

ИСКУССТВО  
ТЕСТИРОВАНИЯ  
ПРОГРАММ

Г. Мауерс

**ИСКУССТВО  
ТЕСТИРОВАНИЯ  
ПРОГРАММ**

**THE ART OF  
SOFTWARE  
TESTING**

**GLENFORD J. MYERS**

**A Wiley — Interscience Publication  
John Wiley & Sons  
New York · Chichester · Brisbane · Toronto**

**Г. Майерс**

# **ИСКУССТВО ТЕСТИРОВАНИЯ ПРОГРАММ**

Перевод с английского под редакцией  
*Б. А. Позина*

Москва «Финансы и статистика» 1982

ББК 32.973  
М14

Переводчики: С. А. Блау, С. Г. Орлов, Б. А. Позин,  
Л. С. Черняк.

**Майерс Г.**

М14 Искусство тестирования программ/Пер. с англ.  
под ред. **Б. А. Позина.** — М.: Финансы и статисти-  
ка, 1982. — 176 с., ил.  
60 к.

В книге широко и достаточно популярно излагаются основные  
принципы методологии тестирования и отладки программ. Рассматри-  
ваются вопросы психологии и экономики тестирования. Значительное  
место отводится методам корректировки программ.

Для специалистов, занимающихся программированием, студентов  
и аспирантов, изучающих ЭВМ.

М 2405000000—098  
010(01)—82 152—82

ББК 32.973  
6Ф7.3

© John Wiley & Sons, Inc., 1979.

© Перевод на русский язык, предисловие, «Финансы и статистика»,  
1982

## ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Книга Г. Майерса «Искусство тестирования программ» посвящена важному и быстро развивающемуся разделу современной технологии программирования. Эта книга может рассматриваться как методическое пособие по тестированию и отладке, рассчитанное на широкий круг специалистов различной квалификации, занимающихся разработкой программ для ЭВМ. Ее основным достоинством является систематичность изложения методов тестирования программ и подготовки программных тестов. Материал иллюстрирован большим числом примеров, способствующих практическому освоению методов тестирования.

Понятия процесса программирования и его объекта разработки — программы качественно изменились за последнее десятилетие. Производство программ приобрело массовый характер, существенно увеличились их средний объем и сложность. Разработка программных комплексов потребовала значительных усилий больших коллективов специалистов. Программы перестали быть только вычислительными и начали выполнять важнейшие функции по управлению и обработке информации в различных отраслях народного хозяйства. Эти и ряд других обстоятельств привели к принципиальному изменению подхода к программам. Программные комплексы не рассматриваются больше лишь как результат научного творчества и искусства. Доминирующим стал подход к программам как к объекту, являющемуся следствием сложного технологического процесса и *программным продуктом*. Появилась необходимость определять качество этого продукта и затраты труда, позволяющие достичь требуемого уровня качества. Возросла необходимость в методах и средствах, обеспечивающих сочетание высокого качества создаваемого программного продукта с короткими сроками разработки и достаточно

высокой производительностью труда коллективов специалистов.

Создание комплексов программ для управления объектами и технологическими процессами, а также для обработки информации в реальном масштабе времени постепенно приобретает характер промышленного производства. В результате возрастает роль технологии разработки программ и различных средств поддержки технологического процесса их проектирования. В состав компонент технологии входят правила и методики организаций коллективов для проектирования; средства методической поддержки процесса проектирования — инструкции по программированию, отладке, документированию и испытаниям программ; средства инструментальной поддержки и автоматизации технологического процесса. Все эти средства позволяют унифицировать разработку отдельных программных компонент и обеспечить их взаимное объединение в структурно упорядоченные комплексы программ с определенной целевой задачей. При этом возможно сбалансированное использование вычислительных ресурсов ЭВМ для решения частных задач с целью наиболее эффективного решения основной целевой задачи данного комплекса программ.

Развитие и применение технологий проектирования комплексов программ приводит к необходимости измерения и сравнения их эффективности прежде всего по степени влияния на качество программного продукта. Показатели качества программ подлежат измерению и численной оценке, для чего они формализуются вводом соответствующих метрик. Применение метрик к комплексам программ позволяет упорядочить их разработку, испытание, эксплуатацию и сопровождение. Взаимодействие разработчика и заказчика (или пользователя) приобретает более четкий характер, и качество программ может оцениваться количественно.

Обеспечение высокого качества сложных комплексов программ связано со значительными затратами труда разработчиков. Средняя производительность труда программистов последние 10 лет практически не увеличивается и остается примерно на уровне 70-х годов, несмотря на совершенствование технологии программирования. Это обусловлено непрерывным возрастанием сложности программных комплексов, вследствие чего стоимость разработки одной команды в программе почти не изме-

няется. Затраты на создание программ быстро увеличиваются при возрастании требований к их качеству. Поэтому на первый план выдвигается задача определения взаимосвязи трудоемкости разработки и различных показателей качества. Оказывается, что для сложных комплексов программ весьма трудно достичь высокого качества функционирования, и после обеспечения общей работоспособности могут потребоваться годы труда для получения программ с необходимыми показателями качества. Одна, казалось бы, незначительная ошибка может резко ухудшить характеристики программы, например ее надежность, и в ряде случаев нужно приложить огромные усилия для обнаружения и устранения этой ошибки. Поэтому уже сегодня требуются методы и средства, которые позволили бы заметно повысить качество программ при относительно небольших затратах труда.

Тестирование — один из наиболее трудоемких этапов создания программ (его трудоемкость составляет от 30 до 60% общей трудоемкости). Кроме того, доля его стоимости в общей стоимости программ имеет тенденцию возрастать при увеличении сложности комплексов программ и повышении требований к их качеству. Однако в научной и учебной литературе по программированию процессу отладки уделяется сравнительно мало внимания. Частично это можно объяснить недостаточным для обобщений опытом в данной области, а частично — традициями в теории и практике программирования, в соответствии с которыми главный акцент делается на языки программирования и средства трансляции программ.

В предлагаемой книге Г. Майерса приводится ряд методических рекомендаций по отладке программ. Автор подчеркивает, что тестирование пока остается в значительной степени искусством. В книге обобщен накопленный в этой области опыт, отработаны методы тестирования, отмечаются пути автоматизации подобных работ, причем внимание концентрируется на *искусстве тестирования* в основном сравнительно небольших групп и автономных программ. Это весьма значительная область технологии разработки программ, и приводимые рекомендации особенно полезны для широкого круга специалистов, еще не овладевших современными методами отладки.

Хотелось бы рекомендовать советским читателям постараться увидеть за рамками рекомендаций в «искус-

стве» определенную науку и принципы построения средств автоматизации технологических процессов. Это особенно важно учитывать не только при разработке небольших программ разового использования, но и при массовом производстве программ разной сложности, подлежащих широкому тиражированию и сопровождению в течение многих лет. В последнем случае необходимо налаживать регламентированный и контролируемый технологический процесс с использованием, в частности, рекомендаций книги Г. Майерса. Отобранные рекомендации могут быть переведены в разряд обязательных технологических правил, выполнение которых целесообразно контролировать автоматизированно.

При отработке технологии проектирования программ следует четко выделять определенное (по возможности не очень большое) число правил отладки, обеспечивающих высокое качество программного продукта и снижающих затраты на его создание. Формализация технологии отладки с тщательным контролем процесса и создаваемого продукта должна способствовать повышению качества комплексов программ и более быстрому росту квалификации специалистов, участвующих в их создании.

Часть рекомендаций автор книги адресует руководителям проектов комплексов программ различной сложности и назначения. Эти рекомендации в ряде случаев весьма просты по форме, тем не менее они могут заметно повысить уровень организации проектирования программ. Известно, что среди многих специалистов — руководителей разработок комплексов программ сохраняется подход к их созданию без глубокой методической основы и упорядоченной регламентированной технологии всего процесса. В результате отдельные проекты разрабатываются долго, с большими затратами труда, и в ряде случаев программы оказываются недостаточно высокого качества.

Методологическая поддержка современного проектирования комплексов при весьма небольших затратах способна значительно влиять на производительность труда коллективов программистов. При этом основными являются психологические трудности, которые появляются при введении регламентирующих методик. С одной стороны, многим руководителям кажется, что их ввести очень просто, а поэтому они представляют собой мало-

эффективные средства, и этим не стоит заниматься. С другой стороны, высококвалифицированные программисты видят в методиках ограничение собственной творческой инициативы и посягательство на их высокое «искусство», а также накопленный опыт работы. В результате они пытаются убедить руководителей в ненужности методического регламентирования технологии проектирования программ. С этих позиций книга Г. Майерса полезна для руководителей не только молодых, но и опытных коллективов.

Предлагаемая книга благодаря упорядоченному изложению принципов отладки программ позволит квалифицированным программистам систематизировать свои знания и опыт. Особенно полезна книга начинающим программистам и студентам. Как уже отмечалось, традиционно внимание студентов акцентируется на языках программирования и трансляционных средствах. Однако при этом не учитывается, что собственно программирование — сравнительно небольшая часть процесса создания комплексов программ (10—20%). Отсутствуют учебные курсы отладки, тестирования, испытаний и сопровождения программ, так же как и курсы, отражающие весь технологический процесс их проектирования. Ряд разделов книги Г. Майерса может послужить методической основой для лекций и практических занятий по автономной отладке программ.

В книге глубоко освещаются вопросы автономной отладки программ и их детерминированного тестирования, т. е. тестирования, при котором известны и контролируются каждая комбинация исходных данных и соответствующие ей результаты исполнения программы. Но современная отладка комплексов программ по своим концепциям и методам значительно шире. Поэтому целесообразно кратко остановиться на методах стохастического и динамического тестирования.

В сложных комплексах программ трудно, а часто и невозможно перебрать все комбинации исходных данных и проверить функционирование программ на каждой из них. В таких случаях применяется *стохастическое тестирование*, при котором исходные тестовые данные задаются множествами случайных величин с определенными распределениями и для сравнения полученных результатов в качестве эталонов используются также распределения случайных величин.

Критерии качества отладки приобретают стохастический характер, и отсутствует анализ каждой реализации теста.

Стохастическое тестирование выполняет преимущественно контролирующие функции при значительно более широком варьировании тестовых значений, чем это доступно для детерминированного тестирования. При этом отдельные ошибки в программах могут быть не обнаружены, если они мало искажают средние статистические значения или распределения. В том случае, когда обнаруживаются области изменения тестовых значений, заметно искажающие статистические распределения результатов, приходится переходить к детерминированному тестированию для диагностики и локализации ошибки в программе.

В системах реального времени необходимо проверять функционирование комплексов программ при оперативном динамическом взаимодействии всех их компонент в процессе обработки различных исходных данных. Для этого применяется *динамическое тестирование* (комплексная динамическая отладка), в ходе которого проверяются исполнение программ и обработка исходных данных с учетом времени их поступления, длительности обработки, приоритетности, динамики использования памяти и взаимодействия с другими программами. Область варьирования тестовых значений еще больше расширяется (по сравнению со стохастическим тестированием) и соответственно усложняется проверка правильности реализации каждого тестового значения. Оценки качества программ становятся интегральными и динамическими. При обнаружении отклонений результатов исполнения программ от предполагавшихся эталонных для локализации ошибки приходится фиксировать время и переходить к статистическому стохастическому или детерминированному тестированию.

Представленная иерархия тестирования и отладки характерна для сложных комплексов программ реального времени. Стохастическое и динамическое тестирования требуют применения средств автоматизации при подготовке тестов и обработке результатов их исполнения. В этих случаях практически неприменимы ручная подготовка тестов и ручная проверка их по программе. Однако следует подчеркнуть высокую эффективность инспекций и ручного анализа, подробно описанных авто-

ром, на этапах детерминированной отладки модулей и групп программ.

При динамической отладке необходимы автоматические имитаторы стохастических наборов тестовых данных, изменяющихся по заданным динамическим законам. Для обработки результатов исполнения программ и сравнения с эталонными значениями широко применяются автоматические средства обработки. Вследствие этого средства обеспечения автоматизации стохастического и динамического тестирования могут быть сложнее, чем комплекс тестируемых программ. Рентабельность средств автоматизации стохастического и динамического тестирования зависит от функционального назначения создаваемого комплекса программ. Так, при отработке программ для управления в авиации и в космосе без таких средств невозможно получить программы требуемого уровня качества.

При переводе книги Г. Майерса возник ряд трудностей, связанных с отсутствием установившейся терминологии в некоторых областях проектирования программ. Во избежание неудачных дословных переводов или транслитераций английских полужаргонных технических терминов переводчики старались либо применять только уже апробированные термины, либо раскрыть существо используемых понятий. Так, вместо термина «тестирование стрессов» (stress testing) здесь приводится термин «тестирование на предельных нагрузках». Аналогично комплексное тестирование (system testing) переводится как тестирование системы и т. д. Список литературы дополнен основными работами советских специалистов и важнейшими публикациями последних лет.

В заключение следует еще раз подчеркнуть, что искусство тестирования программ быстро превращается в область науки проектирования сложных комплексов программ, удовлетворяющих заданному уровню качества. У этой новой области прикладной науки формируются предмет исследований, методы проектирования, технология и средства автоматизации разработки высококачественного программного продукта. Происходит уточнение основных понятий и определений, формируется терминология. Тестирование и отладка приобретают характер коллективных работ по определенной технологии, существенно определяющих количество и качество программного продукта. Методы и средства тестирова-

ния необходимо рассматривать совместно с показателями качества программ, которые могут быть получены при определенных трудовых затратах. Задачи анализа эффективности отладки становятся важнейшими при создании сложных комплексов программ с высокими характеристиками качества.

*Профессор В. В. Липаев*

## ПРЕДИСЛОВИЕ

Как известно, при создании типичного программного проекта около 50% общего времени и более 50% общей стоимости расходуется на проверку (тестирование) разрабатываемой программы или системы. Основываясь на данном факте, можно было бы предположить, что к настоящему времени тестирование программ поднялось до уровня точной науки. Однако это не так. На самом деле тестирование программ освещено, пожалуй, меньше, чем любой другой аспект разработки программного обеспечения. К тому же тестирование было «немодным» предметом, если иметь в виду спорность публикаций по этому вопросу.

Изложенное может служить достаточным обоснованием для написания книги «Искусство тестирования программ». Не раз профессора и преподаватели говорили мне: «Наши выпускники приходят в промышленность, не имея практического представления о том, как приняться за тестирование программы. Более того, в существующих вводных курсах не дается никаких советов студентам по тестированию и отладке их учебных программ».

Таким образом, цель настоящей книги — заполнить этот пробел в знаниях профессиональных программистов и студентов, изучающих вычислительную технику. Как следует из названия, книга представляет собой не теоретические основы тестирования программ, а скорее практическое руководство, позволяющее читателю сравнительно легко освоить рассматриваемый предмет. Поэтому многие вопросы, связанные с тестированием программ, такие, как математическое доказательство их корректности, здесь намеренно не обсуждаются.

В первой главе дается короткий тест для самопроверки, который должен выполнить каждый читатель перед чтением книги. Оказывается, что наиболее важными практическими сведениями, необходимыми для тестирования программ, являются философские и экономические

принципы, обсуждаемые в гл. 2. В гл. 3 развиваются такие важные идеи, как сквозные просмотры и проверки тестов без применения ЭВМ. Хотя основное внимание здесь уделяется процедурному и управленческому аспектам этих методов, они рассмотрены и с точки зрения техники поиска ошибок.

Хорошо осведомленный читатель представляет себе, что искусство тестирования находится в прямой зависимости от умения программиста разрабатывать эффективные тесты; это является предметом рассмотрения гл. 4. В гл. 5 и 6 приводятся соответственно методы тестирования отдельных модулей (или подпрограмм) и их объединений, а в гл. 7 — некоторые практические рекомендации по отладке программ.

Данная книга предназначена для трех основных групп читателей. Профессиональные программисты, образующие первую группу, вряд ли найдут здесь много новой информации, но тем не менее смогут углубить свои знания в части методов тестирования. Если полученные знания позволят кому-либо выявить в программе на одну ошибку больше, то ценность книги возрастет во много раз. Вторую группу читателей составляют руководители программных проектов. Для них эта книга представляет новый практический материал по управлению процессом тестирования. К третьей группе относятся студенты, специализирующиеся по программированию и вычислительной технике. Ознакомившись с материалом книги, они смогут понять проблему тестирования программ и получат набор эффективных методов. Данную книгу можно предложить как дополнение к курсу программирования с тем, чтобы ввести студента в предмет тестирования программного обеспечения за краткое время его обучения.

*Гленфорд Дж. Майерс*

Нью-Йорк, июль 1978 г.

## ГЛАВА 1

### ТЕСТ ДЛЯ САМООЦЕНКИ

Прежде чем приступить к работе над книгой, рекомендуем читателю выполнить следующий простой тест. Задача состоит в том, чтобы проверить некоторую программу.

Эта программа производит чтение с перфокарты трех целых чисел, которые интерпретируются как длины сторон треугольника. Далее программа печатает сообщение о том, является ли треугольник неравносторонним, равнобедренным или равносторонним.

Напишите на листе бумаги набор тестов (т. е. специальные последовательности данных), которые, как вам кажется, будут адекватно проверять эту программу. Построив свои тесты, проанализируйте их.

Следующий шаг состоит в оценке эффективности вашей проверки. Оказывается, что программу труднее написать, чем это могло показаться вначале. Были изучены различные версии данной программы и составлен список общих ошибок. Оцените ваш набор тестов, попытавшись с его помощью ответить на приведенные ниже вопросы. За каждый ответ «да» присуждается одно очко.

1. Составили ли вы тест, который представляет правильный неравносторонний треугольник? (Заметим, что ответ «да» на тесты, со значениями 1, 2, 3 и 2, 5, 10 не обоснован, так как не существует треугольников, имеющих такие стороны.)

2. Составили ли вы тест, который представляет правильный равносторонний треугольник?

3. Составили ли вы тест, который представляет правильный равнобедренный треугольник? (Тесты со значениями 2, 2, 4 принимать в расчет не следует.)

4. Составили ли вы по крайней мере три теста, которые представляют правильные равнобедренные треугольники, полученные как перестановки двух равных сторон треугольника (например, 3, 3, 4; 3, 4, 3 и 4, 3, 3)?

5. Составили ли вы тест, в котором длина одной из сторон треугольника принимает нулевое значение?

