяется соответствие системы не внешним спецификациям, не целям, а исходным требованиям. Тестирование приемлемости — необычный вид тестирования, поскольку он выполняется организацией-покупателем или пользователем, а не разработчиком программного обеспечения.

Для нового программного продукта *любого* типа, будь то продукт, полученный по контракту с компанией-разработчиком или по соглашению с внутренней программистской организацией, или установка новой версии операционной системы, или покупка уже готового программного продукта, организация-пользователь должна спроектировать и написать несколько тестов с целью выяснить, в какой мере продукт не соответствует требованиям. Если эти тесты оказываются «неудачными», то продукт официально принимается к использованию. Тестирование приемлемости всегда представляет собой некоторую комбинацию трех элементов: *тестирование на типовых задачах*, когда выполняются специально подготовленные тесты, *опытная эксплуатация*, когда новый продукт используется ограниченно, на экспериментальных началах, и *тестирование сопоставлением*, когда он устанавливается и работает параллельно с прежней системой.

ТЕСТИРОВАНИЕ НАСТРОЙКИ

Оставшийся тип тестирования программного обеспечения, как и все другие, направлен на обнаружение ошибок, но теперь уже не в самом программном обеспечении. Многие большие системы и программы требуют сложных процедур настройки. Покупатель должен отобрать нужные ему возможности, разместить файлы и библиотеки, иметь в наличии необходимую конфигурацию аппаратного обеспечения и часто еще состыковать новый продукт с другими программами. Цель тестирования настройки — найти любые ошибки, которые были допущены при осуществлении этих процедур.

Тесты настройки должны быть спроектированы и написаны разработчиками программного обеспечения и поставляться вместе с продуктом и документацией на него. Эти тесты выполняются либо покупателем, либо обслуживающим персоналом организации-разработчика после настройки продукта. Тесты настройки проверяют наличие всех необходимых файлов и аппаратных устройств, содержимое первой записи каждого файла, наличие всех частей программного продукта и т. п.

Любая программа должна поставляться вместе с несколькими тестами, способными обнаруживать ошибки настройки.

ЛИТЕРАТУРА

1. OS/VS Utilities, C35-0005, JBM Corp., White Plains, N. Y., 1974.
2. Elmendorf W. R. Cause-Effect Graphs in Functional Testing, TR-00.2487, IBM Systems Development Div., Poughkeepsie, N. Y., 1973.
3. Elmendorf W. R. Functional Analysis Using Cause-Effect Graphs, Proceedings of SHARE XLIII. New York: SHARE, 1974, pp. 567—577.
4. Драммонд Д., Методы оценивания и измерений дискретных вычислительных систем. Пер. с англ.— М.: Мир, 1976.
5. Shelley M. Computer Software Reliability, Fact or Myth?, TR-MMER/RM-73-125, Hill Air Force Base, Utah, 1973.
6. Pullen E. W., Shuttee D. R. MUSE: A Tool for Testing and Debugging a Multiterminal Programming System, Proceedings of the 1968 Spring Joint Computer Conference. Washington: Thompson, 1968, pp. 491—502.
7. The Data Base/Data Communication Driver System: Design Objectives, GH20-4281, IBM Corp., White Plains, N. Y., 1974.
8. James D. L. Remote-Terminal Emulator (Design Verification Model) — Data Structures, Scenario Instructions, and Commands, MTR-2677, Mitre Corp., Bedford, Mass., 1974.
9. Stevens R. T. Testing the NORAD Command and Control System, *IEEE Transactions on Systems Science and Cybernetics*, SSC-4 (1), 47—51 (1968).
10. Jacobs J. H., Dillon T. J. Interactive Saturn Flight Program Simulator, *IBM Systems Journal*, 9(2), 145—158 (1970).

ГЛАВА 13

Отладка

Несмотря на все рекомендации из части 2, некоторые ошибки в программном обеспечении все-таки будут сделаны, и трудная задача его отладки остается. Тем более что правильно тестировать — значит стремиться найти как можно больше ошибок. Тот факт, что эта глава имеет несчастливый номер 13,— весьма красноречивое совпадение.

Отладка — это устранение ошибок: она начинается с обнаружения некоторых признаков, или симптомов ошибки в программном обеспечении, и представляет собой процесс определения ее местоположения и исправления. Причина, по которой отладка считается трудным искусством, в том, что все мы как-то подсознательно предназначаем «это занятие для других, менее совершенных людей». Имеется удивительно мало исследований или рекомендаций по отладке, и это при том, что средний программист тратит на нее больше времени, чем на проектирование и собственно программирование. Появились, правда, некоторые обнадеживающие признаки: книга, частично посвященная этому вопросу [1], и курс, посвященный исключительно предмету отладки [2], который читается по крайней мере в одном университете.

КАК ИСКАТЬ ОШИБКУ

Чтобы устранить ошибку, нужно прежде всего точно установить ее расположение. Приходится начинать с симптомов типа «последний символ отсутствует во всех записях выходного файла», и требуется точно установить, какой именно или какие именно операторы (из 100 000) в нашей программе ошибочны. Встречающаяся во многих книгах по программированию рекомендация «разбросайте по программе побольше операторов печати промежуточных результатов» не лишена смысла, но это — пример бессистемного подхода, и начинать с него не следует. Метод решения задач Пойа [3] можно применить и к отладке, вполне аналогично тому, как в гл. 3 он был приспособлен к особенностям проектирования.

1. *Поймите задачу.* Многие программисты начинают процесс отладки бессистемно, пропуская жизненно важный этап детального

анализа имеющихся данных. Первым делом нужно тщательно исследовать, что в программе выполнено правильно, а что — неправильно, чтобы выработать одну или несколько гипотез о природе ошибки. Одна из самых распространенных причин затруднений при отладке — не учтен какой-нибудь существенный фактор в выходных данных программы. Следует изучить выходные данные и оценить, достаточно ли их для определения местоположения ошибки, так как, если их недостаточно, прежде всего нужно собрать больше данных, а для этого подготовить и выполнить дополнительные тесты. Важно исследовать данные в поисках *противоречий* гипотезе (например, ошибка возникает только в каждой второй записи), потому что это поведет к уточнению гипотезы или, возможно, покажет, что имеется не одна причина ошибки.

Полезный подход к изучению симптомов ошибок — *метод Брауна и Сэмпсона* [1]. Для организации и анализа имеющихся данных используется изображенный на рис. 13.1 бланк. В графе ЧТО описываются основные симптомы, в графе ГДЕ описываются точки, где замечены симптомы, в графе КОГДА — всевозможные догадки, касающиеся того, когда симптомы проявляются, а когда нет, а в графе УТОЧНЕНИЕ указывается область действия симптомов. Столбцы ДА и НЕТ составляют суть метода, поскольку они описывают отличия и противоречия, которые поведут к созданию гипотезы.

Предположим, что ошибка встретилась при использовании разработанной в части 2 программы загрузчика. Общие симптомы — при выполнении модуля, загруженного последним, неожиданно возникает программное прерывание, хотя сам модуль в листинге программы кажется правильным. Прежде всего нужно проверить, достаточно ли имеется фактов, чтобы заполнить все клетки таблицы, изображенной на рис. 13.1. Заполненная таблица приведена на рис. 13.2.

2. *Разработайте план.* Следующий шаг — построить одну или несколько гипотез об ошибке и разработать план проверки этих гипотез. Изучив рис. 13.2, придем к гипотезе, что последний элемент в таблице ТПЕРМ не замкнут (это приводит к переходу «наугад» при выполнении соответствующего оператора CALL в загруженной программе), и ошибка, вероятно, в функции НАСТР_АДР модуля УПР_ТПЕРМ.

3. *Выполните план.* Теперь, следуя своему плану, пытаемся доказать гипотезу. Если план включает несколько шагов, нужно проверить каждый. Чтобы подтвердить гипотезу о загрузчике, следует написать тест и сделать копию состояния ТПЕРМ и загруженной программы после завершения выполнения функции НАСТР_АДР.

4. *Проверьте решение.* Если кажется, что точное местоположение ошибки обнаружено, необходимо выполнить еще несколько про-

?	Да	Нет
Что		
Где		
Когда		
Уточнение		

Рис. 13.1. Таблица для анализа симптомов ошибок.

верок, прежде чем пытаться исправить ошибку. Проанализируйте, может ли предполагаемая ошибка давать в точности известные симптомы. Убедитесь, что найденная причина полностью объясняет все симптомы, а не только часть их. Проверьте, не вызовет ли ее исправление новой ошибки.

Главная причина затруднений при отладке — такая *психологическая установка*, когда разум видит то, что он ожидает увидеть, а совсем не то, что имеет место в действительности [4]. Это, например, главная причина того, что не удается заметить все типографские опечатки в книге. Один из способов преодоления такой установки — скептицизм в отношении всего, что вы изучаете, в особенности — комментариев и документации. Опытные специалисты по отладке, изучая модуль, часто закрывают комментарии, поскольку комментарии нередко описывают, что программа делает, *по мнению ее создателя. Обратный просмотр* (чтение программы в обратном направлении) — еще один полезный тактический прием, поскольку он помогает по-новому взглянуть на алгоритм.

Еще одна трудность при отладке — такое состояние, когда все идеи зашли в тупик и найти местоположение ошибки кажется просто невозможным. Это означает, что либо вы смотрите не туда, куда нужно, и следует еще раз изучить симптомы и построить новую гипотезу, либо подозрения правильные, но разум уже не способен заметить ошибку. Если кажется, что именно так и есть, то лучший принцип — «утро вечера мудренее». Переключите внимание на другую деятельность, и пусть над задачей работает ваше подсознание.

?	Да	Нет
Что	В загруженной программе неправильно выполняется переход	Не похоже на ошибку в логике загруженной программы
Где	Происходит только в последнем загруженном модуле	Судя по листингу, ТВИМ правильна
Когда	Когда в последнем модуле выполняется оператор CALL	Когда выполняется любой другой оператор CALL
Уточнение	Происходит, кажется, только в случае, если последний загруженный модуль содержит оператор CALL	Когда последний загруженный модуль не содержит операторов CALL

Рис. 13.2. Симптомы ошибки в загрузчике.

Многие программисты признают, что самые трудные свои задачи они решают во время бритья или по дороге на работу.

Когда вы найдете и проверите ошибку и убедитесь в том, что нашли ее правильно, не забудьте о явлении, отраженном на рис. 10.9, который показывает, что вероятность других ошибок в этой части программы теперь выше. Изучите программу в окрестности найденной ошибки в поисках новых неприятностей. Проверьте, не была ли сделана такая же ошибка в других местах программы.

Исследования методов отладки вначале концентрировались на сравнении отладки в пакетном и диалоговом режимах, причем большинство исследователей приходило к выводу, что диалоговый режим предпочтительнее [5]. Однако более поздняя работа показала, что, вероятно, самым лучшим является третий режим [6]. Опытным программистам были даны для отладки сложные программы на Фортране (которых они никогда не видели раньше). В некоторых экспериментах им говорилось только, что в программе есть ошибки. В других случаях им давали также ожидаемые и действительные результаты, а в некоторых экспериментах даже сообщали, в каком операторе ошибка. Точное место ошибок обычно устанавливалось очень быстро (и без помощи ЭВМ), причем различия по затратам времени для разных экспериментов были удивительно малы. Один из выводов этого исследования состоит в том, что, вероятно, наилучший способ отладки — просто читать программу и изо всех сил стараться вникнуть в алгоритм, хотя это и требует усердия и собранности.

Важно подчеркнуть, что многие из методов проектирования, обсуждавшиеся в части 2, помогают и в процессе отладки. Такие методы, как композиционное проектирование, структурное программирование и хороший стиль программирования не только уменьшают исходное количество ошибок, но и облегчают отладку, делая программу более простой для понимания. Защитные средства, описанные в гл. 7 и 8, также помогают процессу отладки, обнаруживая симптомы ошибок вскоре после исполнения ошибочных фрагментов и тем самым облегчая определение местоположения ошибки.

КАК ИСПРАВЛЯТЬ ОШИБКИ

После того как точно установлено, где находится ошибка, надо ее исправить. Самая большая трудность на этом шаге — суметь охватить проблему целиком; самая распространенная неприятность — устранить только некоторые симптомы ошибки. Избегайте «экспериментальных» исправлений («Кажется, дело вот в чем; попытаюсь-ка я изменить это так, и посмотрим, что получится»); они показывают, что вы еще недостаточно подготовлены к отладке этой программы, поскольку не понимаете ее.

В деле исправления ошибок очень важно понимать, что оно возвращает нас назад, к стадии проектирования. Обидно, если после завершения хорошо организованного проектирования весь его строгий порядок нарушается, когда вносятся поправки. Исправления должны выполняться по крайней мере так же строго, как первоначальное проектирование программы. Если необходимо, следует обновить документацию, поправки должны проходить сквозной структурный контроль или другие формы контрольного чтения программы. Ни одна поправка не «мала» настолько, чтобы не нуждаться в тестировании. Если ошибка была обнаружена после автономного тестирования, нужно добавить тесты модуля, которые, очевидно, отсутствовали при первоначальном тестировании, и повторно выполнить все автономные тесты.

По самой своей природе исправления всегда имеют некоторое отрицательное влияние на структуру программы и легкость ее чтения. Тот факт, что они делаются в условиях жесткого давления, усиливает это влияние. Опыт показал, что при исправлении довольно высока вероятность внесения в программу новой ошибки (обычно от 20 до 50%). Из этого следует, что отладка должна выполняться лучшими программистами проекта и никогда не должна перекладываться на плечи новичков.

Нужно полностью отказаться от исправлений непосредственно объектного кода в расчете позднее исправить исходный текст. Там, где допускается этот метод, всегда царит глубокая тревога, поскольку легко может оказаться, что объектный код и исходный текст не соответствуют друг другу. Дополнительное время, необходимое для

исправления исходного текста и его компиляции,— это затраты, которые всегда окупятся.

ИНСТРУМЕНТЫ ОТЛАДКИ

Незавидный уровень, на котором находится большинство инструментов отладки, хорошо иллюстрируется недавним рекламным объявлением, объясняющим, почему следует покупать новую книгу по языку ассемблера: «Прежде всего, вам наверняка придется интерпретировать диагностику или результаты компиляции с языка высокого уровня. Если вы не знакомы с базовым языком ассемблера, вас ожидает задача почти неразрешимая». К сожалению, это утверждение рисует реальную картину, характерную для большинства средств отладки. Сегодня уже нельзя мириться с тем, чтобы операционная система, обнаружив ошибку в программе на Фортране, выводила восьмеричные или шестнадцатеричные дампы, безжалостно распечатывала переменные с плавающей точкой в шестнадцатеричном формате (заставляя программиста на Фортране выполнять нетривиальные преобразования), представляла программу на Фортране не в ее исходной форме, а в виде машинных команд и адресов и подавляла программиста избытком не относящихся к делу бесполезных данных, например указывая содержимое управляющих блоков системы.

Основной недостаток большинства инструментов отладки заключается в том, что они вынуждают программиста, работающего на языке высокого уровня, разбираться в деталях машинного языка, и это в ситуации, когда главное назначение языков высокого уровня — скрыть от программиста эти детали. Решение этой проблемы состоит в том, чтобы проектировать все средства отладки в расчете на их общение с программистом исключительно в терминах языка программирования и ни в коем случае не в терминах машины. Описанные в этом разделе инструменты представляют собой некоторые шаги в нужном направлении.

Диалоговые инструменты

В большинстве своем диалоговые инструменты отладки обеспечивают взаимодействие программиста с программой во время ее выполнения. Они позволяют программисту прерывать программу, указывая в ней *точки вклинивания* (заранее намеченные точки, в которых выполнение программы должно быть прервано), а также выводить и изменять значения переменных, обращаясь к ним *по символическим именам* (а не по адресу памяти), проходить по программе оператор за оператором и получать дампы в символической форме.

Одним из таких инструментов является отладочный компилятор (Checkout Compiler) с языка PL/I фирмы IBM [7]. Программист

общается с ним только в терминах *исходного* языка. Он может установить точки вклинивания в конкретных операторах PL/1, изменить значение переменной A, печатая «$A=2$», и получить след пооператорной прокрутки программы. Программист может получить выборочные наглядные дампы по таким запросам, как «все активизированные процедуры», «все файлы», «все переменные-указатели», и состояние всех ветвей и ситуаций прерывания. Компилятор PL/1 в системе Multics фирмы Honeywell представляет аналогичный набор средств отладки [8]. Фирма IBM поставляет подобные средства для языков Фортран и Кобол [9]. Язык АПЛ благодаря присущим ему общности и единообразию представляет собой идеальную отладочную среду, в которой язык программирования и язык отладки совпадают.

Инструменты отладки такого типа полезны при тестировании, но они обычно не применимы при отладке системы, эксплуатируемой в рабочем режиме, так как требуют специальных возможностей компилятора. При разработке больших, сложных программных систем стоит рассмотреть вопрос о включении средств отладки прямо в создаваемую систему. Например, операционные системы OS/VS1 и OS/VS2 фирмы IBM содержат отладочное средство, называемое DSS (Dynamic Support System — система динамической поддержки) [10]. DSS позволяет обслуживающему персоналу взаимодействовать с системой в рабочем режиме. Можно заказать прерывание (точку вклинивания) на конкретной команде, можно потребовать остановить систему при изменении конкретного регистра или слова памяти.

Система EXDAMS

Один из диалоговых инструментов, заслуживающий особого упоминания,— это EXDAMS (Extendable Debugging and Monitoring System — Расширяемая система отладки и контроля) [11]. EXDAMS уникальна и по набору предоставляемых пользователю возможностей, и по способу их реализации.

EXDAMS — диалоговое средство, использующее терминал с дисплеем. Хотя пользователь за терминалом может считать, что он взаимодействует с программой во время ее выполнения, в действительности он работает с хранимой на ленте историей выполнения программы. Прежде всего EXDAMS запоминает в файле исходный текст программы и ее таблицу символов. Затем программа выполняется, и при этом создается лента, содержащая след управления и результаты всех операторов присваивания. Когда пользователь приступает к отладке программы, она «выполняется» за счет «прокручивания» этой ленты. Ниже перечислены примеры некоторых из множества отладочных возможностей.

1. Пользователь может сказать: «Покажите оператор, в котором переменной А присвоено значение 33.3.» EXDAMS показывает на

экране сегмент исходного текста программы, особенно ярко высвечивая соответствующий оператор.

2. Пользователь может указать список переменных, значения которых он желает проверить. Имена этих переменных становятся на экране заголовками столбцов, в которых EXDAMS, прокручивая программу, последовательно показывает их изменяющиеся со временем значения.

3. Пользователь может наблюдать за выполнением своей программы. Очередной выполняемый оператор выделяется яркостью, и если это оператор присваивания, то выдаются также его результаты. При этом пользователь может выполнять свою программу как в прямом, так и в обратном направлении.

Средства языка

В дополнение к упоминавшимся выше средствам отладки аналогичные средства часто включают в языки программирования в качестве их расширений. В PL/1 имеется мощная ситуация прерывания CHECK, которая приводит к прерыванию или выдаче сообщения каждый раз, когда изменяется значение указанной переменной или выполняется оператор с указанной меткой. Например, оператор

```
(CHECK (TRAN, TABLE, I)): DIST: PROC;
```

вызывает выдачу сообщения, как только в процедуре DIST изменяются значения переменных TRAN, TABLE или I.

Дополнительные возможности имеются в PL/C [12], одной из версий PL/1. Кроме того, что в PL/C также имеется конструкция CHECK, PL/C также проверяет, присвоено ли переменной значение до ее употребления и не выходят ли индексы за установленные границы. PL/C позволяет также оставлять в программе отладочные операторы; они рассматриваются либо как комментарии, либо как выполняемые операторы в зависимости от указания компилятору. Например, оператор

```
/* 4 PUT SKIP(3) LIST('HIT POINT 32', ICOUNT); */
```

интерпретируется как комментарий, если только в управляющих картах не указано

```
COMMENTS=(4);
```

в результате чего этот оператор превращается в настоящий оператор PUT. Этот метод можно было бы еще усовершенствовать, позволяя активизировать этот оператор во время выполнения, а не только во время компиляции.

PL/C выдает также полезные сведения в случае ошибки. Они содержат список всех переменных и их текущие значения, список пе-

ременных, на которые не было ссылок, счетчики частоты выполнения всех помеченных операторов и входов, а также список 18 последних передач управления.

Дампы

Как уже отмечалось, трудно оправдать выдачу средствами отладки простого восьмеричного или шестнадцатеричного дампа. Эти примитивные дампы содержат огромное количество не относящихся к делу данных и обычно выводят их в неудобной форме, вследствие чего многим организациям приходится специально организовывать занятия, посвященные исключительно искусству чтения дампов. Дампы в терминах входного языка, наподобие тех, что обеспечивает PL/C, очень желательны, но они требуют тесной связи программы выдачи дампов с компилятором.

Разработанная в университете штата Огайо программа выдачи дампов показала, что можно реализовать разумные программы печати дампов и без связи с компилятором [13]. Такие дампы вырабатываются операционной системой при возникновении прерывания по ошибке в прикладной программе. Их цель — напечатать только те данные, что относятся к делу, преобразовать их в соответствующую числовую форму и помочь программисту, подсказав возможную причину прерывания. В дампе значения всех регистров с плавающей точкой печатаются в десятичном формате с порядком, а не в шестнадцатеричной форме «с избытком 64». Вызвавшая ошибку команда «разбирается» и печатается на языке ассемблера, а не в машинном коде, выводится также описание прерывания.

Существенная особенность этой программы выдачи дампов состоит в том, что она пытается угадать действительную ошибку, которая привела к программному прерыванию. Для этого было проанализировано множество данных об известных ошибках с целью определить, какие типы ошибок обычно вызывают те или иные программные прерывания. Программа выдачи дампов анализирует и тип прерывания, и вызвавшую это прерывание команду, и ее расположение, и, пользуясь этой информацией, высказывает догадку о вероятной ошибке. Например, если код прерывания OC4 (попытка адресовать защищенную область памяти), а общий регистр 15 содержит шестнадцатеричное число 02005000, программа выдачи дампов делает вывод, что прикладная программа пыталась читать из неоткрытого файла, и печатает сообщение об этом, а также информацию об этом файле из управляющего блока.

Разумные программы выдачи дампов, вроде этой, можно расширить, включив распознавание распространенных ошибок при использовании конкретных языков. Например, если подпрограмме на Фортране в качестве аргумента передан массив и этот массив не объявлен в ней в операторе DIMENSION, то большинство компиляторов примет ссылку на элемент этого массива за обращение к функции

и последует попытка выполнить массив как программу. Разумная программа выдачи дампов легко могла бы распознать такую ситуацию и угадать ошибку.

Информационные средства

При отладке большой системы программисту часто бывает нужно получить ответ на вопросы такого рода: «Какие еще модули ссылаются на это поле в этой записи?» или «Какие модули вызывают этот модуль?». Было разработано несколько средств сбора и обработки такой информации, дающих ответ на эти и другие подобные вопросы.

Рассматривавшаяся в гл. 11 система ASES обеспечивает часть этих возможностей, запоминая информацию о программе в базе данных и отвечая на вопросы типа «В каких операторах используются и/или изменяются значения вот этой переменной?». Возможности Cross-Program Analyzer в системе AUTOFLOW II фирмы Applied Data Research позволяют вырабатывать отчеты или отвечать на запросы такого типа, как поиск всех операторов, в которых есть ссылка на данную переменную, поиск нарушений определенных стандартов программирования, поиск всех операторов, в которых есть ссылки на данный файл, или поиск всех модулей, которые вызывают данный модуль.

Средства прослеживания и трассировки

Последний класс инструментов отладки составляют средства самой программной системы, сообщающие программисту информацию в случае ошибки. Средства активного и пассивного обнаружения ошибок, рассматривавшиеся в гл. 7, находят и записывают информацию об ошибках и, таким образом, попадают в эту категорию. В базах данных и системах телеобработки часто ведется журнал, в котором хранятся копии каждого сообщения базы данных или терминала, чтобы обеспечить восстановление в случае отказа аппаратуры или программного обеспечения и сохранить следы попыток нарушить правила секретности. Такие журналы незаменимы также и при отладке.

Крупные системы реального времени должны иметь средства для регистрации внутренних потоков данных в системе. Эти средства могут быть реализованы на различных уровнях, например могут регистрировать все вызовы подсистем, уровней абстракции или даже отдельных модулей. Для каждого вызова составляется запись, содержащая имена вызывающего и вызываемого модулей и передаваемые входные и выходные данные. Записи можно хранить на трассировочной ленте или в кольце буферов таким образом, чтобы сохранились только N (например, 100) последних записей.

ИЗУЧЕНИЕ ПРОЦЕССА ОТЛАДКИ

Один из лучших способов повысить надежность программного обеспечения в нынешних или будущих проектах — очевидный, но часто упускаемый из виду процесс обучения на сделанных ошибках. Каждую ошибку следует внимательно изучить, чтобы понять, почему она возникла и что должно было быть сделано, чтобы ее предотвратить или обнаружить раньше. Редко можно встретить программиста или организацию, которые выполняли бы такой полезный анализ, а когда он проводится, то обычно имеет поверхностный характер и сводится, например, к классификации ошибок: ошибки проектирования, логические ошибки, ошибки сопряжения или другие, не имеющие особого смысла категории.

Нужно уделять время изучению природы *каждой* обнаруженной ошибки. Я должен подчеркнуть, что анализ ошибок должен быть в значительной мере *качественным* и не сводиться просто к упражнению в «подсчете бобов». Чтобы понять причины, лежащие в основе ошибок, и усовершенствовать процессы проектирования и тестирования, нужно ответить на следующие вопросы:

1. Почему возникла именно такая ошибка? В ответе должны быть указаны как первоисточник, так и непосредственный источник ошибки. Например, ошибка могла быть сделана при программировании конкретного модуля, но в ее основе могла лежать неоднозначность в спецификациях или исправление другой ошибки.
2. Как и когда ошибка была обнаружена? Поскольку мы только что добились значительного успеха, почему бы нам не воспользоваться приобретенным опытом?
3. Почему эта ошибка не была обнаружена при проектировании, контроле или на предыдущей фазе тестирования?
4. Что следовало сделать при проектировании или тестировании, чтобы предупредить появление этой ошибки или обнаружить ее раньше?

Собирать эту информацию нужно не только для того, чтобы учиться на ошибках. Официальная форма отчетности об ошибках и об их исправлении необходима и для того, чтобы гарантировать, что обнаруженные ошибки в работающих или тестируемых системах не упущены и что исправления выполнены в соответствии с принятыми нормами. Почти в каждой организации есть такая процедура, поэтому нет необходимости ее описывать. Желательно подкрепить ее программой управления и наблюдения за исправлением ошибок. Например, в IBM используется программа, называемая SMP (System Modification Program — программа модификации систем) [14], которая помогает вносить изменения и отменять их, если они ошибочны, запоминает статус каждого модуля и обеспечивает регистрацию всех модификаций.

Другой способ обучения на ошибках в программном обеспечении — учиться на опыте других организаций, например изучая публикуемые исследования ошибок. К сожалению, это не жизнеспособный вариант, поскольку таких исследований недостаточно по нескольким причинам. Большинство компаний считает информацию об ошибках в программном обеспечении конфиденциальной. Почти все исследования, которые все-таки публикуются, суть простые «подсчеты бобов» (например, «32% наших ошибок были логическими ошибками»), что делает их почти бесполезными. Наконец, в различных исследованиях ошибки подсчитываются по-разному (например, часто бывает неясно, учитываются ли ошибки, обнаруженные при компиляции, сквозном контроле, автономном тестировании, и ошибки, обнаруженные пользователем), что делает сравнение разных проектов почти невозможным. Есть, однако, некоторые исключения. Эндреш [15] дает интересное исследование ошибок, найденных при тестировании выпуска операционной системы DOS/VS IBM. Шуман и Болски [16] анализируют ресурсы, потраченные при тестировании и отладке небольшой программы, а Крейг [17] анализирует ошибки в трех больших программах.

ЛИТЕРАТУРА

1. Brown A. R., Sampson W. A. Program Debugging. London: Macdonald, 1973.
2. Mathis R. F. Teaching Debugging, *SIGCSE Bulletin*, 6 (1), 59—63 (1974).
3. Пойа Д. Как решать задачу. Пер. с англ.— М.: Наука, 1961.
4. Weinberg G. M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.
5. Сакман Г. Решение задач в системе человек — ЭВМ. Пер. с англ.— М.: Мир, 1973.
6. Gould J. D., Drongowski P. A Controlled Psychological Study of Computer Program Debugging, RC 4083, IBM Research Div., Yorktown Heights, N. Y., 1972.
7. OS PL/I Checkout Compiler: Programmer's Guide, SC33-0007, IBM Corp., White Plains, N. Y., 1971.
8. Wolman B. L. Debugging PL/I Programs in the Multics Environment, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 507—514.
9. IBM OS COBOL Interactive Debug and (TSO) COBOL Prompter: General Information, GC28-6454, IBM Corp., White Plains, N. Y., 1974.
10. OS/VS Dynamic Support System, GC28-0640, IBM Corp., White Plains, N. Y., 1973.
11. Balzer R. M. EXDAMS — Extendable Debugging and Monitoring System, Proceedings of the 1969 Spring Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1969, pp. 567—580.
12. Morgan H. L., Wagner R. A. PL/C: The Design of a High-Performance Compiler for PL/I, Proceedings of the 1971 Spring Joint Computer Conference, Montvale, N, J.: AFIPS Press, 1971, pp. 503—510.

13. Kirsch B. M. An Improved Error Diagnostics System for IBM System/360-370 Assembler Program Dumps, OSU-CISRC-TR-74-3, Ohio State University, Columbus, Ohio, 1974.
14. OS/VS System Modification Program (SMP), GC28-0673, IBM Corp. White Plains, N. Y., 1974.
15. Endres A. B. An Analysis of Errors and Their Causes in System Programs, *IEEE Transactions on Software Engineering*, SE-1 (2), 140—149 (1975).
16. Shooman M. L., Bolsky M. I. Types, Distribution, and Test and Correction Times for Programming Errors, Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE, 1975, pp. 347—357.
17. Craig G. R. et al. Software Reliability Study, RADC-TR-74-250, TRW Corp., Redondo Beach, Ca., 1974.