6. Составили ли вы тест, в котором длина одной из сторон треугольника принимает отрицательное значение?
  7. Составили ли вы тест, включающий три положительных целых числа, сумма двух из которых равна третьему? (Другими словами, если программа выдала сообщение о том, что числа 1, 2, 3 представляют собой стороны неравнобедренного треугольника, то такая программа содержит ошибку.)
  8. Составили ли вы по крайней мере три теста с заданными значениями всех трех перестановок, в которых длина одной стороны равна сумме длин двух других сторон (например, 1, 2, 3; 1, 3, 2 и 3, 1, 2)?
  9. Составили ли вы тест из трех целых положительных чисел, таких, что сумма двух из них меньше третьего числа (т. е. 1, 2, 4 или 12, 15, 30)?
  10. Составили ли вы по крайней мере три теста из категории 9, в которых вами испытаны все три перестановки (например, 1, 2, 4; 1, 4, 2 и 4, 1, 2)?
  11. Составили ли вы тест, в котором все стороны треугольника имеют длину, равную нулю (т. е. 0, 0, 0)?
  12. Составили ли вы по крайней мере один тест, содержащий нецелые значения?
  13. Составили ли вы хотя бы один тест, содержащий неправильное число значений (например, два, а не три целых числа)?
  14. Специфицировали ли вы в каждом тесте не только входные значения, но и выходные данные программы?
- Конечно, нет гарантий, что с помощью набора тестов, который удовлетворяет вышеперечисленным условиям, будут найдены все возможные ошибки. Но поскольку вопросы 1—13 представляют ошибки, имевшие место в различных версиях данной программы, адекватный тест для нее должен их обнаруживать. Если вы не программист, то вы не очень хорошо справитесь с составлением теста. Для сравнения отметим, что опытные профессиональные программисты набирают в среднем только 7—8 очков из 14 возможных. Выполненное упражнение показывает нам, что тестирование даже тривиальных программ, подобных приведенной, — непростая задача. И коль скоро это так, рассмотрим трудности тестирования программ размером в 100000 операторов для системы управления воздушным движением, компилятора или программы расчета заработной платы.

## ПСИХОЛОГИЯ И ЭКОНОМИКА ТЕСТИРОВАНИЯ ПРОГРАММ

Тестирование как объект изучения может рассматриваться с различных чисто технических точек зрения. Однако наиболее важными при изучении тестирования представляются вопросы его экономики и психологии разработчика. Иными словами, достоверность тестирования программы в первую очередь определяется тем, кто будет ее тестировать и каков его образ мышления, и уже затем определенными технологическими аспектами. Поэтому, прежде чем перейти к техническим проблемам, мы остановимся на этих вопросах.

Вопросы экономики и психологии освещаются здесь всего на нескольких страницах. Но, разобравшись в них, мы легко решим все остальные проблемы тестирования.

Поначалу может показаться тривиальным жизненно важный вопрос определения термина «тестирование». Необходимость обсуждения этого термина связана с тем, что большинство специалистов используют его неверно, а это в свою очередь приводит к плохому тестированию. Таковы, например, следующие определения: «Тестирование представляет собой процесс, демонстрирующий отсутствие ошибок в программе», «Цель тестирования — показать, что программа корректно исполняет предусмотренные функции», «Тестирование — это процесс, позволяющий убедиться в том, что программа выполняет свое назначение».

Эти определения описывают нечто *противоположное* тому, что следует понимать под тестированием, поэтому они неверны. Оставив на время определения, предположим, что если мы тестируем программу, то нам нужно добавить к ней некоторую новую стоимость (т. е. тести-

рование стоит денег и нам желательно возвратить затраченную сумму путем увеличения стоимости программы). Увеличение стоимости означает повышение качества или возрастание надежности программы. Последнее связано с обнаружением и удалением из нее ошибок. Следовательно, программа тестируется не для того, чтобы показать, что она работает, а скорее наоборот — тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживаются их максимально возможное число. Таким образом, сформулируем наиболее приемлемое определение:

*Тестирование — это процесс исполнения программы с целью обнаружения ошибок.*

Пока все наши рассуждения могут показаться тонкой игрой семантик, однако практикой установлено, что именно ими в значительной мере определяется успех тестирования. Дело в том, что верный выбор цели дает важный психологический эффект, поскольку для человеческого сознания характерна целевая направленность. Если поставить целью демонстрацию отсутствия ошибок, то мы подсознательно будем стремиться к этой цели, выбирая тестовые данные, на которых вероятность появления ошибки мала. В то же время если нашей задачей станет обнаружение ошибок, то создаваемый нами тест будет обладать большей вероятностью обнаружения ошибки. Такой подход заметнее повысит качество программы, чем первый.

Из приведенного определения тестирования вытекает несколько следствий, которые обсуждаются в различных разделах книги. Например, одно из них состоит в том, что тестирование — процесс *деструктивный* (т. е. обратный созидательному, конструктивному). Именно этим и объясняется, почему многие считают его трудным. Большинство людей склонны к конструктивному процессу созидания объектов и в меньшей степени — к деструктивному процессу разделения на части. Из определения следует также, как нужно строить набор тестовых данных и кто должен (а кто не должен) тестировать данную программу.

Для усиления определения тестирования проанализируем два понятия «удачный» и «неудачный» и, в частности, их использование руководителями проектов при оценке результатов тестирования. Большинство руково-

дителей проектов называют тестовый прогон неудачным, если обнаружена ошибка, и, наоборот, удачным, если он прошел без ошибок. Чаще всего это является следствием ошибочного понимания термина «тестирование», так как, по существу, слово «удачный» означает «результативный», а слово «неудачный» — «нежелательный», «нерезультативный». Но если тест не обнаружил ошибки, его выполнение связано с потерей времени и денег, и термин «удачный» никак не может быть применен к нему. (Читатель поймет, что это утверждение справедливо лишь при предварительном обсуждении, поскольку мы не можем до исполнения знать, приведет тест к обнаружению ошибки или нет. Однако данный вопрос рассматривается позднее.)

Тестовый прогон, приведший к обнаружению ошибки, нельзя назвать неудачным хотя бы потому, что, как отмечалось выше, это целесообразное вложение капитала. Отсюда следует, что в слова «удачный» и «неудачный» необходимо вкладывать смысл, обратный общепринятому. Поэтому в дальнейшем будем называть тестовый прогон удачным, если в процессе его выполнения обнаружена ошибка, и неудачным, если получен корректный результат.

Проведем аналогию с посещением больным врача. Если рекомендованное врачом лабораторное исследование не обнаружило причины болезни, не назовем же мы такое исследование удачным — оно неудачно: ведь счет пациента сократился долларов на сорок, а он все так же болен. Если же исследование показало, что у больного язва желудка, то оно является удачным, поскольку врач может прописать необходимый курс лечения. Следовательно, медики используют эти термины в нужном нам смысле. (Аналогия здесь, конечно, заключается в том, что программа, которую предстоит тестировать, подобна больному пациенту.)

Определения типа «тестирование представляет собой процесс демонстрации отсутствия ошибок» порождают еще одну проблему: они ставят цель, которая не может быть достигнута ни для одной программы, даже весьма тривиальной. (Если вы не можете согласиться с этим утверждением, то примите его пока на веру — ниже оно будет обсуждаться.) Результаты психологических исследований показывают, что если перед человеком ставится невыполнимая задача, то он работает хуже. Например, ес-

ли предложить кому-то решить кроссворд в воскресном номере «Нью-Йорк Таймс» за 15 минут, то через 10 минут не будет достигнут значительный успех; ведь понятно, что это невыполнимая задача. Если же на решение отводится четыре часа, то через 10 минут результат окажется лучше. Иными словами, определение тестирования как процесса обнаружения ошибок переводит его в разряд решаемых задач и таким образом преодолевается психологическая трудность.

Другая проблема возникает в том случае, когда для тестирования используется следующее определение: «Тестирование — это процесс, позволяющий убедиться в том, что программа выполняет свое назначение», поскольку программа, удовлетворяющая данному определению, может содержать ошибки. Если программа не делает того, что от нее требуется, то ясно, что она содержит ошибки. Однако ошибки могут быть и тогда, когда она *делает то, что от нее не требуется*. Рассмотрим задачу о треугольнике из гл. 1. Программа, корректно определяющая неравносторонние, равносторонние и равнобедренные треугольники, все же допустит ошибку, если будет делать то, что она не должна делать (например, сообщать, что тройка 1, 2, 3 представляет неравносторонний треугольник, а тройка 0, 0, 0 — равнобедренный). Ошибки этого класса можно обнаружить скорее, если рассматривать тестирование как процесс поиска ошибок, а не демонстрацию корректности работы.

Подводя итог, можно сказать, что тестирование представляется деструктивным процессом попыток обнаружения ошибок в программе (наличие которых предполагается). Набор тестов, способствующий обнаружению ошибки, считается удачным. Естественно, в конечном счете каждый с помощью тестирования хочет добиться определенной степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена, но лучшим средством для достижения этой цели является непосредственный поиск ошибок. Допустим, кто-то обращается к вам с заявлением: «Моя программа великолепна» (т. е. не содержит ошибок). Лучший способ доказать справедливость подобного утверждения — попытаться его опровергнуть, обнаружить неточности, нежели просто согласиться с тем, что программа на определенном наборе входных данных работает корректно.

## ЭКОНОМИКА ТЕСТИРОВАНИЯ

Дав такое определение тестированию, необходимо на следующем шаге рассмотреть возможность создания теста, обнаруживающего *все* ошибки программы. Покажем, что ответ будет отрицательным даже для самых тривиальных программ. В общем случае невозможно обнаружить все ошибки программы. А это в свою очередь порождает экономические проблемы, задачи, связанные с функциями человека в процессе отладки, способы построения тестов.

Тестирование программы как черного ящика

Одним из способов изучения поставленного вопроса является исследование стратегии тестирования, называемой стратегией черного ящика, *тестированием с управлением по данным*, или *тестированием с управлением по входу-выходу*. При использовании этой стратегии программа рассматривается как черный ящик. Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации. Тестовые же данные используются только в соответствии со спецификацией программы (т. е. без учета знаний о ее внутренней структуре).

При таком подходе обнаружение всех ошибок в программе является критерием *исчерпывающего входного тестирования*. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных. Необходимость выбора именно этого критерия иллюстрируется следующим примером. Если в той же задаче о треугольниках один треугольник корректно признан равносторонним, нет никакой гарантии того, что все остальные равносторонние треугольники так же будут корректно идентифицированы. Так, для треугольника со сторонами 3842, 3842, 3842 может быть предусмотрена специальная проверка и он считается неравносторонним. Поскольку программа представляет собой черный ящик, единственный способ удовлетворения приведенному выше критерию — перебор всех возможных входных значений.

Таким образом, исчерпывающий тест для задачи о треугольниках должен включать равносторонние треугольники с длинами сторон вплоть до максимального

целого числа. Это, безусловно, астрономическое число, но и оно не обеспечивает полноту проверки. Вполне вероятно, что останутся некоторые ошибки, например программа может представить треугольник со сторонами 3, 4, 5 неравносторонним, а со сторонами 2,  $A$ , 2—равносторонним. Для того, чтобы обнаружить подобные ошибки, нужно перебрать не только все *разумные*, но и все вообще *возможные* входные наборы. Следовательно, мы приходим к выводу, что для исчерпывающего тестирования задачи о треугольниках требуется бесконечное число тестов.

Если такое испытание представляется сложным, то еще сложнее создать исчерпывающий тест для большой программы. Образно говоря, число тестов можно оценить «числом, большим, чем бесконечность». Допустим, что делается попытка тестирования методом черного ящика компилятора с Кобола. Для построения исчерпывающего теста нужно использовать все множество правильных программ на Коболе (фактически их число бесконечно) и все множество неправильных программ (т. е. действительно бесконечное число), чтобы убедиться в том, что компилятор обнаруживает все ошибки. Только в этом случае синтаксически неверная программа не будет компилирована. Если же программа имеет собственную память (например, операционная система, база данных или система резервирования билетов), то дело обстоит еще хуже. В таких программах исполнение команды (например, задания, запроса в базу данных, выполнение резервирования) зависит от того, какие события ей предшествовали, т. е. от предыдущих команд. Здесь следует перебрать не только все возможные команды, но и все их возможные последовательности.

Из изложенного следует, что построение исчерпывающего входного теста невозможно. Это подтверждается двумя аргументами: во-первых, нельзя создать тест, гарантирующий отсутствие ошибок; во-вторых, разработка таких тестов противоречит экономическим требованиям. Поскольку исчерпывающее тестирование исключается, нашей целью должна стать максимизация результативности капиталовложений в тестирование (иными словами, максимизация числа ошибок, обнаруживаемых одним тестом). Для этого мы можем рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией

достоверности предположения (например, разумно предположить, что если программа сочла треугольник 2, 2, 2 равносторонним, то таким же окажется и треугольник со сторонами 3, 3, 3). Этот вопрос обсуждается в гл. 4 при изучении стратегии построения тестов. Тестирование программы как белого ящика

Стратегия белого ящика, или стратегия тестирования, управляемого логикой программы, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы (к сожалению, здесь часто не используется спецификация программы).

Сравним способ построения тестов при данной стратегии с исчерпывающим входным тестированием стратегии черного ящика. Непосвященному может показаться, что достаточно построить такой набор тестов, в котором каждый оператор выполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь в детали (поскольку они рассматриваются в гл. 4), укажем лишь, что исчерпывающему входному тестированию может быть поставлено в соответствие *исчерпывающее тестирование маршрутов*. Подразумевается, что программа проверена полностью, если с помощью тестов удастся осуществить выполнение этой программы по всем возможным маршрутам ее потока (графа) передач управления.

Последнее утверждение имеет два слабых пункта. Один из них состоит в том, что число не повторяющихся друг друга маршрутов в программе — астрономическое. Чтобы убедиться в этом, рассмотрим представленный на рис. 2.1 граф передач управления простейшей программы. Каждая вершина, или кружок, обозначают участок программы, содержащий последовательность линейных операторов, которая может заканчиваться оператором ветвления. Дуги, оканчивающиеся стрелками, соответствуют передачам управления. По-видимому, граф описывает программу из 10—20 операторов, включая цикл DO, который выполняется не менее 20 раз. Внутри цикла имеется несколько операторов IF. Для того чтобы определить число неповторяющихся маршрутов при исполнении программы, подсчитаем число неповторяющихся маршрутов из точки А в В в предположении, что все приказы взаимно независимы. Это число вычисляется как сумма  $5^{20} + 5^{19} + \dots + 5^1 = 10^{14}$ , или 100 триллионам,

где 5—число путей внутри цикла. Поскольку большинству читателей трудно оценить это число, приведем такой пример: если допустить, что на составление каждого теста мы тратим пять минут, то для построения набора тестов нам потребуется примерно один миллиард лет.

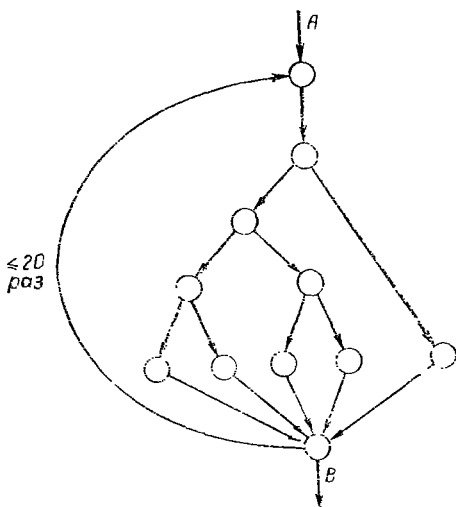


Рис. 2.1. Граф передач управления не-  
большой программы

Конечно, в реальных программах условные переходы не могут быть взаимно независимы, т. е. число маршрутов исполнения будет несколько меньше. С другой стороны, реальные программы значительно больше, чем простая программа, представленная на рис. 2.1. Следовательно, исчерпывающее тестирование маршрутов, как и исчерпывающее входное тестирование, не только невыполнимо, но и невозможно.

Второй слабый пункт утверждения заключается в том, что, хотя исчерпывающее тестирование маршрутов является полным тестом и хотя каждый маршрут программы может быть проверен, сама программа будет содержать ошибки. Это объясняется следующим образом. Во-первых, исчерпывающее тестирование маршрутов не может дать гарантии того, что программа соответствует описанию. Например, вместо требуемой программы сортировки по возрастанию случайно была написана про-

грамма сортировки по убыванию. В этом случае ценность тестирования маршрутов невелика, поскольку после тестирования в программе окажется одна ошибка, т. е. программа неверна. Во-вторых, программа может быть неверной в силу того, что пропущены некоторые маршруты. Исчерпывающее тестирование маршрутов не обнаружит их отсутствия. В-третьих, исчерпывающее тестирование маршрутов не может обнаружить ошибок, *появление которых зависит от обрабатываемых данных.* Существует множество примеров таких ошибок. Приведем один из них. Допустим, в программе необходимо выполнить сравнение двух чисел на сходимость, т. е. определить, является ли разность между двумя числами меньше предварительно определенного числа. Может быть написано выражение

IF ((A — B) < EPSILON) ...

Безусловно, оно содержит ошибку, поскольку необходимо выполнить сравнение абсолютных величин. Однако обнаружение этой ошибки зависит от значений, использованных для A и B, и ошибка не обязательно будет обнаружена просто путем исполнения каждого маршрута программы.

В заключение отметим, что, хотя исчерпывающее входное тестирование предпочтительнее исчерпывающего тестирования маршрутов, ни то, ни другое не могут стать полезными стратегиями, потому что оба они нереализуемы. Возможно, поэтому реальным путем, который позволит создать хорошую, но, конечно, не абсолютную стратегию, является сочетание тестирования программы как черного и как белого ящиков. Этот вопрос обсуждается в гл. 4.

## ПРИНЦИПЫ ТЕСТИРОВАНИЯ

Сформулируем основные принципы тестирования, используя главную предпосылку настоящей главы о том, что наиболее важными в тестировании программ являются вопросы психологии. Эти принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания.

*Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора.*

Нарушение этого очевидного принципа представляет одну из наиболее распространенных ошибок. Ошибочные, но правдоподобные результаты могут быть признаны правильными, если результаты теста не были заранее определены. Здесь мы сталкиваемся с явлением психологии: мы видим то, что мы хотим увидеть. Другими словами, несмотря на то что тестирование по определению — деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных переменных заранее при разработке теста. Поэтому тест должен включать две компоненты: описание входных данных и описание точного и корректного результата, соответствующего набору входных данных.

Необходимость этого подчеркивал логик Копи в работе [1]: «Проблема может быть охарактеризована как факт или группа фактов, которые не имеют приемлемого объяснения, которые кажутся необычными или которые не удается подогнать под наши представления или предположения. Очевидно, что если что-нибудь подвергнется сомнению, то об этом должна иметься какая-то предварительная информация. Если нет предположений, то не может быть и неожиданных результатов».