ЧАСТЬ 4

Дополнительные вопросы надежности программного обеспечения

ГЛАВА 14

Методы руководства и надежность

Если вы администратор и обратились к этой главе в надежде найти здесь сжатое изложение методов управления, которые могут повлиять на надежность программного обеспечения, вы заблуждаетесь. Невозможно разделить проблемы на «чисто управленческие» и «чисто технические», и по этой причине почти в каждой главе книги обсуждаются идеи, которые должны интересовать руководителя проекта. Назначение данной главы — рассмотреть некоторые вопросы руководства, не вписавшиеся естественно в другие главы.

Здесь у читателя, не участвующего в управлении, может возникнуть вопрос: а зачем вообще рассматривать эти проблемы в книге по надежности программного обеспечения? Как то, что делает (или не делает) мой руководитель, повлияет на число ошибок в моей программе? Ответ прост: надежность программного обеспечения требует соблюдения некоторой дисциплины в вопросах руководства в такой же степени, как и в вопросах технических. Исследования программных проектов показали, что грамотное руководство (или его отсутствие) часто является основным фактором, определяющим успех проекта. Недавнее неофициальное исследование в крупной компании показало, что, по мнению программистов, административные проблемы порождают более серьезные ошибки в программном обеспечении, чем недостатки процессов проектирования, реализации или тестирования. (Это мнение, конечно, не свободно от предубеждений.) Тремя основными административными проблемами были названы: (1) давление, оказываемое с целью начать (вследствие напряженного графика) собственно программирование (кодирование) прежде, чем будет закончено проектирование; (2)

недостаточное внимание к обучению программистов (особенно использованию языка программирования), и (3) слабый учет личных качеств программистов при распределении заданий.

ОРГАНИЗАЦИЯ И ПОДБОР КАДРОВ

Самое важное в организации проекта и подборе кадров — свести к минимуму число занятых людей. Хотя большинство администраторов знает, что над многими из наиболее успешных проектов работало удивительно мало людей и что метод «монгольской орды» [1] (попытка управиться с проблемой, щедро добавляя атакующие цепи сотрудников) редко оправдывает себя, не часто встречаются администраторы, по-настоящему преданные этим идеям. Когда проект начинает вызывать беспокойство, обычное «решение» — добавить еще людей вместо более разумного — уменьшить их число. В дополнение к положительному влиянию на стоимость и график работ (см., например, Брукс [2]) минимизация числа участников хорошо сказывается и на надежности благодаря сокращению числа связей и, следовательно, ошибок перевода.

В большинстве проектов требуется, чтобы пик количества занятых участников приходился на фазы проектирования и кодирования модулей. Не следует слишком рано комплектовать штаты в расчете на этот пик. Одна из проблем, которые могут возникнуть в противном случае,— риск, что создаваемой системе подсознательно будет придана структура, отражающая организационную структуру коллектива ее создателей.

В случае преждевременного комплектования штатов большого проекта часто возникает и другая проблема: ответственный за программирование может предложить ответственному за проектирование: «Пусть мои 50 программистов поработают над твоими внешними спецификациями, чтобы мы смогли сэкономить несколько недель». Такое предложение нельзя назвать мудрым по нескольким причинам. Внешнее проектирование должно выполняться минимальным числом людей (в гл. 4 предлагалось использовать не более двух), и оно требует совсем других талантов. Если по некоторым причинам штаты программистской организации укомплектованы на раннем этапе, всегда найдется другая полезная работа, которой можно заняться, не превращаясь в монгольскую орду на стадии внешнего проектирования и разработки архитектуры. К такой работе относятся обучение, эксперименты с различными вариантами проектов логики программы (это возможно до завершения внешнего проектирования), разработка специальных инструментов программирования, оценка сложности проекта и разработка алгоритмов.

Чтобы у читателя не сложилось мнение, что комплектование штатов — дело простое, рассмотрим вопрос с заведомо неоднозначным ответом: должна ли одна и та же группа выполнять все этапы

проектирования? Основываясь на предыдущих рассуждениях, можно ответить «да», поскольку это свело бы к минимуму число занятых в проекте людей и упростило бы проблемы их общения. Есть, однако, несколько более основательных причин, по которым на одну группу не следует возлагать ответственность за все фазы проектирования. Если ответственность за два последовательных этапа проектирования несут разные группы, обычно улучшается качество документации и выполняется двойная проверка проекта (группа, получающая результаты предшествующего этапа проектирования, весьма заинтересована в том, чтобы найти все ошибки). Но, вероятно, самая важная причина — обстоятельство, отмеченное гл. 4: каждая фаза проектирования требует специальной подготовки и опыта. Этапы определения требований, целей и внешнего проектирования требуют глубокого понимания нужд пользователя и квалификации в системном анализе, исследовании операций, психологии взаимодействия человека и машины и т. д. Этапы проектирования архитектуры системы и структуры программы требуют знаний в таких областях, как проектирование файлов, уровни абстракции, общая теория систем, композиционное проектирование. Этапы внешнего и внутреннего проектирования и программирования модулей требуют опыта в области языков программирования, алгоритмов, структурного программирования и т. д. Вряд ли кто-либо один обладает всеми этими талантами.

Когда в работу над каждым этапом проектирования вовлекаются разные люди, важно обеспечить непрерывность этапов. Распространенная ошибка состоит в том, что считается возможным, например, перевести разработчиков внешнего проекта или аналитиков на другой проект, как только внешние спецификации «сделаны». Ведь из-за ошибок и изменений в проекте спецификации редко удается «сделать» прежде, чем закончится вся работа. Основные проектировщики должны быть связаны с проектом до самого конца. Ошибки в спецификациях и неизбежные изменения проекта, появляющиеся в конце работы,— самые рискованные изменения, и они требуют внимания авторов проекта. Кроме того, как указывалось в гл. 12, разработчики внешнего проекта— идеальные члены группы комплексного тестирования.

Еще один вопрос, который необходимо рассмотреть, прежде чем закончить разговор о проектировании,— официальная процедура принятия изменений в проекте. Ее назначение — обеспечить, чтобы решения принимались соответствующими людьми и чтобы были поняты все последствия предлагаемого изменения. Предположим, что мы приняли в проекте некоторое решение X в самом начале работы и что впоследствии десять решений были основаны на выводах из решения X. Если кто-то предлагает изменить решение X, важно рассмотреть это предложение в рамках всего проекта, чтобы выявить и исследовать также и эти десять решений.

Одна из опасностей применения процедуры принятия проектных изменений в большом проекте — в возможности перехватить через край, пытаясь управлять проектированием с ее помощью вплоть до слишком низких уровней. В большом проекте только определение целей, внешнее проектирование, разработка архитектуры системы и проектирование базы данных должны контролироваться такой формальной процедурой.

Процессы тестирования составляют ту часть работы над проектом, которая в наибольшей степени требует от администраторов планирования и внимания. Один из ключевых организационных вопросов — создание отдельных групп тестирования. Как упоминалось в гл. 10, организация-разработчик не должна нести ответственности за выполнение тестирования своей продукции. Такая организация обычно слишком привязана к результатам своего труда, слишком стеснена графиком и бюджетом, вряд ли достаточно скептически относится к своей работе и не слишком заинтересована в нахождении ошибок. Кроме того, тестирование требует навыков, отличающихся от тех, которых требует программирование. Тестирование требует также другой системы мотивации и вознаграждения. В то время как вознаграждение программистов обычно тем выше, чем меньше ошибок в их программах, специалисты по тестированию должны получать вознаграждение за обнаружение как можно большего числа ошибок, вплоть даже до «сдельной» оплаты (например, 100 долларов за обнаружение ошибки серьезности *X*).

При комплектовании штатов для нового проекта руководители все еще зачастую придерживаются распространенной «кастовой» системы в работе программистов. Задания по проектированию верхних уровней рассматриваются как работа только для элиты, т. е. творческих работников, проектирование более низких уровней и кодирование стоят на ступень ниже, тестирование — еще ниже в этой шкале, а работа по сопровождению рассматривается как задание самого низкого класса, подходящее только для не способного к творчеству и неопытного программиста. Такая позиция не только совершенно неправильна по существу (в предшествующих главах уже говорилось, что тестирование — самая творческая работа в создании программного обеспечения и что отладка требует лучших программистов), но благодаря ей люди, от природы одаренные талантами, необходимыми в тестировании и отладке, занимаются другой работой только ради карьеры. Организация с такой кастовой системой (а эта система может существовать, даже если никто об этом не догадывается) должна добиться справедливого баланса в соотношении числа сотрудников на старших и младших должностях во всех видах работ и должна ясно показать, что во всех видах деятельности ее сотрудники имеют равные возможности для продвижения по службе.

Еще один, последний, вопрос, касающийся комплектования шта-

тов,— отличительные особенности хорошего программиста. Важность этого вопроса очевидна при найме программистов, определении программы обучения и при определении соответствия возможностей программиста требованиям дела. Опрашивая группу руководителей о признаках хорошего программиста, можно услышать о таких качествах, как способности к анализу, распознаванию образов, алгоритмическому мышлению, концентрации и дедуктивным рассуждениям. При опросе группы более опытных руководителей могут появиться такие факторы, как профессионализм, подготовка в области оснований информатики, способность работать в напряженных условиях, способность работать с другими людьми, умение приспосабливаться к изменениям, аккуратность, стремление к совершенствованию и даже чувство юмора. Все это, конечно, требуется в работе программиста, но, возможно, самый существенный фактор — это *способность к общению*. Программисты тратят около двух третей своего времени на общение (слушают, читают, разговаривают и пишут), а согласно микромодели трансляции в гл. 2, большинство ошибок в программном обеспечении связано с проблемами взаимного общения между людьми. Этим объясняется интересный феномен: выпускники с дипломами по литературе или музыке (которые в первую очередь требуют мастерства в общении) часто оказываются выдающимися программистами. Главным вопросом при найме и подготовке программ обучения должно стать умение общаться, т. е. умение программиста выразить свои собственные мысли и понимать идеи других.

ПРОГРАММИСТ-БИБЛИОТЕКАРЬ

Относительно недавнее изменение организации работы над проектом связано с появлением концепции *программиста-библиотекаря:* секретаря или парапрофессионала, принимающего участие в работе над программным проектом [3, 4]. Библиотекарь становится посредником между программистом и вычислительной машиной, выполняя значительную часть «конторской» работы (например, ввод и редактирование данных, выполнение тестов), он ведет основную библиотеку листингов, исходных и объектных модулей, документации и тестов, а также архив всей деятельности и переписки, связанной с проектом. Основное достоинство концепции программиста-библиотекаря с точки зрения надежности — централизация контроля за последними версиями программы и документации, устраняющая обычные проблемы, связанные с кипами листингов в шкафах, когда никто с уверенностью не может указать последнюю «официальную» версию или состояние программы. Если используется инструментальная библиотека на машинных носителях, то библиотекарь обычно является единственным ее пользователем, что сокращает число инструментов, которые должны быть освоены каждым программи-

стом. Библиотекарь часто исполняет также обязанности секретаря и чертежника, печатая и регистрируя корреспонденцию и подготавливая по наброскам графическую документацию, например таблицы решений, структурные схемы программы и HIPO-диаграммы.

Помимо достоинств, концепция библиотекаря имеет также несколько недостатков.

1. Программисты подвергаются заслуженной критике за то, что они не имеют достаточного опыта в роли пользователей своей продукции. Например, важно, чтобы каждый программист, создающий новую операционную систему, имел некоторый опыт применения существующих операционных систем как в роли оператора системы, так и пользователя за терминалом, потому что это позволяет ему (или ей) лучше понять, как принимаемые ими решения сказываются на пользователях. Если концепция программиста-библиотекаря доводится до крайности, программисты не будут иметь непосредственного контакта с машиной, что во всяком случае нежелательно.

2. В некоторых проектах на библиотекаря возлагается ответственность за выполнение сложных функций по редактированию (например, «измени в моей программе каждое вхождение переменной XVELOCITY на YVELOCITY») и даже исправлению синтаксических ошибок. Первое опасно, поскольку часто требует профессионального навыка; второе нежелательно потому, что противоречит требованию точности в программировании (профессиональные программисты не должны делать синтаксических ошибок).

3. Не очевидно, что все время, затраченное программистом на «конторскую» работу, потрачено непродуктивно. Поскольку программирование является прежде всего интеллектуальной работой, программист, по-видимому, нуждается в частых переключениях внимания, чтобы изменить угол зрения на проблему и дать поработать своему подсознанию. Программист, которого вы видите шагающим через вычислительный центр за своим листингом, не зря тратит время; возможно, он во время ходьбы решает свою задачу.

4. Трудно добиться постоянной загруженности библиотекаря. На некоторых этапах он (или она) может быть просто завален делами и станет тормозить всю работу над проектом; на других этапах может оказаться, что ему почти нечего делать.

К счастью, ни одна из этих проблем не является неразрешимой и при достаточно предусмотрительном руководстве программист-библиотекарь может стать ценным звеном в организации проекта.

Споры о достоинствах концепции библиотекаря часто воскрешают в памяти старые споры о том, какого типа системы — пакетные или диалоговые — обеспечивают бо́льшую производительность программиста. Если производительность означает «скорость, с ко-

торой программист пишет свои программы», тогда есть свидетельства в пользу диалоговых систем. Если производительность учитывает полную стоимость продукта (и потому, как указывалось в гл. 1, оказывается тесно связанной с надежностью), тогда диалоговые системы программирования, видимо, нежелательны (это мнение разделяют многие, см., например, [5]). Программисты, использующие пакетные системы, вынуждены тщательно планировать свои действия заранее, а в условиях диалоговых систем таким тщательным планированием часто пренебрегают, что ведет к обыкновению «экспериментировать» за терминалом. Диалоговые системы обычно вынуждают программиста думать слишком быстро и небрежно и, судя по всему, способствуют неряшливости программирования.

Конечно, многие из этих вопросов исчезают, если используется программист-библиотекарь, потому что он исполняет роль посредника между программистами и машиной. Библиотекарь не нуждается в диалоговой системе, потому что он редко бывает способен вести «диалог». Используя диалоговую систему, он обычно работает с ней в режиме дистанционной пакетной обработки; он может внести изменения в исходный модуль и начать выполнение теста, но не ведет диалога с системой (т. е. если что-то идет неправильно, то он сообщает об этом программисту). Можно возразить, что это не так в случае диалоговых инструментов отладки, поскольку в этой ситуации с системой должен взаимодействовать программист, а не библиотекарь. Однако в гл. 13 приводились некоторые свидетельства в пользу того, что отлаживать программу, может быть, лучше всего вообще без использования машины.

БРИГАДЫ ПРОГРАММИСТОВ

Еще одно нововведение в организации проекта — мысль, что программисты должны работать официально оформленными бригадами,— кладет конец традиционному взгляду на программирование как на деятельность в основном индивидуальную. Оказалось, что бригады программистов обладают многими существенными достоинствами: улучшается общение между программистами, вырабатываются общие стандарты качества и самоконтроля, члены бригады многому учатся друг у друга, поощряется безличное программирование, поскольку программисты рассматривают свои результаты как результаты бригады, а не индивидуальные, и легче обеспечивается преемственность, когда один из участников проекта неожиданно оставляет работу. Здесь рассматриваются три распространенных варианта концепции бригады программистов: бригада ведущего программиста, хирургическая бригада и демократическая бригада.

Концепция *бригады ведущего программиста* была предложена

Х. Миллсом из IBM [3]. Бригаду возглавляет ведущий программист: опытный программист высокой квалификации. Он выполняет всю работу по проектированию, пишет программы всех основных модулей, а также выполняет сборку и тестирование программ остальных членов бригады. Он также служит основным посредником между группой и внешними организациями, например другими бригадами или заказчиками, сокращая таким образом число линий связи между участниками проекта. Ведущему программисту ассистирует *второй программист:* специалист той же квалификации, который выполняет такую работу, как исследование альтернативных вариантов при проектировании и планировании, а также принимает участие в проектировании, кодировании и тестировании. Другой постоянный член бригады — программист-библиотекарь. Остальные участники бригады меняются в зависимости от конкретного этапа работы. Например, на время внешнего проектирования можно добавить системного аналитика, а на этапе кодирования — от двух до пяти программистов.

Основное достоинство концепции бригады ведущего программиста в том, что она позволяет вернуть специалистов высокого уровня на главное направление в создании программного обеспечения. Во многих организациях, связанных с программированием, специалисты с большим опытом полностью отходят от участия в работе над проектами. Люди часто считают отход от «кодирования» при первой возможности продвижением по службе. Такое отношение привело множество высококвалифицированных в технических вопросах специалистов на должности, связанные с управлением или кадрами, как можно более удаленные от задач непосредственной разработки программного обеспечения. Концепция бригады ведущего программиста пытается исправить эту ситуацию, устанавливая полезные и престижные должности непосредственно в программных проектах.

Бригада ведущего программиста может иметь также и некоторые недостатки. Скептики часто утверждают, что бригады ведущего программиста хороши для ведущих программистов, но не особенно вдохновляют остальных, потому что программистов часто тасуют из бригады в бригаду, что лишает их чувства ответственности и удовлетворения от выполненной работы. Такое отношение к членам бригады выражается различными авторами: «Другие программисты включаются в группу только для кодирования конкретных, хорошо определенных функций» [6] или «программистов можно легко переключать с одного проекта на другой, и при этом они немедленно погружаются в детали работы...» [7].

Чтобы эта идея работала, ведущий программист должен быть программистом высокого уровня, пользующимся в бригаде авторитетом. Организация бригады из программистов с относительно равным опытом и назначение одного из них ведущим часто приво-

дит к неудаче из-за возникающей зависти. Даже если ведущий программист пользуется глубоким уважением, зависть часто появляется из-за впечатления, что он имеет доступ к особой информации. Многие считают, что по самой природе этой должности ведущему программисту необходима подготовка в области управления и в действительности он должен быть администратором [6]. Последнее определенно противоречит основной идее, поскольку это легко может помешать обмену информацией и взаимосвязям внутри бригады.

Второй вариант, *хирургическая бригада*, был предложен Бруксом [2]. Основное отличие хирургической бригады от бригады ведущего программиста в том, что члены бригады остаются в ней в течение всей работы над проектом и каждый из них имеет свою особую роль, учитывающую его особые таланты. Бригадой руководит ведущий программист («хирург»), но здесь он пишет *все* программы и *всю* документацию на продукцию бригады. В бригаде имеется также ассистент, который исследует альтернативные варианты, служит «адвокатом дьявола» для ведущего программиста и взаимодействует с остальными членами бригады. Брукс предлагает также, чтобы в бригаде был администратор: человек, занимающийся исключительно бюджетом, «жилищными» условиями, машинным временем, кадровыми вопросами. Четвертый член бригады — редактор документации. Хотя всю документацию пишет ведущий программист, редактор отвечает за ясность документации и следит за ее подготовкой.

Хирургическая бригада, как и бригада ведущего программиста, включает также программиста-библиотекаря. Еще один член бригады — инструментальщик: человек, отвечающий за проектирование и разработку (или приобретение) всех необходимых инструментов программирования или тестирования. В бригаду входит также специалист по тестированию, который отвечает за подготовку тестов и разработку заглушек или драйверов. Последний член бригады — языковед: человек, который исследует различные методы программирования и алгоритмы.

Третий тип бригады программистов — *демократическая бригада* [8]. Она отличается тем, что не имеет формального лидера или предварительного распределения обязанностей. Отдельные члены бригады стремятся оказаться на высоте положения (быть неформальным лидером), когда бригада переходит к тому этапу, в котором этот ее член наиболее квалифицирован. Бригада сама распределяет обязанности, основываясь на способностях ее членов. Поэтому такая бригада стремится принимать во внимание прежде всего сильные стороны ее членов. Большое различие между нею и бригадой ведущего программиста состоит в том, что демократическая бригада сохраняет свой состав, переходя от проекта к проекту. Когда проект закончен, бригада не распадается; она целиком переходит к новому

проекту. Это означает, что взаимосвязь, деловые взаимоотношения и стандарты внутри бригады сохраняются от проекта к проекту.

Бригада программистов— ценная концепция, поскольку она означает признание того, что программирование — скорее коллективная, нежели индивидуальная деятельность. Однако, как бывает почти всегда, бригады программистов обладают некоторыми недостатками. Индивидуальная работа каждого программиста менее видна руководителю проекта, что усложняет оценку персонала. В бригадах с официальным лидером последний может стать посредником между руководителем и остальными членами группы, что ведет к серьезным этическим проблемам. (Конечно, оба этих недостатка отсутствуют, если ведущий программист является одновременно руководителем). Может понадобиться подобрать и рабочие помещения таким образом, чтобы создать соответствующую обстановку для работы бригады. Вместо обычной ситуации, когда программисты сидят по одному-двое в комнате или когда все находятся в одной огромной комнате, разгороженной на клетушки, можно предусмотреть такое изменение планировки, чтобы каждая бригада размещалась в отдельной комнате достаточных размеров.

Нет единственно «правильной» для всех условий организации бригады, и в действительности во многих компаниях используются различные модификации трех основных методов ее организации. Если нельзя подобрать ведущего программиста, который пользуется всеобщим уважением, то демократическая бригада является наилучшей альтернативой. С другой стороны, возможно, что группа программистов может испытывать трудности при работе в демократической обстановке, без лидера, и тогда это потребует организации бригады ведущего программиста.

ПРИНЦИПЫ ХОРОШЕГО РУКОВОДСТВА

Из всех аспектов руководства проектом, вероятно, самым важным с точки зрения надежности программного обеспечения является *ответственность руководства*. Многие программисты считают, что руководители только на словах интересуются надежностью программного обеспечения и единственное, что действительно заботит руководителя,— как уложиться в график и бюджет. К сожалению, это мнение часто справедливо. Обычно руководитель проекта оценивается в первую очередь по тому, как выдержан график и бюджет, и редко — по числу ошибок в окончательном продукте. Если ситуация именно такова, руководителю ничего не остается, как перенести эти же критерии на оценку своих сотрудников. Один из способов устранения такого положения — оценивать руководителей, так же как и программистов, по надежности результата работы.

Все еще продолжаются споры о том, насколько подготовленным технически должен быть руководитель. Одна крайность — мнение,

что он должен быть только администратором, т. е. управлять персоналом и почти (или вовсе) не обязан ничего понимать в технических вопросах разработки программного обеспечения. Другая крайность — руководитель должен быть ведущим специалистом организации, возможно даже,— ведущим программистом. Один из ключей к этой проблеме дает исследование программистских проектов Министерства обороны, которое показывает, что фактором, наиболее тесно коррелирующим с успехом проекта, была способность руководителя разбираться в технических вопросах [9]. Руководитель должен быть достаточно хорошо осведомлен в технических аспектах разработки программного обеспечения, чтобы ставить цели, принимать компромиссные решения, оценивать использование новых инструментов и методов, а также точно оценивать своих программистов. Конечно, это не следует доводить до крайности, когда руководитель объясняет программисту, как ему делать свое дело. Во всяком случае, от руководителя проекта требуется, чтобы он мог читать и оценивать продукцию тех, кто работает с ним (или с ней), т.е. их программы, тесты и т. д., и не только мог, но и действительно делал это. Когда руководитель проекта читает программы всех своих программистов, это позволяет ему лучше понимать их проблемы, а также, с другой стороны, улучшает качество программ.

В гл. 3 подчеркивалась необходимость явно сформулировать *цели проекта*, определяющие общий характер принимаемых при программировании основных компромиссных решений, связанных с такими факторами, как эффективность, адаптируемость, календарный план, легкость чтения программы и др. Хороший руководитель не только явно указывает цели, но и привлекает к этой деятельности руководимую им организацию программистов, поскольку опыт показывает, что люди охотнее стремятся к достижению тех целей, которые ставят сами, а не кто-то другой.

В большинстве школ или курсов для администраторов значительное внимание уделяется обоснованию того, что руководитель должен быть готов сказать «нет» своему руководителю. Однако многие руководители не всегда осознают, что программисты в свою очередь также имеют право сказать «нет», особенно если они оказываются в ситуации, когда должны пойти на компромисс с собственными этическими принципами; например, когда их просят сдать программную систему, о которой вполне определенно известно, что она все еще содержит серьезные ошибки, или уложиться в нереальный график, жертвуя ради этого качеством собственной работы. Следует также понимать, что программисты могут принадлежать профессиональным организациям, таким, как ACM, IEEE или ACPA, члены которых обязаны соблюдать соответствующие моральные кодексы, и что в определенных обстоятельствах эти кодексы могут входить в конфликт с целями, определенными администрацией.

Как руководитель вы должны учитывать моральные кодексы этих профессиональных организаций!

В качестве заключительного замечания напомним, что программисты — одна из человеческих профессий. Каждому руководителю независимо от его служебного положения необходимо прочитать книгу Вейнберга «Психология программирования для ЭВМ» [8].

ЛИТЕРАТУРА

1. Ogdin J. L. The Mongolian Hordes versus Superprogrammer, *INFOSYSTEMS*, **19** (12), 20—23 (1972).
2. Брукс Ф. П. мл. Как проектируются и создаются программные комплексы.— М.: Наука, 1979.
3. Baker F. T. Chief Programmer Team Management of Production Programming, *IBM Systems Journal*, **11** (1), 56—73 (1972).
4. Yourdon E., Abbott R. Programmers are Paid to Program: Enter Program Librarian, *INFOSYSTEMS*, **21** (12), 28—32 (1974).
5. Martin J. Design of Man-Computer Dialogues. Englewood Cliffs, N. J.: Prentice-Hall, 1973.
6. Baker F. T. Organizing for Structured Programming, in C. E. Hackl, Ed., Programming Methodology. Berlin: Springer-Verlag, 1975, pp. 38—86.
7. McGowan C. L., Kelly J. R. Top-Down Structured Programming Techniques, New York: Petrocelli/Charter, 1975.
8. Weinberg G. M. The Psychology of Computer Programming, New York: Van Nostrand Reinhold, 1971.
9. Boehm B. W. Keynote Address — The High Cost of Software, Proceedings of a Symposium on the High Cost of Software, Menlo Park, Cal.: Stanford Research Institute, 1973, pp. 27—40.

ГЛАВА 15

Языки программирования и надежность

В окончательном виде программная система представляется тысячами или даже миллионами предложений языка программирования. В некотором смысле черты языка программирования сами по себе не имеют решающего значения, поскольку большинство существенных ошибок в проекте будет сделано еще до того, как дело дойдет до собственно программирования. Однако даже самые тривиальные ошибки при использовании языка могут иногда вести к гибельным результатам. В программе на Фортране, управлявшей первым полетом американского корабля на Венеру, программист написал оператор DO вот так:

DO 3 I=1.3

По ошибке он вместо запятой поставил точку. Однако компилятор воспринял это как допустимый оператор присваивания, поскольку в Фортране нет зарезервированных слов, пробелы игнорируются, а переменные не обязательно должны объявляться явно. Хотя очевидно, что это — неправильный оператор DO, компилятор интерпретирует его как присваивание новой переменной DO3I значения 1.3. Результатом такой «тривиальной» ошибки был срыв полета. Конечно, доля ответственности за эту ошибку стоимостью в миллиарды долларов приходится на программиста и группу тестирования, но разве не сыграли здесь свою роль и особенности языка Фортран?

Влияние языка программирования на надежность программного обеспечения определяется главным образом тем, что программисты-профессионалы тратят больше времени на изучение существующих программ, чем на написание новых. (Слово «профессионал» означает здесь всякого, кто занимается в основном изготовлением программ, предназначенных для использования другими людьми). Деятельность по отладке, сопровождению и развитию программ во многом связана с чтением и пониманием программ, написанных на языке программирования. Таким образом, характеристики языка программирования действительно существенно влияют на ошибки в программном обеспечении.

Задача этой главы — отметить недостатки существующих языков программирования и предложить решения для языков будущего. Хотя для иллюстрации большинства недостатков используется PL/1,

из этого не следует, что PL/1 — плохой язык. Разработка PL/1 была начата в 1963 г., до начала «революции в программировании», и многие из недостатков PL/1 имеются также в других языках. PL/1 выбран потому, что этот язык (или некоторый его диалект) изучается во многих учебных программах по информатике (так что читатель, возможно, знаком с ним лучше, чем с другими языками, например с Коболом или RPG), он был разработан относительно недавно по сравнению с большинством других широко распространенных языков и предназначался для широкого спектра приложений.

Один из выводов, к которому мы приходим, состоит в том, что универсальный язык программирования невозможен. Слово «универсальный» не обязательно предполагает применимость к широкому спектру приложений; здесь этот термин означает возможность употребления широким кругом пользователей, таких, как профессиональные программисты, аналитики, любители или «программисты от случая к случаю» (такие, как инженер-химик, который время от времени пишет небольшие программы решения каких-нибудь задач) и студенты, впервые знакомящиеся с концепциями программирования. К языку, разработанному для такого широкого контингента пользователей, предъявляются противоречивые требования. легкость в изучении и удобство при использовании, легкость расширения самого языка и легкость развития написанных на нем программ, сопротивляемость ошибкам программирования, независимость от машины, эффективность объектной программы и эффективность компиляции. Если пытаться удовлетворить всем этим требованиям, язык неизбежно окажется отнюдь не идеальным инструментом для каждой конкретной категории пользователей. Самыми удачными языками программирования являются те, разработчики которых ориентировались на конкретный контингент пользователей (например, БЭЙСИК был разработан для обучения студентов университетов, а RPG — для «коммерческих» программистов со слабой подготовкой в области обработки данных или вовсе без нее).