*Следует избегать тестирования программы ее автором.<sup>1</sup>*

Этот принцип следует из обсуждавшихся ранее в данной главе положений, которые определяют тестирование как деструктивный процесс. После выполнения конструктивной части при проектировании и написании программы программисту трудно быстро (в течение одного дня) перестроиться на деструктивный образ мышления.

---

<sup>1</sup> К сожалению, реализация этого в целом верного принципа не всегда возможна в силу трех факторов: 1) людские ресурсы разработки, как правило, недостаточны; 2) для регулярного применения этого принципа к каждой программе требуется весьма высокая квалификация всех программистов или большой группы программистов, тестирующих все программы, что не всегда осуществимо; 3) необходим высокий уровень формализации ведения разработки: тщательные формализованные спецификации требований к программам и данным, тщательное описание интерфейса и формализация ответственности за качество продукта. В настоящее время проводится значительная работа по созданию и внедрению формализованных методов в большинстве крупных разработок, но опыт подобного ведения разработок пока еще недостаточно массовый. — *Примеч. ред.*

Многие, кому приходилось самому делать дома ремонт, знают, что процесс обрывания старых обоев (деструктивный процесс) не легок, но он просто невыносим, если не кто-то другой, а вы сами первоначально их наклеивали. Вот так же и большинство программистов не могут эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки.<sup>1</sup>

В дополнение к этой психологической проблеме следует отметить еще одну, не менее важную: программа может содержать ошибки, связанные с неверным пониманием постановки или описания задачи программистом. Тогда существует вероятность, что к тестированию программист приступит с таким же недопониманием своей задачи.

Тестирование можно уподобить работе корректора или рецензента над статьей или книгой. Многие авторы представляют себе трудности, связанные с редактированием собственной рукописи. Очевидно, что обнаружение недостатков в своей деятельности противоречит человеческой психологии.

Отсюда вовсе не следует, что программист *не может* тестировать свою программу. Многие программисты с этим вполне успешно справляются. Здесь лишь делается вывод о том, что тестирование является более эффективным, если оно выполняется кем-либо другим. Заметим, что все наши рассуждения не относятся к отладке, т. е. к исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

---

<sup>1</sup> Это действительно сильный психологический фактор при коллективной разработке. Программист, тщательно отлаживающий программу, незвольно может работать медленнее, что становится известно другим участникам разработки. С другой стороны, он вынужден запрашивать дополнительное машинное время на отладку у своего непосредственного руководителя. Тем самым итоги тестирования оказываются уже не просто делом одного человека, тестирующего программу (пока в большинстве случаев ее автора), но и информацией, возбуждающей общественный интерес (и оценку!) участников разработки, в том числе ее руководителей. Перспектива создать о себе мнение как о специалисте, делающем много ошибок, не воодушевляет программиста, и он подсознательно снижает требования к тщательности тестирования. В такой ситуации от руководителей разработки всех рангов требуется большое чувство такта и понимание процессов, чтобы поощрять специалистов, проводящих тщательное тестирование, и уметь различить и ограничить деятельность программистов, прикрывающих свою нерадивость трудностями тестирования. — *Примеч. ред.*

*Программирующая организация не должна сама тестировать разработанные ею программы.*

Здесь можно привести те же аргументы, что и в предыдущем случае. Во многих смыслах проектирующая или программирующая организация подобна живому организму с его психологическими проблемами. Работа программирующей организации или ее руководителя оценивается по их способности производить программы в течение заданного времени и определенной стоимости. Одна из причин такой системы оценок состоит в том, что временные и стоимостные показатели легко измерить, но в то же время чрезвычайно трудно количественно оценить надежность программы. Именно поэтому в процессе тестирования программирующей организации трудно быть объективной, поскольку тестирование в соответствии с данным определением может быть рассмотрено как средство уменьшения вероятности соответствия программы заданным временным и стоимостным параметрам.

Как и ранее, из изложенного не следует, что программирующая организация не может найти свои ошибки; тестирование в определенной степени может пройти успешно. Мы утверждаем здесь лишь то, что экономически более целесообразно выполнение тестирования каким-либо объективным, независимым подразделением<sup>1</sup>.

*Необходимо досконально изучать результаты применения каждого теста.*

По всей вероятности, это наиболее очевидный принцип, но и ему часто не уделяется должное внимание. В экспериментах, проверенных автором, многие испытываемые не смогли обнаружить определенные ошибки, хотя их признаки были совершенно явными в выходных листингах. Представляется достоверным, что значительная часть всех обнаруженных в конечном итоге ошибок могла быть выявлена в результате самых первых тестовых прогонов, но они были пропущены вследствие недостаточного тщательного анализа результатов первого тестового прогона.

---

<sup>1</sup> В некоторых организациях подобная практика существует, но только на этапах комплексной отладки. Подобный способ тестирования чрезвычайно сложно реализовать из-за организационных трудностей и пока еще недостаточного массового опыта формализованного ведения разработок. — *Примеч. ред.*

*Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных.*

При тестировании программ имеется естественная тенденция концентрировать внимание на правильных и предусмотренных входных условиях, а неправильным и непредусмотренным входным данным не придавать значения. Например, при тестировании задачи о треугольниках, приведенной во второй главе, лишь немногие смогут привести в качестве теста длины сторон 1, 2 и 5, чтобы убедиться в том, что треугольник не будет ошибочно интерпретирован как неравносторонний. Множество ошибок можно также обнаружить, если использовать программу новым, не предусмотренным ранее способом. Вполне вероятно, что тесты, представляющие неверные и неправильные входные данные, обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным.

*Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать.*

Это логически просто вытекает из предыдущего принципа. Необходимо проверить программу на нежелательные побочные эффекты. Например, программа расчета зарплаты, которая производит правильные платежные чеки, окажется неверной, если она произведет лишние чеки для работающих или дважды запишет первую запись в список личного состава.

*Не следует выбрасывать тесты, даже если программа уже не нужна.*

Эта проблема наиболее часто возникает при использовании интерактивных систем отладки. Обычно тестирующий сидит за терминалом, на лету придумывает тесты и запускает программу на выполнение. При такой практике работы после применения тесты пропадают. После внесения изменений или исправления ошибок необходимо повторять тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы. В результате повторное тестирование бывает менее тщательным, чем первоначальное, т. е. если модификация затронула функциональную часть программы и при этом была допущена ошибка, то она зачастую может остаться необнаруженной.

*Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены.*

Такую ошибку обычно допускают руководители проекта, использующие неверное определение тестирования как процесса демонстрации отсутствия ошибок в программе, корректного функционирования программы.

*Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.*

Этот принцип, не согласующийся с интуитивным представлением, иллюстрируется рис. 2.2. На первый

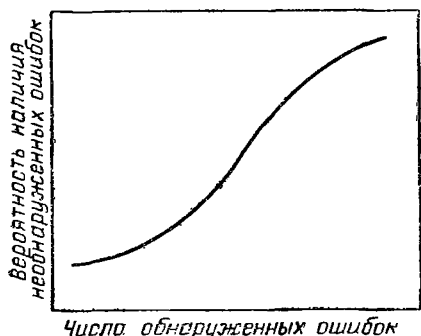


Рис. 2.2. Неожиданное соотношение числа оставшихся и числа обнаруженных ошибок

взгляд он лишен смысла, но тем не менее подтверждается многими программами. Например, допустим, что некоторая программа состоит из модулей или подпрограмм А и В. К определенному сроку в модуле А обнаружено пять ошибок, а в модуле В — только одна, причем модуль А не подвергался более тщательному тестированию.

Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле А больше, чем в модуле В. Справедливость этого принципа подтверждается еще и тем, что для ошибок свойственно располагаться в программе в виде неких скоплений, хотя данное явление пока никем еще не объяснено. В качестве примера можно рассмотреть операционные системы IBM S/370. В одной из версий операционной системы 47% ошибок, обнаруженных пользователями, приходилось на 4% модулей системы<sup>1</sup>.

Преимущество рассматриваемого принципа заключается в том, что он позволяет ввести обратную связь в

<sup>1</sup> Интуитивно понятно, что ошибки могут группироваться в частях программы (модулях), разрабатываемых программистами низкой квалификации, или в модулях, слабо проработанных идеологически. — *Примеч. ред.*

процесс тестирования. Если в какой-нибудь части программы обнаружено больше ошибок, чем в других, то на ее тестирование должны быть направлены дополнительные усилия.

*Тестирование — процесс творческий.*

Вполне вероятно, что для тестирования большой программы требуется больший творческий потенциал, чем для ее проектирования. Выше было показано, что нельзя дать гарантию построения теста, обнаруживающего все ошибки. В дальнейшем здесь будут обсуждаться методы построения хороших наборов тестов, но применение этих методов должно быть творческим.

Чтобы подчеркнуть некоторые мысли, высказанные в настоящей главе, приведем еще раз три наиболее важных принципа тестирования.

*Тестирование — это процесс выполнения программ с целью обнаружения ошибок.*

*Хорошим считается тест, который имеет высокую вероятность обнаружения еще не выявленной ошибки.*

*Удачным считается тест, который обнаруживает еще не выявленную ошибку.*

## ЛИТЕРАТУРА

1. Copi I. M. Introduction to Logic. New York, Macmillan, 1968.

## ГЛАВА 3

### ИНСПЕКЦИИ, СКВОЗНЫЕ ПРОСМОТРЫ И ОБЗОРЫ ПРОГРАММЫ

В течение многих лет большинство программистов были убеждены в том, что программы пишутся исключительно для выполнения их на машине и не предназначены для чтения человеком, а единственным способом тестирования программы является ее исполнение на ЭВМ. Это мнение стало изменяться в начале 70-х годов в значительной степени благодаря книге Вейнберга «Психология программирования для ЭВМ» [1]. Вейнберг показал, что программы должны быть удобочитаемыми и что их просмотр должен быть эффективным процессом обнаружения ошибок.

По этой причине прежде чем перейти к обсуждению традиционных методов тестирования, основанных на применении ЭВМ, рассмотрим процесс тестирования без применения ЭВМ («ручного тестирования»), являющийся темой настоящей главы. Эксперименты показали, что методы ручного тестирования достаточно эффективны с точки зрения нахождения ошибок, так что один или несколько из них должны использоваться в каждом программном проекте. Описанные здесь методы предназначены для периода разработки, когда программа закодирована, но тестирование на ЭВМ еще не началось. Аналогичные методы могут быть получены и применены на более ранних этапах процесса создания программ (т. е. в конце каждого этапа проектирования), но рассмотрение этого вопроса выходит за рамки данной книги. Некоторые из таких методов приводятся в работах [2] и [3].

Следует заметить, что из-за неформальной природы методов ручного тестирования (неформальной с точки зрения других, более формальных методов, таких, как математическое доказательство корректности программ)

первой реакцией часто является скептицизм, ощущение того, что простые и неформальные методы не могут быть полезными. Однако их использование показало, что они не «уводят в сторону». Скорее эти методы способствуют существенному увеличению производительности и повышению надежности программы. Во-первых, они обычно позволяют раньше обнаружить ошибки, уменьшить стоимость исправления последних и увеличить вероятность того, что корректировка произведена правильно. Во-вторых, психология программистов, по-видимому, изменяется, когда начинается тестирование на ЭВМ. Возрастает внутреннее напряжение и появляется тенденция «исправлять ошибки так быстро, как только это возможно». В результате программисты допускают больше промахов при корректировке ошибок, уже найденных во время тестирования на ЭВМ, чем при корректировке ошибок, найденных на более ранних этапах.

## ИНСПЕКЦИИ И СКВОЗНЫЕ ПРОСМОТРЫ

Инспекции исходного текста и сквозные просмотры являются основными методами ручного тестирования. Так как эти два метода имеют много общего, они рассматриваются здесь совместно. Различия между ними обсуждаются в последующих разделах.

Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Эти методы развиты из идей Вейнберга [1]. Оба метода предполагают некоторую подготовительную работу. Завершающим этапом является «обмен мнениями» — собрание, проводимое участниками проверки. Цель такого собрания — нахождение ошибок, но не их устранение (т. е. тестирование, а не отладка).

Инспекции и сквозные просмотры широко практикуются в настоящее время, но причины их успеха до сих пор еще недостаточно выяснены. Не удивительно, что они связаны с некоторыми принципами гл. 2. Заметим, что данный процесс выполняется группой лиц (оптимально три-четыре человека), лишь один из которых является автором программы. Следовательно, программа, по существу, тестируется не автором, а другими людьми, которые руководствуются изложенными выше принципами, обычно не эффективными при тестировании собственной программы. Фактически «инспекция» и «сквоз-

ной просмотр» — просто новые названия старого метода «проверки за столом» (состоящего в том, что программист просматривает свою программу перед ее тестированием), однако они гораздо более эффективны опять-таки по той же причине: в процессе участвует не только автор программы, но и другие лица. Результатом использования этих методов, по-видимому, также является более низкая стоимость отладки (исправления ошибок), так как во время поиска ошибок обычно точно определяют их природу. Кроме того, с помощью данных методов обнаруживают группы ошибок, что позволяет в дальнейшем корректировать сразу несколько ошибок. С другой стороны, при тестировании на ЭВМ обычно выявляют только *симптомы* ошибок (например, программа не закончилась или напечатала бессмысленный результат), а сами они определяются поодиночке.

Эксперименты по применению этих методов показали, что с их помощью для типичных программ можно находить от 30 до 70% ошибок логического проектирования и кодирования. (Однако эти методы не эффективны при определении ошибок проектирования «высокого уровня», например, сделанных в процессе анализа требований.) Так, экспериментально установлено, что при проведении инспекций и сквозных просмотров определяются в среднем 38% общего числа ошибок в учебных программах [4]. При использовании инспекций исходного текста в фирме IBM эффективность обнаружения ошибок составляет 80% [5] (в данном случае имеется в виду не 80% общего числа ошибок, поскольку, как отмечалось в гл. 2, общее число ошибок в программе никогда не известно, а 80% всех ошибок, найденных к моменту окончания процесса тестирования). Конечно, можно критиковать эту статистику в предположении, что ручные методы тестирования позволяют находить только «легкие» ошибки (те, которые можно просто найти при тестировании на ЭВМ), а трудные, незаметные или необычные ошибки можно обнаружить только при тестировании на машине. Однако проведенное исследование показало, что подобная критика является необоснованной [6]. В работе [4] делается также вывод о том, что при нахождении определенных типов ошибок ручное тестирование *более* эффективно, чем тестирование на ЭВМ, *в то время как для других типов ошибок верно противоположное утверждение.* Подразумевается, что инспекции, сквозные просмотры и

тестирование, основанное на использовании ЭВМ, дополняют друг друга; эффективность обнаружения ошибок уменьшится, если тот или иной из этих подходов не будет использован.

Наконец, хотя эти методы весьма важны при тестировании новых программ, они представляют не меньшую ценность при тестировании модифицированных программ. Опыт показал, что в случае модификации существующих программ вносится большее число ошибок (измеряемое числом ошибок на вновь написанные операторы), чем при написании новой программы. Следовательно, модифицированные программы также должны быть подвергнуты тестированию с применением данных методов.

## **ИНСПЕКЦИИ ИСХОДНОГО ТЕКСТА**

Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении (чтении) текста группой специалистов [7]. При рассмотрении инспекций исходного текста внимание будет сосредоточено в основном на процедурах, формах выполнения и т. д.; после краткого изложения основной процедуры мы изучим существующие методы обнаружения ошибок.

Инспектирующая группа включает обычно четыре человека, один из которых выполняет функции председателя. Председатель должен быть компетентным программистом, но не автором программы; он не должен быть знаком с ее деталями. В обязанности председателя входят подготовка материалов для заседаний инспектирующей группы и составление графика их проведения, ведение заседаний, регистрация всех найденных ошибок и принятие мер по их последующему исправлению. Председателя можно сравнить с инженером отдела технического контроля. Членами группы являются автор программы, проектировщик (если он не программист) и специалист по тестированию.

Общая процедура заключается в следующем. Председатель заранее (например, за несколько дней) раздает листинг программы и проектную спецификацию остальным членам группы. Они знакомятся с материалами, предшествующими заседанию. Инспекционное заседание разбивается на две части:

1. Программиста просят рассказать о логике работы программы. Во время беседы возникают вопросы, преследующие цель обнаружения ошибки. Практика показала, что даже только чтение своей программы слушателям представляется эффективным методом обнаружения ошибок и многие ошибки находит сам программист, а не другие члены группы.

2. Программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования (такой список будет рассмотрен в следующем разделе).

Председатель является ответственным за обеспечение результативности дискуссии. Ее участники должны сосредоточить свое внимание на нахождении ошибок, а не на их корректировке. (Корректировка ошибок выполняется программистом после инспекционного заседания.)

По окончании заседания программисту передается список найденных ошибок. Если список включает много ошибок или если эти ошибки требуют внесения значительных изменений, председателем может быть принято решение о проведении после корректировки повторной инспекции программы. Список анализируется и ошибки распределяются по категориям, что позволяет совершенствовать его с целью повышения эффективности будущих инспекций.

В большинстве примеров описания процесса инспектирования утверждается, что во время инспекционного заседания ошибки не должны корректироваться. Однако существует и другая точка зрения [8]: «Вместо того, чтобы сначала сосредоточиться на основных проблемах проектирования, необходимо решить второстепенные вопросы. Два или три человека, включая разработчика программы, должны внести очевидные исправления в проект с тем, чтобы впоследствии решить главные задачи. Однако обсуждение второстепенных вопросов сконцентрирует внимание группы на частной области проектирования. Во время обсуждения наилучшего способа внесения изменений в проект кто-либо из членов группы может заметить еще одну проблему. Теперь группе придется рассматривать две проблемы по отношению к одним и тем же аспектам проектирования, объяснения будут полными и быстрыми. В течение нескольких минут целая область проекта может быть полностью исследована и любые проблемы станут очевидными».

ми... Как упоминалось выше, многие важные проблемы, возникавшие во время обзоров блок-схем, были решены в результате многократных безуспешных попыток решить вопросы, которые на первый взгляд казались тривиальными».

Время и место проведения инспекции должны быть спланированы так, чтобы избежать любых прерываний инспекционного заседания. Его оптимальная продолжительность, по-видимому, лежит в пределах от 90 до 120 мин. Так как это заседание является экспериментом, требующим умственного напряжения, увеличение его продолжительности ведет к снижению продуктивности. Большинство инспекций происходит при скорости, равной приблизительно 150 операторам в час. При этом подразумевается, что большие программы должны рассматриваться за несколько инспекций, каждая из которых может быть связана с одним или несколькими модулями или подпрограммами.

Для того чтобы инспекция была эффективной, должны быть установлены соответствующие отношения. Если программист воспринимает инспекцию как акт, направленный лично против него, и, следовательно, занимает оборонительную позицию, процесс инспектирования не будет эффективным. Программист должен подходить к нему с менее эгоистических позиций [1]; он должен рассматривать инспекцию в позитивном и конструктивном свете: объективно инспекция является процессом нахождения ошибок в программе и таким образом улучшает качество его работы. По этой причине, как правило, рекомендуется результаты инспекции считать конфиденциальными материалами, доступными только участникам заседания. В частности, использование результатов инспекции руководством может нанести ущерб целям этого процесса.

Процесс инспектирования в дополнение к своему основному назначению, заключающемуся в нахождении ошибок, выполняет еще ряд полезных функций. Кроме того что результаты инспекции позволяют программисту увидеть сделанные им ошибки и способствуют его обучению на собственных ошибках, он обычно получает возможность оценить свой стиль программирования и выбор алгоритмов и методов тестирования. Остальные участники также приобретают опыт, рассматривая ошибки и стиль программирования других программистов.

Наконец, инспекция является способом раннего выявления наиболее склонных к ошибкам частей программы, позволяющим сконцентрировать внимание на этих частях в процессе выполнения тестирования на ЭВМ (один из принципов тестирования гл. 2).

### **СПИСОК ВОПРОСОВ ДЛЯ ВЫЯВЛЕНИЯ ОШИБОК ПРИ ИНСПЕКЦИИ**

Важной частью процесса инспектирования является проверка программы на наличие общих ошибок с помощью списка вопросов для выявления ошибок. Концентрация внимания в предлагаемом списке (как, например, в работе [9]) на рассмотрении стиля, а не ошибок (вопросы типа «Являются ли комментарии точными и информативными?» и «Располагаются ли символы THEN/ELSE и DO/END по одной вертикали друг под другом?») представляется неудачной, так же как и наличие неопределенности в списке, уменьшающее его полезность (вопросы типа «Соответствует ли текст программы требованиям, выдвигаемым при проектировании?»). Список, приведенный в данном разделе, был составлен автором после многолетнего изучения ошибок программного обеспечения. В значительной мере он является независимым от языка; это означает, что большинство ошибок встречается в любом языке программирования. Читатель может дополнить этот список вопросами, позволяющими выявить ошибки, специфичные для того языка программирования, который он использует, и обнаруженные им в результате выполнения процесса инспектирования.

#### **Ошибки обращения к данным**

1. Существуют ли обращения к переменным, значения которым не присвоены или не инициализированы? Наличие переменных с неустановленными значениями — наиболее часто встречающаяся программная ошибка, она возникает при различных обстоятельствах. Для каждого обращения к единице данных (например, к переменной, элементу массива, полю в структуре) попытайтесь неформально «доказать», что ей присвоено значение в проверяемой точке.

2. Не выходит ли значение каждого из индексов за границы, определенные для соответствующего измерения при всех обращениях к массиву?

3. Принимает ли каждый индекс целые значения при всех обращениях к массиву? Нецелые индексы не обязательно являются ошибкой для всех языков программирования, но представляют практическую опасность.

4. Для всех обращений с помощью указателей или переменных-ссылок память, к которой производится обращение, уже распределена? Наличие переменных-ссылок представляет собой ошибку типа «подвешенного обращения». Она возникает в ситуациях, когда время жизни указателя больше, чем время жизни памяти, к которой производится обращение. Например, к такому результату приводит ситуация, когда указатель задает локальную переменную в теле процедуры, значение указателя присваивается выходному параметру или глобальной переменной, процедура завершается (освобождая адресуемую память), а программа затем пытается использовать значение указателя. Как и при поиске ошибок первых трех типов, попытайтесь неформально доказать, что для каждого обращения, использующего переменную-указатель, адресуемая память существует.

5. Если одна и та же область памяти имеет несколько псевдонимов (имен) с различными атрибутами, то имеют ли значения данных в этой области корректные атрибуты при обращении по одному из этих псевдонимов? Ошибки типа некорректных атрибутов у псевдонимов могут возникнуть при использовании атрибута `DEFINED` или базированной памяти в `PL/1`, оператора `EQUIVALENCE` в Фортране и глагола `REDEFINES` в Коболе. В качестве примера можно привести программу на Фортране, содержащую вещественную переменную *A* и целую переменную *B*. Обе величины размещены на одном и том же месте памяти с помощью оператора `EQUIVALENCE`. Если программа записывает величину *A*, а обращается к переменной *B*, то, вероятно, произойдет ошибка, поскольку машина будет использовать битовое представление числа с плавающей точкой в данной области памяти как целое.

6. Отличаются ли типы или атрибуты переменных величин от тех, которые предполагались компилятором? Это может произойти в том случае, когда программа на `PL/1` или Коболе считывает записи из памяти и обращается к ним как к структурам, но физическое представление записей отлично от описания структуры.

7. Существуют ли явные или неявные проблемы ад-

ресации, если в машине будут использованы единицы распределения памяти, меньшие, чем единицы адресации памяти? Например, в PL/1 в системе IBM S/370 битовые строки фиксированной длины не обязательно начинаются с границ байтов, а адресами могут быть только границы байтов. Если программа вычисляет адрес битовой строки и впоследствии обращается к строке по этому адресу, то может произойти ошибочное обращение к памяти. Такое же положение может сложиться при передаче подпрограмме битовой строки в качестве аргумента.

8. Если используются указатели или переменные-ссылки, то имеет ли адресуемая память атрибуты, предполагаемые компилятором? Примером несоответствия атрибутов может служить случай, когда указатель PL/1, по которому базируется структура данных, имеет в качестве значения адрес другой структуры.

9. Если к структуре данных обращаются из нескольких процедур или подпрограмм, то определена ли эта структура одинаково в каждой процедуре?

10. Не превышены ли границы строки при индексации в ней?

11. Существуют ли какие-нибудь другие ошибки в операциях с индексацией или при обращении к массивам по индексу?

#### Ошибки описания данных

1. Все ли переменные описаны явно? Отсутствие явного описания не обязательно является ошибкой, но служит общим источником беспокойства. Так, если в подпрограмме на Фортране используется элемент массива и отсутствует его описание (например, в операторе DIMENSION), то обращение к массиву (например,  $X=A(I)$ ), будет интерпретироваться как вызов функции. Последнее приведет к тому, что машина будет пытаться обработать массив как программу. Если отсутствует явное описание переменной во внутренней процедуре или блоке, следует ли понимать это так, что описание данной переменной совпадает с описанием во внешнем блоке<sup>1</sup>?

2. Если не все атрибуты переменной явно присутствуют в описании, то понятно ли их отсутствие? Например,

---

<sup>1</sup> При разработке больших программных изделий неявное описание данных (описание данных по умолчанию) зачастую запрещают методически, чтобы упростить поиск ошибок при комплексной отладке. — *Примеч. ред.*

отсутствие атрибутов, считающееся общепринятым в PL/1, часто является источником неожиданных осложнений.

3. Если начальные значения присваиваются переменным в операторах описания, то правильно ли инициализируются эти значения? Во многих языках программирования присвоение начальных значений массивам и строкам представляется довольно сложным и, следовательно, является возможным источником ошибок.

4. Правильно ли для каждой переменной определены длина, тип и класс памяти (например, `STATIC`, `AUTOMATIC`, `BASED` или `CONTROLLED` в PL/1)?

5. Согласуется ли инициализация переменной с ее типом памяти? Например, если в подпрограмме на Фортране необходимо устанавливать начальные значения переменной каждый раз при вызове подпрограммы, переменная должна быть инициализирована в операторе описания, отличном от оператора `DATA`. Если в PL/1 описывается инициализация величины и начальное значение необходимо устанавливать каждый раз при вызове процедуры, то для этой переменной должен быть определен класс памяти `AUTOMATIC`, а не `STATIC`.

6. Есть ли переменные со сходными именами (например, `VOLT` и `VOLTS`)? Наличие сходных имен не обязательно является ошибкой, но служит признаком того, что имена могут быть перепутаны где-нибудь внутри программы.

#### Ошибки вычислений

1. Имеются ли вычисления, использующие переменные недопустимых (например, неарифметических) типов данных?

2. Существуют ли вычисления, использующие данные разного вида? Например, сложение переменной с плавающей точкой и целой переменной. Такие случаи не обязательно являются ошибочными, но они должны быть тщательно проверены для обеспечения гарантии того, что правила преобразования, принятые в языке, понятны. Это особенно важно для языков со сложными правилами преобразования (например, для PL/1). Например, следующий фрагмент программы на PL/1:

```
DECLARE A BIT(1);  
A=1;
```

определяет значение `A` равным битовому 0, а не 1.

3. Существуют ли вычисления, использующие переменные, имеющие одинаковый тип данных, но разную длину? Такой вопрос справедлив для PL/1 и возник он из этого языка. Например, в PL/1 результатом вычисления выражения  $25 + 1/3$  будет 5.333..., а не 25.333...<sup>1</sup>

4. Имеет ли результирующая переменная оператора присваивания атрибуты, описывающие ее с меньшей длиной, чем в атрибутах выражения в правой части?

5. Возможны ли персполнение или потеря результата во время вычисления выражения? Это означает, что конечный результат может казаться правильным, но промежуточный результат может быть слишком большим или слишком малым для машинного представления данных.

6. Возможно ли, чтобы делитель в операторе деления был равен нулю?

7. Если величины представлены в машине в двоичной форме, получают ли какие-нибудь результаты неточными? Так,  $10 \times 0,1$  редко равно 1,0 в двоичной машине.

8. Может ли значение переменной выходить за пределы установленного для нее диапазона? Например, для операторов, присваивающих значение переменной ВЕРОЯТНОСТЬ, может быть произведена проверка, будет ли полученное значение всегда положительным и не превышающим 1,0.

9. Верны ли предположения о порядке оценки и следования операторов для выражений, содержащих более чем один оператор?

10. Встречается ли неверное использование целой арифметики, особенно деления? Например, если  $I$  — целая величина, то выражение  $2*I/2=I$  зависит от того, является значение  $I$  четным или нечетным, и от того, ка-

---

<sup>1</sup> Автор здесь не вполне корректен: при вычислении этого выражения возникает ситуация FIXEDOVERFLOW. Действительно, рассмотрим атрибуты, присваиваемые компилятором константам, входящим в состав данного выражения. Константа 25 имеет тип DECIMAL FIXED (2,0), константы 1 и 3 — при DECIMAL FIXED (1,0), величина  $1/3$  — тип DECIMAL FIXED (15,14), поскольку ее значение 0,33...3, а выражение  $25 + 1/3$  — атрибуты DECIMAL FIXED (15,14). При таких атрибутах старший десятичный разряд результата не может быть размещен верно — возникает ситуация FIXEDOVERFLOW, т. е. результат неверный, он вовсе не равен 5.333..., как пишет автор. Верный результат достигается при записи этого выражения, например, в виде  $25 + \emptyset\emptyset 1/3$ . — *Примеч. ред.*

кое действие — умножение или деление — выполняется первым.

Ошибки при сравнениях

1. Сравняются ли в программе величины, имеющие несовместимые типы данных (например, строка символов с адресом)?

2. Сравняются ли величины различных типов или величины различной длины? Если да, то проверьте, правильно ли интерпретируются (поняты) правила преобразования.

3. Корректны ли операторы сравнения? Программисты часто путают такие отношения, как *наибольший*, *наименьший*, *больше чем*, *не меньше чем*, *меньше или равно*.

4. Каждое ли булевское выражение сформулировано так, как это предполагалось? Программисты часто делают ошибки при написании логических выражений, содержащих операции «И», «ИЛИ», «НЕ».

5. Являются ли операнды булевских выражений булевскими? Существуют ли ошибочные объединения сравнений и булевских выражений? Они представляют другой часто встречающийся класс ошибок. Примеры нескольких типичных ошибок приведены ниже. Если величина  $I$  определена как лежащая в интервале между 2 и 10, выражение  $2 < I < 10$  является неверным. Вместо него должно быть написано выражение  $(2 < I) \& (I < 10)$ . Если же величина  $I$  определена как большая, чем  $X$  или  $Y$ , то выражение  $I > X | Y$  является неверным; оно должно быть записано в виде  $(I > X) | (I > Y)$ . При сравнении трех чисел на равенство выражение  $IF (A = B = C)$  означает совсем другое. В случае необходимости проверить математическое отношение  $X > Y > Z$  правильным будет выражение  $(X > Y) \& (Y > Z)$ .

6. Сравняются ли в программе мантиссы или числа с плавающей запятой, которые представлены в машине в двоичной форме? Это является иногда источником ошибок из-за усечения младших разрядов и из-за точного равенства чисел в двоичной и десятичной формах представления.

7. Верны ли предположения о порядке оценки и следовании операторов для выражений, содержащих более одного булевского оператора? Иными словами, если задано выражение  $(A = 2) \& (B = 2) | (C = 3)$ , понятно ли, какая из операций выполняется первой: И или ИЛИ?

8. Влияет ли на результат выполнения программы способ, которым конкретный компилятор выполняет булевские выражения? Например, оператор

$$\text{IF } (X \neq 0) \ \& \ ((Y/X) > Z)$$

является приемлемым для некоторых компиляторов PL/1 (т. е. компиляторов, которые заканчивают проверку, как только одно из выражений в функции И окажется ложным), но приведет к делению на 0 при использовании других компиляторов.

Ошибки в передачах управления

1. Если в программе содержится переключатель (например, вычисляемый оператор GO TO в Фортране), то может ли значение индекса когда-либо превысить число возможных переходов? Например, всегда ли *I* будет принимать значение 1, 2 или 3 в операторе Фортрана GO TO (200, 300, 400), *I*?

2. Будет ли каждый цикл в конце концов завершен? Придумайте неформальное доказательство или аргументы, подтверждающие их завершение.

3. Будет ли программа, модуль или подпрограмма в конечном счете завершена?

4. Возможно ли, что из-за входных условий цикл никогда не сможет выполняться? Если это так, то является ли это оплошностью? Например, что произойдет для циклов, начинающихся операторами

$$\begin{aligned} &\text{DO WHILE (NOTFOUND)} \\ &\text{DO I=X TO Z} \end{aligned}$$

если первоначальное значение NOTFOUND *ложь* или если *X* больше *Z*?

5. Для циклов, управляемых как числом итераций, так и булевым условием (например, цикл для организации поиска), какова последовательность «погружения в тело цикла»? Например, что произойдет с циклом, имеющим заголовок

$$\text{DO I=1 TO TABLESIZE WHILE (NOTFOUND)}$$

если NOTFOUND никогда не принимает значение «ложь»?

6. Существуют ли какие-нибудь ошибки «отклонения от нормы» (например, слишком большое или слишком малое число итераций)?

7. Если язык программирования содержит понятие

группы операторов (например, DO — группы в PL/1, ограниченные операторами DO, END), то имеется ли явный оператор END для каждой группы и соответствуют ли операторы END своим группам?

8. Существуют ли решения, подразумеваемые по умолчанию? Например, пусть ожидается, что входной параметр принимает значения 1, 2 или 3. Логично ли тогда предположить, что он должен быть равен 3, если он не равен 1 или 2? Коль скоро это так, то является ли предположение правильным?

#### Ошибки интерфейса

1. Равно ли число параметров, получаемых рассматриваемым модулем, числу аргументов, передаваемых каждым из вызывающих модулей? Правильен ли порядок их следования?

2. Совпадают ли атрибуты (например, тип и размер) каждого параметра с атрибутами соответствующего ему аргумента?

3. Совпадают ли единицы измерения каждого параметра с единицами измерения соответствующих аргументов? Например, нет ли случаев, когда значение параметра выражено в градусах, а аргумента — в радианах?

4. Равно ли число аргументов, передаваемых из рассматриваемого модуля другому модулю, числу параметров, ожидаемых в вызываемом модуле?

5. Соответствуют ли атрибуты каждого аргумента, передаваемого другому модулю, атрибутам соответствующего параметра в рассматриваемом модуле?

6. Совпадают ли единицы измерения каждого аргумента, передаваемого другому модулю, с единицами измерения соответствующего параметра в рассматриваемом модуле?

7. Если вызываются встроенные функции, правильно ли заданы число, атрибуты и порядок следования аргументов?

8. Если модуль имеет несколько точек входа, передается ли параметр всегда вне зависимости от точки входа? Такая ошибка присутствует во втором операторе присваивания следующей программы на PL/1:

```
A: PROCEDURE(W,X);  
    W=X+1;  
    RETURN;  
B: ENTRY(Y,Z);  
    Y=X+Z;  
    END;
```

9. Не изменяет ли подпрограмма параметр, который должен использоваться только как входная величина?

10. Если имеются глобальные переменные (например, переменные в PL/I с атрибутом EXTERNAL, переменные, указанные в операторах COMMON Фортрана), имеют ли они одинаковые определения и атрибуты во всех модулях, которые к ним обращаются?

11. Передаются ли в качестве аргументов константы? В некоторых реализациях Фортрана такие операторы, как

CALL SUBX (J, 3)

являются опасными, поскольку, если подпрограмма SUBX присвоит значение второму параметру, значение константы 3 будет изменено.

**Ошибки ввода-вывода**

1. Являются ли правильными атрибуты файлов, описанных явно?

2. Являются ли правильными атрибуты оператора OPEN?

3. Согласуется ли спецификация формата с информацией в операторах ввода-вывода? Например, согласуется ли каждый оператор FORMAT (с точки зрения числа элементарных данных и их атрибутов) с соответствующими операторами READ и WRITE в программе, написанной на Фортране? То же самое применимо к проверке соответствия между списком данных и списком форматов в операторах ввода-вывода PL/I.

4. Равен ли размеру записи размер области памяти для ввода-вывода?

5. Все ли файлы открыты перед их использованием?

6. Правильно ли обнаруживаются и трактуются признаки конца файла?

7. Правильно ли трактуются ошибочные состояния ввода-вывода?

8. Существуют ли смысловые или грамматические ошибки в тексте, выводимом программой на печать или экран дисплея?

**Другие виды контроля**

1. Если компилятор выдает таблицу перекрестных ссылок идентификаторов, проверьте величины, на которые в этом списке нет ссылок или есть только одна ссылка.