В настоящей главе мы рассмотрим языки, созданные для использования профессиональными программистами в тех проектах, где надежность важнее всего. Для этих языков такие цели, как легкость изучения и применения, скорость кодирования программ и эффективность компилятора, несущественны. Основными целями становятся легкость чтения написанных на языке программ, отсутствие особенностей языка, провоцирующих ошибки, и возможность ясно и просто выражать функцию и логику программы. Подобные цели ставились в нескольких проектах разработки новых языков. Например, системный язык для проекта SUE (язык для реализации операционных систем) проектировался на основе следующих двух принципов [1]:

1. Основная функция языка — общение между людьми, и существенно, чтобы такое общение было ясным, удобным и недвусмысленным. Чем больше система, тем более решающим является это общение.

2. Важность языков программирования высокого уровня для конструирования систем заключена не только в мощных операциях, работу с которыми они делают удобной, но и в способности предупреждать ошибки и «умничанье».

Как и почти в каждой сфере деятельности, в области разработки программного обеспечения следующие ниже предложения, касающиеся создания языков, основаны скорее на интуиции и опыте, чем на достоверных экспериментальных данных. Разработка языка — одна из немногих сфер деятельности, где управляемые эксперименты осуществимы, но выполнено их на удивление мало. Двумя замечательными исключениями являются эксперимент Руби для ВВС США [2] и эксперименты Гэннона в университете Торонто [3]. Целью эксперимента Руби было сравнение PL/1 с тремя другими языками. Было выбрано семь различных прикладных задач, и для каждой написано по две программы, одна из которых — на PL/1, а другая — на JOVIAL, Фортране или Коболе, в зависимости от того, какой из языков казался наиболее подходящим. Общий вывод состоял в том, что PL/1 не уступает ни в одной области применений, но подлинная ценность этого эксперимента заключается в обилии количественных и качественных данных о возникавших ошибках программирования.

Исследование Гэннона представляло собой статистический эксперимент с 51 программистом (студенты университета, не имеющие серьезного опыта) и двумя языками. Первый из них — TOPPS, язык «выражений», спроектированный для обучения работе с асинхронными процессами. Второй язык, TOPPS II, был специально спроектирован для этого эксперимента. Предполагалось, что различные конструкции TOPPS провоцируют ошибки, и такие конструкции в TOPPS II были переделаны, чтобы статистически проверить различия. В этом исследовании были проанализированы причины и сопротивляемость (время до обнаружения) 1248 ошибок. Оба исследования подтверждают многие из предположений, высказанных в данной книге.

В течение многих лет основным вопросом разработки языка считалась разработка его грамматики (или синтаксиса). Например, спецификации многих форм Алгола в большей степени касаются синтаксиса и в меньшей — семантики, а некоторые вопросы семантики явно оставлены «зависящими от реализации». Важно, однако, понимать, что в полной разработке языка («языковой системы» по работе [4]) грамматика — только один из многих аспектов. Другие факторы, такие, как семантика, листинги компилятора, диагностические сообщения, средства диагностики в процессе счета, средства тестирования и отладки, документация, средства периода компиляции и межмодульные ссылки (например, редактирование связей), заслуживают равного внимания при разработке языка.

ЕДИНООБРАЗИЕ

Один из самых важных принципов синтаксиса и семантики языка — единообразие. Языковая конструкция, появляющаяся в разных контекстах, должна иметь одинаковые синтаксис и семантику (например, если и операторы ввода-вывода, и операторы присваивания могут содержать индексные выражения, правила должны быть одинаковыми в обоих случаях). Одна и та же конструкция не должна иметь несколько смыслов (например, во многих языках оператор I=A(B) может означать «положить I равным значению B-го элемента массива A» или «передать B функции A в качестве аргумента и положить I равным значению, возвращенному A»). Синтаксис разных языковых конструкций должен быть по возможности похожим (все операторы PL/1, *кроме* оператора присваивания, начинаются с ключевого слова).

Язык должен также подчиняться неформальному закону «минимальной неожиданности», т. е. он не должен «подводить» программиста. Одним из самых сбивающих с толку нарушений этого закона являются правила умолчания PL/1. Для того чтобы сохранить некоторую совместимость с Фортраном (цель сама по себе сомнительная), многие компиляторы PL/1 в соответствии с правилами умолчания считают переменные, начинающиеся с букв от I до N, целыми двоичными переменными, как показано в первых двух строках приведенного ниже примера. Заметьте, однако, что происходит, если программист явно объявляет часть атрибутов. Правила умолчания неожиданно изменяются так, что в одном случае он получает десятичное число с точностью в пять десятичных цифр, а в другом — число с плавающей точкой и точностью в 21 двоичную цифру.

Вид оператора DECLARE	Приписанные атрибуты
—	AUTOMATIC FIXED BINARY(15,0) REAL
DECLARE I;	AUTOMATIC FIXED BINARY(15,0) REAL
DECLARE I FIXED;	AUTOMATIC FIXED DECIMAL(5,0) REAL
DECLARE I BINARY;	AUTOMATIC FLOAT BINARY(21) REAL
DECLARE I STATIC;	STATIC FIXED BINARY(15,0) REAL

В большинстве языков имеются поразительные примеры отклонений от единообразия. Косая черта "/" в Фортране используется как разделитель в операторе DATA и в объявлении блока COMMON, в то время как в операторе EQUIVALENCE в качестве разделителя используются скобки. Для большинства описателей полей в объявлении формата FORMAT длина поля указывается после символа кода преобразования (например F8.0, A6,I5), но в двух случаях код преобразования следует за длиной поля (например, 4*X*). В реализации PL/1 фирмы IBM имена переменных могут иметь длину до 31 сим-

вола, но имена процедур и глобальных переменных должны содержать только до 7 символов (а чтобы еще усложнить дело, для имен длиной более 7 символов лишние символы отбрасываются, причем используются четыре первых и три последних). Программистов часто учат использовать лишние скобки, чтобы сделать выражение более понятным, поскольку лишние скобки игнорируются компилятором. В PL/1 так обычно и происходит, но, к сожалению, есть и исключения. Следующие два оператора CALL семантически различны; первый соответствует «вызову по ссылке», второй — «вызову по значению».

```
CALL AMOD(X);
CALL AMOD((X));
```

Язык можно усложнить и тем, что одному и тому же знаку операции приписывается разный смысл, подобно тому как в PL/1 используется знак равенства и для присваивания, и для сравнения. Оператор «$A=B=C$» означает «присвоить A значение 1, если B равно C, и нуль в противном случае». Заметьте также, как по-разному это выражение интерпретируется в разных контекстах:

```
A=B=C;   — то же, что A=(B=C);
IF A=B=C — то же, что IF (A=B)=C
```

Этот оператор IF означает совсем не то, что можно ожидать. Выражение после IF истинно, если A равно B и C имеет битовое значение «единица» либо если A не равно B, а C имеет битовое значение «нуль».

В цикле DO язык PL/1 допускает задание списка выражений, так что цикл выполняется по разу для каждого из них, причем параметр цикла при i-й итерации принимает значение i-го выражения. Таким образом, можно было бы ожидать, что цикл

```
DO I=0,I=1;
    PUT DATA(I);
END;
```

будет выполнен дважды, причем напечатаны будут значения 0 и 1. Эти надежды оправдываются только наполовину; цикл выполняется дважды, но оба раза печатается значение 0. Объяснение причины я оставляю читателям в качестве упражнения (она связана с отсутствием единообразия в употреблении знака равенства).

Некоторые языки позволяют программисту употреблять сокращения. Этой практики следует избегать, потому что она усложняет чтение программы, вводит несколько конструкций с одинаковым смыслом, а кроме того, сокращения не всегда единообразны. Напри-

мер, для ключевых слов PROCEDURE, REDUCIBLE и VARYING используются сокращения PROC, RED и VAR соответственно, но слова SEQUENTIAL, FIXEDOVERFLOW и CONTROLLED сокращаются до SEQL, FOFL и CTL.

Последняя область, в которой нарушается единообразие, относится не к синтаксису или семантике, а к объектному коду. Предположим, что мы разрабатываем новую программу на PL/1 (или, если угодно, на любом другом языке), и обнаруживаем, что у нас уже есть некоторые подпрограммы на Фортране или Коболе, которые можно было бы включить в нее. Смогли бы мы вызвать их из программы на PL/1? Обычный ответ — нет, без больших усилий не смогли бы. Требования к среде, в которой выполняются объектные программы на разных языках, существенно расходятся между собой, даже если компиляторы написаны для одной и той же вычислительной системы, причем одной и той же компанией.

Одно из решений большинства этих проблем — применить идею концептуальной целостности, обсуждавшуюся в гл. 4, к проектированию языка. Концептуальная целостность предполагает, что язык должен иметь единый гармоничный набор согласованных функций и конструкций. Любая «особенность», которая не согласуется с этим набором с синтаксической, семантической или любой другой точки зрения, должна быть исключена из языка. В гл. 4 указывалось также, что для достижения концептуальной целостности требуется, чтобы группа разработчиков была очень маленькой (предлагалось иметь не более двух участников, имеющих право принимать решения). В подтверждение этого можно отметить, что три наиболее единообразных языка — АПЛ, Паскаль и системный язык для проекта SUE — были созданы одним-двумя разработчиками.

ПРОСТОТА

Точности в программировании можно добиться, лишь в совершенстве владея своим языком программирования. Язык должен быть достаточно мал, чтобы можно было полностью понимать его, и притом он должен способствовать наиболее ясному выражению мыслей программиста. Язык не должен ни поощрять запутанности, ни увеличивать вероятность ошибки программиста. Число возможных альтернатив должно быть сведено к минимуму, чтобы у программиста не было слишком много способов выразить одно и то же.

Один из предлагавшихся для достижения простоты способов [4] предполагает опираться в разработке языка на «магическое число семь» (которое, как оказалось, тесно связано со способностью человека различать объекты и хранить их в кратковременной памяти). При записи предложения на хорошем языке программист в каждый момент времени должен иметь дело не более чем с семью различ-

ными вариантами. Это значит, что должно быть не более семи типов операторов, семи возможных ключевых слов для каждого оператора, семи типов данных, семи операций и т. д. Это достигается исключением лишних альтернатив или введением в язык дополнительных уровней иерархии.

PL/1 является наилучшим примером языка, который не отвечает большинству упоминавшихся выше требований (следом за ним идет Кобол). Невероятно, чтобы какой-нибудь программист, работающий на PL/1, претендовал на знание этого языка в совершенстве. В PL/1 имеется более 30 основных типов данных, и некоторые из них, кажется, скорее вредны, чем полезны. Один из таких видов — переменные типа LABEL, позволяющие программе динамически изменять свою структуру, в результате чего она становится непостижимой. Следующий пример показывает, как может быть сконструирован «динамический GO TO»:

```
DECLARE XLAB LABEL;
        .
        .
XLAB=ALAB;
        .
        .
XLAB=BLAB;
        .
        .
GO TO XLAB;
```

Такой оператор GO TO много хуже обычного, потому что без прокрутки программы читатель не может узнать даже, на какие операторы возможен переход.

В PL/1 имеется много полезных особенностей, но беда в том, что их слишком много. Можно заподозрить, что PL/1 мог бы быть исключительно хорошим языком, если бы только сделать три изменения: отказаться от 80% его возможностей, совсем исключить автоматическое преобразование данных (подробнее о нем ниже) и лучше написать и организовать документацию.

Простота языка не предполагает бесконечной общности; она означает ясность и целостность концепций языка. Измерять простоту вполне разумно объемом руководства по языку. Ведь простой язык — это не обязательно язык менее мощный; вероятно, наилучший пример этого — АПЛ. Системный язык проекта SUE включает такие мощные конструкции, как блочная структура, разнообразные типы данных, в том числе массивы и структуры, макросредства периода компиляции, операторы IF THEN ELSE, DO, CASE и ASSERT, а также явное размещение в памяти, и при этом формальное описание его синтаксиса занимает только три страницы.

ОБЪЯВЛЕНИЕ ДАННЫХ

Важным вопросом при проектировании языка является способ, которым программист объявляет или определяет данные в своей программе. Указание атрибутов (например, размеров и типа) каждой переменной важно, потому что это позволяет компилятору обнаруживать определенные ошибки (например, попытку сложить символьную строку с десятичным числом) и служит важной формой документирования программы (понимание данных — основа понимания программы).

Основное правило здесь формулируется в виде требования, чтобы все переменные были объявлены явно. АПЛ слаб в этом отношении, потому что он не допускает объявлений данных в какой бы то ни было форме. В Фортране (и во многих других языках) всякая переменная, не объявленная явно, считается объявленной по умолчанию. Например, опечатка в операторе

```
INDEX = INDEZ + 1
```

который должен был бы увеличивать значение INDEX, не будет обнаружена большинством компиляторов. Компиляторы решат, что INDEZ — еще одна переменная, и она, вероятно, будет иметь непредсказуемое значение, благодаря чему ошибку будет трудно локализовать. Если одна из программ на Фортране передает другой в качестве аргумента массив и соответствующий параметр в вызываемой программе не объявлен явно, компилятор рассматривает ссылку на элемент массива во второй подпрограмме как вызов подпрограммы-функции, что ведет к попытке выполнить массив!

Интересное расширение правила явных объявлений введено в системном языке для проекта SUE [1]. В этом языке запрещено использовать без объявлений любые константы, значение которых больше 2. Всякая константа, превосходящая 2, должна быть явно объявлена, и ей должно соответствовать символическое имя. Таким образом в программе минимизируется использование «мистических чисел» — пример того, как язык поощряет хороший стиль программирования.

Вторая рекомендация по объявлению данных — исключение правил умолчания (чтобы программист был обязан явно указывать все атрибуты переменной), либо в противном случае правила должны быть крайне простыми. Я уже приводил пример сложных правил умолчания в PL/1. Эти удивительные правила умолчания были очень распространенной причиной ошибок в эксперименте Руби [2]. PL/1 усложняет ситуацию еще и другими средствами. Имеется оператор DEFAULT, позволяющий программисту изменять правила умолчания. Некоторые компиляторы PL/1 идут даже дальше, позволяя

не выводить листинги, показывающие, как применены правила умолчания! Справедливости ради следует сказать, что правила умолчания PL/1 были предназначены для того, чтобы облегчить жизнь любителям и непрофессиональным программистам. Последнее еще раз подтверждает то положение, что хороший универсальный язык, вероятно, невозможен.

Связанная с этим проблема проектирования языка описывается как *раздвоение* [4]; ситуация, когда программа делает не то, что она, как кажется, должна делать. Самый распространенный случай — когда ссылка на одну переменную в действительности относится к другой переменной. Например, нет гарантии, что оператор Фортрана

```
A(J) = 1.0
```

изменяет значение элемента массива *A*. Он может изменять значение элемента другого массива, если значение *J* выходит за границы массива *A*. Конструкция ON в PL/1 — второй пример раздвоения, поскольку во время выполнения конкретного оператора может возникнуть соответствующая ситуация прерывания, вследствие чего конструкция ON получит управление и может изменить значения некоторых дополнительных переменных (не указанных явно в этом операторе). Третий пример раздвоения — использование одной общей памяти для нескольких переменных за счет таких конструкций, как EQUIVALENCE в Фортране, REDEFINES в Коболе или атрибут DEFINED в PL/1. Всякий раз, когда оператор изменяет значение одной из переменных, автоматически изменяются другие. Нет необходимости говорить, что конструкции, ведущие к раздвоению, запутывают программу, и в хороших языках их следует избегать.

Принцип единообразия относится также и к объявлению данных. В PL/1 есть полезная возможность применять операции к целым массивам и структурам так же, как к отдельным переменным. Например, следующий оператор присваивания полагает все десять элементов массива *A* равными нулю:

```
DECLARE A(10) FIXED DECIMAL(5,0);
A = 0;
```

В операторе DECLARE переменным можно также присвоить начальные значения, так что можно было бы ожидать, что следующий оператор имеет тот же эффект:

```
DECLARE A(10) FIXED DECIMAL(5,0) INITIAL(0);
```

К сожалению, это не так; будет инициализирован только первый элемент. Для инициализации всех элементов нужен атрибут INITIAL(10(0)). Заметьте, от нас требуют дважды указать «10», что может привести к случайной ошибке при изменении размеров массива.

ТИПЫ И ПРЕОБРАЗОВАНИЯ ДАННЫХ

В языках программирования, по-видимому, наметилась тенденция давать программисту широкий выбор вариантов представления данных. Например, в PL/1 имеется более 30 типов данных, а если учесть, что многие из этих типов допускают различную точность (число битов или цифр в элементе данных), то программист оказывается перед выбором из нескольких тысяч альтернатив. Эта тенденция должна быть изменена, поскольку она ведет к ненужной сложности, уменьшению независимости от машины, принятию программистом произвольных решений и патологическим правилам преобразования данных.

В арифметике действительных (не комплексных) чисел PL/1 предлагает 240 различных вариантов точности десятичных чисел с фиксированной точкой, 992 варианта точности двоичных чисел с фиксированной точкой, 33 различных варианта точности десятичных чисел с плавающей точкой и 109 вариантов точности двоичных чисел с плавающей точкой. Почти все эти возможности можно выбросить безо всяких потерь. Например, двоичные числа с плавающей точкой (вида 11010.11011E2B) можно исключить, равно как и двоичные числа с фиксированной точкой.

Нет необходимости давать программисту так много возможностей управлять внутренним представлением данных (двоичные или десятичные) и точностью чисел, нет даже нужды указывать, представляется ли число с фиксированной или плавающей точкой. Доказательством этого служит язык АПЛ; в то время как в PL/1 — тысячи числовых форм, в АПЛ— ровно одна. Данные этого единственного арифметического типа языка АПЛ хранятся в одной из трех внутренних форм (битовой, двоичной или с плавающей точкой), в зависимости от того, как они используются в программе. Если переменная принимает значения, равные только нулю или единице, она хранится в битовой форме. Если она имеет только целочисленные значения, хранится в виде двоичного целого числа. Если переменная имеет дробные значения, она хранится в виде числа с плавающей точкой. Более того, эта внутренная форма динамически изменяется в зависимости от того, как переменная используется в программе.

Главная причина простоты АПЛ — ограниченность набора его типов данных. Есть только два типа (числа и литеры) и два способа организации данных (скаляры и массивы). В АПЛ, однако, простота

заходит слишком далеко. Полезная концепция *структуры* — используемый в PL/1 и Коболе принцип организации данных, позволяющий представить запись с неоднородными полями,— отсутствует в АПЛ. В качестве примера, возможно чрезмерной общности, можно упомянуть, что в АПЛ такие объекты, как метки, представляются скалярными переменными, поэтому программист может выполнять трюки вроде использования метки в качестве переменной в операторе присваивания [1]).

Важная сторона языка высокого уровня — устранение концепции линейной последовательной памяти и замена ее концепцией непересекающихся переменных. Этому противоречит, например, *переменная типа указатель*, значение которой есть адрес другой переменной. Концепция указателя в том виде, как она воплощена в PL/1, чревата ошибками, поскольку она дает программисту прямой доступ к памяти машины. Если указатель ссылается на область данных в памяти и эта область освобождается, то указатель не прекращает «указывать», он ссылается на то же место в памяти. Это приводило к тонким ошибкам и поощряло трюкачество в программах. Хоар [6] считает, что введение переменных типа указатель — «такой шаг назад, от которого мы можем никогда не оправиться».

Если в языке имеется такая конструкция, она должна быть ограничена по крайней мере в двух отношениях: когда область данных освобождается, все ссылающиеся на нее указатели должны быть «свернуты» (им присваиваются нулевые значения), а при объявлении указателей необходимо объявлять также, на какие конкретные переменные или типы данных они могут ссылаться, что позволит компилятору обнаруживать некоторые ошибки при присваиваниях. В Алголе 68 и системном языке для проекта SUE реализовано последнее ограничение.

Еще лучше — вообще отказаться от концепции указателя. Переменная типа указателя обычно используется для реализации линейных списков (таблицы, стеки и очереди), многосвязных списков и деревьев. Вместо того чтобы давать программисту в руки опасную гибкость указателя, можно было бы прямо включить эти структуры в язык. Например, в системном языке для проекта SUE имеется ценная структура данных, называемая *множеством*, и связанные с ней операции, такие, как объединение, пересечение, дополнение, включение и равенство [7].

Одно из преимуществ наличия в языке различных типов данных состоит в том, что компилятор может обнаружить такие ошибки, как, например, сложение числа и символьной строки. PL/1 в этом отношении страдает тремя серьезными недостатками. Язык не только содержит тысячи типов данных, но и не требует проверки

---

[1]) Метки в АПЛ — локальные скалярные константы, их значения используются в выражении, но не могут быть изменены присваиванием.— *Прим. перев.*

их согласованности, а правила преобразования в нем невероятно сложны. В PL/1 можно перемешивать действительные и комплексные числа, строки битов и символов в одном операторе присваивания, хотя результат этого зачастую трудно предсказать.

Автоматическое преобразование данных возникает в PL/1 всякий раз, когда в выражении используются переменные, различающиеся типом или точностью. Правила преобразования занимают многие страницы в руководстве по языку [8] и так сложны, что даже опытные программисты на PL/1, вероятно, не угадают ответов на приведенные ниже примеры. Записанное на PL/1 выражение 25+1/3 вырабатывает значение 5.333... , а выражение 25+01/3 дает правильный ответ 25.333... , иллюстрируя, таким образом, что убежденность в бессмысленности ведущих нулей не всегда оправдана. В следующей программе:

```
DECLARE (I,J) FIXED BINARY (31,5);
I = 1;
J = 1;
I = I + .1;
J = J + .1000;
```

окончательное значение *I* равно 1.0625, а значение *I* равно 1.09375, откуда видно, что *заключительные* нули также важны.

Куда передается управление в следующей программе?

```
DECLARE B BIT(1);
B=1;
IF B=1 THEN GO TO Y;
       ELSE GO TO X;
```

К этому моменту читатель может решить делать ставки на то, что противоречит интуиции, и, если так, он прав; управление передается по метке *X*. Причина в том, что константа 1 (десятичная, с фиксированной точкой и точностью (1,0)) преобразуется к битовому типу в присваивании и к двоичному с фиксированной точкой при сравнении. В мою задачу не входит объяснение правил преобразования, я только иллюстрирую их эффект. В качестве еще одного примера попробуйте угадать, чему равно значение *B* после выполнения

```
DECLARE B BIT (5);
B=1
```

Значение *B* — строка битов '00010'B.

Преобразования данных при вызове процедуры в PL/1 могут быть крайне опасны возможными ошибками. Предположим, что

процедура *A* вызывает процедуру *B*, передавая ей аргумент *X*. Обычный механизм передачи параметров — вызов по ссылке (адрес *X* передается *B*). Если, однако, тип или точность аргумента отличаются от соответствующих атрибутов параметра процедуры *B*, выполняется автоматическое преобразование *X*, преобразованное значение присваивается промежуточной переменной и адрес этой промежуточной переменной передается *B*. Возможно, предполагается, что *B* изменит значение *X*, однако *X* никогда не изменится, изменится только значение промежуточной переменной. Это происходит без всякого предупреждения со стороны компилятора и ведет к чрезвычайно тонким ошибкам.

Все эти примеры свидетельствуют о том, что в языке не должно быть никаких автоматических преобразований данных. Всякая попытка смешивать данные разных типов в выражении должна отмечаться компилятором. Конечно, может случиться, что возникнет необходимость преобразовать переменную к другому типу, но для этого в языке должны быть функции, позволяющие явно потребовать преобразования. Алгол не следует этой рекомендации в точности, но предлагает разумный компромисс. В Алголе 68 разрешено только «расширение» [9]. Целое значение **integer** может быть автоматически преобразовано к типу **real** (с плавающей точкой) или **complex**, и значение типа **real** может быть преобразовано к типу **complex**, но неявные преобразования в противоположном направлении запрещены.

ПРОЦЕДУРЫ И ОБЛАСТИ ДОСТУПНОСТИ ДАННЫХ

Еще одна важная характеристика языка программирования — в какой степени он помогает воплощению желательных свойств структуры системы или программы, обсуждавшихся в гл. 5 и 6. Такие концепции, как уровни абстракции, высокая прочность модуля и слабое сцепление, не могут быть полностью реализованы без надлежащего оформления языковыми средствами.

Одна из основных конструкций, которые должны существовать в языке,— модуль, или внешняя процедура. Модуль представляет собой замкнутую процедуру, которую можно отдельно компилировать и вызывать из любого места внутри программы. Большинство языков обеспечивает эту возможность; одним из примечательных исключений является Алгол.

Важная концепция, которая должна естественно вписываться в язык — информационно прочный модуль, рассматривавшийся в гл. 6. Такой модуль имеет несколько точек входа, причем каждый из этих входов соответствует замкнутой подпрограмме. Эти входы должны быть не зависимыми друг от друга (например, имена переменных и параметров должны быть локальными для текста соответствующей подпрограммы), но это не касается той структуры

данных, которую этот модуль «скрывает»; она должна быть известна всем входам. PL/1 наилучшим образом воплощает эту концепцию, но, изучая рис. 8.2 и 8.7, можно обнаружить некоторые недостатки. Один из входов должен быть произвольно назван «главным» (имя процедуры), что противоречит идее равноправия всех точек входа. Единственный способ изолировать текст подпрограммы, соответствующий точке входа,— это заключить его в блок BEGIN, что несколько уменьшает эффективность при выполнении. Кроме того, язык не позволяет заключать в блок BEGIN операторы ENTRY, поэтому имена параметров не будут локальными для входа.

Есть тип данных, который связан с использованием информационно прочных модулей и поэтому обязательно должен присутствовать в языке,— это *имя*. Заметьте, что в загрузчике, спроектированном в гл. 6, имя ТВИМ передается многим модулям, но только один из них знает что-то о структуре ТВИМ. Единственный способ реализовать это в PL/1 — объявить ТВИМ переменной типа указатель, но последнее может привести к некоторым проблемам вследствие излишней общности концепции указателя. Чтобы надлежащим образом способствовать применению методов композиционного проектирования, языки должны содержать *имена* как тип данных, т. е. необходим метод передачи между модулями переменной или структуры данных без какой бы то ни было информации о ее атрибутах или формате. Модулю, в котором переменная объявляется как *имя*, не должно быть разрешено изменять эту переменную или использовать ее иначе, чем в качестве аргумента или параметра.

Оператор CALL в языке может быть значительно усовершенствован. В процессе проектирования сопряжения модуля параметры делятся на входные и/или выходные, но в большинстве языков такое различие между параметрами никак не отражено. Можно было бы усовершенствовать оператор CALL, изменяя его форму таким образом:

```
CALL FINDTOK IN(XSTRING) OUT(SIZE,TOKEN)
```

Это не только облегчает чтение программы, но и позволяет обнаружить некоторые ошибки. Например, все аргументы, объявленные входными, но не выходными, можно было бы защитить от изменения в вызываемом модуле [1]).

В языке следует также ограничить другую опасную практику использования оператора CALL. Следующие два оператора дают еще один пример раздвоения, поскольку и для M, и для N в вызванном оператором CALL модуле назначена одна и та же область памяти.

---

[1]) Это реализовано, например, в языке ИНФ; см. Клещёв А. С., Тёмов В. Л. Язык программирования ИНФ и его реализация.— Л.: Наука, 1973.— *Прим. ред.*

```
CALL X (I,I);
.
.
X: PROCEDURE (M,N);
```

Компилятор должен сигнализировать о всех случаях неоднократного вхождения одного и того же имени аргумента в оператор CALL.

Последняя важная сфера, в которой проявляется связь между языком и структурой системы,— это правила определения *области доступности* данных: правила, в соответствии с которыми данные, не передаваемые как аргументы, могут одновременно использоваться разными модулями и процедурами. В гл. 5 и 6 объяснялось, что использование глобальных данных в программе нежелательно, но это не обязательно означает, что из языка следует исключить соответствующие понятия. Использование глобальных данных иногда оправдано, но должно допускаться при тщательно контролируемых условиях.

Основное правило состоит в том, что переменная должна быть локальной для модуля или процедуры, если программист не отрицает этого явно. АПЛ неудачен в этом отношении, поскольку в нем принят противоположный принцип; если программист явно не указывает, что переменная локальна для данного модуля (функции АПЛ), она становится глобальной. Это распространенная причина коварных ошибок, когда программист последовательно использует во многих функциях АПЛ имя переменной, например I, в качестве управляющей переменной цикла, но в одной или нескольких функциях он по небрежности забывает объявить эту переменную I локальной.

В большинстве других языков все переменные по умолчанию локальны, но там, где имеется концепция *внутренней процедуры*, т. е. замкнутой подпрограммы внутри модуля (например, в PL/1, Коболе, Алголе), это правило обычно не соблюдается. Если переменная встречается во внутренней процедуре, но не объявлена там явно, ею совместно пользуются внутренняя и охватывающая ее процедуры. Вулф и Шоу [10] привели серьезные аргументы против такой практики определения областей доступности. Конечно, если принять предложение, что все переменные должны быть объявлены явно, эта проблема исчезает.

Нужно принять следующие принципы определения областей доступности данных и глобальных переменных. Все данные должны быть локальными для процедуры, в которой они объявлены, если только программист явно не потребует противного. Ни одна переменная не должна быть доступна двум процедурам, если только обе они не указывают этого явно. Каждая глобальная переменная должна «принадлежать» одному модулю, который может указывать,

какие еще модули имеют доступ к этой переменной и какого типа доступ (например, только чтение) разрешен каждому из этих модулей.

УПРАВЛЯЮЩИЕ СТРУКТУРЫ

С возникновением идеи структурного программирования большинство исследователей в области языков сосредоточили свое внимание на управляющем графе программы, в первую очередь — на исключении оператора GO TO. Мода на такого рода исследования зашла так далеко, что появилось даже сатирическое предложение ввести оператор COME FROM (принять управление из) [11]. Хотя оператор GO TO часто является причиной чрезмерного усложнения программы, не все случаи его употребления следует порицать, а иногда его использование может оказаться и желательным [12].