2. Если компилятор выдает список атрибутов, проверьте атрибуты каждой величины для обеспечения га-

рантии того, что в программе нет никаких неожиданных и отсутствующих атрибутов.

3. Если программа оттранслирована успешно, но компилятор выдаст одно или несколько «предупреждений» или «информационных» сообщений, внимательно проверьте каждое из них.

Предупреждение свидетельствует о «подозрениях» компилятора в отношении правильности ваших действий. Все эти «подозрения» должны быть рассмотрены. В информационных сообщениях могут перечисляться неописанные переменные или конструкции языка, которые препятствуют оптимизации кода.

4. Является ли программа (или модуль) достаточно устойчивой? Иными словами, проверяет ли она правильность своих входных данных?

5. Не пропущена ли в программе какая-нибудь функция?

Сводный список вопросов для выявления ошибок приведен на рис. 3.1 и 3.2.

#### ОБРАЩЕНИЕ К ДАННЫМ

1. Используются ли переменные с неустановленными значениями?
2. Лежат ли индексы вне заданных границ?
3. Есть ли нецелые индексы?
4. Есть ли «подвешенные» обращения?
5. Корректны ли атрибуты при всех псевдонимах?
6. Соответствуют ли атрибуты записи и структуры?
7. Вычислимы ли адреса битовых строк? Передаются ли битовые строки в качестве аргументов?
8. Корректны ли атрибуты базированной памяти?
9. Соответствуют ли друг другу определения структуры, данные ей в различных процедурах?
10. Превышены ли границы строки?
11. Существуют ли какие-нибудь другие ошибки в операциях с индексацией или при обращении к массивам по индексу?

#### ВЫЧИСЛЕНИЯ

1. Производятся ли вычисления неарифметических переменных?
2. Производятся ли вычисления с использованием данных разного вида?
3. Существуют ли вычисления переменных разной длины?
4. Не меньше ли длина результата, чем длина вычисляемого значения?
5. Возможно ли переполнение или потеря промежуточного результата при вычислении?
6. Есть ли деление на нуль?
7. Существуют ли неточности при работе с двоичными числами?
8. Не выходит ли значение переменной за пределы установленного диапазона?
9. Понятен ли порядок следования операторов?
10. Правильно ли осуществляется деление целых чисел?

## ОПИСАНИЕ ДАННЫХ

1. Все ли переменные описаны?
2. Понятно ли отсутствие атрибутов?
3. Правильно ли инициализированы массивы и строки?
4. Правильно ли определены размер, тип и класс памяти?
5. Согласуется ли инициализация с классом памяти?
6. Нельзя ли обойтись без переменных со сходными именами?

## СРАВНЕНИЕ

1. Сравниваются ли величины несовместимых типов?
2. Сравниваются ли величины различных типов?
3. Корректны ли отношения сравнения?
4. Корректны ли булевские выражения?
5. Объединяются ли сравнения и булевские выражения?
6. Сравниваются ли дробные величины, представленные в двоичной форме?
7. Понятен ли порядок следования операторов?
8. Понятна ли процедура разбора компилятором булевских выражений?

Рис. 3.1. Сводный список вопросов для выявления ошибок при инспекции (часть 1)

## ПЕРЕДАЧА УПРАВЛЕНИЯ

1. Может ли значение индекса в переключателе превысить число переходов?
2. Будет ли завершен каждый цикл?
3. Будет ли завершена программа?
4. Существует ли какой-нибудь цикл, который не выполняется из-за входных условий?
5. Корректны ли возможные погружения в цикл?
6. Есть ли ошибки отклонения числа итераций от нормы?
7. Соответствуют ли друг другу операторы DO и END?
8. Существуют ли неявные решения?

## ВВОД-ВЫВОД

1. Правильны ли атрибуты файлов?
2. Правильны ли операторы OPEN?
3. Соответствует ли формат спецификации операторам ввода-вывода?
4. Соответствует ли размер буфера размеру записи?
5. Открыты ли файлы перед их использованием?
6. Обнаруживаются ли признаки конца файла?
7. Обнаруживаются ли ошибки ввода-вывода?
8. Существуют ли какие-нибудь текстовые ошибки в выходной информации?

## ИНТЕРФЕЙС

1. Равно ли число входных параметров числу аргументов?
2. Соответствуют ли атрибуты параметров и аргументов?

## ДРУГИЕ ВИДЫ КОНТРОЛЯ

1. Есть ли в таблице перекрестных ссылок какие-нибудь переменные, на которые нет ссылок?

3. Соответствуют ли единицы измерения параметров и аргументов?
  4. Равно ли число аргументов, передаваемых вызываемым модулям, числу параметров?
  5. Соответствуют ли атрибуты аргументов, передаваемых вызываемым модулям, атрибутам параметров?
  6. Совпадают ли единицы измерения аргументов, передаваемых вызываемым модулям, единицам измерения параметров?
  7. Правильно ли заданы число, атрибуты и порядок следования аргументов для встроенных функций?
  8. Существуют ли какие-нибудь обращения к параметрам, не связанным с текущей точкой входа?
  9. Не изменяет ли подпрограмма аргументы, являющиеся только входными?
  10. Согласуются ли определения глобальных переменных во всех использующих их модулях?
  11. Передаются ли в качестве аргументов константы?
2. Список атрибутов такой, который и ожидался?
  3. Есть ли какие-нибудь предупреждения или информационные сообщения?
  4. Осуществляется ли контроль правильности входных данных?
  5. Нет ли пропущенных функций?

Рис. 3.2. Сводный список вопросов для выявления ошибок при инспекции (часть 2)

## СКВОЗНЫЕ ПРОСМОТРЫ

Сквозной просмотр, как и инспекция, представляет собой набор процедур и способов обнаружения ошибок, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но их процедуры несколько отличаются и, кроме того, здесь используются другие методы обнаружения ошибок.

Подобно инспекции сквозной просмотр проводится как непрерывное заседание, продолжающееся один или два часа. Группа по выполнению сквозного просмотра

состоит из 3—5 человек. В нее входят председатель, функции которого подобны функциям председателя в группе инспектирования, секретарь, который записывает все найденные ошибки, и специалист по тестированию. Мнения о том, кто должен быть четвертым и пятым членами группы, расходятся. Конечно, одним из них должен быть программист. Относительно пятого участника имеются следующие предположения: 1) высококвалифицированный программист; 2) эксперт по языку программирования; 3) начинающий (на точку зрения которого не влияет предыдущий опыт); 4) человек, который будет в конечном счете эксплуатировать программу; 5) участник какого-нибудь другого проекта; 6) кто-либо из той же группы программистов, что и автор программы.

Начальная процедура при сквозном просмотре такая же, как и при инспекции: участникам заранее, за несколько дней до заседания, раздаются материалы, позволяющие им ознакомиться с программой. Однако процедура заседания отличается от процедуры инспекционного заседания. Вместо того чтобы просто читать текст программы или использовать список ошибок, участники заседания «выполняют роль вычислительной машины». Лицо, назначенное тестирующим, предлагает собравшимся небольшое число написанных на бумаге тестов, представляющих собой наборы входных данных (и ожидаемых выходных данных) для программы или модуля. Во время заседания каждый тест мысленно выполняется. Это означает, что тестовые данные подвергаются обработке в соответствии с логикой программы. Состояние программы (т. е. значения переменных) отслеживается на бумаге или доске.

Конечно, число тестов должно быть небольшим и они должны быть простыми по своей природе, потому что скорость выполнения программы человеком на много порядков меньше, чем у машины. Следовательно, тесты сами по себе не играют критической роли, скорее они служат средством для первоначального понимания программы и основой для вопросов программисту о логике проектирования и принятых допущениях. В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

Как и при инспекции, мнение участников является решающим фактором. Замечания должны быть адресо-

ваны программе, а не программисту. Другими словами, ошибки не рассматриваются как слабость человека, который их совершил. Они свидетельствуют о сложности процесса создания программ и являются результатом все еще примитивной природы существующих методов программирования.

Сквозные просмотры должны протекать так же, как и описанный ранее процесс инспектирования. Побочные эффекты, получаемые во время выполнения этого процесса (установление склонных к ошибкам частей программы и обучение на основе анализа ошибок, стиля и методов) характерны и для процесса сквозных просмотров.

## **ПРОВЕРКА ЗА СТОЛОМ**

Третьим методом ручного обнаружения ошибок является применявшаяся ранее других методов «проверка за столом». Проверка за столом может рассматриваться как проверка исходного текста или сквозные просмотры, осуществляемые одним человеком, который читает текст программы, проверяет его по списку ошибок и (или) пропускает через программу тестовые данные.

Большей частью проверка за столом является относительно непродуктивной. Это объясняется прежде всего тем, что такая проверка представляет собой полностью неупорядоченный процесс. Вторая, более важная причина заключается в том, что проверка за столом противопоставляется одному из принципов тестирования гл. 2, согласно которому программист обычно неэффективно тестирует собственные программы. Следовательно, проверка за столом наилучшим образом может быть выполнена человеком, не являющимся автором программы (например, два программиста могут обмениваться программами вместо того, чтобы проверять за столом свои собственные программы), но даже в этом случае такая проверка менее эффективна, чем сквозные просмотры или инспекции. Данная причина является главной для образования группы при сквозных просмотрах или инспекциях исходного текста. Заседание группы благоприятствует созданию атмосферы здоровой конкуренции: участники хотят показать себя с лучшей стороны при нахождении ошибок. При проверке за столом этот, безусловно, ценный эффект отсутствует. Короче

говоря, проверка за столом, конечно, полезна, но она гораздо менее эффективна, чем инспекция исходного текста или сквозной просмотр.

## ОЦЕНКА ПОСРЕДСТВОМ ПРОСМОТРА

Последний ручной процесс обзора программы не связан с ее тестированием (т. е. целью его не является нахождение ошибок). Однако описание этого процесса приводится здесь потому, что он имеет отношение к идее чтения текста.

Оценка посредством просмотра [10] является методом оценки анонимной программы в терминах ее общего качества, ремонтпригодности, расширяемости, простоты эксплуатации и ясности. Цель данного метода — обеспечить программиста средствами самооценки. Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор в свою очередь отбирает приблизительно 6—20 участников (6— минимальное число для сохранения анонимности). Предполагается, что участники должны быть одного профиля (например, в одну группу не следует объединять программистов, использующих Кобол, и системных программистов, пишущих на Ассемблере). Каждого участника просят представить для рассмотрения две свои программы: наилучшую (с его точки зрения) и низкого качества.

Отобранные программы случайным образом распределяются между участниками. Им дается на рассмотрение по четыре программы. Две из них являются «наилучшими», а две — «наихудшими», но рецензенту не сообщают о том, какая программа к какой группе относится. Каждый участник тратит на просмотр одной программы 30 мин и заполняет анкету для ее оценки. После просмотра всех четырех программ оценивается их относительное качество. В анкете для оценки проверяющему предлагается оценить программу по семибалльной шкале (1 означает определенное «да», 7 — определенное «нет») при ответе, например, на следующие вопросы:

Легко ли было понять программу?

Оказались ли результаты проектирования высокого уровня очевидными и приемлемыми?

Оказались ли результаты проектирования низкого уровня очевидными и приемлемыми?

Легко ли для вас модифицировать эту программу? Испытывали бы вы чувство удовлетворения, написав такую программу?

Проверяющего просят также дать общий комментарий и рекомендации по улучшению программы. После просмотра каждому участнику передают анонимную анкету с оценкой двух его программ. Участники получают статистическую сводку, которая содержит общую и детальную классификацию их собственных программ в сравнении с полным набором программ, и анализ того, насколько оценки чужих программ совпадают с оценками тех же самых программ, данными другими проверяющими. Цель такого просмотра — дать возможность программистам самим оценить свою квалификацию. Этот способ представляется полезным как для промышленного, так и для учебного применения.

#### ЛИТЕРАТУРА

1. Weinberg G. M. The Psychology of Computer Programming. New York, Van Nostrand Reinhold, 1971.
2. Myers G. J. Software Reliability: Principles and Practices. New York, Wiley-Interscience, 1976. Русский перевод: Майерс Г. Надежность программного обеспечения. М., Мир, 1980.
3. Myers G. J. Composite/Structured Design. New York, Van Nostrand Reinhold, 1978.
4. Myers G. J. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. — Commun. ACM, 1978, 21(9), p. 760—768.
5. Perriens M. P. An Application of Formal Inspections to Top-Down Structured Program Development. — RADC—TR—77—212, IBM Federal System Div., Gaithersburg, Md., 1977 (NTIS AD/A—041645).
6. 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, p. 347—357.
7. Fagan M. E. Design and Code Inspections to Reduce Errors in Program Development. — IBM Systems J., 1976, 15(3), p. 182—211.
8. Freeman R. D. An Experiment in Software Development. — The Bell System Technical Journal, Special Safeguard Supplement, 1975, p. S199—S209.
9. Ascoly J. et al. Code Inspection Specification. — TR—21. 630, IBM System Communication Division, Kingston, N. Y., 1976.
10. Anderson N. and Shneiderman B. Use of Peer Ratings in Evaluating Computer Program Quality. — IFSM—TR—20, University of Maryland, 1977.

## ГЛАВА 4

### ПРОЕКТИРОВАНИЕ ТЕСТА

Результаты психологических исследований, обсуждавшиеся в гл. 2, показывают, что наибольшее внимание при тестировании программ уделяется проектированию или созданию эффективных тестов. Это связано с невозможностью «полного» тестирования программы, т. е. тест для любой программы будет обязательно неполным (иными словами, тестирование не может гарантировать отсутствия всех ошибок). Стратегия проектирования заключается в том, чтобы попытаться уменьшить эту «неполноту» на столько, на сколько это возможно.

Если ввести ограничения на время, стоимость, машинное время и т. п., то ключевым вопросом тестирования становится следующий:

*Какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?*

Изучение методологий проектирования тестов дает ответ на этот вопрос.

По-видимому, наихудшей из всех методологий является тестирование со случайными входными значениями (стохастическое) — процесс тестирования программы путем случайного выбора некоторого подмножества из всех возможных входных величин. В терминах вероятности обнаружения большинства ошибок случайно выбранный набор тестов имеет малую вероятность быть оптимальным или близким к оптимальному подмножеством. В настоящей главе рассматривается несколько подходов, которые позволяют более разумно выбирать тестовые данные. В гл. 2 было показано, что исчерпывающее тестирование по принципу черного или белого ящика в общем случае невозможно. Однако при этом отмечалось, что приемлемая стратегия тестирования может обладать элементами обоих подходов. Таковой яв-

ляется стратегия, излагаемая в этой главе. Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе чёрного ящика, а затем дополнить его проверкой логики программы (т. е. с привлечением методов белого ящика).

Методологии, обсуждаемые в настоящей главе, представлены ниже.

### *Чёрный ящик*

Эквивалентное разбиение  
Анализ граничных значений  
Применение функциональных диаграмм  
Предположение об ошибке

### *Белый ящик*

Покрытие операторов  
Покрытие решений  
Покрытие условий  
Покрытие решений/условий  
Комбинаторное покрытие условий

Хотя перечисленные методы будут рассматриваться здесь по отдельности, при проектировании эффективного теста программы рекомендуется использовать если не все эти методы, то по крайней мере большинство из них, так как каждый метод имеет определенные достоинства и недостатки (например, возможность обнаруживать и пропускать различные типы ошибок). Правда, эти методы весьма трудоемки, поэтому некоторые специалисты, ознакомившись с ними, могут не согласиться с данной рекомендацией. Однако следует представлять себе, что тестирование программы — чрезвычайно сложная задача. Для иллюстрации этого приведем известное изречение: «Если вы думаете, что разработка и кодирование программы — вещь трудная, то вы еще ничего не видели».

Рекомендуемая процедура заключается в том, чтобы разрабатывать тесты, используя методы черного ящика, а затем как необходимое условие — дополнительные тесты, используя методы белого ящика. Методы белого ящика, получившие более широкое распространение, обсуждаются первыми.

## **ТЕСТИРОВАНИЕ ПУТЕМ ПОКРЫТИЯ ЛОГИКИ ПРОГРАММЫ**

Тестирование по принципу белого ящика характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Как показано

в гл. 2, исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе; но поскольку в программе с циклами выполнение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается в данной книге как перспективное.

Если отказаться полностью от тестирования всех путей, то можно показать, что критерием покрытия является выполнение каждого оператора программы по крайней мере один раз. К сожалению, это слабый критерий, так как выполнение каждого оператора по крайней мере один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика (рис. 4.1). Предположим, что на рис. 4.1 представлена небольшая программа, которая должна быть протестирована. Эквивалентная программа, написанная на языке PL/1, имеет вид:

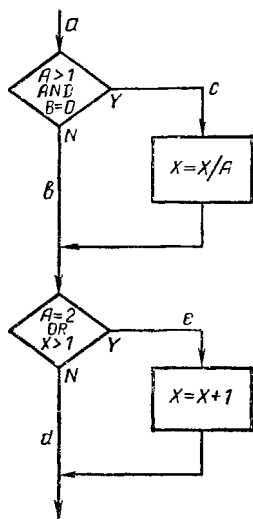


Рис. 4.1. Небольшая программа, которая должна быть протестирована

```

M: PROCEDURE (A, B, X);
  IF ((A > 1) & (B = 0)) THEN DO;
    X = X / A;
  END;
  IF ((A = 2) | (X > 1)) THEN DO;
    X = X + 1;
  END;
END;
  
```

Можно выполнить каждый оператор, записав единственный тест, который реализовал бы путь *ace*. Иными словами, если бы в точке *a* были установлены значения  $A=2$ ,  $B=0$  и  $X=3$ , каждый оператор выполнялся бы один раз (в действительности  $X$  может принимать любое значение).

К сожалению, этот критерий хуже, чем он кажется на

первый взгляд. Например, пусть первое решение записано как *или*, а не как *и*. При тестировании по данному критерию эта ошибка не будет обнаружена. Пусть второе решение записано в программе как  $X > 0$ ; эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором  $X$  не изменяется (путь *abd*). Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

Более сильный критерий покрытия логики программы известен как *покрытие решений*, или *покрытие переходов*. Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение *истина* и *ложь* по крайней мере один раз. Иными словами, каждое направление перехода должно быть реализовано по крайней мере один раз. Примерами операторов перехода или решений являются операторы DO (или PERFORM UNTIL в Коболе), IF, многовыходные операторы GO TO.

Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существует по крайней мере три исключения. Первое — патологическая ситуация, когда программа не имеет решений. Второе встречается в программах или подпрограммах с несколькими точками входа; данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Третье исключение — операторы внутри ON-единиц; выполнение каждого направления перехода не обязательно будет вызывать выполнение всех ON-единиц. Так как покрытие операторов считается необходимым условием, покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом значения *истина* и *ложь* и при этом каждый оператор выполнялся бы по крайней мере один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значения *истина* и *ложь* и что каждой точке входа (включая ON-единицы) долж-

но быть передано управление при вызове программы по крайней мере один раз.

Изложенное выше предполагает только двузначные решения или переходы и должно быть модифицировано для программ, содержащих многозначные решения. Примерами таких программ являются программы на PL/I, включающие операторы SELECT (CASE) <sup>1</sup> или операторы GO TO, использующие метку-переменную, программы на Фортране с арифметическими операторами IF, вычисляемыми операторами GO TO или операторами GO TO по предписанию, и программы на Коболе, содержащие операторы GO TO вместе с ALTER или операторы GO—TO—DEPENDING—ON. Критерием для них является выполнение каждого возможного результата всех решений по крайней мере один раз и передача управления при вызове программы или подпрограммы каждой точке входа по крайней мере один раз.

В программе, представленной на рис. 4.1, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути *ace* и *abd*, либо пути *acd* и *abe*. Если мы выбираем последнее альтернативное покрытие, то входами двух тестов являются  $A=3$ ,  $B=0$ ,  $X=3$  и  $A=2$ ,  $B=1$ ,  $X=1$ .

Покрытие решений — более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где  $X$  не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50%. Если во втором решении существует ошибка (например,  $X < 1$  вместо  $X > 1$ ), то ошибка не будет обнаружена двумя тестами предыдущего примера.

Лучшим критерием по сравнению с предыдущим является *покрытие условий*. В этом случае записывают число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз. Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что каждой точке входа в программу или подпрограмму, а также ON-единицам

---

<sup>1</sup> Таких операторов в языке PL/I нет, но средствами PL/I их можно реализовать. Способы реализации изложены в кн.: Хьюз Дж., Мичтом Дж. Структурный подход к программированию. М., Мир, 1980. — *Примеч. пер.*

должно быть передано управление при вызове по крайней мере один раз. Например, оператор цикла

DO K=0 TO 50 WHILE (J+K<QUEST);

содержит два условия:  $K$  меньше или равно 50 и  $J+K$  меньше, чем QUEST. Следовательно, здесь требуются тесты для ситуаций  $K \leq 50$ ,  $K > 50$  (т.е. выполнение последней итерации цикла),  $J+K < \text{QUEST}$  и  $J+K \geq \text{QUEST}$ .

Программа рис. 4.1 имеет четыре условия:  $A > 1$ ,  $B = 0$ ,  $A = 2$  и  $X > 1$ . Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где  $A > 1$ ,  $A \leq 1$ ,  $B = 0$  и  $B \neq 0$  в точке  $a$  и  $A = 2$ ,  $A \neq 2$ ,  $X > 1$  и  $X \leq 1$  в точке  $b$ . Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

1.  $A = 2$ ,  $B = 0$ ,  $X = 4$  *ace*.
2.  $A = 1$ ,  $B = 1$ ,  $X = 1$  *abd*.

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано, покрытие условий обычно лучше покрытия решений, поскольку оно *может* (но не всегда) вызвать выполнение решений в условиях, не реализуемых при покрытии решений. Например, оператор

DO K=0 TO 50 WHILE (J+K<QUEST);

представляет собой двузначный переход (либо выполняется тело цикла, либо выход из цикла). Если использовать тестирование решений, то достаточно выполнить цикл при изменении  $K$  от 0 до 51 *без проверки случая, когда условие в WHILE ложно*. Однако при критерии покрытия условий необходим тест, который реализовал бы результат *ложь* условия  $J+K < \text{QUEST}$ .

Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Если тестируется решение IF ( $A \& B$ ), то при критерии покрытия условий требовались бы два теста —  $A$  есть *истина*,  $B$  есть *ложь* и  $A$  есть *ложь*,  $B$  есть *истина*. Но в этом случае не выполнялось бы THEN — предложение оператора IF. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста

1.  $A = 1$ ,  $B = 0$ ,  $X = 3$ .
2.  $A = 2$ ,  $B = 1$ ,  $X = 1$

покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь *abe* и, следовательно, не выполняют результат *истина* первого решения и результат *ложь* второго решения).

Очевидным следствием из этой дилеммы является критерий, названный *покрытием решений/условий*. Он требует такого достаточного набора тестов, чтобы все

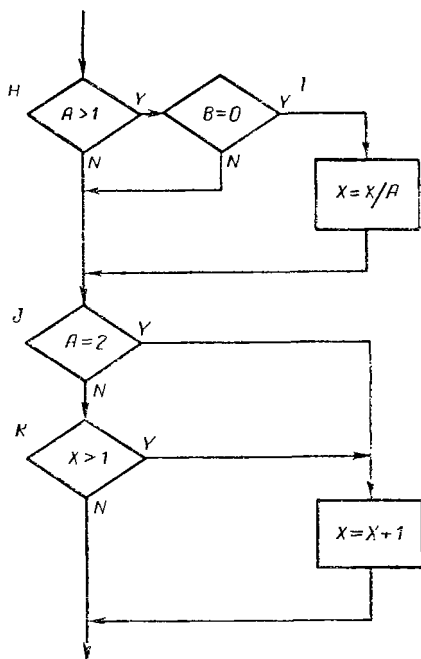


Рис. 4.2. Машинный код программы, изображенной на рис. 4.1

возможные результаты каждого условия в решении выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и каждой точке входа передавалось управление по крайней мере один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий; часто подобное выполнение имеет место вследствие того, что определенные условия скрыты другими условиями. В качестве примера рассмотрим приведенную на рис. 4.2 схему передач управления в машинном коде программы рис. 4.1. Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство машин не имеет команд, реализующих решения с многими исходами. Наиболее полное покрытие тестами в этом случае осуществляется таким образом, чтобы выполнялись все возможные результаты каждого простого решения. Два предыдущих теста критерия покрытия решений не выполняют этого; они недостаточны для выпол-

нения результата *ложь* решения Н и результата *истина* решения К. Набор тестов для критерия покрытия условий такой программы также является неполным; два теста (которые случайно удовлетворяют также и критерию покрытия решений/условий) не вызывают выполнения результата *ложь* решения I и результата *истина* решения К.

Причина этого заключается в том, что, как показано на рис. 4.2, результаты условий в выражениях *и* и *или* могут скрывать и блокировать действие других условий. Например, если условие *и* есть *ложь*, то никакое из последующих условий в выражении не будет выполнено. Аналогично если условие *или* есть *истина*, то никакое из последующих условий не будет выполнено. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

Критерием, который решает эти и некоторые другие проблемы, является *комбинаторное покрытие условий*. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись по крайней мере один раз. Например, в приведенной ниже последовательности операторов существуют четыре ситуации, которые должны быть протестированы:

```
NOTFOUND='1' B;  
DO I=1 TO TABSIZE WHILE (NOTFOUND); /* поиск в таблице */  
последовательность операторов, реализующая процедуру поиска,  
END;
```

1.  $I \leq \text{TABSIZE}$  и NOTFOUND есть *истина*.
2.  $I \leq \text{TABSIZE}$  и NOTFOUND есть *ложь* (обнаружение необходимого искомого значения до достижения конца таблицы).
3.  $I > \text{TABSIZE}$  и NOTFOUND есть *истина* (достижение конца таблицы без обнаружения искомого значения).
4.  $I > \text{TABSIZE}$  и NOTFOUND есть *ложь* (искомое значение является последней записью в таблице).

Легко видеть, что набор тестов, удовлетворяющий критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

По этому критерию в программе рис. 4.1 должны быть покрыты тестами следующие восемь комбинаций:

1.  $A > 1, B = 0.$
2.  $A > 1, B \neq 0.$
3.  $A \leq 1, B = 0.$
4.  $A \leq 1, B \neq 0.$

5.  $A = 2, X > 1.$
6.  $A = 2, X \leq 1.$
7.  $A \neq 2, X > 1.$
8.  $A \neq 2, X \leq 1.$

Заметим, что комбинации 5—8 представляют собой значения второго оператора IF. Поскольку  $X$  может быть изменено до выполнения этого оператора, значения, необходимые для его проверки, следует восстановить исходя из логики программы с тем, чтобы найти соответствующие входные значения.

Для того чтобы протестировать эти комбинации, не обязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

$A = 2, B = 0, X = 4$	покрывает 1,5
$A = 2, B = 1, X = 1$	покрывает 2,6
$A = 1, B = 0, X = 2$	покрывает 3,7
$A = 1, B = 1, X = 1$	покрывает 4,8

То, что четырем тестам соответствуют четыре различных пути на рис. 4.1, является случайным совпадением. На самом деле представленные выше тесты не покрывают всех путей, они пропускают путь  $acd$ . Например, требуется восемь тестов для тестирования следующей программы:

```
IF ((X=Y) & (LENGTH(Z)=0) & END) THEN J=1;
                               ELSE I=1;
```

хотя она покрывается лишь двумя путями. В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого 1) вызывает выполнение всех результатов каждого решения по крайней мере один раз и 2) передает управление каждой точке входа (например, точке входа, ON-единице) по крайней мере один раз (чтобы обеспечить выполнение каждого оператора программы по крайней мере один раз). Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных

комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы по крайней мере один раз. [Слово «возможных» употреблено здесь потому, что некоторые комбинации условий могут быть нереализуемыми; например, в выражении  $(A > 2) \& (A < 10)$  могут быть реализованы только три комбинации условий.]

## ЭКВИВАЛЕНТНОЕ РАЗБИЕНИЕ

В гл. 2 отмечалось, что хороший тест имеет приемлемую вероятность обнаружения ошибки и что исчерпывающее входное тестирование программы невозможно. Следовательно, тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Тогда, конечно, хотелось бы выбрать для тестирования самое подходящее подмножество (т. е. подмножество с наивысшей вероятностью обнаружения большинства ошибок).

Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

а) уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;

б) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число *классов эквивалентности* так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же

самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться).

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как *эквивалентное разбиение*. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое — для разработки минимального набора тестов, покрывающих эти условия.

Примером класса эквивалентности для программы о треугольнике (см. гл. 1) является набор «трех равных чисел, имеющих целые значения, большие нуля». Определяя этот набор как класс эквивалентности, устанавливая, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом набора. Иными словами, в этом случае время тестирования лучше затратить на что-нибудь другое (на тестирование других классов эквивалентности).

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: 1) выделение классов эквивалентности и 2) построение тестов.

**Выделение классов эквивалентности**

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Для проведения этой операции используют таблицу, изображенную на рис. 4.3. Заметим, что

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности

Рис. 4.3. Форма таблицы для перечисления классов эквивалентности

различают два типа классов эквивалентности: *правильные классы эквивалентности*, представляющие правиль-

ные входные данные программы, и *неправильные классы эквивалентности*, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). Таким образом, придерживаются одного из принципов гл. 2 о необходимости сосредоточивать внимание на неправильных или неожиданных условиях.

Если задаться входными или внешними условиями, то выделение классов эквивалентности представляет собой в значительной степени эвристический процесс. При этом существует ряд правил:

1. Если входное условие описывает *область* значений (например, «целое данное может принимать значения от 1 до 999»), то определяются один правильный класс эквивалентности ( $1 \leq \text{значение целого данного} \leq 999$ ) и два неправильных (значение целого данного  $< 1$  и значение целого данного  $> 999$ ).

2. Если входное условие описывает *число* значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяются один правильный класс эквивалентности и два неправильных (ни одного и более шести человек).

3. Если входное условие описывает *множество* входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, ПЕШКОМ или МОТОЦИКЛЕ»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «НА ПРИЦЕПЕ»).

4. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ — буква) и один неправильный (первый символ — не буква).

5. Если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

Этот процесс ниже будет кратко проиллюстрирован.  
**Построение тестов**

Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор пока все неправильные классы эквивалентности не будут покрыты тестами.

Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами. Например, спецификация устанавливает «тип книги при поиске (ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ПРОГРАММИРОВАНИЕ или ОБЩИЙ) и количество (1-9999)». Тогда тест

XYZ 0

отображает два ошибочных условия (неправильный тип книги и количество) и, вероятно, не будет осуществлять проверку количества, так как программа может ответить: «XYZ — НЕСУЩЕСТВУЮЩИЙ ТИП КНИГИ» и не проверять остальную часть входных данных.

#### Пример

Предположим, что при разработке компилятора для подмножества языка Фортран требуется протестировать синтаксическую проверку оператора DIMENSION. Спецификация приведена ниже. (Этот оператор не является полным оператором DIMENSION Фортрана; спецификация была значительно сокращена, что позволило сделать ее «учебным примером». Не следует думать, что тестирование реальных программ так же легко, как в примерах данной книги.) В спецификации элементы, написанные латинскими буквами, обозначают синтаксические единицы, которые в реальных операторах должны быть заменены соответствующими значениями, в квадратные скобки заключены необязательные элементы, многоточие показывает, что предшествующий ему элемент может быть повторен подряд несколько раз.

Оператор DIMENSION используется для определения массивов. Форма оператора DIMENSION:

DIMENSION *ad* [*ad*] ...,

где *ad* есть описатель массива в форме

$n(d[d] \dots)$ ,

*n* — символическое имя массива, а *d* — индекс массива. Симво-

лические имена могут содержать от одного до шести символов — букв или цифр, причем первой должна быть буква. Допускается от одного до семи индексов. Форма индекса

$[lb : ]ub,$

где  $lb$  и  $ub$  задают нижнюю и верхнюю границы индекса массива. Граница может быть либо константой, принимающей значения от —65534 до 65535, либо целой переменной (без индексов). Если  $lb$  не определена, то предполагается, что она равна единице. Значение  $ub$  должно быть больше или равно  $lb$ . Если  $lb$  определена, то она может иметь отрицательное, нулевое или положительное значение. Как и все операторы, оператор DIMENSION может быть продолжен на нескольких строках. (Конец спецификации.)

Первый шаг заключается в том, чтобы идентифицировать входные условия и по ним определить классы эквивалентности (табл. 4.1). Классы эквивалентности в таблице обозначены числами.

Следующий шаг — построение теста, покрывающего один или более правильных классов эквивалентности. Например, тест

DIMENSION A(2)

покрывает классы 1, 4, 7, 10, 12, 15, 24, 28, 29 и 40. Далее определяются один или более тестов, покрывающих оставшиеся правильные классы эквивалентности. Так, тест

DIMENSION A12345(I,9,J4XXXX,65535,1,KLM,  
X 100),BBB (—65534 : 100,0 : 1000,10 : 10,I : 65535)

покрывает оставшиеся классы. Перечислим неправильные классы эквивалентности и соответствующие им тесты:

(3) : DIMENSION	
(5) : DIMENSION	(10)
(6) : DIMENSION	A234567 (2)
(9) : DIMENSION	A,1 (2)
(11) : DIMENSION	1A (10)
(13) : DIMENSION	B
(14) : DIMENSION	B (4,4,4,4,4,4,4)
(17) : DIMENSION	B (4,A (2))
(18) : DIMENSION	B (4,,7)
(21) : DIMENSION	C (1,,10)
(23) : DIMENSION	C (10,1J)
(25) : DIMENSION	D (—65535:1)
(26) : DIMENSION	D (65536)
(31) : DIMENSION	D (4:3)
(37) : DIMENSION	D (A (2):4)
(38) : DIMENSION	D (.: 4)

Эти классы эквивалентности покрываются 18 тестами. Читатель может при желании сравнить данные тесты с набором тестов, полученным каким-либо специальным методом.

Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т. е. пропускает определенные типы высокоэффективных тестов). Следующие два метода — анализ граничных значений и использование функциональных диаграмм (диаграмм причинно-следственных связей cause-effect graphing) — свободны от многих недостатков, присущих эквивалентному разбиению.

## Классы эквивалентности

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Число описателей массивов	один (1), > одного (2)	ни одного (3)
Длина имени массива	1—6 (4)	0 (5), > 6 (6)
Имя массива	имеет в своем составе буквы (7) и цифры (8)	содержит что-то еще (9)
Имя массива начинается с буквы	да (10)	нет (11)
Число индексов	1—7 (12)	0 (13), > 7 (14)
Верхняя граница	константа (15), целая переменная (16)	имя элемента массива (17), что-то иное (18)
Имя целой переменной	имеет в своем составе буквы (19), и цифры (20)	состоит из чего-то еще (21)
Целая переменная начинается с буквы	да (22)	нет (23)
Константа	—65534—65535 (24)	<—65534 (25), >65535 (26)
Нижняя граница определена	да (27), нет (28)	
Верхняя граница по отношению к нижней границе	больше (29), равна (30)	меньше (31)
Значение нижней границы	отрицательное (32), ноль (33), > 0 (34)	
Нижняя граница	константа (35), целая переменная (36)	имя элемента массива (37), что-то иное (38)
Оператор расположен на нескольких строках	да (39), нет (40)	

## АНАЛИЗ ГРАНИЧНЫХ ЗНАЧЕНИЙ

Как показывает опыт, тесты, исследующие *граничные условия*, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия — это ситуации, возникающие непосредственно на, выше или ниже границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и *пространство результатов* (т. е. выходные классы эквивалентности).

Трудно описать «кухню» анализа граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме. (Следовательно, анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта.) Тем не менее приведем несколько общих правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть  $-1,0 - +1,0$ , то написать тесты для ситуаций  $-1,0$ ,  $1,0$ ,  $-1,001$  и  $1,001$ .

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0,1, 255 и 256 записей.

3. Использовать правило 1 для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет 0,00 дол., а максимум — 1165,25 дол., то построить тесты, которые вызывают расходы с 0,00 дол. и 1165,25 дол. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165,25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но тем не менее стоит рассмотреть эту возможность.

4. Использовать правило 2 для каждого выходного

условия. Например, если система информационного поиска отображает на экране терминала наиболее релевантные рефераты в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

6. Попробовать свои силы в поиске других граничных условий.

Чтобы проиллюстрировать необходимость анализа граничных значений, можно использовать программу анализа треугольника, приведенную в гл. 1. Для задания треугольника входные значения должны быть целыми положительными числами, и сумма любых двух из них должна быть больше третьего. Если определены эквивалентные разбиения, то целесообразно определить одно разбиение, в котором это условие выполняется, и другое, в котором сумма двух целых не больше третьего. Следовательно, двумя возможными тестами являются 3—4—5 и 1—2—4. Тем не менее здесь есть вероятность пропуска ошибки. Иными словами, если выражение в программе было закодировано как  $A+B \geq C$  вместо  $A+B > C$ , то программа ошибочно сообщала бы нам, что числа 1—2—3 представляют правильный равносторонний треугольник. Таким образом, существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие *на и вблизи границ эквивалентных разбиений*.