Д. Кнут считает, что «слишком много внимания было уделено исключению GO TO вместо действительно важных вопросов». Эксперимент Руби [2], видимо, подтверждает это. Кроме ошибок в пунктуации, было обнаружено 266 ошибок, и самыми распространенными из них (119) были ошибки в описании данных, файлов и форматов. Неясно также, насколько отсутствие GO TO могло бы предупредить остальные 147 ошибок; ведь только 32 ошибки были непосредственно связаны с неправильным управлением или принятием решений.

Популярным расширением традиционного набора управляющих структур является оператор CASE. Это, в сущности, условный оператор с несколькими ветвями, позволяющий выполнить одну из групп операторов в зависимости от значения индексной переменной. Например, в ALGOL-W имеется оператор CASE, в котором выбирается один из n блоков в соответствии со значением индексной переменной. Определение языка требует, однако, чтобы значение индексной переменной было меньше или равно n. Конечно, нет гарантии, что это требование будет соблюдаться, и поведение программы в таком случае не определено. Оператор CASE всегда должен содержать условие ELSE, указывающее, какой блок должен быть выполнен в том случае, когда не выбран никакой другой блок.

Хотя оператор CASE и весьма популярен, он, видимо, имеет немного преимуществ перед последовательностью операторов IF-THEN. Кроме того, использование числовых индексов от 1 до n не очень удобно во многих реальных ситуациях (например, может оказаться желательным выполнить действия четырех типов в зависимости от того, имеет ли TRANCODE символьное значение «ADD», «UPD», «DEL» или «DIS») [1]).

---

[1]) Оператор CASE в Паскале и некоторых других языках предоставляет такие возможности (см. например, Вирт Н. Систематическое программирование. Введение.— М.: Мир, 1977).— *Прим. ред.*

Проблемы, связанные с управляющими структурами, могут быть простыми или весьма тонкими. В PL/1 имеется оператор END, который используется, чтобы обозначить конец группы операторов, например цикла, группы DO, блока BEGIN. Компилятор может обнаружить «недостачу» операторов END, сравнивая количество DO и BEGIN с количеством END, и это часто полезно. Однако PL/1 ослабляет эту возможность с помощью *«кратных» END*. Если за словом «END» следует метка (например, имя процедуры), то компилятор генерирует все недостающие «END» и помещает их непосредственно перед отмеченным (отсюда название «кратные» END). Последствия такой акции часто губительны; это было, в частности, распространенной причиной ошибок в эксперименте Руби. Кратным END не должно быть места в языках программирования; они только поощряют неряшливость в программировании. Обобщая описанный пример, можно сказать, что компиляторы должны избегать догадок или предположений относительно намерений программиста.

Язык должен запрещать программе изменять саму себя, но большинство языков в той или иной форме позволяет делать это (например, как упоминалось ранее, переменная типа метки есть способ модификации оператора GO TO). Кобол также содержит неприятное нарушение этого принципа — конструкцию ALTER. Предложение Кобола

```
ALTER ROUTINE-1 TO PROCEED TO END-ROUTINE
```

требует, чтобы предложение GO TO в параграфе ROUTINE-1 понималось как GO TO END-ROUTINE.

Нет нужды говорить, что язык должен содержать конструкции, необходимые для структурного программирования (например, DO-WHILE и IF-THEN-ELSE) и что эти управляющие структуры должны быть спроектированы просто и единообразно. В этом отношении уязвим АПЛ; в языке не только отсутствуют управляющие структуры такого рода, но и те, что в нем есть, довольно непонятны. Например, первое из предложений на АПЛ означает «если X равно Y, перейти к предложению 6», а второе — «если X равно Y, перейти к предложению 6, иначе — возврат из функции».

```
→6 × ⍳ (X=Y)
→6 × (X=Y)
```

В качестве эксперимента была предпринята попытка исправить эти недостатки, дополнив АПЛ обычными управляющими структурами [13].

ДЕЙСТВИЯ С ДАННЫМИ

Конечной целью языка программирования, естественно, является выполнение операций над некоторым набором данных. Мощность операций языка положительно влияет на надежность программирования, сокращая объем программы, которую должен написать программист. PL/1 позволяет выполнять операции над целыми массивами или структурами или над подмассивами (например, отдельной строкой или столбцом), что дает программисту возможность одним оператором сделать то, что могло бы потребовать целого цикла в программе на Фортране. Мощь АПЛ во многом объясняется тем, что его операции единообразно применяются к векторам и массивам так же, как и к скалярам.

Распространенным источником ошибок является употребление логических операций, таких, как И и ИЛИ. Например, если нужно определить, не равно ли значение *I* 1 или 2, возникает сильное побуждение написать

```
IF I = (1 OR 2)
```

что имеет эффект, в точности противоположный тому, который можно было бы ожидать. Приведенное выше выражение означает следующее: константы 1 и 2 будут преобразованы в строки битов, к которым будет применена операция OR (она выработает битовую строку, представляющую числовое значение 3), результат будет преобразован к типу переменной *I* и только затем выполнится сравнение. Оно выработает значение «истина», если значение *I* было равно 3, и «ложь», если *I* равнялось единице! Разумно предложение [3] — исключить операции И и ИЛИ и заменить их встроенными функциями ALL и ANY [1]). Тогда проверку *I* можно было бы переписать так:

```
IF I = ANY (1,2)
```

До языка АПЛ существовало общепринятое соглашение о выполнении выражений слева направо. В АПЛ выражение выполняется справа налево, так что значение 3—1+2 равно 0, а не 4. Приверженцы АПЛ скажут, что правило «справа налево» более естественно, но одно из оснований для этого соглашения — разрешить вложение присваиваний друг в друга, что само по себе является сомнительным приемом. Поскольку во многих других областях (например, письмо, математика и т. д.) используется правило «слева направо», последнее кажется менее провоцирующим ошибки, что подтверждается некоторыми экспериментами [3].

---

[1]) ALL — все, ANY — некоторые.— *Прим. перев.*

ОБНАРУЖЕНИЕ ОШИБОК ПРИ КОМПИЛЯЦИИ

В гл. 8 утверждалось, что профессиональный программист должен писать свою программу так, чтобы он мог рассчитывать на ее успешную компиляцию с первой попытки. Одна из причин этого кроется в том, что немногие компиляторы способны обнаружить все синтаксические ошибки. Всякая незамеченная синтаксическая ошибка становится логической ошибкой. Похоже, что создатели языков и разработчики компиляторов исходят из предположения, что программисты совершенны. Всякий раз, когда в описании языка используется фраза «должно быть», необходимо указать действия в случае, если это «должно быть» не имеет места. Однако такие указания встречаются редко. Разработчики компиляторов слишком часто свято соблюдают букву такого рода описаний, вследствие чего компилятор предпринимает неопределенные и непредсказуемые действия всякий раз, когда компилируемая программа нарушает требование типа «должно быть».

В качестве примера отметим, что в описании языка Фортран требуется, чтобы начальное значение параметра цикла DO было больше нуля. Это означает, что следующая программа должна рассматриваться как неправильная:

```
      L = 10
      DO 20 I = 0,L
   20 CONTINUE
```

Компилятор Фортрана уровня H фирмы IBM не обнаруживает такую ошибку, и в генерируемой им программе тело цикла выполнится один раз. С другой стороны, компилятор WATFOR сигнализирует об этом как об ошибке [14].

Наилучшее, что компилятор может сделать для надежности программного обеспечения,— обнаруживать все синтаксические ошибки. Проекты создания компиляторов выиграли бы, если бы в них была принята следующая цель разработки компилятора Ditran [15]: «Всякое условие, запрещенное описанием языка или реализацией, должно обнаруживаться и не должно приводить к неопределенным результатам». Не менее 56% ошибок периода выполнения в эксперименте Руби могли быть обнаружены компилятором, если бы тот строго проверял соблюдение всех спецификаций языка и если бы языки имели лучше защищающий от ошибок синтаксис и требовали полного и явного объявления, а также инициализации всех переменных [16].

Компилятор будет иметь все возможности обнаружить синтаксические ошибки, если спроектировать язык таким образом, что распространенные синтаксические ошибки превратят программу в синтаксически неправильную. Часто совершаемые синтаксиче-

ские ошибки не должны приводить к тому, что оператор опознается как синтаксически правильный, но имеющий другой смысл, отличный от того, который имел в виду программист (например, точка вместо запятой в примере с оператором в начале этой главы превратила его в допустимый оператор присваивания).

Компиляторы для профессиональных программистов (в отличие от программистов-любителей) никогда не должны пытаться автоматически исправлять синтаксические ошибки или просто оставлять без внимания ошибочный оператор, с тем чтобы сгенерировать объектную программу, которую программист может выполнить, если пожелает. Такие попытки исправления часто неправильны, к тому же они способствуют неряшливости при программировании. Следует также исключить многие возможности, которые в нынешних компиляторах предлагаются программисту на выбор. Например, программисту не нужно позволять не печатать части листинга или сообщения об ошибках, выдаваемых компилятором.

Оптимизирующие компиляторы хороши, но только не в случаях, когда они вносят в компилируемую программу ошибки. Например, компилятор Фортрана уровня Н фирмы IBM оптимизирует следующую программу, вынося вызов функции SQRT и помещая его перед оператором IF

```
      DO 11 I = 1,5
      DO 12 J = 1,5
      IF (B(I).LT.0) GO TO 11
   12 C(J) = SQRT(B(I))
   11 CONTINUE
```

Кроме того, многие ситуации, потенциально подлежащие оптимизации,— случаи, когда программист в действительности совершил ошибку. Например, оптимизирующий компилятор мог бы проверять, не совпадают ли между собой альтернативы THEN и ELSE оператора IF, собираясь в этом случае выбросить условие и одну из альтернатив, как, например, в следующем фрагменте:

```
IF A = B THEN A = A/2;
         ELSE A = A/2;
```

Однако это скорее ошибка в программе, а не кандидат для оптимизации. Хороший оптимизирующий компилятор мог бы выдавать сообщение: «Я оптимизирую этот участок программы, но он кажется мне подозрительным. Пожалуйста, проверьте, то ли это, чего вы хотели».

Хороший компилятор может также другими способами анализировать возможность ошибки в исходной программе. Например,

компилятор может обнаруживать использование переменной до инициализации. Можно распознавать случаи, когда переменной присваивают одно значение, а затем, не использовав первое, присваивают другое (хотя это не обязательно ошибка). Например, последовательность операторов, начинающаяся с присваивания X некоторого значения, заканчивающаяся другим присваиванием этой же переменной и не содержащая никаких ссылок на X и переходов ни из этой последовательности, ни внутрь ее,— такая последовательность операторов, вероятно, содержит ошибку.

Компиляторы могут обнаруживать и другие ошибки, если операторы объявления данных расширены так, чтобы в них указывались приемлемые значения переменных. Например, системный язык для проекта SUE допускает следующие объявления данных: [1])

```
TYPE angle=(0 TO 359);
TYPE direction=(north,east,south,west);
DECLARE angle a;
DECLARE angle b;
DECLARE direction movement;
```

Это позволяет компилятору обнаруживать недопустимые присваивания переменным *a*, *b* и *movement*. Конечно, компилятор не может обнаружить все неправильные присваивания (например, такие, в которых используются параметры, входные/выходные данные или сложные вычисления), так что такая проверка должна быть распространена на период выполнения.

Один из недостатков раздельно компилируемых модулей заключается в том, что компилятор не может обнаружить несогласованность сопряжений. Однако большинство вычислительных систем содержит средство (называемое редактором связей), которое строит программу из объектных модулей. Редактор связей должен проверять такие ошибки межмодульных связей, как несовпадение числа аргументов и соответствующих параметров или различие типов аргумента и соответствующего параметра.

ОБНАРУЖЕНИЕ ОШИБОК ПРИ ВЫПОЛНЕНИИ ПРОГРАММ

Сказанное ранее о «должно быть» применимо также и к периоду выполнения программы. Если описание языка утверждает «должно быть» относительно некоторого семантического условия, можно было бы ожидать, что всякое его нарушение будет обнаружено при выполнении программы, но так бывает редко. В большинстве язы-

---

[1]) Аналогичные возможности имеются в языке Паскаль, см. прим. на стр. 290. — *Прим. ред.*

ков утверждается, что индекс в ссылке на элемент массива не может выходить за границы массива, но случаи, когда это все-таки происходит, редко обнаруживаются и обычно приводят к непредсказуемым результатам. В Фортране утверждается, что значение индексной переменной в вычисляемом GO TO не может превосходить числа меток в этом операторе. Однако пять компиляторов Фортрана различных фирм — изготовителей ЭВМ порождают пять различных эффектов для следующей программы:

```
I = 5
GO TO (11,12,13,14),I
```

Один компилятор генерирует бессмысленный переход, другой— переход на оператор 14, еще один — переход на оператор, следующий за GO TO, четвертый — переход на оператор 11, и только пятый обнаруживает ошибку и прекращает выполнение программы.

Некоторые компиляторы генерируют дополнительные проверки ошибочных ситуаций, когда программа компилируется в «отладочном режиме», но убирают проверки при компиляции в «рабочем режиме» (из соображений «эффективности»). Как считают Д. Кнут и Ч. Хоар, это напоминает моряка, который носит спасательный жилет при тренировках на суше, но снимает его, отправляясь в море [12].

Очень полезно для обнаружения ошибок во время выполнения *утвердительное предложение*, которое может принимать, например, следующий вид:

```
ASSERT (0<INDEX<100,ORDER<ONHAND+INTRANS)
```

Это предложение может использоваться для того, чтобы заставить компилятор генерировать защитные фрагменты в начале каждого модуля и в других критических точках программы. Для тех случаев, когда утверждение несправедливо, может быть определена стандартная реакция на ошибку. Такие утверждения полезны также в качестве документации при изменениях или отладке программы [1]).

ЛИТЕРАТУРА

1. Clark B. L., Horning J. J. Reflections on a Language Designed to Write an Operating System, *SIGPLAN Notices*, 8(9), 52—56 (1973).

---

[1]) В связи с материалом этой главы полезно обратить внимание читателя на язык Ада, который в ближайшее время станет стандартом министерства обороны США, а возможно, и международным стандартом. По своим целям и качествам этот язык очень хорошо согласуется с положениями автора книги. См. SIGPLAN Notices, v, 14 (5), 1979, part A and B (1979). — *Прим. ред.*

2. Rubey R. J. et al. Comparative Evaluation of PL/I, ESD-TR-68-150, U. S. Air Force, Bedford, Mass., 1968.
3. Gannon J. D., Horning J. J. Language Design for Programming Reliability, *IEEE Transactions on Software Engineering*, SE-1 (2), 179—191(1975).
4. Nicholls J. E. Complexity and Duplicity in Programming Languages TR-12.101, IBM United Kingdom Laboratories, Hursley, England, 1972.
5. Clark B. L., Horning J. J. The System Language for Project SUE, *SIGPLAN Notices*, **6** (9), 79—88 (1971).
6. Hoare C. A. R. Hints on Programming Language Design, STAN-CS-73-403, Stanford University, 1973.
7. Clark B. L., Ham F. J. B. The Project SUE System Language Reference Manual, CSRG-42, Computer Systems Research Group, University of Toronto, 1974.
8. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, SC33-0009, IBM Corp., White Plains, N. Y., 1972.
9. Valentine S, H. Comparative Notes on ALGOL 68 and PL/I, *The Computer Journal*, **17** (4), 325—331 (1974).
10. Wulf W., Shaw M. Global Variable Considered Harmful, *SIGPLAN Notices*, 8 (2), 28—34 (1973).
11. Clark R. L. A Linguistic Contribution to GOTO-less Programming, *DATAMATION*, **19** (12), 62—63 (1973).
12. Knuth D. E. Structured Programming with go to Statements, *Computing Surveys*, **6** (4), 261—301 (1974).
13. Kelley R. A. APLGOL, an Experimental Structured Programming Language, *IBM Journal of Research and Development*, **17**(1), 69—73 (1973).
14. Siegel S. WATFOR,,, Speedy Fortran Debugger, *DATAMATION*, **17** (22), 22—26 (1971).
15. Moulton P. G., Muller M. E. DITRAN — A Compiler Emphasizing Diagnostics, *Communications of the ACM*, **10** (1), 45—52 (1967).
16. Kosy D. K. Approaches to Improved Program Validation Through Programming Language Design, in W. C. Hetzel, Ed., Program Test Methods, Englewood Cliffs, N. J.: Prentice-Hall, 1973, pp. 75—92.

ГЛАВА 16

Архитектура ЭВМ и надежность

Архитектура ЭВМ (на уровне машинных команд) влияет на надежность косвенно, но весьма существенно. В этой книге читатель неоднократно сталкивался с ситуациями, когда надежность и эффективность вступают в конфликт. Проектируя ЭВМ надлежащим образом, можно ослабить этот конфликт; языки программирования и программные системы могут разрабатываться с ориентацией прежде всего на надежность, а базовую ЭВМ можно тогда проектировать таким образом, чтобы программное обеспечение выполнялось на ней оптимально.

Прежде чем развивать эту мысль, заметим, что есть связь между аппаратурой и программным обеспечением, которая так проста и очевидна, что часто остается без внимания:

лучший способ повысить надежность программного обеспечения за счет развития аппаратуры — продолжать увеличивать ее скорость и снижать стоимость.

Дело в том, что в условиях противоречия между надежностью и скоростью ради скорости часто пренебрегают мерами по повышению надежности, например опуская защитные фрагменты, поскольку это замедляет программу, избегая высокой степени модульности из-за накладных расходов на выполнение оператора CALL, отказываясь на время выполнения от организуемых компилятором проверок из-за накладных расходов на них и используя запутанные программистские трюки ради того, чтобы сэкономить в программе одну-две миллисекунды. Если бы аппаратный процессор был бесконечно быстрым и стоил всего лишь несколько долларов, это противоречие было бы устранено, поскольку для беспокойства об эффективности почти не осталось бы оснований.

Технологическое прогнозирование — всегда дело рискованное, но имеющиеся прогнозы рисуют обнадеживающую картину будущего [1]. Видимо, вполне уверенно можно утверждать, что через 15 лет отношение стоимость/производительность для процессоров уменьшится в 100 раз. В качестве признаков этого можно отметить, что 1975 г.— год, когда появились центральные процессоры с себестоимостью ниже 100 долларов. В недалеком будущем процессоры

должны стать настолько дешевыми, что концепция мультипрограммирования (выполнения нескольких программ одним процессором), вероятно, исчезнет; вместо этого каждой выполняемой программе будет отдаваться один или несколько специализированных процессоров. Эта тенденция изменения отношения стоимость/производительность благотворно скажется на надежности программного обеспечения, сделав эффективность программы менее существенным фактором.

В прошлом конструкторы ЭВМ разрабатывали процессоры, как будто бы слабо или совсем не представляя себе, что такое программа, какие языки будут использоваться или как будут конструироваться компиляторы. Исследования показали, что многие компиляторы используют лишь небольшое подмножество полного набора команд машины. Короче говоря, большинство вычислительных систем разработано восходящим методом. Это привело к тому, что Гальярди называет *семантическим разрывом* [2]; данные и операции, обеспечиваемые машиной, редко бывают тесно связаны с данными и операциями языков программирования. Архитекторы ЭВМ начинают теперь понимать, что разрыв между языками высокого уровня и наборами машинных команд должен быть ликвидирован; архитектура машины должна ориентироваться на языки программирования. Это положительно сказывается не только на эффективности, но и на надежности программного обеспечения, как будет показано ниже. В качестве вывода можно сформулировать два следующих основных архитектурных принципа.

1. Архитектуру ЭВМ нельзя рассматривать изолированно. Первый шаг в проектировании вычислительной системы — выбор или проектирование языков программирования, подходящих для намеченных приложений. Второй шаг — проектирование архитектуры машины совместно с компиляторами и интерпретаторами, с учетом неизбежных компромиссных решений в распределении функций между аппаратурой и программным обеспечением. При этом действительно важным вопросом является не богатство набора команд машины или скорость выполнения какой-то отдельной команды, а связь архитектуры машины с языками и компиляторами.

2. Программирование в кодах машины или на языке ассемблера должно быть запрещено. В таком случае многие нынешние вопросы архитектуры ЭВМ, такие, как виртуальные машины и механизмы защиты памяти, оказываются почти (или вовсе) несущественными, потому что большинство проблем, которые пытаются решать с их помощью, никогда не возникнет, если все программы пишутся на языке высокого уровня. Если разрешено программирование на машинном языке, компромиссы между программным и аппаратным обеспечением, выбранные в соответствии с первым принципом,

сводятся на нет, потому что программист может их нарушить. Читатель, вероятно, скажет, что невозможно полностью защититься от программирования на машинном языке; нет — возможно, если такое программирование очень затруднено и если устранены мотивы, побуждающие его применять. Производители вычислительных машин вполне могут предотвратить программирование в кодах, относясь к нему так же, как многие из них сейчас относятся к микропрограммированию: никакой поддержки, никаких ассемблеров, заниматься этим можно только «на свой страх и риск».

В соответствии с первым принципом об архитектуре вычислительной машины ничего конкретного нельзя сказать прежде, чем будет рассмотрен общий проект системы, и в частности — языки программирования. Однако следующий тезис признается почти всеми: *машины фоннеймановского типа более нежелательны.*

Модель фон Неймана имеет две основные характерные особенности: во-первых, память ее представляет собой единое линейное последовательное запоминающее устройство и, во-вторых, машина не различает команды и данные. Последнее означает, что программа может модифицировать себя и что интерпретация слова памяти как элемента данных или как команды полностью зависит от выполняемой программы. Подавляющее большинство сегодняшних ЭВМ базируется на модели фон Неймана.

Для судьбы фоннеймановской модели важно прежде всего то, что большинство языков высокого уровня не нуждается в ней. Концепция дискретных именованных переменных устранила необходимость в последовательной памяти. Не предполагается, что программы на языке высокого уровня способны модифицировать себя. Есть четкое различие между данными программы (которые не могут быть выполнены) и операциями (на которые нельзя ссылаться, как на данные). Семантический разрыв между наборами команд и языками программирования во многом объясняется ориентацией архитектуры машин на модель фон Неймана и отказом от этой концепции в языках программирования. В следующих разделах рассматриваются методы ликвидации этого разрыва.

СТРУКТУРА ПАМЯТИ

В машине фон Неймана со словом памяти не связывается никакой явный смысл; смысл неявно привносится программой. Если в программе говорится «перейти по этому адресу», то машина интерпретирует слово по этому адресу как команду. Если в программе говорится «выполнить сложение с плавающей точкой над этими двумя словами», то машина интерпретирует эти слова в соответствии с форматом с плавающей точкой и выполняет над ними указанную операцию. Однако во многих языковых системах компилятор должен хранить и обрабатывать явные описания данных программы

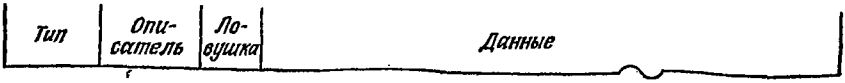


Рис. 16.1. Самоидентифицируемые данные.

для выполнения преобразований, контроля ошибок во время выполнения, символической отладки и символических дампов. Похоже, что разумным компромиссом было бы сконструировать такую машину, в которой было бы удобно обрабатывать эти описания, что приблизило бы структуру памяти машины к языку программирования.

Первый шаг в этом направлении — сделать так, чтобы каждое слово памяти содержало собственную спецификацию [3], как это изображено на рис. 16.1. Термин *слово* использован здесь для обозначения адресуемого элемента памяти. Как следует из рис. 16.1, слова не обязательно должны иметь фиксированную или единую для всех длину.

Как показано на рис. 16.1, слово памяти имеет четыре основные компоненты. (Учтите, пожалуйста, что этот формат используется для обсуждения общих принципов; это не какой-то конкретный проект машины). Одна из компонент слова, *данные*, имеет переменную длину и является значением этого слова. Когда программа ссылается на слово в памяти (например, при вычислении или для ввода-вывода), она видит только эту компоненту. Компонента *тип* указывает интерпретацию (тип данных) компоненты «данные». Эти типы должны соответствовать типам данных в языках программирования; примерами могут быть целое число, число с плавающей точкой, символьная строка, указатель, команда и модуль. Фьюстел [4] предлагает 32 возможных типа данных.

Компонента *описатель* задает длину компоненты «данные» и указывает, имеет ли она осмысленное значение (т. е. эта компонента является индикатором «пусто-непусто»). Если машина должна распознавать более сложные структуры данных (массивы, структуры или списки), то должна быть представлена также информация о соответствующей организации компоненты «данные».

Компонента *ловушка* вводит идею программно-ориентированного (в отличие от системно-ориентированного) прерывания. Эта компонента могла бы указывать, что при выборке или изменении этого слова программой или при его выполнении (если это слово — команда) управление должно быть передано специальной программе. Она используется средствами отладки, а также для трассировки и отображения адресов.

Выполняемая программа работает с компонентами «данные». Для запоминания и проверки трех других компонент имеются специальные команды машины, но они используются только компилятором или загрузчиком, инструментами отладки и программами

выдачи дампов. Для того чтобы обеспечить такие структуры, как строки команд в слове-модуле или неоднородные структуры, желательно иметь возможность *вкладывать* слова памяти друг в друга.

Прежде чем продолжать анализ этого проекта, рассмотрим некоторые его достоинства.

1. Все в памяти само себя описывает. Такие ошибки в программах, как попытка передать управление массиву или послать данные по адресу команды, выявливаются машиной. Легко обнаруживаются также попытки выполнить арифметические операции над данными несогласующихся типов или несоответствие между аргументами и параметрами.

2. Вместо того чтобы иметь отдельные машинные команды для каждого типа данных (например, логическое сложение, сложение с плавающей точкой, десятичное сложение), в машине имеются *обобщенные* команды (например, только одна команда сложения). Конкретный тип вычислений определяется компонентой «тип» операндов. Это упрощает процессы генерации кода в компиляторах и при соответствующем развитии языка программирования могло бы означать, что можно изменять представление данных программы без ее перекомпиляции (это аналогично концепции *независимости данных* в системах управления базами данных).

3. Становятся возможными осмысленные дампы памяти, поскольку программа выдачи дампов может представлять значение каждого слова в соответствии с его типом.

4. Концепция ловушки позволяет эффективно реализовать многие из рассматривавшихся в части 3 средств (например, трассировку и точки вклинивания для отладки).

Хотя большинство существующих ЭВМ соответствует модели фон Неймана, в некоторых машинах реализована идея самоидентифицируемых данных, хотя и не в такой степени, как рассматривалось выше. Например, слово машины B6500 фирмы «Барроуз» имеет 48-битовую компоненту данных и 3-битовую компоненту типа [5]. Каждое слово ЭВМ R-2 Rice Research имеет 54 бита данных, 4 бита типа и 2 интерпретируемых программой бита ловушки [6]. Каждое слово системы ISPL имеет три бита типа и бит ловушки [7].

Следующей принципиально важной частью структуры памяти является метод адресации слов. Термин «*указатель*» используется здесь для обозначения ссылки на слово памяти, но он не предполагает последовательной или линейной упорядоченности слов памяти. Формат указателя показан на рис. 16.2.

Адресная компонента указателя — имя или значение, которое говорит машине, на какое слово памяти происходит ссылка. Адрес здесь может быть именем сегмента памяти, числом или любой другой уникальной комбинацией битов. Биты адресной компоненты

Адрес	Полномочия	Смещение

Рис. 16.2. Содержание указателя.

не имеют абсолютно никакого смысла вне взаимодействия с конкретной машиной. Программы не могут ни создавать адреса, ни изменять их каким-либо образом. Когда программа требует, чтобы машина разместила в памяти слово, машина возвращает программе некоторый адрес. И это единственный способ породить адрес.

Указатель содержит также компоненту *полномочия*. Полномочия могут включать право читать и писать или право только читать при ссылке на данное слово. Машина позволяет программе копировать указатель и при желании устанавливать для копии более слабые права.

Третья часть указателя — компонента *смещение*. Она описывает способ ссылки на конкретную позицию внутри компоненты «данные» указываемого слова, как, например, на символ в строке символов или на элемент массива.

Указатель можно хранить в слове памяти или на регистре; в этом случае он имеет тип данных «указатель» и формат, изображенный на рис. 16.3.

Указатель — единственный механизм, необходимый для защиты памяти. Программа не может ссылаться на слово, если у нее нет на него указателя, а самостоятельная генерация указателей или манипуляции с ними запрещены машиной. Это означает, что программа (или модуль) может ссылаться на такое слово, которое она породила или на которое получила указатель от других программ. Отнесение прав к указателям, а не к самим данным, позволяет наделять разные программы или модули разными полномочиями. С помощью этого механизма легко реализовать усовершенствованный оператор CALL из гл. 15. Аргументы передаются модулю в виде списка указателей. Каждому указателю на аргумент, являющийся только входным, присваиваются права, позволяющие только читать.

Описанный механизм указателей имеет один серьезный недостаток. Ничто не мешает программе запомнить указатель на слово, которое было освобождено, и затем, намеренно или случайно, использовать этот указатель для ссылки на память. Такая ситуация демонстрирует недостатки архитектуры одновременно и в обнаружении ошибок, и в обеспечении безопасности данных. Одно из решений этой проблемы — проектировать машину таким образом, чтобы каждый создаваемый ею указатель был уникальным. Машина (за все время ее жизни) никогда не создает дважды одно и то же значение указателя (конечно, будет повторно использовано то же физическое пространство, но машина сама должна обеспечивать

Тип	Описатель	Ловушка	Адрес	Полномочия	Смещение

Рис. 16.3. Представление указателя в памяти.

отображение указателей в физические ячейки). Уникальность указателей может быть достигнута включением в адресное поле при назначении указателя отметки о времени в какой-либо форме. Она устраняет упомянутый ранее недостаток, позволяя машине обнаруживать все случаи применения вышедших из употребления указателей.

Идея связывать с адресами права была реализована в некоторых проектах ЭВМ. В системе ISPL каждый указатель содержит разрешение читать/писать [7]. В системе SYSTEM 250 Плесси [8] имеется концепция, названная *способностью* (capability),— она определяет для модуля указанный тип доступа к указанной области памяти.