В качестве примера применения метода анализа граничных значений рассмотрим следующую спецификацию программы.

MTEST есть программа, которая сортирует различную информацию об экзаменах. Входом программы является файл, названный OCR, который содержит 80-символьные записи. Первая запись представляет название; ее содержание используется как заголовок каждого выходного отчета. Следующее множество записей описывает правильные ответы на экзамене. Каждая запись этого множества содержит «2» в качестве последнего символа. В первой записи в колонках 1—3 задается число ответов (оно принимает значения от 1

до 999). Колонки 10—59 включают сведения о правильных ответах на вопросы с номерами 1—50 (любой символ воспринимается как ответ). Последующие записи содержат в колонках 10—59 сведения о правильных ответах на вопросы с номерами 51—100, 101—150 и т. д. Третье множество записей описывает ответы каждого сту-

Название									
1									80
Число вопросов		Правильные ответы 1-50						2	
1	3 4	9 10						59 60	79 80
		Правильные ответы 51-100						2	
1	9 10								59 60 79 80
⋮									
Идентификатор студента		ответы студента 1-50						3	
1	9 10								59 60 79 80
		ответы студента 51-100						3	
1	9 10								59 60 79 80
⋮									
Идентификатор студента		ответы студента 1-50						3	
1	9 10								59 60 79 80

Рис. 4.4. Структуры входных записей для программы MTEST

дента; любая запись этого набора имеет число «3» в восьмидесятой колонке. Для каждого студента первая запись в колонках 1—9 содержит его имя или номер (любые символы); в колонках 10—59 помещены сведения о результатах ответов студентов на вопросы с номерами 1—50. Если в тесте предусмотрено более чем 50 вопросов, то последующие записи для студента описывают ответы 51—100, 101—150 и т. д. в колонках 10—59. Максимальное число студентов — 200. Форматы входных записей показаны на рис. 4.4. Выходными записями являются: 1) отчет, упорядоченный в лексикографическом порядке идентификаторов студентов и показывающий качество ответов каждого студента (процент правильных ответов) и его ранг; 2) аналогичный отчет, но упорядоченный по качеству; 3) отчет, показывающий среднее значение, медиану и среднее квадратическое отклонение качества ответов; 4) отчет, упорядоченный по номерам вопросов и показывающий процент студентов, отвечающих правильно на каждый вопрос. (Конец спецификации.)

Начнем методичное чтение спецификации, выявляя входные условия. Первое граничное входное условие есть пустой входной файл. Второе входное условие — карта (запись) названия; граничными условиями явля-

ются отсутствие карты названия, самое короткое и самое длинное названия. Следующими входными условиями служат наличие записей о правильных ответах и наличие поля числа вопросов в первой записи ответов. 1—999 не является классом эквивалентности для числа вопросов, так как для каждого подмножества из 50 записей может иметь место что-либо специфическое (т. е. необходимо много записей). Приемлемое разбиение вопросов на классы эквивалентности представляет разбиение на два подмножества: 1—50 и 51—999. Следовательно, необходимы тесты, где поле числа вопросов принимает значения 0, 1, 50, 51 и 999. Эти тесты покрывают большинство граничных условий для записей о правильных ответах; однако существуют три более интересные ситуации — отсутствие записей об ответах, наличие записей об ответах типа «много ответов на один вопрос» и наличие записей об ответах типа «мало ответов на один вопрос» (например, число вопросов — 60<sup>1</sup>, и имеются три записи об ответах в первом случае и одна запись об ответах во втором). Таким образом, определены следующие тесты:

1. Пустой входной файл.
  2. Отсутствует запись названия.
  3. Название длиной в один символ.
  4. Название длиной в 80 символов.
  5. Экзамен из одного вопроса.
  6. Экзамен из 50 вопросов.
  7. Экзамен из 51 вопроса.
  8. Экзамен из 999 вопросов.
  9. 0 вопросов на экзамене.
  10. Поле числа вопросов имеет нечисловые значения.
  11. После записи названия нет записей о правильных ответах.
  12. Имеются записи типа «много правильных ответов на один вопрос».
  13. Имеются записи типа «мало правильных ответов на один вопрос».
- Следующие входные условия относятся к ответам студентов. Тестами граничных значений в этом случае, по-видимому, должны быть:
14. 0 студентов.

---

<sup>1</sup> При выбранных значениях поля числа вопросов интереснее рассмотреть не 60, а 50 вопросов. — *Примеч. пер.*

15. 1 студент.
16. 200 студентов.
17. 201 студент.
18. Есть одна запись об ответе студента, но существуют две записи о правильных ответах.
19. Запись об ответе вышеупомянутого студента первая в файле.
20. Запись об ответе вышеупомянутого студента последняя в файле.
21. Есть две записи об ответах студента, но существует только одна запись о правильном ответе.
22. Запись об ответах вышеупомянутого студента первая в файле.
23. Запись об ответах вышеупомянутого студента последняя в файле.

Можно также получить набор тестов для проверки выходных границ, хотя некоторые из выходных границ (например, пустой отчет 1) покрываются приведенными тестами. Граничными условиями для отчетов 1 и 2 являются:

- 0 студентов (так же, как тест 14);
- 1 студент (так же, как тест 15);
- 200 студентов (так же, как тест 16).
24. Оценки качества ответов всех студентов одинаковы.
25. Оценки качества ответов всех студентов различны.
26. Оценки качества ответов некоторых, но не всех студентов одинаковы (для проверки правильности вычисления рангов).
27. Студент получает оценку качества ответа 0.
28. Студент получает оценку качества ответа 100.
29. Студент имеет идентификатор наименьшей возможной длины (для проверки правильности упорядочения).
30. Студент имеет идентификатор наибольшей возможной длины.
31. Число студентов таково, что отчет имеет размер, несколько больший одной страницы (для того чтобы посмотреть случай печати на другой странице).
32. Число студентов таково, что отчет располагается на одной странице.

Граничные условия отчета 3 (среднее значение, медиана, среднее квадратическое отклонение):

33. Среднее значение максимально (качество ответов всех студентов наивысшее).
34. Среднее значение равно 0 (качество ответов всех студентов равно 0).
35. Среднее квадратическое отклонение равно своему максимуму (один студент получает оценку 0, а другой — 100).
36. Среднее квадратическое отклонение равно 0 (все студенты получают одну и ту же оценку).

Тесты 33 и 34 покрывают и границы медианы. Другой полезный тест описывает ситуацию, где существует 0 студентов (проверка деления на 0 при вычислении математического ожидания), но он идентичен тесту 14.

Проверка отчета 4 дает следующие тесты граничных значений:

37. Все студенты отвечают правильно на первый вопрос.
38. Все студенты неправильно отвечают на первый вопрос.
39. Все студенты правильно отвечают на последний вопрос.
40. Все студенты отвечают на последний вопрос неправильно.
41. Число вопросов таково, что размер отчета несколько больше одной страницы.
42. Число вопросов таково, что отчет располагается на одной странице.

Опытный программист, вероятно, согласится с той точкой зрения, что многие из этих 42 тестов позволяют выявить наличие общих ошибок, которые могут быть сделаны при разработке данной программы. Кроме того, большинство этих ошибок, вероятно, не было бы обнаружено, если бы использовался метод случайной генерации тестов или специальный метод генерации тестов. Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако он часто оказывается неэффективным из-за того, что внешне выглядит простым. Читатель должен понимать, что граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют комбинаций входных условий. Например, пусть программа MTEST предыдущего раздела не выполняется, если произведение числа вопросов и числа студентов превышает некоторый предел (например, объем памяти). Такая ошибка не обязательно будет обнаружена тестированием граничных значений.

Тестирование комбинаций входных условий — непростая задача, поскольку даже при построенном эквивалентном разбиении входных условий число комбинаций обычно астрономически велико. Если нет систематического способа выбора подмножества входных условий, то, как правило, выбирается произвольное подмножество, приводящее к неэффективному тесту.

Метод функциональных диаграмм или диаграмм причинно-следственных связей [1] помогает систематически выбирать высокорезультативные тесты. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Диаграмме можно сопоставить цифровую логическую цепь (комбинаторную логическую сеть), но для ее описания используется более простая нотация (форма-записи), чем обычная форма записи, принятая в электронике. Для уяснения метода функциональных диаграмм вовсе не обязательно знание электроники, но желательно понимание булевой логики (т. е. логических операторов *и*, *или* и *не*). Построение тестов этим методом осуществляется в несколько этапов.

1. Спецификация разбивается на «рабочие» участки. Это связано с тем, что функциональные диаграммы становятся слишком громоздкими при применении данного метода к большим спецификациям. Например, когда тестируется система разделения времени, рабочим участком может быть спецификация отдельной команды. При тестировании компилятора в качестве рабочего участка можно рассматривать каждый отдельный оператор языка программирования.

2. В спецификации определяются причины и следствия. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Например, если сообщение программе приводит к обновлению основного файла, то изменение в нем и является преобразованием системы; подтверждающее сообщение было бы выходным условием. Причины и следствия определяются путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждому причине и следствию присписывается отдельный номер.
3. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия. Это и есть функциональная диаграмма.
4. Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.
5. Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
6. Столбцы таблицы решений преобразуются в тесты.

Базовые символы для записи функциональных диаграмм показаны на рис. 4.5. Каждый узел диаграммы может находиться в двух состояниях — 0 или 1; 0 обозначает состояние «отсутствует», а 1 — «присутствует». Функция *тождество* устанавливает, что если значение  $a$  есть 1, то и значение  $b$  есть 1; в противном случае значение  $b$  есть 0. Функция *не* устанавливает, что если  $a$  есть 1, то  $b$  есть 0; в противном случае  $b$  есть 1. Функция *или* устанавливает, что если  $a$ , или  $b$ , или  $c$  есть 1, то  $d$  есть 1; в противном случае  $d$  есть 0. Функция *и* устанавливает, что если и  $a$ , и  $b$  есть 1, то и  $c$  есть 1; в противном случае  $c$  есть 0. Последние две функции разрешают иметь любое число входов. Для иллюстрации изложенного рассмотрим диаграмму, отображающую спецификацию:

Символ в колонке 1 должен быть буквой «А» или «В», а в колонке 2 — цифрой. В этом случае файл обновляется. Если первый сим-

вол неправильный, то выдается сообщение X12, а если второй символ неправильный — сообщение X13.

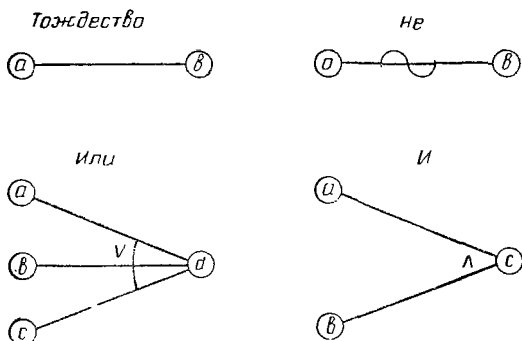


Рис. 4.5. Базовые логические отношения функциональных диаграмм

Причинами являются

- 1 — символ «А» в колонке 1;
- 2 — символ «В» в колонке 1;
- 3 — цифра в колонке 2,

а следствиями —

- 70 — файл обновляется;
- 71 — выдается сообщение X12;
- 72 — выдается сообщение X13.

Функциональная диаграмма показана на рис. 4.6. Отметим, что здесь создан промежуточный узел 11. Читателю следует убедиться в том, что диаграмма действительно отображает данную спецификацию, задавая причинам все возможные значения и проверяя, принимают ли при этом следствия правильные значения. Для читателей, знакомых с логическими диаграммами, на рис. 4.7 показана эквивалентная логическая схема.

Хотя диаграмма рис. 4.6 отображает спецификацию, она содержит невозможную комбинацию причин — причины 1 и 2 не могут быть установлены в 1 одновременно. В большинстве программ определенные комбинации причин невозможны из-за синтаксических или внешних ограничений (например, символ не может принимать значения «А» и «В» одновременно). В этом случае используются дополнительные логические ограничения, изображенные на рис. 4.8. Ограничение *E* устанавливает, что *E* должно быть истинным, если хотя бы одна из причин — *a* или *b* — принимает значение 1 (*a* и *b* не могут принимать значение 1 одновременно). Ограничение *I* устанавливает, что по крайней мере одна из величин *a*, *b* или *c* всегда должна быть равной 1 (*a*, *b* и *c* не могут

принимать значение 0 одновременно). Ограничение  $O$  устанавливает, что одна и только одна из величин  $a$  или  $b$  должна быть равна 1. Ограничение  $R$  устанавливает, что если  $a$  принимает значение 1, то и  $b$  должна принимать значение 1

(т. е. невозможно, чтобы  $a$  было равно 1, а  $b$  — 0).

Часто возникает необходимость в ограничениях для следствий. Ограничение  $M$  на рис. 4.9 устанавливает, что если следствие  $a$  имеет

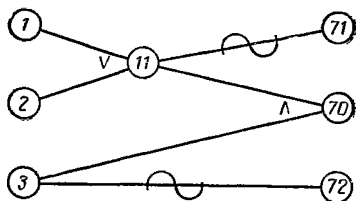


Рис. 4.6. Пример функциональной диаграммы

значение 1, то следствие  $b$  должно принять значение 0.

Как видно из рассмотренного выше примера, физически невозможно, чтобы причины 1 и 2 присутствовали одновременно, но возможно,

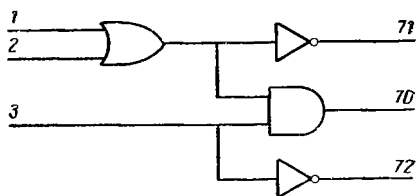


Рис. 4.7. Логическая схема, эквивалентная диаграмме рис. 4.6

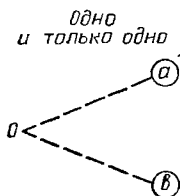
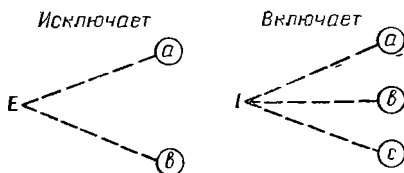


Рис. 4.8. Символы ограничений

чтобы присутствовала одна из них. Следовательно, они связаны ограничением  $E$  (рис. 4.10).

Проиллюстрируем использование функциональных диа-

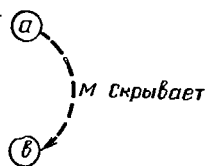
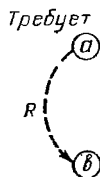


Рис. 4.9. Символ для «скрытого» ограничения

грамм для получения тестов. С этой целью воспользуемся спецификацией на команду отладки в интерактивной системе.

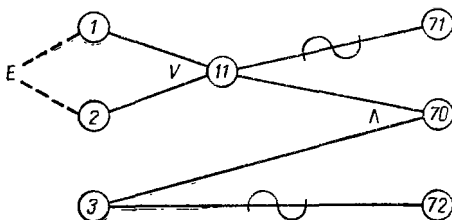


Рис. 4.10. Пример функциональной диаграммы с ограничением «исключает»

Команда **DISPLAY** используется для того, чтобы отобразить на экране распределение памяти. Синтаксис команды показан на рис. 4.11. Скобки представляют альтернативные необязательные операнды. Прописные буквы обозначают ключевые слова операндов, а буквы с предшествующими точками — значения операндов



Рис. 4.11. Синтаксис команды **DISPLAY**

(т. е. действительные значения операндов, которые должны быть подставлены). Подчеркнутые операнды соответствуют стандартным значениям (т. е. если операнд опущен, то принимается стандартное значение).

Первый операнд (*адрес 1*)<sup>1</sup> определяет адрес первого байта области памяти, содержимое которой должно быть отображено на экран. Длина адреса задается одной — шестью шестнадцатеричными цифрами (0—9, A—F). Если первый операнд не определен, то предполагается, что адрес равен 0. Адрес должен принимать значение из действительной области памяти машины.

Второй операнд определяет объем памяти, который должен быть отображен. Если *адрес 2* определен, то он в свою очередь определяет адрес последнего байта области памяти, которую необходимо отобразить на экран. Длина этого адреса задается одной — шестью шестнадцатеричными цифрами. Он должен быть больше или равен начальному адресу (*адрес 1*). Аналогично *адрес 2* обязан принимать значения из действительной области памяти машины. Если в качестве второго операнда определено **END**, то память отображается до последнего действительного адреса машины. Если же в качестве операнда определен *счетчик байтов*, то он в свою очередь определяет число байтов памяти, которые нужно отобразить (начиная

<sup>1</sup> *Адрес 1* и *адрес 2* — шестнадцатеричные. — Примеч. ред.

с байта с адресом *адрес 1*). Операнд *счетчик байтов* является шестнадцатеричным целым числом (длиной от одной до шести цифр). Сумма значений операндов *счетчик байтов* и *адрес 1* не должна превышать действительного размера памяти плюс единица, а *счетчик байтов* должен по крайней мере иметь значение 1. Состояние памяти отображается на экран терминала в виде одной или нескольких строк следующего формата:

*xxxxxx* = слово 1 слово 2 слово 3 слово 4,

где *xxxxxx* есть шестнадцатеричный адрес *слово 1*.

Всегда отображается полное число слов (четыре байтовых последовательностей, где адрес первого байта в слове кратен четырем) независимо от значения операнда *адрес 1* или отображаемого объема памяти. Все выходные строки всегда содержат четыре слова (16 байт). Первый байт отображаемой области памяти находится в пределах первого слова.

Могут иметь место следующие сообщения об ошибках:

M1 НЕПРАВИЛЬНЫЙ СИНТАКСИС КОМАНДЫ  
M2 ЗАПРАШИВАЕТСЯ АДРЕС БОЛЬШИЙ ДОПУСТИМОГО

M3 ЗАПРАШИВАЕТСЯ ОБЛАСТЬ ПАМЯТИ С  
НУЛЕВЫМ ИЛИ ОТРИЦАТЕЛЬНЫМ АДРЕС-  
СОМ

Примеры команды DISPLAY:

DISPLAY

отображает первые четыре слова в памяти (стандартное значение начального адреса 0, а стандартное значение счетчика байтов 1);

DISPLAY 77F

отображает слово, содержащее байт с адресом 77F, и три последующих слова;

DISPLAY 77F — 407A

отображает слова, содержащие байты с адресами от 77F до 407A;

DISPLAY 77F.6

отображает слова, содержащие шесть байт, начиная с адреса 77F;

DISPLAY 50FF — END

отображает слова, содержащие байты с адреса 50FF до конца памяти.