СТРУКТУРА ПРОГРАММЫ

Второе вероятное направление усилий по ликвидации семантического разрыва между машинами и языками — это учет машиной структуры программы. В большинстве существующих ЭВМ не отражено никакого представления о программах; они соответствуют лишь идее простого «исполнителя команд». Например, единственный аспект архитектуры Системы 370 фирмы IBM, как-то связанный с концепцией программы,— примитивная команда перехода с запоминанием адреса возврата, применяемая для вызова подпрограммы.

Поскольку во многих современных системах модульность требует дополнительных затрат, сказывающихся на эффективности,

Область сохранения регистров программы
Адрес предыдущего блока в стеке
Адрес следующего блока в стеке
Адрес возврата в этот модуль
Область памяти, в которой размещаются локальные переменные и промежуточные результаты

Рис. 16.4. Блок активации.

а в предыдущих главах на модульность обращалось особое внимание, одно из возможных усовершенствований состоит в том, чтобы машина учитывала концепцию модуля и взяла на себя сложный процесс связывания модулей. Для этого нужны три новые команды машины: CALL, ACTIVATE и RETURN. Команды CALL и RETURN соответствуют операторам CALL и RETURN/END большинства языков программирования. Команда ACTIVATE соответствует операторам PROCEDURE, SUBROUTINE или ENTRY; она генерируется компилятором у каждого входа каждого модуля и в начале каждого блока в языках с блочной структурой. Все действия по сохранению состояния, передаче аргументов и распределению памяти будут выполняться машиной, а не генерируемыми компилятором связками.

Для иллюстрации этого подхода предположим, что машина при каждом вызове модуля сохраняет информацию, как это изображено на рис. 16.4. Это так называемый *блок активации*; он аналогичен программно обрабатываемым связям в системе MULTICS, называемым *стековой структурой* [9], с той разницей, что блоки активации обрабатываются аппаратурой и никогда не видны программному обеспечению. Блок активации содержит заголовок фиксированных размеров и тело переменной длины. В заголовке находится область сохранения регистров, адреса предыдущего и следующего блоков активации и адрес команды, на которую вызванный модуль должен вернуть управление. Тело содержит всю память, необходимую для локальных переменных и для запоминания промежуточных данных модуля.

Когда выполняется команда CALL, машина запоминает регистры и адрес возврата (обычно адрес команды, следующей за командой CALL) в текущем блоке активации модуля (рекурсивный модуль или модуль, выполняемый более чем одной программой, могут иметь несколько блоков активации). Первой командой вызванного модуля должна быть команда ACTIVATE. По этой команде размещается новый блок активации надлежащих размеров и связывается с предыдущим. Когда выполняется команда RETURN, то восстанавливаются значения регистров из предыдущего блока активации, текущий блок активации уничтожается и машина возобновляет выполнение с адреса возврата в предыдущий модуль.

Этот механизм удобен также и для обработки машинных прерываний. Когда возникает прерывание, машина имитирует команду CALL, сохраняя таким образом состояние прерванной программы. Для возобновления выполнения прерванной программы операционная система выдает команду RETURN.

Для некоторых особо важных систем реального времени необходимо иметь возможность осуществлять их сопровождение, не прерывая их функционирования. Нужен метод, позволяющий вставлять «отремонтированные» модули в программу, не загружая

и не инициализируя систему заново. Для этого необходимо, помимо прочего, исключить традиционный процесс «редактирования связей» (связывания модулей перед выполнением). Модули должны связываться динамически; преобразование имени модуля в указатель должно осуществляться при выполнении команды CALL. Динамическое связывание можно было бы делать программными средствами (например, с помощью LINK в OS/360), но, видимо, эффективность потребует аппаратного или микропрограммного решения. Для этого машинная команда CALL могла бы иметь два операнда: символическое имя вызываемого модуля и список указателей для аргументов. Машина сама обеспечивает соответствие между именами и адресами, так что она может динамически преобразовывать имя в адрес.

Этот механизм ликвидирует необходимость в редакторе связей, но по-прежнему не решает полностью проблему оперативной замены модулей. Например, невозможно заменить модуль, когда он выполняется, или в процессе вызова его другими модулями. Для замены модуля машине необходима команда «приостановить» модуль. Когда модуль приостановлен, все вызовы его временно перехватываются, а все его активации разрешено закончить. После этого новая версия модуля загружается в память, машина обновляет свою таблицу отображения имен в адреса и разрешается продолжить выполнение остановленных программ. (Конечно, могут существовать модули, постоянно активные при работе программы или системы; в этих случаях единственный выход — остановить всю систему).

СРЕДСТВА ОТЛАДКИ

Структура памяти, описанная выше, представляет эффективный механизм для реализации большинства распространенных средств отладки. Ссылка на слово, ловушка которого запрещает свободное чтение, вызовет выполнение заранее определенной программы, передачу ей адреса данных, имени модуля и номера (или смещения) команды, в которой выполнена эта ссылка, и отметки времени. Модификация слова с ловушкой, запрещающей запись, вызовет такие же действия, и, кроме того, этой программе будут переданы старое и новое значения компоненты «данные». Если установлен запрет исполнять конкретную команду, то при попытке выполнить ее программе передаются имя модуля, номер команды и время. Этот простой механизм предоставляет возможность устанавливать точки вклинивания при выполнении определенных команд и выборке или изменении определенного слова памяти, трассировать или регистрировать поток выполнения программы и обращения программы к данным. Кроме того, концепция данных, содержащих собственные описания, дает аппаратуре возможность

обнаруживать ссылки на переменные, начальные значения которых не определены, а также вычисления с несогласующимися типами данных, несоответствия между атрибутами аргументов и параметров и выходы за границы массивов. Программная система получает также возможность выдавать осмысленные дампы памяти.

Другое возможное аппаратное средство отладки — автоматическая проверка области изменения переменных. При размещении слова машине может быть сообщено его нижнее и верхнее граничные значения, что позволяет ей обнаруживать все выходы значений за область изменения. Поскольку, однако, это существенно увеличило бы необходимый объем памяти (в худшем случае в три раза) и сказалось бы на производительности машины (или потребовало бы дополнительной параллельно работающей аппаратуры), это предложение в известной степени проблематично и требует дальнейших исследований.

МАШИНЫ С ЯЗЫКОМ ВЫСОКОГО УРОВНЯ

Вероятно, крайняя мера в развитии архитектуры ЭВМ — полная ликвидация семантического разрыва за счет превращения языка высокого уровня в язык машины. Вместо традиционных машинных команд, таких, как загрузка и запоминание регистра, сложение регистров, командами машины могли бы быть операторы PL/1 DECLARE, присваивание, DO и т.д.

В этом подходе нет ничего невозможного, он был реализован в нескольких экспериментальных машинах. При этом исходная программа не компилируется в примитивные машинные команды, а интерпретируется и интерпретатор реализован микропрограммно, а не обычными программными средствами. На самом деле машина с языком высокого уровня не выполняет программу непосредственно в «сыром» исходном виде; считается, что машина имеет в качестве языка язык высокого уровня, если есть взаимно однозначное соответствие между командами машины и операторами языка высокого уровня (аналогично тому, как обычная машина может рассматриваться как машина с языком ассемблера). Как правило, имеется простой реализованный программно транслятор или «ассемблер» для обнаружения синтаксических ошибок, удаления пробелов, преобразования имен в адреса и перевода операторов в более удобную, например постфиксную, форму.

Наиболее широкоизвестной машиной с языком высокого уровня является, вероятно, SYMBOL 2R [10, 11]. Она была специально спроектирована для особого языка высокого уровня SPL, который представляет собой смесь характерных черт PL/1, Алгола и АПЛ. Машина SYMBOL не имеет операционной системы; все традиционные функции операционной системы выполняются машиной (т. е.

операционная система представляет собой аппаратную логическую схему).

Можно было бы спроектировать машину PL/1, и все было бы хорошо, если бы и операционная система была написана на PL/1, и все прикладные программы были бы программами на PL/1, но как быть с выполнением программ на Коболе или РПГ на такой машине? Эта проблема была решена в новаторской разработке — машине B1700 фирмы «Барроуз» [12]. Машинный язык в ней не зафиксирован. Когда операционная система назначает к выполнению программу на Коболе, эта машина преобразуется в машину с Коболом в качестве машинного языка. Когда назначается программа на Фортране, B1700 становится фортранной машиной. Когда выполняется операционная система, набором команд машины становится язык, на котором написана операционная система. Эти преобразования наборов команд выполняются подключением одного из многих микропрограммных интерпретаторов. Измерения показали, что выполнение программы в такой форме требует меньше места в памяти и более эффективно, чем выполнение скомпилированной программы на выходном языке низкого уровня на сравнимых машинах [13].

Читателю, который заинтересован в более детальном изучении концепции машины с языком высокого уровня, лучше всего начать с работы [14] [1]).

ЛИТЕРАТУРА

1. Turn R. Computers in the 1980s. New York: Columbia Univ. Press, 1974.
2. Gagliardi U. O. Software-Related Advances in Computer Hardware, Proceedings of a Symposium on the High Cost of Software. Menlo Park, Cal.: Stanford Research Institute, 1973, pp. 99—119.
3. Айлиф Дж. Принципы построения базовой машины.— М: Мир, 1973.
4. Feustel E. A. On the Advantages of Tagged Architecture, *IEEE Transactions on Computers*, C-22 (7), 644—656 (1973).
5. Burroughs B6500 Information Processing Systems Reference Manual. Burroughs Corp., Detroit, Mich., 1969.
6. Feustel E. A. The Rice Research Computer — A Tagged Architecture, Proceedings of the 1972 Spring Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 369—377.
7. Balzer R. M. An Overview of the ISPL Computer System Design, *Communications of the ACM*, **16** (2), 117—122 (1973).
8. Hamer-Hodges K. J. Fault Resistance and Recovery Within SYSTEM 250, Proceedings of the First International Conference on Computer Communications. New York: IEEE, 1972, pp. 290—296.

---

[1]) Многие из обсуждавшихся в этой главе идей, а также ряд новых, реализованы в МВК «Эльбрус». (См. Бурцев В. С. Принципы построения многомашинных вычислительных комплексов «Эльбрус». Препринт ИТМ и ВТ АН СССР, 1977, № 1, а также Бабаян Б. А. Основные принципы программного обеспечения МВК «Эльбрус». Препринт ИТМ и ВТ АН СССР, 1977, № 3).— *Прим. ред.*

9. Organick E. I. The MULTICS System: An Examination of Its Structure. Cambridge, Mass.: MIT Press, 1972.
10. Richards H., Wright C. Introduction to the SYMBOL 2R Programming Language, ACM-IEEE Symposium on High-Level-Language Computer Architecture. New York: IEEE, 1973, pp. 27—33.
11. Hutchinson P. C., Ethington K. Program Execution in the SYMBOL 2R Computer, ACM-IEEE Symposium on High-Level-Language Computer Architecture. New York: IEEE, 1973, pp. 20—25.
12. Wilner W. T. Design of the Burroughs B1700, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 489—497.
13. Wilner W. T. Burroughs B1700 Memory Utilization, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 579—586.
14. ACM-IEEE Symposium on High-Level-Language Computer Architecture, New York: IEEE, 1973.

ГЛАВА 17

Доказательство правильности программ

Вероятно, главным недостатком всех методов тестирования программного обеспечения является невозможность гарантировать отсутствие ошибок в программах. Как говорилось в гл. 10, исчерпывающее тестирование даже самых тривиальных программ — задача невыполнимая, и лучшее, на что можно надеяться в реалистических попытках тестирования,— найти высокий процент остающихся ошибок. Таким образом, возникает вопрос, есть ли какой-то метод, отличный от тестирования, способный гарантировать, что программа не содержит ошибок.

Ответ, основанный на данном ранее определении ошибки в программном обеспечении,— твердое «нет», и не представляется вероятным, чтобы он изменился в будущем. Есть, однако, школа, утверждающая, что «правильность» программы может быть доказана, если выразить намерения программиста в некоторой формальной логической системе, сформулировать математические теоремы о программе, используя это описание и исходный текст программы, и затем доказать теоремы. Если допустить, что это возможно (а так оно и есть), тогда, очевидно, центральной здесь становится проблема определения смысла слова «правильность». Если оно означает правильность по отношению к нуждам пользователей, тогда такой подход был бы исключительно ценным. Если оно означает правильность по отношению к внешним спецификациям, подход также был бы крайне полезным, хотя и не гарантировал бы отсутствие всех ошибок. Если «правильность» означает правильность по отношению к спецификациям низкого уровня, таким, как внешние спецификации модуля, подход по-прежнему заслуживает внимания, хотя и в меньшей степени.

Идея математического доказательства правильности программы в настоящее время концентрируется вокруг доказательства того, что подпрограмма правильна по отношению к спецификациям ее сопряжения. Теперь читатель должен уже понимать, что это не гарантирует отсутствия всех ошибок. Более того, как показано в последующих разделах, может быть и так, что программа, правильность которой по отношению к ее спецификациям доказана, выполняется не в соответствии с ними.

Теперь возникает вопрос: какова же польза такого доказательства, если оно не заменяет тестирования? Ответ прост: попытка математического доказательства правильности программы — замечательный способ обнаружения ошибок в программе, а всякий метод, который может раскрыть ошибки, обязательно имеет некоторую ценность. Более того, эта концепция уже сейчас существенно воздействует на многие аспекты разработки программного обеспечения, такие, как проектирование языка, методология проектирования, а то и просто заставляет программиста мыслить в более формальных и точных терминах.

Интересно, что по данному вопросу имеется больше литературы, чем по любому другому аспекту надежности программного обеспечения. Настоящая глава представляет собой только частичный обзор этой области; более полную картину можно найти в литературе, перечисленной в конце главы, а также в книгах и статьях, на которые там есть ссылки. В этих работах имеется много доказательств правильности программ, включая алгоритмы сортировки и поиска, простейшие компиляторы, подпрограммы численного анализа, ядра операционных систем и программы редактирования.

МЕТОД ИНДУКТИВНЫХ УТВЕРЖДЕНИЙ

Самый распространенный метод доказательства правильности программ — неформальный метод индуктивных утверждений, независимо введенный Флойдом [1] и Науром [2]. Он предполагает выявление ряда потенциальных теорем (часто называемых *условиями верификации*), доказательство которых подтверждает правильность программы.

Первый шаг — написать *утверждения* о входных условиях и выходных результатах программы. Эти утверждения обычно формулируются в некоторой формальной логической системе; одна из наиболее широко применяемых — исчисление предикатов первого порядка. В исчислении предикатов утверждения (или предложения) являются логическими переменными или выражениями, имеющими значение «истина» или «ложь». Подробное обсуждение исчисления предикатов выходит за рамки настоящей книги, но неформальные операции, необходимые в этом разделе, представлены в таблице на рис. 17.1. Для читателя, который желает изучить этот вопрос глубже, имеются учебники по математической логике или искусственному интеллекту, большинство из которых содержит полное и формальное описание исчисления предикатов первого порядка.

Прежде чем продолжать, рассмотрим пример, иллюстрирующий процесс доказательства. На рис. 17.2 приведена подпрограмма, вычисляющая значение $Z=A^B$, где B — неотрицательное целое число (этот пример взят из работы Кинга [3]). Вместо того чтобы

Связка	Пример	Значение
Дизъюнкция	$p \wedge q$	p и q
Конъюнкция	$p \vee q$	p или q
Отрицание	$\neg p$	не p
Импликация	$p \supset q$	p влечет q. То же, что $(\neg p) \vee q$
Эквивалентность	$p \equiv q$	p эквивалентно q. То же, что $(p \supset q) \wedge (q \supset p)$
Квантор всеобщности	$\forall x(p(x))$	Для всех значений x $p(x)$ истинно
Квантор существования	$\exists x(p(x))$	Существует значение x, для которого $p(x)$ истинно

Рис. 17.1. Связки исчисления предикатов.

просто перемножить A на себя B раз, подпрограмма выполняет возведение в степень, вычисляя последовательно A^{2n} и используя двоичное представление B для отбора нужных степеней. Каждый раз, когда в цикле вычисляется очередная степень A (A, A^2, A^4, A^8 и т. д.) и соответствующий бит B равен 1, степень A входит в результат как множитель. Например, если значение B равно 13 (в двоичном коде 1101), Z вычисляется как произведение $A^8 \times A^4 \times \times A^1$. Программа выполняет для этого семь умножений вместо 13 (есть и другие умножения и деления, но вторым операндом здесь служит двойка, так что они компилируются в простые команды сдвига вправо или влево). Оператор Y=Y/2 сдвигает показатель вправо на один бит, а оператор IF проверяет, не равен ли самый правый бит единице. Поскольку алгоритм довольно хитрый (почему и требуется доказать его правильность), читателю, прежде чем двигаться дальше, полезно проследить за его работой на простом примере.

Чтобы нагляднее показать пути выполнения этой программы, на рис. 17.3 приведена ее блок-схема. Входное и выходное утверждения относятся соответственно к точкам A1 и A2. Входное утверждение описывает все необходимые входные условия для программы, а выходное утверждение описывает ожидаемый результат. Эти ут-

```
POWER: PROCEDURE (A,B,Z)
Z = 1
X = A
Y = B
DO WHILE (Y≠0)
   IF (Y≠(2*(Y/2))) THEN Z = Z*X   IS Y AN ODD NUMBER?
   Y = Y/2
   X = X*X
END
END
```

Рис. 17.2. Программа.

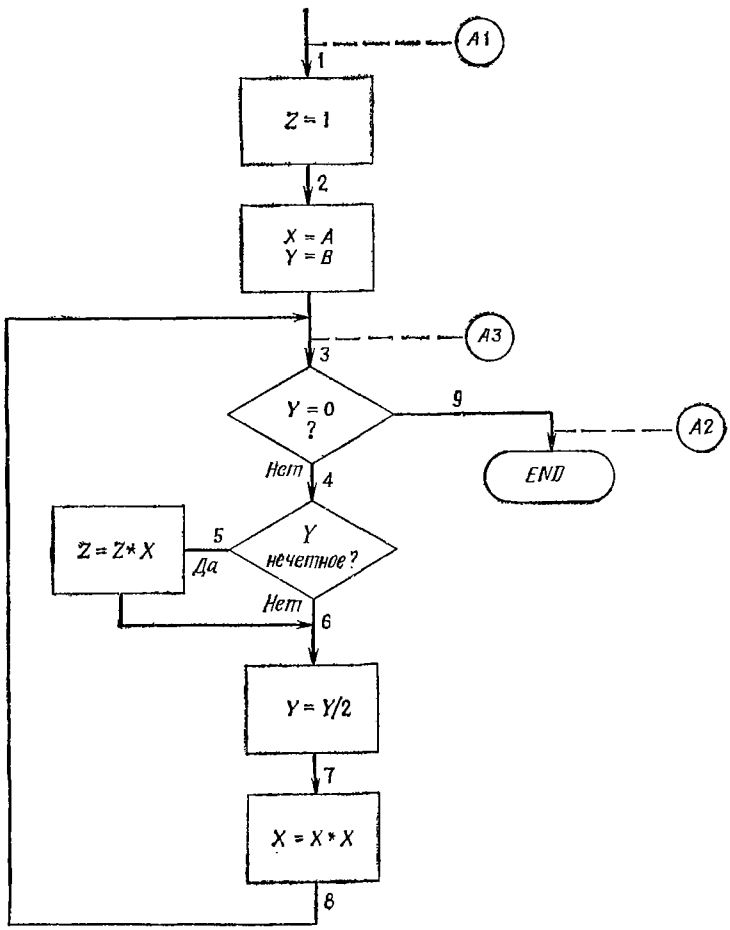


Рис. 17.3. Блок-схема программы.

верждения можно представить следующими предложениями:

A1: $(B \geqslant 0) \wedge (B$ целое$)$
A2: $(Z = A^B) \wedge (A,\ B$ не изменены$)$

На втором шаге этого метода доказательства нужно сформулировать теоремы, или условия верификации, подлежащие доказательству. Первый способ сделать это — предположить, что A1 истинно в точке 1, воспользоваться семантикой оператора, расположенного в программе между точками 1 и 2 для преобразования A1 в промежуточное утверждение в точке 2, и продолжать так до тех пор, пока промежуточное утверждение (выведенное из исходного A1) не будет построено для точки 9. Для этой точки построено два утверждения: первое — выведенное утверждение (входное утверждение, преобразованное в соответствии с семантикой программы), а второе — A2, т. е. выходное утверждение. Теперь можно сформулировать теорему о том, что из выведенного в точке 9 утверждения логически следует A2.

Второй способ — начать с выходного утверждения и двигаться по программе в *обратном направлении* до тех пор, пока не получится выведенное утверждение для точки 1. Теорема (условие верификации) теперь формулируется так: из A1 следует утверждение, выведенное для точки 1.

Чтобы сформулировать условия верификации, необходимо аксиоматическое описание семантики языка программирования. Гипотетический язык, на котором написана изображенная на рис. 17.2 программа, сводится к операторам двух типов (присваивание и принятие решения), так что нужно описать, как должны изменяться утверждения при рассмотрении этих операторов. Предпочтительнее *обратный* метод формулировки условий верификации, поскольку он дает новую точку зрения на выполнение программы (можно считать, что он моделирует выполнение программы в обратном направлении). Рис. 17.4 иллюстрирует соответствующие аксиомы. Эти аксиомы должны отвечать на такой вопрос: если некоторое утверждение истинно после оператора программы, какое утверждение должно быть истинным перед оператором? Если утверждение в точке 2 на рис. 17.4 таково: $(S = X + 4) \wedge (X \geqslant 0)$, а оператор присваивания имеет вид $X = X - 2$, то в точке 1 будет выведено утверждение $(S = X + 2) \wedge (X \geqslant 2)$. Другими словами, вы «отменяете» присваивание, заменяя в утверждении все вхождения переменной, указанной слева от знака присваивания, выражением справа. Чтобы «отменить» принятие решения, применяется вторая аксиома, изображенная на рис. 17.4. Если в точке 5 имеем утверждение $X = 0.5\ MV^2$, а условие g: $V > 1000$, то утверждение в точке 3 будет выглядеть так: $(V > 1000) \supset (X = 0.5\ MV^2)$.

Здесь может показаться, что у нас достаточно информации для построения условия верификации, но остается одна проблема:

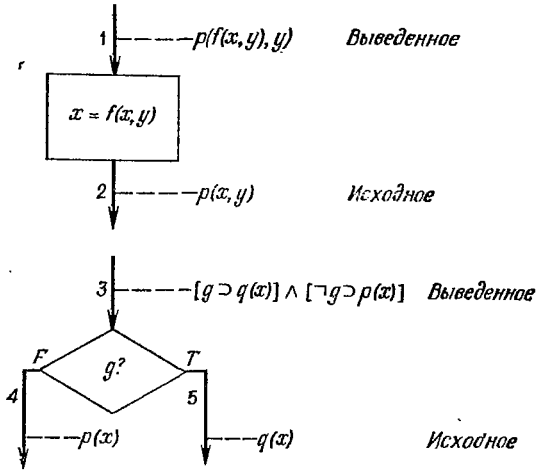


Рис. 17.4. Аксиомы преобразования утверждений при обратном просмотре.

как быть с циклом? Число повторений цикла может быть различным в зависимости от значения B. Поэтому нам необходимо третье утверждение для точки внутри цикла (называемое *индуктивным* утверждением или *инвариантом цикла*), чтобы мы могли применить для доказательства правильности метод математической индукции. Индуктивное утверждение должно описывать все свойства программы, которые инвариантны для выбранной точки в цикле. После тщательного изучения программы читатель должен суметь доказать, что следующее утверждение истинно в точке 3 на рис. 17.3.

$$\text{A3: } (Z*X^Y=A^B)\wedge(Y\geqslant 0)\wedge(Y \text{ — целое})\wedge(A,\ B \text{ не изменены})$$

Сложность с формулировкой индуктивных утверждений в том, что нужно описать все осмысленные инвариантные условия. Общепризнанно, что это самый трудный шаг в процессе доказательства. Обычно имеется тенденция опускать некоторые инвариантные свойства, но это легко обнаруживается при попытке доказать условие верификации. Определение индуктивных утверждений оказывается полезным даже тогда, когда вы не собираетесь доказывать правильность программы, потому что оно заставляет формализовать интуитивные представления о логике цикла.

Теперь у нас имеется три утверждения (или больше, если было несколько циклов). Вместо того чтобы строить одно подлежащее доказательству условие верификации, мы построим несколько таких условий следующим образом. Рассмотрим граф управления программы и совокупность утверждений и выделим в программе

все пути, которые начинаются и заканчиваются утверждениями и не содержат утверждений внутри. Такие пути называются *выделенными путями*, на рис. 17.3 их четыре

1. От А1 к А3
2. От А3 к А2.
3. От А3 к А3 для Y нечетного.
4. От А3 к А3 для Y четного.

Для каждого выделенного пути должны быть построены условия верификации; их доказательство подтверждает правильность программы. Для построения условия верификации следует взять утверждение в конце выделенного пути и трансформировать его, возвращаясь к началу. Чтобы детальнее познакомиться с этим процессом, построим условия верификации для третьего выделенного пути. Утверждение в точке 8 совпадает с утверждением в точке 3 (которым является А3), поскольку между точкой 8 и точкой 3 нет никаких операторов. В точке 7, заменяя X на $X*X$, получаем

$$(Z*X^{2Y}=A^B)\wedge(Y\geqslant 0)\wedge(Y\text{ — целое})\wedge(A,\ B\text{ не изменены})$$

В точке 6, заменяя Y на $Y/2$, имеем

$$(Z*X^{2\,(Y/2)}=A^B)\wedge(Y/2\geqslant 0)\wedge(Y/2\text{ — целое})\wedge(A,\ B\text{ не изменены})$$

Двойки в показателе степени не сокращаются, потому что целое деление в машине — не то же самое, что деление в математике ($2*(Y/2)$ не всегда равно Y). В точке 5 выведенное утверждение получает вид

$$(Z*X^{1+2\,(Y/2)}=A^B)\wedge(Y/2\geqslant 0)\wedge(Y/2\text{ — целое})\wedge(A,\ B\text{ не изменены})$$

В точке 4 имеем

$$(Y\text{ нечетно})\supset[\text{утверждение для точки 5}]$$

а в точке 3 —

$$(Y\neq 0)\supset\ \{(Y\text{ нечетно})\supset[\text{утверждение для точки 5}]\}$$

Итак, мы получили условие верификации, согласно которому утверждение А3 влечет за собой выписанное выше утверждение. Три других условия верификации могут быть построены аналогично; все четыре условия приведены на рис. 17.5.

Следующий шаг — доказать эти четыре утверждения. Для этого мы должны воспользоваться правилами вывода в исчислении предикатов, но обычно достаточно неформальных рассуждений. Все четыре теоремы имеют вид $p\supset q$, а обычный метод доказательства таких теорем сводится к тому, чтобы показать, что q истинно всякий раз, когда истинно p (можно также пытаться доказать, что q всегда истинно или что p всегда ложно). Читатель не должен встре-

1. $[(B \geqslant 0) \wedge (B \text{ целое})] \supset$
$[(1 * A^B = A^B) \wedge (B \geqslant 0) \wedge (B \text{ целое}) \wedge (A,\ B \text{ не изменены})]$

2. $[(Z * X^Y = A^B) \wedge (Y \geqslant 0) \wedge (Y \text{ целое}) \wedge (A,\ B \text{ не изменены})] \supset$
$\{(Y = 0) \supset [(Z = A^B) \wedge (A,\ B \text{ не изменены})]\}$

3. $[(Z * X^Y = A^B) \wedge (Y \geqslant 0) \wedge (Y \text{ целое}) \wedge (A,\ B \text{ не изменены})] \supset$
$\{(Y \neq 0) \supset \{(Y \text{ четно}) \supset [(Z * X^{1+2(Y/2)} = A^B) \wedge (Y/2 \geqslant 0) \wedge$
$(Y/2 \text{ целое}) \wedge (A,\ B \text{ не изменены})]\}\}$

4. $[(Z * X^Y = A^B) \wedge (Y \geqslant 0) \wedge (Y \text{ целое}) \wedge (A,\ B \text{ не изменены})] \supset$
$\{(Y \neq 0) \supset \{(Y \text{ четно}) \supset [(Z * X^{2(Y/2)} = A^B) \wedge (Y/2 \geqslant 0) \wedge$
$(Y/2 \text{ целое})\ (A,\ B \text{ не изменены})]\}\}$

Рис. 17.5. Четыре условия верификации, подлежащие доказательству.

тить затруднений, рассматривая два первых условия верификации. Два последних доказательства хитрее, поэтому поясним доказательство третьего условия. Утверждения для $Y/2$, A и B должны быть очевидны, поэтому нам остается показать, что $Z * X^Y = A^B$ влечет за собой $Z * X^{1+2(Y/2)} = A^B$, когда Y нечетно. Примем, что в языке (как в большинстве других языков) округление до целого производится с недостатком, если результат деления — дробный (например, 5/2 равно 2). Поэтому $2(Y/2)$ равно $Y-1$, поскольку Y — нечетно, и доказательство закончено ($Y = 1 + Y - 1$). Четвертое доказательство можно провести аналогично.

Мы показали, что программа правильна по отношению к входному и выходному утверждениям, но только если ее выполнение заканчивается. Последнее (т. е. что программа не будет циклиться бесконечно) еще не доказано. Нет общего метода доказательства завершаемости программ, но обычно бывает достаточно неформальных соображений. В нашем примере программа заканчивается только тогда, когда Y равно нулю. Согласно входному утверждению, значение Y первоначально неотрицательно, и изменяется только в одном месте: в цикле, где уменьшается по крайней мере на единицу при каждом его повторении. Таким образом, неформально устанавливается, что выполнение этой программы закончится.

Подводя итоги, можно сформулировать следующие шаги неформального доказательства методом индуктивных утверждений:

1. Постройте блок-схему программы (не обязательно, но это поможет выделять пути).
2. Выпишите входное и выходное утверждения.
3. Найдите все циклы и сформулируйте индуктивные утверждения для каждого из них.

4. Составьте список выделенных путей.

5. Используя семантику операторов программы, постройте условия верификации.

9. Докажите каждое условие верификации. Если это не удается, причиной может быть одно из трех обстоятельств: либо вы недостаточно изобретательны, либо одно из ваших утверждений неполно, либо вы обнаружили ошибку на этом выделенном пути.

7. Докажите, что выполнение программы закончится.

Скептики видят в этом методе доказательства лишь причудливую маскировку обычных процессов чтения текста программы или сквозного контроля. В значительной мере так оно и есть, единственное различие — степень формализации. Но уровень формализации часто остается вопросом личного вкуса; например, в работе [4] Лондон доказывает правильность пяти программ более неформально и бегло, чем в приведенном примере.

Может показаться, что этот метод доказательства рассчитан на численные программы. В некоторой степени это верно, но при определенной изобретательности его можно применять и к широкому классу нечисленных программ. Например, выходное утверждение для программы сортировки элементов вектора M могло бы состоять в том, что для всех I: $M(I) \leqslant M(I+1)$ и что окончательное значение M — перестановка исходного значения M. Для доказательства правильности алгоритма поиска в таблице T индекса I того элемента, значение которого равно X, выходное утверждение могло бы быть таким:

$$[(X \in T) \supset X = T(I)] \wedge [(\neg(X \in T)) \supset I = 0] \wedge [T \text{ не изменено}]$$

Читатель, вероятно, понимает, что доказательство правильности — длинный процесс. Доказательство всегда гораздо длиннее самой программы, и самые большие программы, за которые пока брались, содержат не более нескольких сотен операторов. В одном из следующих разделов рассматриваются некоторые методы автоматизации шагов доказательства, разрабатываемые для того, чтобы сократить время, необходимое для выполнения этого процесса.

Описанный в данном разделе метод доказательства может быть распространен на всю программу, если доказывать ее модуль за модулем. При этом первый шаг — определение входных и выходных утверждений для каждого модуля. Когда при построении условий верификации внутри модуля встречается оператор CALL вызова другого модуля, утверждения, касающиеся вызываемого модуля, используются для определения семантики оператора CALL.

ЧТО МОЖНО И ЧЕГО НЕЛЬЗЯ ДОКАЗАТЬ С ПОМОЩЬЮ ДОКАЗАТЕЛЬСТВ

Как уже говорилось выше, доказательство правильности не может гарантировать отсутствия ошибок в программном обеспечении. Бывали случаи, когда в программах, правильность которых была доказана, обнаруживались ошибки. Последнее, однако, не помешало исследователям, работающим в данной области, провозгласить этот метод «заменой тестирования» и утверждать, что с его помощью можно «окончательно доказать отсутствие ошибок в программе». Чтобы оценить перспективы такого метода, рассмотрим проблемы, недостатки и достоинства математического доказательства правильности программ. Пока число недостатков превышает число достоинств, но это не означает, что методы доказательства бесполезны. Многие (но не все) недостатки будут устранены благодаря ведущимся в настоящее время исследованиям.

Основные недостатки метода — в том, что программы, правильность которых доказана, могут все же содержать ошибки; кроме того, доказательства сложны, утомительны и требуют много времени, а многие конструкции широко распространенных языков программирования не поддаются доказательству. Ниже приводится более детальный список имеющихся трудностей.

1. Правильность программы доказывается только по отношению к входному и выходному утверждениям. Если выходное утверждение неправильно или неполно, может быть доказана правильность программы, а ошибки в ней останутся необнаруженными. Кроме того, ошибки, сделанные на более ранних этапах проектирования (цели, внешний проект), не будут обнаружены при доказательстве.

2. Входное утверждение описывает исходные предположения и среду, в которой работает программа. Нет гарантии, что они будут справедливы в момент, когда программа действительно будет выполняться (например, доказательство ничего не говорит о поведении программы в случае, если реальные входные данные не удовлетворяют входному утверждению).

3. Ошибки в сопряжении (например, порядок параметров не соответствует порядку аргументов или аргумент в радианах передается параметру в градусах) обычно не обнаруживаются при доказательстве.

4. Само доказательство может быть неправильным. Человек, доказывающий условие верификации, легко может допустить ошибку и показать его истинность, в то время как на самом деле это не так. Может быть, это исключительный случай, но в профессиональном журнале [5] была опубликована программа из 25 строк, правильность которой была доказана и в которой после этого было обнаружено 7 ошибок. Все семь ошибок были бы, вероятно, найдены, если бы эта программа была тестирована.

5. Вероятно, самым большим недостатком является то обстоятельство, что легко можно неправильно интерпретировать семантику языка. Конкретный оператор программы может иметь совсем не тот смысл, который подразумевается в доказательстве. Если человек допускает ошибку, неправильно понимая оператор при кодировании программы, та же ошибка, вероятно, будет сделана и при ее доказательстве.

6. Системные и машинные ограничения (ошибки округления, условия переполнения и другие следствия ограниченности машинной арифметики) обычно опускаются в доказательствах. Есть способы учесть эти ограничения, но это в огромной степени усложняет доказательство.

7. Современные методы доказательства неадекватны в применении к программам, выполняющим операции ввода-вывода (например, программы печати отчетов или обновления базы данных). Как отмечалось в гл. 15, значительный процент ошибок в программном обеспечении (45%, согласно исследованию Руби) связан с операциями ввода-вывода и определением данных.

8. Доказательства правильности нечисленных задач гораздо сложнее, чем доказательства правильности программ числовых расчетов. Например, доказательство правильности небольшой подпрограммы MATCHES из гл. 8 было бы чрезвычайно сложным.

9. Доказательства правильности не обнаруживают нежелательных побочных эффектов или посторонних результатов. «Правильная» программа может все же иметь ошибочный побочный эффект, например разрушение глобальной области данных.

10. Еще не разработаны методы доказательства, позволяющие работать с такими языковыми средствами, как параллелизм, сложные структуры данных, глобальные переменные, правила определения области действия имен.

11. Сложность доказательства пропорциональна богатству и сложности языка программирования. Доказательство правильности типичной программы на PL/1 было бы почти неосуществимо из-за тысяч типов данных и соответствующих тонких преобразований в PL/1.

Несмотря на все эти проблемы, математические доказательства имеют некоторые существенные достоинства. При доказательстве могут быть обнаружены новые ошибки, и притом такие, которые, возможно, не удалось бы найти методами тестирования. Рассмотрим программу, в которой равенство трех переменных проверяется с помощью выражения IF (A=(A+B+C)/3). Для некоторых значений B и C это выражение не вырабатывает нужный результат. Такая ошибка, вероятно, не была бы обнаружена обычными методами тестирования, поскольку она проявляется только для небольшой доли всех возможных значений переменных. Возможно, что

она не была бы обнаружена и при неформальном чтении текста или структурном контроле. Однако очень вероятно, что при попытке доказать правильность программы эта ошибка была бы найдена.

Попытки доказать правильность заставляют программиста очень детально исследовать и программу, и спецификации, а также формализовать свое понимание программы. Процесс подготовки входных и выходных утверждений (даже если программист не имеет в виду их доказывать) полезен и сам по себе как метод формального выражения спецификаций и способ определения необходимых защитных фрагментов в модуле (т. е. фрагментов, выполняющих сравнение условий на входе с входным утверждением).

Доказательства правильности программ приносят и косвенную пользу. Поскольку полнота спецификаций становится необходимым исходным требованием, программист, если он собирается в дальнейшем доказывать правильность программы, вынужден будет писать свои спецификации более точно. Благодаря использованию формализма общее стремление к точности программирования часто усиливается, а это благотворно влияет также на стиль и структуру программы. Математические доказательства уже оказывают влияние на разработку языков программирования, помогая понять семантику языка и разрабатывать более ясные языковые конструкции.

Поскольку методы доказательств все еще далеки от совершенства, а также из-за перечисленных выше проблем, этот метод нельзя рекомендовать для употребления в большинстве современных проектов. В тех случаях, когда он применяется, его можно использовать вместо процессов структурного контроля и автономного тестирования, но он не может заменить остальные процессы тестирования, такие, как тестирование сборки, внешних функций и комплексное тестирование, а также тестирование приемлемости. Доказательства, однако, могут и сегодня представлять ценность в определенных критических ситуациях, когда в разработку проекта возможно вложить дополнительные затраты ради повышения достоверности результата. Если бы я разрабатывал операционную систему, содержащую несколько модулей, жизненно важных для работы системы, я бы рассмотрел возможность доказательства правильности этих модулей в дополнение к обычным методам тестирования. То же самое можно сказать о прикладной системе, такой, как система энергоснабжения, с критическими модулями, вычисляющими счета или реализующими алгоритм управления распределением энергии между компаниями.

Для читателя, намеревающегося применять метод доказательства правильности программ, можно следующим образом развить одну из аксиом тестирования из гл. 10: не следует пытаться доказывать правильность программы, которую написали вы сами. Это связано с пятой из обсуждавшихся проблем; программист, сделавший ошибку при составлении программы, вероятно, сделает эту же

ошибку при доказательстве. Кто-то иной, а не сам программист, должен указать входное и выходное утверждения, формулировать и доказывать условия верификации. Однако, так как индуктивные утверждения требуют глубокого понимания логики программы, представляется разумным, чтобы программист сам формулировал индуктивные утверждения и передавал их тому, кто выполняет доказательство.

ФОРМАЛЬНЫЕ И АВТОМАТИЧЕСКИЕ ДОКАЗАТЕЛЬСТВА

Опираясь на формальный метод индуктивных утверждений, представленный в предыдущем разделе, можно было бы сделать следующий логический шаг: формализовать его настолько, чтобы можно было применять его с помощью программы, а не вручную. Другими словами, можно ли написать программу, которая получала бы на входе другую программу, а также ее входные и выходные утверждения и выдавала бы на выходе необходимые доказательства (или контрпример)? Ответ: и да, и нет. В этом направлении проводились плодотворные исследования, но результаты не находят практического применения сегодня и вряд ли найдут его в близком будущем.

Большинство подходов к этой проблеме начинает с требования, чтобы программист задал входное, выходное и индуктивные утверждения в языке исчисления предикатов первого порядка (может оказаться возможной генерация индуктивных утверждений и самой системой). Синтаксис и семантика языка программирования хранятся в такой системе в виде аксиом на языке исчисления предикатов. Система построения доказательств определяет выделенные пути в исследуемой программе и затем использует саму программу, утверждения и аксиомы для генерации условий верификации. Затем она использует правила вывода исчисления предикатов в сочетании с дедуктивными и эвристическими принципами рассуждения, пытаясь доказать каждое условие верификации. Заметьте, что существование такой системы сняло бы многие из рассматривавшихся ранее проблем, например ошибки человека при интерпретации семантики языка и доказательстве условий верификации.

Самая основная компонента доказывающей системы — генератор условий верификации. Он должен уметь манипулировать логическими предикатами, упрощать их, интерпретировать операторы программы, но самая сложная задача — снабдить систему необходимыми аксиомами. Не только все детали синтаксиса и семантики, но и множество банальных принципов, таких, как коммутативный и дистрибутивный законы для сложения, должны быть описаны в виде аксиом. Несколько генераторов условий верификации было написано для языков блок-схем и подмножеств АПЛ, Алгола и

Фортрана. Однако описание набором аксиом сложного языка вроде Кобола или PL/1 — необъятная задача, которая, вероятно, не будет решена в этом столетии.

Вторая компонента — система «механического» доказательства теорем. Такая система относится к области искусственного интеллекта, поскольку человек при доказательстве теорем использует дедуктивные приемы рассуждения и методы определения целей. Доказывающая система могла бы действовать методом проб и ошибок, если бы располагала неограниченными временем и памятью, но это, конечно, практически неприемлемо. Принцип резолюций Робинсона [6] теоретически может существенно улучшить процесс доказательства теорем, но даже это слишком медленно, когда число аксиом велико (что обязательно имело бы место в реальных языках программирования).

Классическая работа в этой области — верификатор программ Дж. Кинга [7], который, кажется, является первой реализацией полностью автоматизированного генератора условий верификации и системы доказательства теорем. Система доказывает правильность программ, написанных на очень маленьком подмножестве языка Алгол, она была использована для доказательства правильности нескольких небольших (менее 100 операторов) программ. Хотя эта работа ознаменовала собой определенный этап, она далека от того, чтобы иметь какое-либо практическое применение.

Ведется работа над многими исследовательскими проектами со сходными целями; одна из систем, заслуживающих упоминания,— система Д. Гуда, Р. Лондона и У. Бледсоу [8]. Эта система пытается строить и доказывать условия верификации для программ на языке Паскаль. Следует отметить такую ее особенность: человек может взаимодействовать с системой в процессе доказательства теорем. Устанавливается лимит времени доказательства условий верификации. Если система не может завершить доказательство за это время, она останавливается и спрашивает совета у пользователя. Пользователь может просмотреть незаконченное доказательство и предоставить дополнительную информацию или направить рассуждения системы по другому руслу.

Эта идея диалога в процессе доказательства теорем представляется необходимой для того, чтобы обеспечить доказывающим системам какую-то возможность практического применения в этом столетии. Полностью автоматические доказательства реальных программ, написанных на распространенных языках программирования,— цель, все еще недостижимая в обозримом будущем.

В первом разделе отмечалось, что формулировка индуктивных утверждений внутри каждого цикла — это, вероятно, самое тяжелое бремя в доказательстве правильности программы. Несколько исследовательских работ было посвящено попытке автоматизировать генерацию индуктивных утверждений. Примером может служить

система VISTA [9]. Имея исходную программу (на языке блок-схем), а также входное и выходное утверждения, VISTA генерирует или уточняет некоторые инварианты циклов. Делается это различными приемами. Система исследует семантику каждого цикла, чтобы выявить определенные инвариантные условия. Она также переносит другие утверждения внутрь цикла и добавляет их к инвариантам цикла. Заключительный шаг — генерация условий верификации и попытка доказать их. Если доказать не удается, VISTA исследует причины неудачи, надлежащим образом подправляет нужное инвариантное утверждение и снова пытается доказывать. Главная трудность этого подхода состоит, однако, в том, что для правильной работы генератора утверждений необходимо, чтобы программа с самого начала была правильной.

Вполне осуществимым и уже сегодня практичным автоматизированным средством является система *проверки доказательств*. Если ей сообщить шаги доказательства, выполненного человеком, она может проверить соответствие каждого шага известным ей правилам вывода. Таким образом она может искать ошибки в подготовленных человеком доказательствах. Однако остальные доказывающие системы (за исключением систем проверки доказательств) еще долгое время вряд ли будут применимы к реальным программам.

ДРУГИЕ ПРИМЕНЕНИЯ МЕТОДОВ ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ

От основной идеи математического доказательства правильности отпочковалось множество новаторских работ в самых различных направлениях. Одним из них является направление, связанное с концепцией *символического выполнения* [10] — методом, промежуточным между традиционным тестированием и доказательством правильности. При символическом выполнении программа действительно в некотором смысле «выполняется», но при этом каждый оператор интерпретируется как преобразование символов, обозначающих реальные входные данные. Например, при символическом выполнении программы, изображенной на рис. 17.6, после выполнения первого оператора переменной S будет присвоено алгебраическое выражение B+C*A, после оператора 2 переменная T будет иметь значение C*A—B, а при возврате из процедуры переменной D присвоено выражение 2*B. Таким образом, программист теперь имеет символическое представление работы подпрограммы.

Одна из проблем, немедленно возникающих при символическом выполнении,— как обрабатывать условные операторы, например оператор IF, посредством которого в зависимости от *конкретного значения* одной или нескольких переменных выбирается

```
        CALL CALC(A,B,C,D);
          .
          .
          .
1       CALC: PROCEDURE(W,X,Y,Z);
2       S = X + Y*W;
3       T = S - 2*X;
4       Z = S - T;
5       END;
```

Рис. 17.6. Исходная программа для символического выполнения.

один из двух путей. Решение состоит в том, чтобы выбирать оба пути. Например, если встречается оператор

```
IF P>N THEN DO...
          ELSE DO...
```

то выполнение продолжается по обоим путям, поскольку нет способа узнать, истинно или ложно отношение P>N. При обработке альтернативы THEN сохраняется символическое состояние всех переменных, а к состоянию программы добавляется условие P>N. При обработке альтернативы ELSE текущее состояние всех переменных также сохраняется и к состоянию программы добавляется условие ¬(P>N). Как нетрудно видеть, процесс символического выполнения почти идентичен процессу генерации условий верификации.

EFFIGY [11] — это диалоговая система символического выполнения программ, написанных на узком подмножестве PL/1. Когда встречается развилка, EFFIGY запоминает текущее состояние программы и запрашивает пользователя, какую ветвь исследовать. После выполнения ветви пользователь может восстановить состояние системы и исследовать другое направление. EFFIGY предоставляет также ряд возможностей для символической отладки, таких, как трассировка и точки вклинивания.

Достоинством метода является то, что единственное выполнение программы эквивалентно целому классу тестов с несимволическими данными. Конечно, метод символического выполнения стоит перед теми же проблемами, что и генерация условий верификации, и основная среди них — описание синтаксиса и семантики языка программирования в виде набора аксиом.

Второе приложение математических методов доказательств — использование доказательств *в процессе* проектирования и кодирования, а не после них. Исходная концепция структурного программирования по Дейкстре включает в себя идеи структурирования текста программы (возможность чтения программы сверху вниз и использование ограниченного набора управляющих конструк-

ций), пошаговой детализации (получение окончательного текста программы в результате ряда небольших шагов детализации) и неформального подтверждения правильности каждого шага детализации [12, 13]. Как читатель мог заметить, к двум последним аспектам структурного программирования раньше не привлекалось особого внимания.

Один из способов использовать методы доказательства на этапе кодирования — сначала писать утверждения, а затем вставлять между ними соответствующий текст программы. Это основная идея методологии, называемой *формальной разработкой* [14, 15]. Предлагается формализм, в рамках которого правильность алгоритма программы можно устанавливать по мере ее написания.

Идея интеграции процессов доказательства и программирования представляет особый интерес, поскольку она способствует точной фиксации программистом своих идей. К сожалению, метод обладает пока еще недостаточной общностью, чтобы можно было легко применять его к распространенным языкам программирования. Читателю, заинтересовавшемуся этой идеей, рекомендуется ознакомиться с примерами применения этого подхода (например, в работах [16—18]).

Конечной целью разработки методов доказательства является скорее всего автоматический синтез программ. Может быть, удастся создать систему, которая генерировала бы текст программы и доказывала ее правильность просто по заданным программистом входному и выходному утверждениям; это устранило бы необходимость в этапе кодирования программы человеком. Побудительной причиной разработки такой системы является предположение, что программисту легче описать, *что* должна делать программа (ее входное и выходное утверждения), чем *как* ей следует это делать (ее алгоритм).

Синтезатор программ — это еще более далекая перспектива, чем система автоматизированного доказательства, потому что синтезатор требует достаточно мощного искусственного интеллекта. Манна и Уолдингер [19] рассматривают необходимые такому синтезатору знания и способность к рассуждению. Синтезатор, например, должен выбирать алгоритмы, строить циклы и проверки условий, отлаживать конструируемые им программы. Вместо того чтобы порождать каждую новую программу с самого начала, с нуля, он должен также уметь пользоваться результатами прежних синтезов. Уже разработано несколько примитивных синтезаторов программ. Например, система PROW умеет писать программы на Лиспе по спецификациям, заданным на языке исчисления предикатов [20].

Даже если синтез программ окажется осуществимым, он может быть не всегда желательным, поскольку его использование ликвидирует ценную избыточность доказательств post factum. В нынешней ситуации программист по спецификациям пишет текст про-

граммы, а написать по этим же спецификациям утверждения и попытаться доказать правильность программы может кто-нибудь другой. Если условия верификации не могут быть доказаны,— значит, возможно, обнаружена ошибка (в тексте программы или в доказательстве). Эта избыточность (два разных перевода спецификаций) — основная ценность доказательств, но именно она устраняется при автоматическом синтезе.

Если не считать идеи неформальных «ручных» доказательств, концепция доказательства правильности программ в целом пока не используется на практике, но важно быть в курсе событий в этой области, если вы желаете идти в ногу со временем, поскольку она связана с изменением условий программирования в будущем. Более полный обзор и библиографию можно найти в работе [21].

ЛИТЕРАТУРА

1. Floyd R. W. Assigning Meanings to Programs, in J. T. Schwartz, Ed., Mathematical Aspects of Computer Science. Providence, R. I.: American Mathematical Society, 1967, pp. 19—32.
2. Naur P. Proof of Algorithms by General Snapshots, *BIT*, **6** (4), 310—316 (1966).
3. King J. C. A Verifying Compiler, in R. Rustin, Ed., Debugging Techniques in Large Systems. Englewood Cliffs, N. J.: Prentice-Hall, 1971, pp. 17—40.
4. London R. L. Proving Programs Correct: Some Techniques and Examples, *BIT*, **10** (2), 168—182 (1970).
5. Goodenough J. B., Gerhart S. L. Toward a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, **SE-1**(2), 156—178 (1975).
6. Robinson J. A. A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM*, **12** (1), 23—41 (1965).
7. King J. C. A Program Verifier, Ph. D. thesis, Carnegie-Mellon Univ., 1969.
8. Good D. I., London R. L., Bledsoe W. W. An Interactive Program Verification System, *IEEE Transactions on Software Engineering*, **SE-1** (1), 59—67 (1975).
9. German S. M., Wegbreit B. A Synthesizer of Inductive Assertions, *IEEE Transactions on Software Engineering*, **SE-1** (1), 68—75 (1975).
10. King J. C. A New Approach to Program Testing, in C. E. Hackl, Ed., Programming Methodology. Berlin: Springer-Verlag, 1975, pp. 278—290.
11. King J. C. Symbolic Execution and Program Testing, RC-5082, IBM Research Div., Yorktown Heights, N. Y., 1974.
12. Дал У., Дейкстра Э., Хоор К. Структурное программирование. Пер. с англ.— М.: Мир, 1975.
13. Dijkstra E. W. A Constructive Approach to the Problem of Program Correctness, *BIT*, **8**(3), 174—186(1968).
14. Jones C. B. Formal Definition in Program Development, in C. E. Hackl, Ed., Programming Methodology. Berlin: Springer-Verlag, 1975, pp. 387—443.
15. Date C. J., McMorran M. A., Sharman G. C. H. Program Proving and Formal Development: A Tutorial Introduction, TR-12.127, IBM United Kingdom Laboratories, Hursley, England, 1974.
16. Floyd R. W. Toward Interactive Design of Correct Programs, Proceedings of the 1971 IFIP Congress. Amsterdam: North-Holland, 1971, pp. 1—4.

17. Hoare C. A. R. Proof of a Program: FIND, Communications of the ACM, 14(1), 39—45 (1971).
18. Jones C. B. Formal Development of Correct Algorithms: An Example Based on Earley's Recogniser, Proceedings of an ACM Conference on Proving Assertions About Programs, *SIGPLAN Notices*, 7(1), 150—169 (1972).
19. Manna Z., Waldinger R. J. Knowledge and Reasoning in Program Synthesis, in C. E. Hackl, Ed., Programming Methodology. Berlin: Springer-Verlag, 1975, pp. 236—277.
20. Waldinger R. J., Lee R. C. T. PROW: A Step Toward Automatic Program Writing, in D. E. Walker and L. M. Norton, Eds., Proceedings of the International Joint Conference on Artificial Intelligence. Bedford, Mass.: Mitre Corp., 1969, pp. 241—252.
21. Elspas B., Levitt K. N., Waldinger R. J., Waksman A. An Assessment of Techniques for Proving Program Correctness, *Computing Surveys*, 4(2), 97—147 (1972).

ГЛАВА 18

Модели надежности

В области надежности аппаратуры достигнут уровень, когда уже создан ряд математических методов, позволяющих инженеру предсказывать надежность его продукта. Эти математические методы (главным образом в форме вероятностных моделей) широко применяются во многих инженерных областях с такими различными целями, как определение вероятности успешного запуска космического корабля или оценка срока службы электрической лампочки. Так как теория надежности аппаратуры развита довольно хорошо, естественно попытаться применить ее и к надежности программного обеспечения.

Из всех неизвестных параметров надежности программного обеспечения, вероятно, самым важным является число ошибок, оставшихся в программе. Если бы разумная его оценка была известна при тестировании, это помогло бы решить, когда можно закончить процесс. Если знать число оставшихся ошибок в устанавливаемой системе, можно было бы оценить стоимость работ по сопровождению и определить уровень доверия к программе. Другие параметры, для которых желательно иметь оценки,— это надежность программы (вероятность, что программа будет выполняться в течение данного интервала времени, прежде чем обнаружится ошибка заданной степени серьезности) и среднее время между отказами программы. Показатели сложности программы также были бы полезны для того, чтобы оценить качество проектирования и «эффект ряби» (какое влияние вызовет изменение данного модуля на другие части программы).

В настоящей главе рассматривается несколько моделей надежности. Несколько первых моделей тесно связано с теорией надежности аппаратуры и существенно опирается на определенные предположения о распределении вероятности отказов программного обеспечения. Следующий ряд моделей дает сходные результаты, но не связан с теорией надежности аппаратуры. Последние несколько моделей предназначены для предсказания сложности программных систем. Поскольку все эти модели вероятностные, предполагается, что читатель знаком с основными понятиями теории вероятностей и статистики.

МОДЕЛЬ РОСТА НАДЕЖНОСТИ

Вероятно, самой известной моделью надежности является модель, разработанная Джелински и Морандой [1] и Шуманом [2]. Поскольку она опирается на теорию надежности аппаратуры, необходимо ненадолго отвлечься, чтобы ввести некоторые понятия этой теории.

Пусть $R(t)$ — функция надежности, т. е. вероятность того, что ни одна ошибка не проявится на интервале от 0 до t; $F(t)$ — функция отказов: вероятность того, что ошибка проявится на интервале от 0 до t. Очевидно, $F(t)=1-R(t)$. Плотностью вероятности для $F(t)$ является функция $f(t)$, такая, что

$$f(t)=\frac{dF(t)}{dt}=-\frac{dR(t)}{dt}.$$

Полезно ввести *функцию риска* $z(t)$: условную вероятность того, что ошибка проявится на интервале от t до $t+\Delta t$, при условии что до момента t ошибок не было. Если T — время появления ошибки, то

$$z(t)\Delta t=P\{t<T<t+\Delta t|T>t\},$$

что, согласно теории вероятностей, эквивалентно

$$z(t)\,\Delta t=\frac{P\{t<T<t+\Delta t\}}{P\{T<t\}}=\frac{F(t+\Delta t)-F(t)}{R(t)}.$$

Поделив обе части на Δt и устремив Δt к нулю, переходим к пределу и получаем, что $z(t)=f(t)/R(t)$ или $z(t)=[-dR(t)/dt]/R(t)$. Теперь это дифференциальное уравнение можно разрешить относительно $R(t)$. Выбирая в качестве начального условия $R(0)=1$, мы получаем

$$R(t)=\exp\left(-\int_0^t z(x)\,dx\right),$$

а среднее время между отказами (СВМО) дается формулой

$$\text{СВМО}=\int_0^\infty R(t)\,dt.$$

Один из способов оценки СВМО — наблюдение за поведением программы в течение некоторого периода времени и нанесение на график значений времени между последовательными ошибками. Можно надеяться, что при этом будет обнаружено явление *роста надежности*; по мере того как ошибки обнаруживаются и исправляются, время между последовательными ошибками становится больше. Экстраполируя эту кривую в будущее, можно предсказать СВМО в любой момент времени и предсказать полное число ошибок

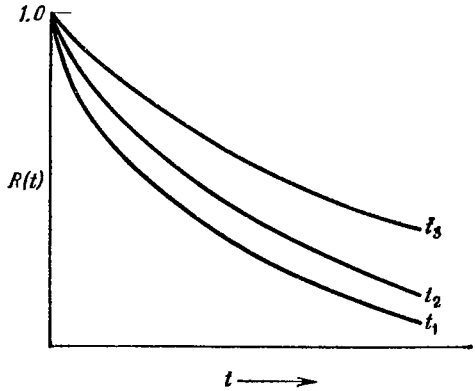


Рис. 18.1. Рост надежности программного обеспечения.

(оценивая число ошибок, которые проявятся раньше, чем СВМО станет бесконечным). Такая экстраполяция, однако, в слишком большой степени основана на догадках и обычно уводит в сторону. Было бы лучше опираться на какое-то априорное представление об имеющемся распределении вероятностей ошибок, затем использовать сведения о найденных ошибках для оценки параметров этого распределения и только потом использовать эту модель для предсказания событий в будущем.

Разработка такой модели начинается с уточнения поведения функции $z(t)$. В большинстве моделей аппаратного обеспечения $z(t)$ сначала уменьшается со временем (этап, когда обнаруживаются и исправляются ошибки проектирования и производства), затем остается постоянной в течение большей части срока службы системы (соответствует случайным отказам) и в конце полезного срока службы системы увеличивается (см. выше рис. 1.1). В теории надежности аппаратуры в основном рассматривается средний период, где функция риска постоянна и потому имеет место знакомое уравнение «без памяти» $R(t)=\exp(-ct)$, где c — некоторая константа. Однако предположение о постоянстве функции риска представляется не соответствующим реальности в случае программного обеспечения, для него эта функция должна уменьшаться по мере обнаружения и исправления ошибок. Поэтому, как показано на рис. 18.1, программное обеспечение характеризуется не единственной кривой $R(t)$, а их семейством; при обнаружении ошибок функции $R(t)$ меняется (улучшается).

Первое существенное предположение состоит в том, что $z(t)$ постоянно (время между сбоями — экспонента с отрицательным коэффициентом в показателе) до обнаружения и исправления ошибки, после чего $z(t)$ опять становится константой, но уже с другим, меньшим, значением. Это означает, что $z(t)$ пропорционально числу

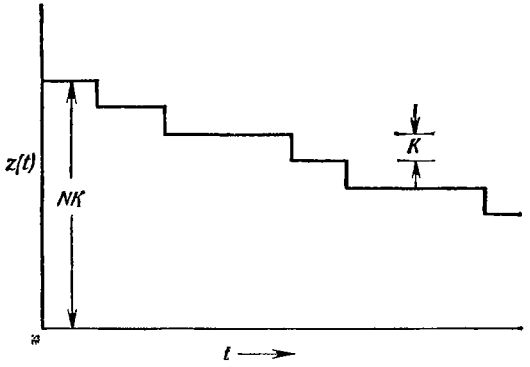


Рис. 18.2. Предполагаемая функция риска.

оставшихся ошибок. Второе предположение состоит в том, что $z(t)$ *прямо* пропорционально числу оставшихся ошибок, т. е. что $z(t) = K(N-i)$, где N — неизвестное первоначальное число ошибок, i — число обнаруженных ошибок, а K — некоторая неизвестная константа. Поведение $z(t)$ изображено на рис. 18.2. Каждый раз, когда ошибка обнаруживается (модель предполагает, что задержка между обнаружением ошибки и ее исправлением отсутствует, $z(t)$ уменьшается на некоторую величину K. На оси времени может быть представлено календарное время или время работы программы (последнее, возможно, нормировано с учетом интенсивности использования программы).

Параметры N и K можно оценить, если некоторое количество ошибок уже обнаружено (например, если фаза тестирования уже частично пройдена). Предположим, что обнаружено n ошибок, а $x[1], x[2], \ldots, x[n]$ — интервалы времени между этими ошибками. В предположении что $z(t)$ постоянно между ошибками, плотность вероятности для $x[i]$ равна

$$p(x[i]) = K(N-i)\exp(-K(N-i)x[i]).$$

Полагая T равным сумме x-ов и используя функцию максимального правдоподобия для этого уравнения, получаем следующую пару уравнений:

$$\sum_{i=1}^{n} \frac{1}{N-i} = n/\ (N-(1/T)\left(\sum_{i=1}^{n} x[i]\, i\right)) = KT;$$

K и N в этих уравнениях — приближения для фигурировавших выше K и N. Получилось два уравнения с двумя неизвестными N и K. Зная, что n ошибок обнаружено с интервалами $x[i]$ между ними, эти уравнения можно решить относительно N и K с помощью простой программы численного анализа. Значение N дает основной результат: оценку полного числа ошибок. Знание параметра K

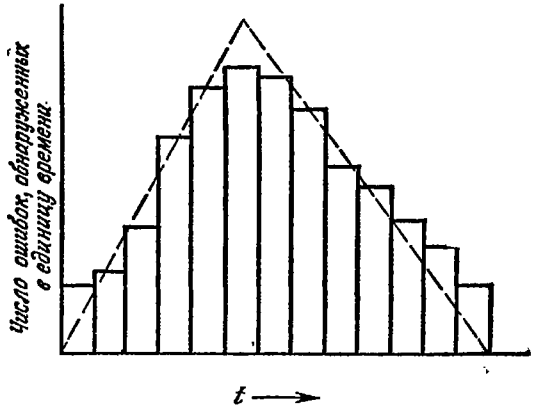


Рис. 18.3. Приближение треугольником.

позволяет использовать уравнения для предсказания времени до появления $(n+1)$-й, $(n+2)$-й и т. д. ошибки.

Эта основная модель может быть развита в различных направлениях. Например, частота отказов нередко увеличивается после завершения некоторого начального периода (по мере того как разрабатываются тесты или программа начинает использоваться интенсивнее) — это показано на рис. 18.3 [3]. Функция $z(t)$ тогда принимает вид

$$z(t) = K\left(N - \int_0^t p(x)\,dx\right),$$

где $p(x)$ — частота обнаружения ошибок. С помощью метода наименьших квадратов можно аппроксимировать $p(x)$ треугольником, как это изображено на рис. 18.3. Однако прогнозирование с помощью такого расширения модели, по-видимому, менее плодотворно, так как трудно постулировать $p(x)$ до того, как будет обнаружено большинство ошибок.

Критика модели

Для понимания и применения модели в первую очередь требуется понимать лежащие в ее основе предположения. Ведь она строится на многих предположениях, и все они — спорны. Некоторые (но не все) проблемы можно устранить развитием модели.

Одно из первых предположений состоит в том, что все ошибки одинаково серьезны (например, отказ системы и орфографическая ошибка в сообщении одинаково важны). Такое предположение легко снять, если разбить все ошибки на классы в соответствии с их серьезностью и дать разные оценки N и K для всех классов.

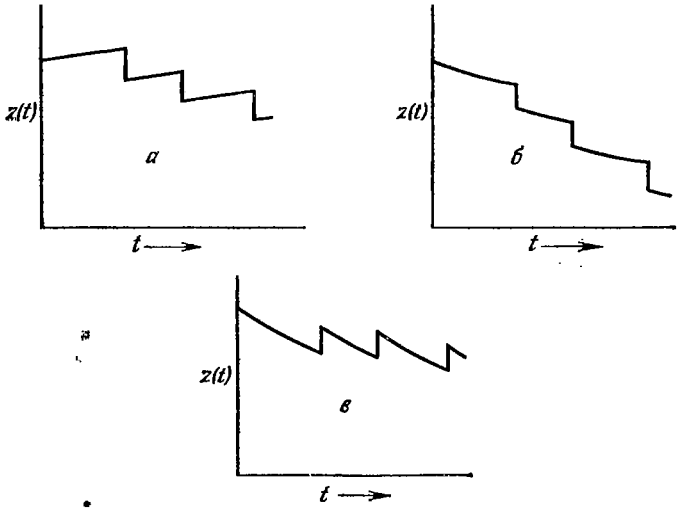


Рис. 18.4. Другие функции риска.

Второе предположение — что ошибка исправляется немедленно (или программа не используется до тех пор, пока найденная ошибка не будет исправлена). Предполагается также, что программа не изменяется (за исключением исправления ошибок). Четвертое предположение — при всех исправлениях найденные ошибки устраняются, и новых ошибок не вносится. Миямото [4], имея в виду это предположение, учел и фактор внесения ошибки. Он также показал эмпирически, что $z(t)$ примерно пропорционально числу оставшихся ошибок, но измерял его числом ошибок, обнаруженных к концу периода тестирования (делая, таким образом, нереалистическое предположение, что при тестировании обнаружены все ошибки).

Конечно, основное предположение — это то, что для всех программ функция $z(t)$ подобна изображенной на рис. 18.2. Тем самым предполагается, что каждая ошибка уменьшает $z(t)$ на постоянную величину K — феномен, вероятно, нереальный, но, по-видимому, с такого предположения вполне разумно начать. Более интересно предположение о том, что $z(t)$ постоянно между ошибками. Хотя желание выразить надежность программного обеспечения некоторой функцией времени вполне разумно, следует понимать, что в действительности она от времени не зависит. Надежность программного обеспечения является функцией числа ошибок, их серьезности и их расположения, а также того, как система используется. Сторонники постоянства $z(t)$ соглашаются, что входные данные системы не являются случайными и равновероятными, но утверждают, что они выглядят как случайные, поскольку их область из-

менения велика. Другие, однако, утверждают, что $z(t)$ возрастает за время между ошибками (рис. 18.4а), аргументируя это тем, что входные данные системы постепенно становятся «ближе» к тем, которые обнаружат оставшиеся ошибки. Есть и такие, кто верит [5], что $z(t)$ уменьшается со временем (рис. 18.4б), и утверждает, что, чем дольше программа выполняется без ошибок, тем меньше вероятность того, что одна из них будет обнаружена. Основываясь на аксиоме, в соответствие с которой при обнаружении ошибки возрастает вероятность появления еще одной, можно было бы постулировать, что $z(t)$ уменьшается между ошибками и возрастает всякий раз, когда ошибка обнаруживается (рис. 18.4в).

Все эти гипотезы о поведении $z(t)$ на рис. 18.2 и 18.4 демонстрируют некоторый рост надежности, но это не всегда имеет место в реальных программах даже в случаях, когда исправления не увеличивают число ошибок. Частота отказов в большой системе при измерении в течение нескольких лет может иметь тенденцию к понижению, но внутри этого периода возможны большие колебания. Анализ частоты обнаружения ошибок при тестировании 14 различных систем в работе [6] показал, что частота эта в пяти проектах достигала пика в начале работы, в пяти — в середине и в четырех — в конце.

Чтобы еще более усложнить этот вопрос, можно утверждать, что каждая программа имеет свое собственное, уникальное распределение $z(t)$ и даже что для каждой установки каждой программы $z(t)$ — свое. Ричардс [7] отмечает также, что функция $z(t)$ для одной программы может меняться со временем или при обнаружении каждой ошибки.

Отметим, наконец (совсем не желая создать излишне неблагоприятное впечатление, поскольку работы в этой области действительно заслуживают внимания), что описанная выше модель кажется чересчур оптимистичной. Например, при $n=10$ (десять обнаруженных ошибок) с интервалами между отказами (в минутах)

9, 17, 21, 54, 32, 78, 82, 33, 57, 82

модель предсказывает, что осталось 3.3 ошибки ($N=13.3$). Интервалы времени между 10, 11, 12 и 13 ошибками оцениваются в 135, 245 и 1265 минут.

ДРУГИЕ ВЕРОЯТНОСТНЫЕ МОДЕЛИ

Кроме обсуждавшейся в предыдущем разделе основной модели, было построено несколько других. Предлагалась байесова модель, учитывающая возможность того, что $z(t)$ может не уменьшаться при каждом исправлении ошибки из-за возможного внесения при этом новых ошибок [5]. Шнейдевинд [8] отказывается от предположения, что исправления выполняются сразу после обнаружения, и предлагает модель, в которой распределение исправлений

пропорционально распределению обнаружений ошибок, но отстает во времени. Было предложено несколько моделей на основе марковского процесса, состояния которого изменяются при всяком обнаружении или исправлении ошибки, а вероятности переходов представлены частотами обнаружения и исправления ошибок [9].

Более общий подход был предложен Шнейдевиндом [10, 11]. Он выполнил статистический анализ данных об ошибках в 19 программах и обнаружил, что эти данные не соответствовали какому-либо единственному распределению вероятностей. Анализ проверки дисперсии СВМО одной программы в семи различных условиях показал, что средние значения были не одинаковы, из чего следует, что среда (т. е. условия, в которых программа используется) — более существенный фактор, чем число оставшихся ошибок. Это привело к выводу, что модель надежности должна быть своей для каждой программы и конкретных условий ее использования. Для этого нужно собрать некоторые данные об ошибках, выработать на основе данных о частоте ошибок гипотезы о функции надежности, оценить параметры модели и выполнить тесты, показывающие, насколько эта модель подходит. Затем определяются доверительные интервалы для параметров и модели, после чего модель используется для предсказания $R(t)$ и доверительных интервалов для нее. Наконец, нужно сравнить вычисленные значения $R(t)$ и данные о новых ошибках.

СТАТИСТИЧЕСКАЯ МОДЕЛЬ МИЛЛСА

Модель совершенно другого типа разработал Миллс [12]. В ней не используется никаких предположений о поведении функции риска $z(t)$, эта модель строится на твердом статистическом фундаменте. Сначала программа «засоряется» некоторым количеством известных ошибок. Эти ошибки вносятся в программу случайным образом, а затем делается предположение, что для ее собственных и внесенных ошибок вероятность обнаружения при последующем тестировании одинакова и зависит только от их количества. Тестируя программу в течение некоторого времени и отсортировывая собственные и внесенные ошибки, можно оценить N — первоначальное число ошибок в программе.

Предположим, что в программу было внесено s ошибок, после чего разрешено начать тестирование. Пусть при тестировании обнаружено $n+v$ ошибок, причем n — число найденных собственных ошибок, а v — число найденных внесенных ошибок. Тогда оценка для N по методу максимального правдоподобия будет такой:

$$N=\frac{sn}{v}.$$

Например, если в программу внесено 20 ошибок и к некоторому моменту тестирования обнаружено 15 собственных и 5 внесенных ошибок, значение N можно оценить в 60. В действительности N можно оценивать после обнаружения каждой ошибки; Миллс [12] предлагает во время всего периода тестирования отмечать на графике число найденных ошибок и текущие оценки для N.

Вторая часть модели связана с выдвижением и проверкой гипотез об N. Примем, что в программе имеется не более k собственных ошибок, и внесем в нее еще s ошибок. Теперь программа тестируется, пока не будут обнаружены все внесенные ошибки, причем в этот момент подсчитывается число обнаруженных собственных ошибок (обозначим его n). Уровень значимости C вычисляется по следующей формуле:

$$C=\begin{cases} 1 & \text{при } n>k, \\ \dfrac{s}{s+k+1} & \text{при } n\leqslant k. \end{cases}$$

Величина C является мерой доверия к модели; это вероятность того, что модель будет правильно отклонять ложное предположение. Например, если мы утверждаем, что в программе нет ошибок ($k=0$), и, внеся в программу 4 ошибки, все их обнаруживаем, не встретив ни одной исходной ошибки, то $C=0.80$. Чтобы достичь уровня 95%, нам надо было бы внести в программу 19 ошибок. Если мы утверждаем, что в программе не более трех исходных ошибок, и, внеся шесть ошибок, обнаруживаем их все и не более трех исходных, уровень значимости равен 60%. Формула для C имеет под собой прочные статистические основания; выведена она Миллсом [12].

Эти две формулы для N и C образуют полезную модель ошибок; первая предсказывает число ошибок, а вторая может использоваться для установления доверительного уровня прогноза. Слабость этой формулы в том, что C нельзя предсказать до тех пор, пока не будут обнаружены все внесенные ошибки (а это, конечно, может не произойти до самого конца этапа тестирования). Чтобы справиться с этой трудностью, можно модифицировать формулу для C так, чтобы C можно было оценить после того, как найдено j внесенных ошибок ($j\leqslant s$) [7]:

$$C=\begin{cases} 1 & \text{при } n>k, \\ \dbinom{s}{j-1}\Big/\dbinom{s+k+1}{k+j} & \text{при } n\leqslant k. \end{cases}$$

В предыдущем примере, где $k=3$, а $s=6$, если найдены 5 из 6 внесенных ошибок, C опускается с 60 до 33%. Еще один график, который полезно строить во время тестирования,— текущее значение верхней границы k для некоторого фиксированного доверительного уровня, например 90%.

Модель Миллса одновременно математически проста и интуитивно привлекательна. Легко представить себе программу внесения ошибок, которая случайным образом выбирает модуль, вносит логическую ошибку, изменяя или убирая операторы, и затем заново его компилирует. Природа внесенной ошибки должна сохраняться в тайне, но все их следует регистрировать, чтобы впоследствии можно было разделять ошибки на собственные и внесенные.

Процесс внесения ошибок в настоящее время является самым слабым местом модели, поскольку предполагается, что для собственных и внесенных ошибок вероятность обнаружения одинакова (но неизвестна). Из этого следует, что внесенные ошибки должны быть «типичными» образцами ошибок, но мы еще недостаточно хорошо понимаем программирование, чтобы сказать, какими именно должны быть типичные ошибки. Однако по сравнению с проблемами, стоящими перед другими моделями надежности, эта проблема кажется относительно несложной и вполне разрешимой.

Наконец, отметим еще одно достоинство внесения ошибок: оно может оказывать положительное психологическое влияние на группу тестирования [13]. У программистов возникают затруднения при отладке своих программ, например, потому, что они склонны считать каждую обнаруженную ошибку последней. Внесение ошибок может помочь в этом деле, поскольку теперь программист знает, что в его программе есть еще не обнаруженные ошибки.

ПРОСТЫЕ ИНТУИТИВНЫЕ МОДЕЛИ

В поисках средств прогнозирования надежности программного обеспечения было разработано и несколько чрезвычайно простых моделей для оценки числа ошибок. Из-за их простоты им часто уделяется недостаточно внимания, но они основаны на более слабых предположениях, чем сложные модели, и могут оказаться очень полезными.

Один простой метод предлагает начинать тестирование двумя совершенно независимыми группами, использующими независимые наборы тестов. Этим двум группам (или двум сотрудникам — в зависимости от размеров проекта) в течение некоторого времени позволяется тестировать систему параллельно, а затем их результаты собирают и сравнивают. Обозначим через N_1 и N_2 число ошибок, обнаруженных каждой из групп соответственно, а через N_{12} — число ошибок, обнаруженных дважды (т. е. обеими группами). Это отношение изображено на рис. 18.5.

Пусть N обозначает неизвестное полное число ошибок в программе. Можно установить эффективность тестирования каждой из групп: $E_1=N_1/N$, $E_2=N_2/N$. Предполагая, что возможность обнаружения для всех ошибок одинакова (серьезное предположение, но не лишенное смысла), мы можем рассматривать каждое под-

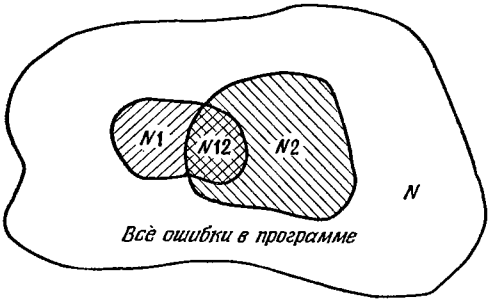


Рис. 18.5. Множества ошибок, обнаруженных на независимых тестах.

множество пространства N как аппроксимацию всего пространства. Если первая группа обнаружила 10% всех ошибок, она должна была найти примерно 10% всякого случайным образом выбранного подмножества, например подмножества N_2. В частности, мы могли бы сказать, что $E_1=(N_1/N)=(N_{12}/N_2)$. Выполняя подстановку для N_2, получаем $E_1=N_{12}/(E_2\times N)$, или

$$N=\frac{N_{12}}{E_1\times E_2};$$

N_{12} известно, а E_1 и E_2 можно оценить как N_{12}/N_2 и N_{12}/N_1 соответственно, откуда мы получаем приближение для N.

Например, предположим, что две группы нашли по 20 и 30 ошибок соответственно, и, сравнивая их, мы замечаем, что восемь ошибок из них — общие. Имеем $E_1=0.27$, $E_2=0.4$, что дает оценку $N=74$ и примерно 32 необнаруженные ошибки (74—20—30+8).

Вероятно, самый простой метод оценки числа ошибок — строить оценки, основываясь на исторических данных, в частности на среднем числе ошибок, приходящемся на один оператор в предыдущих проектах. В литературе сведения о частоте ошибок программистов довольно немногочисленны, но на основании имеющихся данных представляется, что в среднем по «отрасли» на каждую тысячу операторов программы после автономного тестирования остается примерно 10 ошибок. Таким образом, если нет более точных данных, можно предположить, что в программе из 32 000 операторов после автономного тестирования еще остается 320 ошибок.

Этими данными следует пользоваться с осторожностью, поскольку это всего лишь средние оценки, основанные на сведениях, собранных в основном до распространения новых методов программирования, например структурного программирования. Сторонники этого метода заявляют о значительно более низких оценках, но вследствие хорошо известной тенденции сообщества программистов к чрезмерному оптимизму может оказаться безопаснее опираться на пессимистические оценки.

Интересная интуитивная модель была найдена в IBM для оценки числа ошибок в выпусках OS/360 и ее преемников OS/VS1 и OS/VS2. Исследование данных об ошибках для всех компонент OS/360 показывает исключительно хорошее соответствие уравнению

$$ИЗМ=23\ (МИМ)+2\ (ИМ);$$

ИЗМ — полное число изменений модуля из-за ошибок, МИМ (многократно исправляемые модули) — число модулей, которые потребовали 10 и более исправлений, ИМ — число модулей, потребовавших одного или нескольких исправлений. Эта формула, как оказалось, дает замечательно точное предсказание числа ошибок, если можно оценить параметры МИМ и ИМ. При применении формулы к OS/VS1 ИМ оценивалось в 90% новых модулей и 15% числа модифицированных «старых» модулей. МИМ часто оценивалось в 15% числа новых модулей плюс 6% измененных. (Все эти числа основаны на реальных данных о предыстории.) Конечно, не утверждается, что эта модель применима к другим программам или к другим условиям.

Из-за неопределенностей во всех обсуждавшихся моделях пока самый разумный подход — воспользоваться несколькими моделями сразу и объединить их результаты. Данные по прежним проектам можно использовать для грубой оценки числа ошибок в сегодняшних проектах. Применима приведенная выше формула для N, если предприняты две параллельные попытки тестирования. Можно вручную внести в программу ошибки и затем оценить общее число ошибок и уровень достоверности по модели Миллса. Для получения четвертого прогноза можно воспользоваться также моделью Джелински, Моранды и Шумана.

МОДЕЛИ СЛОЖНОСТИ

Так как между сложностью и надежностью существует тесная связь, проблем надежности косвенным образом касается еще одна группа моделей — это модели, предназначенные для оценки сложности. Параметры этих моделей не являются чисто статистическими — они учитывают определенные атрибуты конкретной программы.

Одна из моделей для оценки сложности программы основана на параметрах композиционного проектирования из гл. 6 [14]. Для применения этой модели следует сначала определить прочность каждого модуля программы и сцепление каждой пары модулей. Имеется таблица, которая каждой категории прочности и сцепления ставит в соответствие определенное число между нулем и единицей (высокая прочность и слабое сцепление характеризуются значениями, близкими к нулю). Затем предполагается, что *зависимость первого порядка* между каждой парой модулей задается

по формуле

$$D_{ij} = \begin{cases} 0.15\,(S_i + S_j) + 0{,}7\,C_{ij}, & \text{если } C_{ij} \neq 0, \\ 0, & \text{если } C_{ij} = 0, \\ 1, & \text{если } i = j. \end{cases}$$

Величина D_{ij} описывает вероятность того, что модуль j придется изменить, когда изменится модуль i, если рассматривать модули i и j вне контекста всей программы (отношение считается симметричным). Величины S_i и S_j обозначают прочность этих модулей, а C_{ij}— сцепление (D_{ij}=0, если сцепления нет). Матрица D может быть представлена неориентированным графом, вершинами которого служат модули, а ребрами — ненулевые значения зависимости первого порядка. Эта матрица не изображает полную модель программы, так как в ней не отражены явно влияния зависимостей более высоких порядков (т. е. зависимости второго, третьего и т. д. порядков). Если изменяется модуль A, то модуль B может потребовать изменения, когда на графе зависимости между A и B имеется ребро или их связывает некоторая комбинация ребер. Для полного изображения взаимозависимостей между любой парой модулей вычисляется другая матрица — *полная матрица зависимостей*. Это делается с помощью следующих шагов.

1. Найти все пути в графе (исключая циклы) между парой модулей.
2. Вычислить вероятность для этих путей (произведение вероятностей для соответствующих дуг).
3. Вычислить зависимость между модулями, используя вероятности для путей, но не считая пути взаимно исключающими. В результате получаем вероятность того, что необходимо будет изменить один из модулей в случае изменения другого (снова предполагается, что отношение симметрично).

Эти вычисления даже для небольших программ довольно длинны, поэтому для их выполнения были написаны специальные программы.

Пример полной матрицы зависимостей приведен на рис. 18.6. Ее элементы — отношения между модулями. Например, если изменяется модуль A, вероятность того, что модуль F потребует изменений, равна 0.11. Суммируя элементы любой строки, можно вычислить число модулей, которые должны быть изменены при изменении соответствующего модуля программы. Например, если изменяется модуль C, ожидаемое число других изменений равно 1.22. Исключив диагональ, вычитая каждый элемент строки из 1.0 и перемножая результаты, получаем вероятность того, что изменение модуля не повлечет изменения других модулей программы. Например, для модуля A эта вероятность равна 0.32.

Модули	A	B	C	D	E	F
A	1.00	0.21	0.32	0.30	0.04	0.11
B	0.21	1.00	0.11	0.08	0.20	0.21
C	0.32	0.11	1.00	0.53	0.02	0.24
D	0.30	0.08	0.53	1.00	0.02	0.13
E	0.04	0.20	0.02	0.02	1.00	0.04
F	0.11	0.21	0.24	0.13	0.04	1.00

Рис. 18.6. Полная матрица зависимостей.

Суммируя все элементы матрицы и деля сумму на число модулей, можно получить грубую оценку сложности программы.

В этой модели привлекает возможность получать ответы на вопросы подобного рода, а также тот факт, что она основана на ключевых свойствах структуры программы, но все же модель основывается на многих недостаточно выверенных предположениях. Модель не была проверена, поскольку не удалось придумать, как это разумно сделать (эмпирическая проверка модели намного сложнее, чем может показаться). Близкая модель, основанная на методах линейной алгебры, была предложена Хейни [15], но она требует, чтобы пользователь сам ввел определение взаимоотношений между модулями.

В настоящее время делаются попытки моделировать сложность программы, опираясь на свойства ее текста. Салливен [16] предлагает несколько моделей, основанных на предположении, что сложность понимания каждого конкретного участка программы пропорциональна числу понятий, активных на этом участке. Предложено несколько способов измерять эту сложность, опираясь на представление структуры управления и структуры данных программы в виде ориентированного графа.

ЛИТЕРАТУРА

1. Jelinski Z., Moranda P. B. Software Reliability Research, in W. Freiberger, Ed., Statistical Computer Performance Evaluation. New York: Academic Press, 1972, pp. 465—484.
2. Shooman M. L. Operational Testing and Software Reliability Estimation during Program Development, Record of the 1973 IEEE Symposium on Computer Software Reliability, New York: IEEE, 1973, pp. 51—57.
3. Shooman M. L. Probabilistic Models for Software Reliability Prediction, in W. Freiberger, Ed., Statistical Computer Performance Evaluation. New York: Academic Press, 1972, pp. 485—502.
4. Miyamoto I. Software Reliability in Online Real Time Environment, Proceedings of the 1975 International Conference on Reliable Software, New York: IEEE, 1975, pp. 194—203.

Littlewood B., Verrall J. L. A Bayesian Reliability Growth Model for Computer Software, Record of the 1973 IEEE Symposium on Computer Software Reliability. New York: IEEE, 1973, pp. 70—77.

Craig G. R. et al. Software Reliability Study, RADC-TR-74-250, TRW Corp., Redondo Beach, Ca., 1974.

Richards F. R. Computer Software: Testing, Reliability, Models, and Quality Assurance, NPS-55RH74071A, Naval Postgraduate School, Monterey, Ca., 1974.

Schneidewind N. F. Analysis of Error Processes in Computer Software, Proceedings of the 1975 International Conference on Reliable Software. New York: IEEE, 1975, pp. 337—346.

Trivedi A. K., Shooman M. L. A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters, Proceedings of the 1975 International Conference on Reliable Software, New York: IEEE, 1975, pp. 208—215.

Schneidewind N. F. A Methodology for Software Reliability Prediction and Quality Control, NPS-55SS72111A, Naval Postgraduate School, Monterey, Ca., 1972.

Schneidewind N. F. An Approach to Software Reliability Prediction and Quality Control, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 837—847.

Mills H. D. On the Statistical Validation of Computer Programs, FSC-72-6015, IBM Federal Systems Div., Gaithersburg, Md., 1972.

Weinberg G. M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.

Myers G. J. Reliable Software Through Composite Design. New York: Petrocelli/Charter, 1975.

Haney F. M. Module Connection Analysis — A Tool for Scheduling Software Debugging Activities, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, N. J.: AFIPS Press, 1972, pp. 173—179.

Sullivan J. E. Measuring the Complexity of Computer Software, MTR-2648-V, Mitre Corp., Bedford, Mass., 1973.

ГЛАВА 19

Инструментальные системы

В предыдущих главах говорилось о многих инструментах, разработанных специально для того, чтобы помочь программисту в различных сферах его деятельности. Сюда относятся многообразные средства: от коммерчески доступных программных продуктов до инструментов, разработанных отдельными организациями для внутренних исследовательских целей, в том числе средства тестирования (например, средства статического анализа программ, мониторы, имитаторы терминалов), отладки, доказательства правильности программ. При виде такого многообразия инструментов в голову приходят два вопроса: остались ли еще аспекты разработки программного обеспечения, нуждающиеся в автоматизации, и не существует ли какого-либо метода объединения этих средств в единую инструментальную систему?

БИБЛИОТЕКИ ОБЕСПЕЧЕНИЯ РАЗРАБОТКИ

В глазах многих из нас представление о рабочем помещении программистов ассоциируется с общим впечатлением беспорядка: шкафы до потолка, хранящие тысячи перфокарт, груды спецификаций и листингов на столах, закапанные кофе коробки с перфокартами, блок-схемы и выпуски документации по проекту, сваленные в кучу на полу. Такая обстановка, хотя она по разным причинам постепенно исчезает (одна из причин — противопожарная безопасность), ведет к дополнительным и отнюдь не обязательным ошибкам. В таких условиях никто не знает точно, какова последняя версия программы. Руководство слабо представляет себе состояние проекта. Если программист заболеет, его часть проекта нередко исчезает из виду. Кроме того, возникает множество ошибок при самых обычных манипуляциях с программой. Никогда не забуду, как, проверяя программу в ночь перед ее демонстрацией, я в нетерпении кинулся к АЦПУ, поднял крышку, чтобы посмотреть результаты тестирования, и застыл в ужасе, наблюдая, как единственная колода из 3000 перфокарт программы на Фортране разлетается по всему вычислительному центру,— я положил ее на АЦПУ!

Концепция программиста-библиотекаря — определенный шаг к решению этих проблем. Однако просто добавление библиотекаря для выполнения ручной работы — не полное решение. При этом все еще возможно потерять или перепутать колоду карт, не туда положить листинги, все еще остается риск пожара или внезапного исчезновения библиотекаря. Одно из решений этой проблемы — автоматизированная библиотека, или система обеспечения разработки. При хранении всех рабочих материалов по проекту (например, исходных текстов, объектных программ, тестовых данных, документации) в базе данных возможность ошибок, связанных с ручными манипуляциями, перемещением и т. п., уменьшается. Контроль за состоянием проекта может быть обеспечен с помощью генераторов отчетов, анализирующих состояние базы данных. Если все инструменты программирования, имеющиеся в организации, можно включить в эту единую систему, проект получает в свое распоряжение все достоинства единой интегрированной базы данных и стандартизации условий (технологии) программирования.

В настоящем разделе дан обзор некоторых автоматизированных библиотечных систем. Он ни в какой степени не претендует на полноту; наша цель здесь — проиллюстрировать достоинства таких средств и рассмотреть некоторые особо интересные решения, позволяющие устранить отдельные источники ошибок в программах.

Система PANVALET

Система PANVALET, созданная фирмой Pansophic Systems, — широко распространенная библиотечная система. Она работает в диалоговом или пакетном режимах на любой операционной системе IBM 360 или 370.

База данных PANVALET содержит файлы, хранящие исходные программы, объектные программы, тексты на языке управления заданиями и данные для программ. Для работы с этими файлами имеются разнообразные команды редактирования, например добавление, стирание или изменение записей, копирование одного файла в другой, печать файла. Для поиска и/или изменения строки символов в одном файле, во всех файлах данного типа (например, во всех файлах с исходными программами на Коболе) или во всей библиотеке может использоваться функция глобального сканирования/подстановки.

Особенно полезной чертой системы PANVALET является концепция *версий файла*. Всякий файл (например, исходный текст процедуры на PL/1) может иметь несколько версий. Таким образом, одна версия будет представлять собой текущую рабочую (готовую) копию модуля, а вторая версия может использоваться программистом для тестирования модификации этого модуля. Одна из версий каждого файла в библиотеке может быть отмечена как

готовая версия. Нет абсолютно никакой возможности изменить готовую версию файла. Единственный способ модификации — создать новую версию и скопировать готовую версию в нее. Это очень полезно для контроля за модификациями программы.

Файлы могут быть иерархически структурированы с помощью *включения*. Это означает, что файл *A* может содержать указание, что он включает содержимое файлов *B*, *C* и *D*. Всякий раз, когда файл *A* используется (например, компилируется), он расширяется посредством логического копирования в него файлов *B*, *C* и *D*.

Еще одна существенная часть базы данных — каталог сведений обо всех версиях каждого файла. Каждый элемент каталога содержит статус версии (например, «готовая»), даты последнего обращения и последнего обновления версии, число операторов, тип последнего действия, выполненного над версией. Предусмотрено несколько вариантов отчетов, таких, как распечатки каталога и перекрестных ссылок, указывающих, какие файлы включают другие файлы, а какие — включены в другие.

При сопровождении программы часто возникает необходимость изучить объектный код из дампа или листинг компиляции, и при этом нет возможности сопоставить их с соответствующей версией исходной программы. PANVALET решает эту проблему, включая номер версии и дату последнего изменения в листинг компилятора и в объектные модули.

PANVALET содержит также ряд операций и вариантов выдачи отчетов для сопровождения библиотеки. Периодически можно делать запасные копии библиотеки для защиты ее от разрушения. Стираемые файлы автоматически копируются на запасную ленту, прежде чем затирается освобождаемое место, чтобы можно было их восстановить, если окажется, что этот файл исключен случайно или преждевременно. Редко используемые файлы переписываются на архивную ленту, откуда их при необходимости выбирают. Для предупреждения несанкционированного использования файла ему может быть присвоен один из уровней секретности.

Система построения систем SBS

SBS — библиотечная система, разработанная для применения внутри IBM [1]. Она включает файлы с исходным текстом, объектным кодом, информацией для редактирования связей, данные, документы и объекты, называемые *группами*.

Как и PANVALET, система SBS предоставляет пользователю концепцию версий файла, но она решает проблему синхронизации исходной и объектной программы иначе. Чтобы понять, как это делается, сначала следует рассмотреть понятие файла типа *группа*. Такой файл — это, по существу, опись всего, что связано с программой. Как показано на рис. 19.1, в ней просто перечисляются

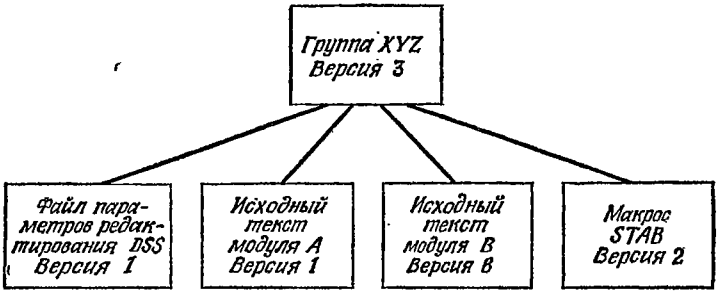


Рис. 19.1. Группа в системе SBS.

объекты, нужные программе (в данном случае — два модуля, один макрос, нужный для компиляции одного из них или, может быть, обоих, и директивы редактора связей). Когда пользователь требует от SBS построить версию 3 программы XYZ, система SBS автоматически компилирует соответствующие версии модулей (если файлы объектного кода для них пусты) и выполняет редактирование связей. Синхронизация исходного и объектного кода выполняется следующим образом: всякий раз, когда обновляется файл исходной программы, SBS автоматически стирает файл объектного кода этой версии, а также объектный код для любой группы, элементом которой является эта версия файла. Тем самым всегда обеспечивается соответствие объектного кода версии модуля или группы (программы) последнему варианту исходной программы.

Фабрика программного обеспечения

Две предыдущие системы обеспечивают ряд удобств при сортировке, обновлении, компиляции программ, но главным образом они ориентированы на этапы собственно программирования и сопровождения. В частности, они не содержат средств, специально предназначенных для других фаз, таких, как определение требований, проектирование и тестирование. Фабрика программного обеспечения,— вероятно, самый честолюбивый проект из разработанных пока систем — представляет собой попытку интеграции инструментов для всего процесса разработки в единую систему.

Фабрика программного обеспечения была разработана для использования в корпорации System Development [2]. Лучше всего начать описание системы с ее базы данных, изображенной в центре рис. 19.2. В системе имеются две базы данных: вместительная библиотека (база данных разработки) и база отношений (база данных управления проектом). Библиотека разработки состоит из файлов, содержащих исходный текст, объектный код, загрузочные модули, операторы управления редактированием связей и тестовые данные; эти файлы могут иметь по нескольку версий. База данных управ-

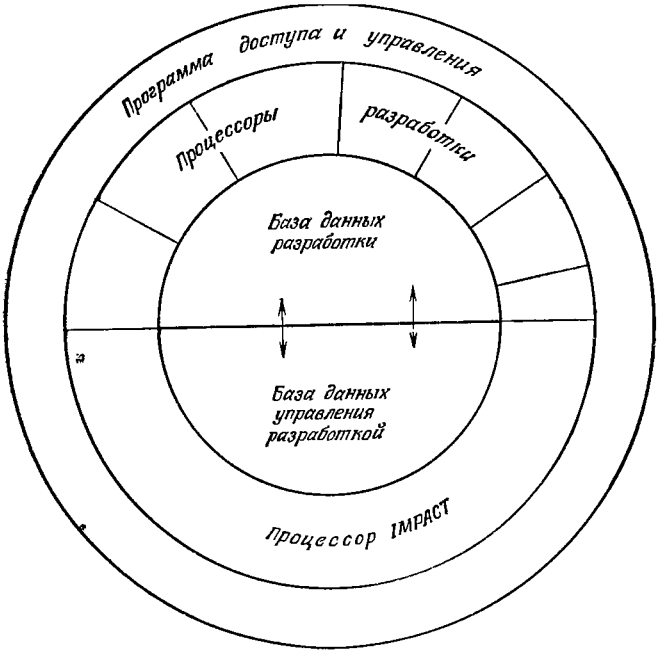


Рис. 19.2. Разрез фабрики программного обеспечения.

ления проектом содержит информацию, связанную с руководством проектом и общей структурой программы. В этой базе данных хранятся календарные планы и оценки необходимых ресурсов для всех этапов проекта, описания требований пользователя и иерархическая структура изготавливаемой программы, регистрируются проектные изменения и исправления ошибок, зависимость от других проектов и т. д. Самое важное — обеспечена обработка отношений между всеми этими элементами, что позволяет делать соответствующие запросы. Руководитель проекта может определить, какие требования пользователя к какому конкретному программному модулю относятся, какие изменения в проектировании влияют на данный модуль, какие модули программы разрабатываются на конкретном этапе работы над проектом, число ошибок, сделанных конкретным программистом, и т. д.

Базу данных окружает ряд инструментов разработки. Сюда входят компиляторы, редакторы связей, обслуживающие программы, средства документирования и тестирования, а также специальный процессор, называемый IMPACT. Над этими процессорами имеется надстройка в виде компоненты FACE. Она обеспечивает взаимодействие между пользователем и всеми процессорами. Кроме того она собирает управляющую информацию и данные о со-

стоянии от процессоров и включает ее в базу данных управления проектом.

Основное назначение процессора IMPACT (Integrated Management, Project Analysis and Control Technique — Объединенные методы руководства, анализа проекта и управления) — помогать руководителю проекта планировать производство и наблюдать за выполнением планов. Этот процесс можно использовать для разработки календарного плана проекта и исследования альтернатив по методу критических путей. Во время осуществления проекта он прослеживает продвижение по плану и расход ресурсов, давая руководителю более наглядное представление о состоянии проекта. IMPACT используется также для запросов к базе данных управления проектом в терминах описанных выше отношений и для ответов на более простые вопросы, касающиеся размеров модуля, числа его компиляций, числа выявленных в нем ошибок или состояния тестирования модуля.

Монитор Simon

Simon — это средство разработки, созданное корпорацией Mitre [3]. В отличие от других систем Simon не является библиотечной системой; это *монитор реализации*, цель которого — сбор данных о программных проектах, для того чтобы заглянуть внутрь процесса программирования и чтобы изучать эффекты экспериментов. Simon расположен между программистом и его инструментами (например, компиляторами, редакторами текстов, средствами тестирования). Он анализирует каждое сообщение и обновляет соответствующий файл, один из трех. *Файл элементов* содержит информацию о проектах, программистах, модулях программы, документах, тестовых данных, макросах и ошибках в программах. *Файл отношений* включает данные об отношениях между компонентами файла элементов, показывая, например, какие ошибки найдены в каких модулях, какие модули вызываются другими модулями, и какими именно, и кто их написал. *Файл событий* регистрирует всякое событие, которое происходит в системе (например, обновление модуля, компиляция, выполнение теста). Накопление всех этих данных ничего не дает для надежности выполняемого в данный момент проекта, но они используются для совершенствования проектов в будущем, помогая формулировать и проверять гипотезы о путях улучшения процессов программирования.

Другие желательные свойства инструментальных систем

Описанные выше системы предоставляют много ценных возможностей, но можно ожидать, что они могли бы уметь еще больше. Ниже перечисляется ряд идей, которые было бы желательно реализовать в будущих инструментальных системах.

Ключевой компонентой всякой системы обеспечения разработки является интегрированная база данных. Элементы этой базы данных содержат исходный текст, объектный код, макросы, спецификации, тестовые данные и программы (ср. элемент-*группа* в SBS). Отношения между элементами должны храниться и обрабатываться так, чтобы, имея один элемент (например, отдельный модуль), легко можно было получить соответствующие ему элементы (например, спецификации и тестовые данные). Отношения в базе данных должны также отражать структуру программы таким образом, чтобы можно было задавать вопросы типа: кто вызывает модуль *X*, кого модуль *X* вызывает, где используется макрос Y, какие макросы использует модуль *X*. Каждая сконструированная версия программы должна иметь опись, так чтобы можно было определить, какие версии каких модулей и макросов ее составляют. Средства сборки программ (например, редактор связей) должны обеспечивать уверенность в том, что версии всех частей программы согласованы. Например, если программист строит программу, в которой два модуля используют один и тот же макрос, система должна убедиться, что оба модуля скомпилированы с одной и той же версией макроса.

При наличии такой интегрированной базы данных возникают идеи других полезных инструментов. Можно просматривать исходный текст программы в поисках нарушений стандартов программирования или употребления чреватых ошибками конструкций. Можно написать программу, которая бы рисовала структуру вызовов для любой программы или прослеживала бы и анализировала поток данных между модулями. (Такая программа должна допускать изменения имен при передаче данных между аргументами и параметрами и перекрытие областей памяти. Аллен [4] описывает экспериментальную программу такого типа, работающую с программами на PL/1.) Можно предусмотреть программу контроля сопряжения модулей, которая бы анализировала исходный текст вызывающего и вызываемого модулей и определяла бы, согласуются ли аргументы и параметры по числу и типу.

Если система содержит информацию обо всех проектах и о сотрудниках, работающих над ними, она могла бы избирательно распространять информацию для участников проектов, на которых может повлиять то или иное событие. Например, программист может быть уведомлен о том, что в модуле, связанном с его модулем (вызываемом им или вызывающем его), найдена ошибка. Руководитель проекта может быть оповещен, если некоторая запланированная работа не была завершена в срок. В свою очередь системе может быть приказано разослать оповещения после завершения некоторой работы. Например, программист *A* мог бы сказать системе: «Дай мне знать, когда программист *B* закончит автономное тестирование модуля *X*».

С этой идеей связана интересная концепция *критерия повышения статуса*. В начале работы над проектом руководитель может указать системе критерии такого рода: «для того чтобы модуль мог получить статус автономно отлаженного, все автономные тесты должны пройти успешно, при тестировании должны быть проверены 100% развилок, успешным должен быть контроль соответствия стандартам и контроль сопряжений и модуль должен быть скомпилирован в режимах *A*, *B* и *C*». Как только модуль получает этот статус, он становится доступным для использования в последующих процессах, таких, как тестирование функций. Для разных типов элементов, например, модулей и спецификаций, могут быть разработаны разные правила присвоения статусов (модуль может быть даже «понижен», если найдена ошибка).

СРЕДСТВА ПРОЕКТИРОВАНИЯ

Примечательно отсутствие в предыдущем разделе каких бы то ни было упоминаний о средствах автоматизации процессов проектирования программного обеспечения. Причина этого совершенно проста: довольно трудно автоматизировать операции, которые еще недостаточно хорошо понимаются человеком.

Однако можно начать автоматизацию процессов проектирования с разработки инструментов для проверки завершенного проекта на неполноту и противоречивость. Трудно представить себе средство, которое могло бы делать это на уровне целей проекта или внешних спецификаций (спецификации пользователя), но для последующих процессов, таких, как проектирование структуры программы (гл. 6) и внешних спецификаций модуля (гл. 8), это кажется уже вполне осуществимым.

Одно из таких средств — система DACC (Design Assertion Consistency Checker — Проверка согласуемости проектных утверждений), разработанная корпорацией TRW для проверки межмодульных сопряжений [5]. Для каждого сопряжения DACC сравнивает аргументы и параметры, проверяя число и порядок элементов данных и их типы, формат, единицы измерения (например, фунты, метры в секунду, доллары) и области допустимых значений. Система DACC была использована для проверки проектных спецификаций большой бортовой космической системы; она нашла семь явных ошибок (например, аргумент задавался в градусах, а соответствующий параметр измерялся в радианах) при затратах в 30 долларов за машинное время и несколько сотен долларов за ввод этих спецификаций в базу данных.

Экспериментальное средство для проверки некоторых свойств структуры программы было разработано в Институте системных исследований IBM. Проектировщик вводит свой проект структуры в базу данных, описывая каждое сопряжение. В этом описании

указываются имена вызывающего и вызываемого модулей, а также входные и выходные аргументы. Аргументы описываются в абстрактных терминах; например, проектировщик может указать, что входными данными для конкретного сопряжения служат «код района и список имен покупателей». Он вводит также характеристику прочности каждого модуля и сцепления модулей.

Программа выполняет несколько типов проверок. Она исследует все сопряжения каждого модуля, чтобы определить, совпадает ли везде число входных и выходных аргументов для каждого сопряжения. Она исследует, правильно ли описано сцепление модулей для каждой программы. Например, когда модули *A* и *B* вызывают модуль *C*, система выдает предостережение, если тип сцепления *A* и *C* отличается от типа сцепления *B* и *C*.

Последний тип проверки — попытка обнаружить неполные сопряжения. Система исследует каждый модуль и окружающие его сопряжения. Формируется описание *полного входного пространства* модуля, содержащее входные аргументы всех вызовов этого модуля плюс выходные аргументы всех модулей, вызываемых из него. Описание *полного выходного пространства* формируется из выходных аргументов всех модулей, вызывающих данный, и входных аргументов модулей, вызываемых им. Каждый элемент выходного пространства сопоставляется с входным пространством, чтобы определить, может ли он быть получен по входным данным. При этом используется алгоритм выделения подмножеств, позволяющий системе понять, что выход «имя покупателя» получается из входного элемента «список имен покупателей».

Очевидно, в результате может быть выдано много сообщений о потенциальных проблемах, которые в действительности проблемами не являются. Например, программа не знает, что выходной элемент «номер человека» получается по входному элементу «запись о сотруднике». Однако цель предостережения — дать проектировщику список потенциальных упущений для тщательной проверки. Если проектировщик желает, он может заранее ввести в систему соответствующие отношения, например «номер человека может быть получен по записи о сотруднике».

При наличии такой базы данных, описывающей структуру программ, могут быть разработаны и другие инструменты проектирования. Например, может быть добавлена возможность задавать такие вопросы: «Какие модули вызывают модуль *X*?», «Каково сопряжение модуля *Y*?»

Упоминавшиеся выше инструменты проектирования применимы лишь к очень небольшой части всего процесса проектирования; можно представить себе средства, более непосредственно участвующие в этом процессе. Каугер [6] и Тейчроев [7] анализируют попытки автоматизировать процессы разработки требований и анализа систем. Окончательной целью разработки средств проекти-

рования является такая система, которая по функциональным спецификациям высокого уровня автоматически строит текст программы и документацию. Для большинства из нас это звучит как отрывок из научно-фантастического романа, но Фримен [8] серьезно рассматривает основные препятствия на этом пути и описывает исследования в этой области. Для читателя, заинтересованного в более глубоком изучении этих вопросов, Виноград [9] предлагает еще кое-какие идеи о том, какого рода услуги могли бы предоставлять пользователю средства программирования в будущем. Тем, кто разрабатывает взаимодействие пользователя со средствами программирования, Овертон [10] дает некоторые полезные советы по части психологических факторов.

ЛИТЕРАТУРА

1. DeJong S. P. The System Building System (SBS), RC-4486, IBM Research Div., Yorktown Heights, N. Y., 1973.
2. Bratman H. The Software Factory, *Computer*, 8 (5), 28—37 (1975).
3. Clapp J. A., Sullivan J. E. Automated Monitoring of Software Quality, Proceedings of the 1974 National Computer Conference Montvale, N. J.: AFIPS Press, 1974, pp. 337—341.
4. Allen F. E. Interprocedural Analysis and the Information Derived by It, in C. E. Hackl, Ed., Programming Methodology. Berlin: Springer-Verlag, 1975, pp. 291—321.
5. Boehm B. W., McClean R. K., Urfrig D. B. Some Experience with Automated Aids to the Design of Large-Scale Reliable Software, *IEEE Transactions on Software Engineering*, SE-1(1), 125—133 (1975).
6. Cougar J. D. Evolution of Business System Analysis Techniques, *Computing Surveys*, 5 (3), 167—198 (1973).
7. Teichroew D. A Survey of Languages for Stating Requirements for Computer-Based Information Systems, Proceedings of the 1972 Fall Joint Computer Conference. Montvale, N. J.: AFIPS Press, 1972, pp. 1203—1224.
8. Freeman P. Automating Software Design, *Computer*, 7(4), 33—38 (1974).
9. Winograd T. Breaking the Complexity Barrier Again, SIGPLAN Notices, 10(1), 13—30 (1975).
10. Overton R. K. et al. Developments in Computer-Aided Software Maintenance, ESD-TR-74-307, AMS Corp., Claremont, Ca., 1974.

Предметный указатель

Абстрактная машина 79
Абстракции уровень 79—83
Автоматические доказательства правильности программ 322—324
Адаптируемость 55, 60, 96
Активации блок 305
Архитектура системы 38, 78—90, 265, 266
Аттестация 174

Базы данных проектирование 39, 266
Безличное программирование 147—149, 269
Безопасность 56, 59, 117, 124, 303
Библиотека обеспечения разработки 344—351
Библиотекарь, программист 267—269, 270 271, 345
Большого скачка метод тестирования 188, 191
Блок-схема 131, 149—150, 200, 317
Бригада программистов 269—272
— ведущего программиста 269—271
— демократическая 271—272
— хирургическая 269—272

Венера, операционная система 81—83
Версии файла 345—346
Виртуальная машина 245
Вклинивания точки 255, 302, 306
Внешнее проектирование модуля 39, 127—129
— — программной системы 37, 62—75, 264, 265, 266
Внешние спецификации 62, 68—75
Восходящий метод тестирования 180—183, 191
Второй программист 270
Выделенный путь 316

Глобальные данные 81, 96—97, 100, 289, 320
Граничные условия 113, 202, 203
Группы 346—347
Гэннона эксперимент 277

Дамп 255, 258—259, 302, 346
Диагностический монитор 119
Диаграмма управления 206, 215
Динамическое чтение программы 152—153
Доказательство 174

Единообразие 278—280, 283—284

Заглушка 184—186, 189
Защита данных 56, 59, 117, 124, 303
Защитное программирование 141—146, 220

Иерархическая структура 40, 79
Изменения проекта процесс 76
Изоляция ошибок 36, 56, 122—124
Имитатор терминала 243
Инвариант цикла 315
Индуктивных утверждений метод 311—327
Интеграция системы 180—192, 229—233
— — непрерывная 230—231
Исправление ошибок 33, 34, 120—121
Испытание 174

Календарный план 56
Когнитивный диссонанс 147—149
Комментарии 140, 150—151, 164—165
Композиционный анализ 100—112
Контролируемый пользователем проект 49
Контроль 174
Конфигурация системы 84
— — прямая 84
— — непрямая 84
Концептуальная целостность 41, 62, 116, 280

Логика модуля 93

Макроэффективность 160—164
Математическое программное обеспечение 194
Матрица зависимостей полная 341—342
Машина с языком высокого уровня 307—308
МетаКобол 216
Метод Брауна — Сэмпсона 251
Методология проектирования 78
Микроэффективность 160—164
Мобильность 82
Модуль 92, 287, 305
Модуля внешнее проектирование 39, 127—129, 202
Модуля логики проектирование 39, 127

Надежности функция 330
Надежность аппаратуры 15, 329—330
— программного обеспечения 13
Наивысшей абстракции точка 101
Независимость 40, 93
Независимый от пользователя проект 49
Нисходящий метод тестирования 183—186
— модифицированный метод тестирования 186—188

Обнаружение ошибок 33—34, 114—119
— — активное 114, 118—119
— — пассивное 114—116
Общность 54
Опытная эксплуатация 239
Отказов функция 330
Отладка 175, 250—262
Ошибка в программном обеспечении 10—12

Перевода макромодель 22—26
— микромодель 27—28
— ошибка 22—28, 70, 193
Побочный эффект 41, 211, 229, 320
«Подпорка» 231
Позиция программиста 147—149
Пользователь, взаимодействие 64—68
Порт 83—88
— входной вторичный 84—88
— — первичный 83—88
— выходной 84—88
Посылки/получения механизм 83—88
Пошаговая детализация 79, 131, 138—140
Предсказуемый модуль 99
Предупреждение ошибок 33
Преобразование данных автоматическое 167, 284—287, 302
Проект исследовательский 72
— разработка 72
Проектирования ошибки 15
Производственный дефект 15
Промежуточная переменная 157
Процесс 83—88
Прочность модуля 93—96
— — информационная 95
— — коммуникационная 95
— — по классу 94
— — по логике 94
— — по совпадению 94

— — процедурная 94
— — функциональная 95
Проявление связей 41
Психологические факторы 55

Разбиение STS 101—102
— операционное 101—102
— функциональное 101—102
Раздвоение 283
Регистрация ошибок 116
Регрессии тест 220, 232
Решение задачи 43—46
Риска функция 330
Руби эксперимент 277, 282, 290, 293, 320

Сандвича метод тестирования 188—189
— модифицированный метод тестирования 189
Сбой 15
Сборочный узел 231
Семантический разрыв 299, 300, 304
Символическое выполнение 324—336
Синтез программ автоматический 326—327
Сквозной контроль 73—74, 90, 113, 152—153
Сложность 40, 340—342
Совместимость 59
Сопровождение 17, 55
Спин 230
Среднее время между отказами (СВМО) 11, 60, 330, 331, 336
Стандарты 59, 151, 350
Статическое чтение программы 112—113, 152
Стоимость программного обеспечения 9, 17—18, 56
Структурное программирование 41, 131 134—138, 290
Сцепление модулей 96—99
— — по внешним данным 97
— — по данным 98
— — по общей области 96
— — по содержимому 96
— — по управлению 97
— — по формату 98

Таблицы решений 70, 225—228
Тестирование автономное 175, 119—219
— внешних функций 175, 219—248
— защиты 235
— комплексное 175, 233—245
— конфигурации 235
— надежности/готовности 236
— настройки 175, 236, 248—249
— объема 234—235
— приемлемости 175, 248
— производительности 236
— психологических факторов 237
— публикаций 237
— регрессии 220, 232
— совместимости 235
— сопряжений 175, 229—233
— средств восстановления 236
— стрессов 234
— требований к памяти 236
— удобства обслуживания 237
— удобства эксплуатации 237
Требования к программному обеспечению 37, 49—53, 248, 265
Тэг 115

Управляемый пользователем проект 49
Условия верификации 311—327
Устойчивость к ошибкам 33, 35, 120—121
Утверждение, утвердительное предложение 296, 311

Фабрика программного обеспечения 347—348
Фоннеймановская модель ЭВМ 300—308
Функциональные границы 203
— диаграммы 75, 221—229
Функция модуля 93

Цели программного обеспечения 37, 45, 53—62
— продукта 54, 59—60
— проекта 54, 60—61

Чувствительность к данным 179—180, 202—203, 209

Эффективность 57, 59, 69, 160—161, 236, 298—299
Эшкрофта — Манны метод 137

Языки программирования 134—137, 146—147, 159—160, 257—258, 275—296, 320

ADS 50
ASES 215—216, 259
AUT 212—214
AUTOFLOW 259

B1700 308
B6500 302

Cross-Program Analizer 259

DACC 351
DB/DC 244
DOS/VS 261
DSS 256

EFFIGY 325—327
EXDAMS 256

GO TO 134, 137—138, 281, 290—291

HIPO-диаграммы 53, 150, 268

IEBDG 219
Information Algebra 50
ISPL 302, 304

MIL-S 212
MTS 214
Multics, 147, 256, 305
MUSE 244
NORAD 244
n плюс — минус один 46—47, 61, 73, 90, 112

OS/VS1 256, 340
OS/VS2 116, 231, 256, 340
OS/360 97, 119, 123, 124, 196, 230, 236

PACE 216
PANVALET 345—346
PERT-диаграммы 183
PET 216
PL/C 257
PRIME 116—118
PROW 326

RESPOND 244
RXVP 215
R-2 302

SAGE 17
SBS 346—347
Series—J 216
Simon 349
SMP 260
SUE 280, 285, 295
SYMBOL 307
SYSTEM-250 304

TAG 50
TESTMASTER 214—215
THE 81—82
TITAN 179
TSO 239
TSPS 121—122
TSS 161

VISTA 323—324
VM/370 245

SOFTWARE RELIABILITY

PRINCIPLES AND PRACTICES

Glenford J. Myers

Staff Member,
IBM Systems Research Institute
Lecturer in Computer Science,
Polytechnic Institute of New York

A WILEY-INTERSCIENCE PUBLICATION
JOHN WILEY & SONS
NEW YORK • LONDON • SYDNEY • TORONTO
1976

Г. Майерс

Надежность программного обеспечения

Перевод с английского
Ю. Ю. Галимова

под редакцией
В. Ш. Кауфмана

Издательство «Мир». Москва 1980

УДК 681.142.2

Первая на русском языке монография по надежности программного обеспечения. Разработка программного комплекса рассматривается в книге во всей полноте — от подготовки технического задания до документации отдельных модулей. Ценно, что теоретическое всследование каждой проблемы сопровождается конкретными рекомендациями по ее решению. Каждая глава завершается библиографией, содержащей важнейшие работы по затронутым в главе вопросам.

Многие специалисты — от программистов до руководителей программистских коллективов — извлекут из этого издания несомненную пользу.

Редакция литературы по математическим наукам

1502000000

М $\frac{20204\text{-}026}{041\ (01)\text{-}80}$ 26-80

© 1976 by John Wiley and Sons, Inc. All Rights Reserved.
Authorized translation from English language edition published by John Wiley and Sons, Inc.
© Перевод на русский язык, «Мир», 1980

Г. МАЙЕРС

Надежность программного обеспечения

Научные редакторы И. А. Маховая, С. В. Чудов. Мл. научный редактор Л. С. Суркова
Художник В. А. Медников. Художественный редактор В. И. Шаповалов
Технический редактор Е. С. Потапенкова. Корректор В. И. Постнова

ИБ № 2087

Сдано в набор 28.02.80. Подписано к печати 25.06.80. Формат 60×90/16. Бумага типографская № 2. Гарнитура латинская. Печать высокая. Объем 11,25 бум. л. Усл. печ.л 22,50. Уч.-изд. л. 22,93. Изд. № 1/0418. Тираж 23 тыс. экз. Зак. № 756. Цена 1 р. 80 к.

ИЗДАТЕЛЬСТВО «МИР» Москва, 1-й Рижский пер., 2.

Отпечатано в Ленинградской типографии № 2 головном предприятии ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29 с матриц ордена Октябрьской Революции и ордена Трудового Красного Знамени Первой Образцовой типографии имени А. А. Жданова Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. Москва, М-54, Валовая, 28