Первый шаг заключается в тщательном анализе спе-

цификации с тем, чтобы идентифицировать причины и следствия. Причинами являются:

1. Наличие первого операнда.
2. Операнд *адрес 1* содержит только шестнадцатеричные цифры.
3. Операнд *адрес 1* содержит от одного до шести символов.
4. Операнд *адрес 1* находится в пределах действительной области памяти.
5. Второй операнд есть END.
6. Второй операнд есть *адрес 2*.
7. Второй операнд есть *счетчик байтов*.
8. Второй операнд отсутствует.
9. Операнд *адрес 1* содержит только шестнадцатеричные цифры.
10. Операнд *адрес 2* содержит от одного до шести символов.
11. Операнд *адрес 2* находится в пределах действительной области памяти.
12. Операнд *адрес 2* больше или равен операнду *адрес 1*.
13. Операнд *счетчик байтов* содержит только шестнадцатеричные цифры.
14. Операнд *счетчик байтов* содержит от одного до шести символов.
15.  $\text{Счетчик байтов} + \text{адрес 1} \leq \text{размер памяти} + 1$ .
16.  $\text{Счетчик байтов} \geq 1$ .
17. Запрашиваемая область памяти настолько велика, что требуется много строк на экране.
18. Начало области не выравнено на границу слова.

Каждой причине соответствует произвольный единственный номер. Заметим, что для описания второго операнда необходимы четыре причины (5—8), так как второй операнд может принимать значения 1) END, 2) *адрес 2*, 3) *счетчик байтов*, 4) может отсутствовать и 5) неопределенное значение, т. е. ни одно из указанных выше.

Следствия:

91. На экран отображается сообщение M1.
92. На экран отображается сообщение M2.
93. На экран отображается сообщение M3.
94. Память отображается на одной строке.
95. Для отображения состояния памяти требуется много строк.

96. Первый байт отображаемой области памяти выравнен на границу слова.

97. Первый байт отображаемой области памяти не выравнен на границу слова.

Второй шаг — разработка функциональной диаграммы.

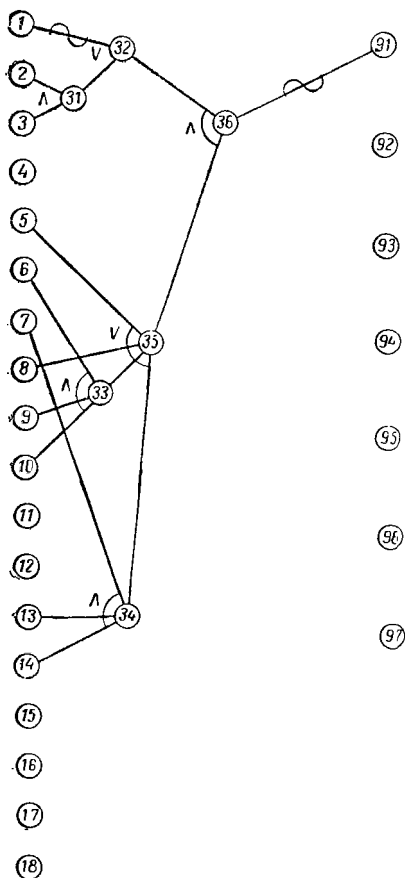


Рис. 4.12. Начальная версия функциональной диаграммы команды DISPLAY

Узлы причин перечислены по вертикали у левого края страницы; узлы следствий собраны по вертикали у ее правого края. Тщательно анализируется семантическое содержание спецификации с тем, чтобы связать причины и следствия (т. е. показать, при каких условиях имеет место следствие).

На рис. 4.12 приведена начальная версия диаграммы. Промежуточный узел 32 представляет синтаксически правильный первый операнд, узел 35 — синтаксически правильный второй операнд, а узел 36 — синтаксически правильную команду. Если значение узла 36 есть 1, то следствие 91 (сообщение об ошибке) отсутствует. Если значение узла 36 есть 0, то следствие 91 имеет место.

На рис. 4.13 изображена полная функциональная диаграмма. Читатель должен тщательно проверить эту диаграмму и убедиться в том, что она точно отображает спецификацию.

Если диаграмму рис. 4.13 непосредственно использовать для построения тестов, то создание многих из них на самом деле окажется невозможным. Это объясняется тем, что определенные комбинации причин не могут иметь места из-за синтаксических ограничений. Например, причины 2 и 3 не могут присутствовать без причи-

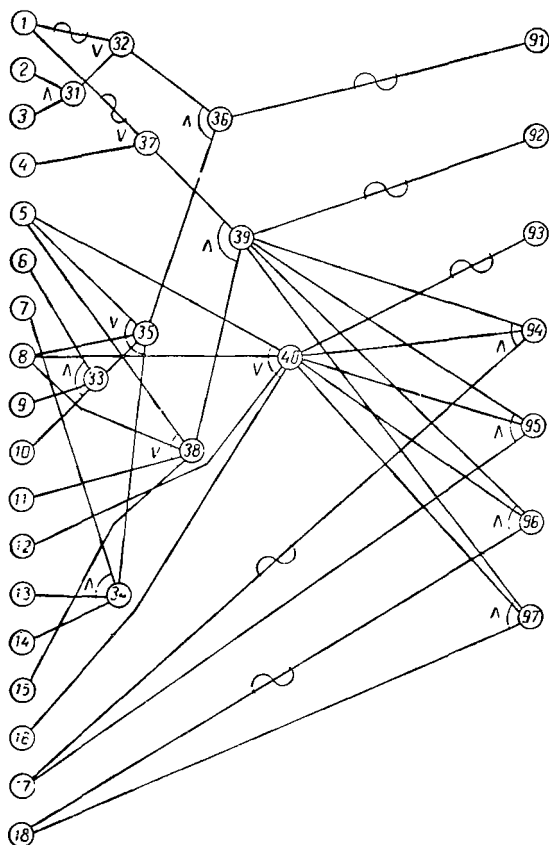


Рис. 4.13. Полная функциональная диаграмма без ограничений

ны 1. Причина 4 не может присутствовать, если нет причин 2 и 3. На рис. 4.14 показана окончательная диаграмма со всеми дополнительными ограничениями. Заметим,

что может присутствовать только одна из причин 5, 6, 7 или 8. Другие ограничения причин являются условиями типа *требуется*. Причина 17 (много строк на экране) и причина 8 (второй операнд отсутствует) связаны отношением *не*; причина 17 может присутствовать только в отсут-

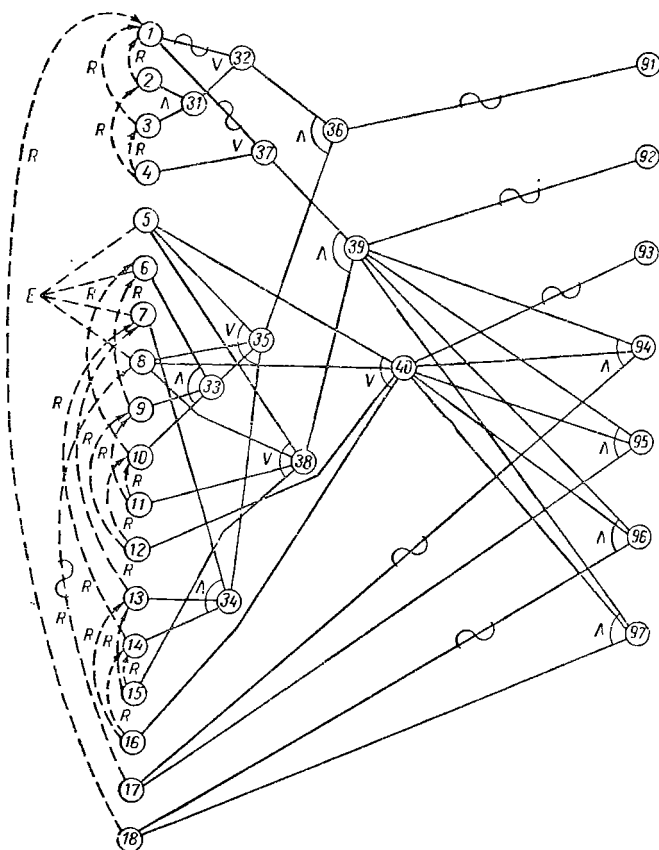


Рис. 4.14. Окончательная функциональная диаграмма команды DISPLAY

ствие причины 8. Читатель опять-таки должен тщательно исследовать все ограничения, налагаемые на условия.

Третьим шагом является генерация таблицы решений с ограниченными входами. Для читателей, знакомых с таблицами решений, причины есть условия, а следствия

есть действия. Процедура генерации заключается в следующем:

1. Выбрать некоторое следствие, которое должно быть в состоянии 1.
2. Найти все комбинации причин (с учетом ограничений), которые установят это следствие в 1, прокладывая из этого следствия обратную трассу через диаграмму.
3. Построить столбец в таблице решений для каждой комбинации причин.
4. Для каждой комбинации причин определить состояния всех других следствий и поместить их в соответствующий столбец таблицы решений.

При выполнении этого шага необходимо руководствоваться тремя положениями:

1. Если обратная трасса прокладывается через узел *или*, выход которого должен принимать значение 1, то одновременно не следует устанавливать в 1 более одного входа в этот узел. Такое ограничение на установку входных значений называется *чувствительностью* пути. Цель данного правила — избежать пропуска определенных ошибок из-за того, что одна причина маскируется другой.
2. Если обратная трасса прокладывается через узел *и*, выход которого должен принимать значение 0, то все комбинации входов, приводящие выход в 0, должны быть в конечном счете перечислены. Однако когда исследуется ситуация, где один вход есть 0, а один или более других входов есть 1, не обязательно перечислять все условия, при которых остальные входы могут быть 1.
3. Если обратная трасса прокладывается через узел *и*, выход которого должен принимать значение 0, то необходимо указать лишь одно условие, согласно которому *все* входы являются нулями. (Когда узел *и* находится в середине графа, и его входы исходят из других промежуточных узлов, может существовать чрезвычайно большое число ситуаций, при которых все его входы принимают значения 0.)

Эти положения кратко поясняются рис. 4.15. Рис. 4.16 приведен в качестве примера функциональной диаграммы. Пусть требуется так задать входные условия, чтобы установить выходное состояние в 0. Согласно положению 3 следует рассматривать только один случай, когда

узлы 5 и 6 — нули. По положению 2 для состояния, при котором узел 5 принимает значение 1, а узел 6 — значение 0, следует рассматривать только один случай, когда узел 5 принимает значение 1 (не перечисляя другие возможные случаи, когда узел 5 может принимать значе-

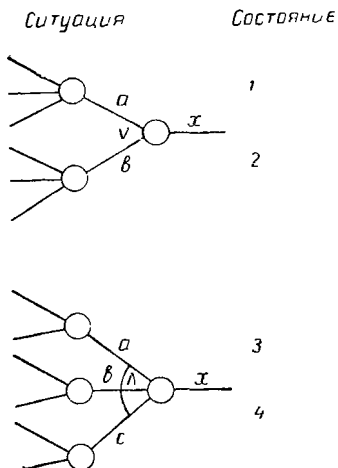


Рис. 4.15. Положения, используемые при прокладке обратной трассы через диаграмму:

1. Если  $X$  должен быть равен 1, то не следует рассматривать ситуацию, где  $a=b=1$  (положение 1).

2. Если  $X$  должен быть равен 0, то перечислить все ситуации, где  $a=b=0$ .

3. Если  $X$  должен быть равен 1, то перечислить все ситуации, где  $a=b=c=1$ .

4. Если  $X$  должен быть равен 0, то рассмотреть только одну ситуацию, где  $a=b=c=0$  (положение 3).

Для состояний  $a, b$  и  $c$  001, 010, 100, 011, 101 и 110 рассмотреть только одну, любую из этих ситуаций (положение 2).

ние 1). Аналогично для состояния, при котором узел 5 принимает значение 0, а узел 6 — значение 1, следует рассматривать только один случай, когда узел 6 принимает значение 1 (хотя в данном примере он является единственным). В соответствии с положением 1 если узел 5 должен быть установлен в состояние 1, то не рекомендуется устанавливать узлы 1 и 2 в состояние 1 одновременно. Таким образом, возможны пять состояний узлов 1—4, например значения

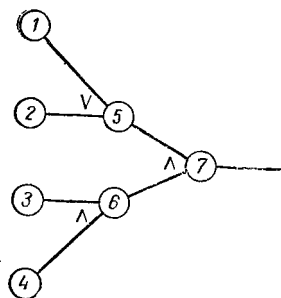


Рис. 4.16. Пример функциональной диаграммы для иллюстрации обратной трассировки

0 0 0 0	(5=0, 6=0)
1 0 0 0	(5=1, 6=0)
1 0 0 1	(5=1, 6=0)
1 0 1 0	(5=1, 6=0)
0 0 1 1	(5=0, 6=1)

а не 13, которые приводят к выходному состоянию 0.

На первый взгляд эти положения могут показаться необъективными, но они преследуют важную цель: уменьшить комбинаторику диаграммы. Их применение позволяет избежать ситуаций, которые приводят к получению малорезультативных тестов. Если не исключать малорезультативные тесты, то общее число тестов, порождаемых по большой функциональной диаграмме, получается астрономическим. Если же и без них число тестов все еще оказывается большим, то выбирается некоторое подмножество тестов, но при этом не гарантируется, что малорезультативные тесты будут исключены. Поэтому самое лучшее — исключить их в процессе анализа диаграммы.

Преобразуем функциональную диаграмму, изображенную на рис. 4.14, в таблицу решений. Выберем первым следствием 91. Следствие 91 имеет место, если узел 36 принимает значение 0. Значение узла 36 есть 0, если значение узлов 32 и 35 есть 0,0, 0,1 или 1,0 и применимы положения 2 и 3. Хотя подобное преобразование — трудоемкий процесс, но можно проложить обратную трассу от следствий к причинам, учесть при этом ограничения причин и найти комбинации последних, которые приводят к следствию 91.

Результирующая таблица решений при условии, что имеет место следствие 91, показана на рис. 4.17 (столбцы 1—11). Столбцы (тесты) 1—3 представляют условия, где узел 32 есть 0, а узел 35 есть 1. Столбцы 4—10 представляют условия, где узел 32 есть 1, а узел 35 есть 0. С помощью положения 3 определена только одна ситуация (колонка 11) из 21 возможной, когда узлы 32 и 35 есть 0. Пробелы представляют «безразличные» ситуации (т. е. состояние причины несущественно) или указывают на то, что состояние причины очевидно вследствие состояний других зависимых причин (например, для столбца 1 известно, что причины 5, 7 и 8 должны принимать значения 0, так как они связаны ограничением «одно и только одно» с причиной 6).

Столбцы 12—15 представляют ситуации, при которых

имеет место следствие 92, а столбцы 16 и 17 — ситуации, при которых имеет место следствие 93. На рис. 4.18 показана оставшаяся часть таблицы решений.

Последний шаг заключается в том, чтобы преобразовать таблицу решений в 38 тестов. Набор из 38 тестов

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1
4												1	1	0	0	1	1
5				0										1			
6	1	1	1	0	1	1	1				1	1			1	1	
7				0				1	1	1			1				1
8				0													
9	1	1	1		1	0	0				0	1			1	1	
10	1	1	1		0	1	0				1	1			1	1	
11												0			0	1	
12																0	
13								1	0	0			1				1
14								0	1	0			1				1
15													0				
16																	0
17																	
18																	
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рис. 4.17. Первая половина результирующей таблицы решений

представлен ниже. Числа возле каждого теста обозначают следствия, которые, как ожидается, должны здесь иметь место. Предположим, что последний используемый адрес памяти машины есть 7FFF.

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	
1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1
2	1	1	1	1					1	1	1	1	1	1	1				1	1	1	
3	1	1	1	1					1	1	1	1	1	1	1				1	1	1	
4	1	1	1	1					1	1	1	1	1	1	1				1	1	1	
5	1				1				1				1			1			1			
6			1				1				1			1			1			1		
7				1				1				1			1			1			1	
8		1				1				1												
9			1				1				1			1			1			1		
10			1				1				1			1			1			1		
11			1				1				1			1			1			1		
12			1				1				1			1			1			1		
13				1				1				1			1			1			1	
14				1				1				1			1			1			1	
15				1				1				1			1			1			1	
16				1				1				1			1			1			1	
17	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
18	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
94	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	
95	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
96	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	
97	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	

Рис. 4.18. Вторая половина результирующей таблицы решений

- |                              |         |
|------------------------------|---------|
| 1. DISPLAY 234AF74—123       | (91)    |
| 2. DISPLAY 2ZX4—3000         | (91)    |
| 3. DISPLAY HHHHHHHH—2000     | (91)    |
| 4. DISPLAY 200 200           | (91)    |
| 5. DISPLAY 0 — 22222222      | (91)    |
| 6. DISPLAY 1 — 2X            | (91)    |
| 7. DISPLAY 2 — ABCDEFGHI     | (91)    |
| 8. DISPLAY 3.1111111         | (91)    |
| 9. DISPLAY 44.\$42           | (91)    |
| 10. DISPLAY 100.\$\$\$\$\$\$ | (91)    |
| 11. DISPLAY 10000000—M       | (91)    |
| 12. DISPLAY FF — 8000        | (92)    |
| 13. DISPLAY FFF.7001         | (92)    |
| 14. DISPLAY 8000 — END       | (92)    |
| 15. DISPLAY 8000 — 8001      | (92)    |
| 16. DISPLAY AA — A9          | (93)    |
| 17. DISPLAY 7000.0           | (93)    |
| 18. DISPLAY 7FF9 — END       | (94,97) |

19. DISPLAY	1	(94,97)
20. DISPLAY	21—29	(94,97)
21. DISPLAY	4021.A	(94,97)
22. DISPLAY	— END	(94,96)
23. DISPLAY		(94,96)
24. DISPLAY	— F	(94,96)
25. DISPLAY	.E	(94,96)
26. DISPLAY	7FF8 — END	(94,96)
27. DISPLAY	6000	(94,96)
28. DISPLAY	A0—A4	(94,96)
29. DISPLAY	20.8	(94,96)
30. DISPLAY	7001 — END	(95,97)
31. DISPLAY	5—15	(95,97)
32. DISPLAY	4FF.100	(95,97)
33. DISPLAY	— END	(95,96)
34. DISPLAY	—20	(95,96)
35. DISPLAY	.11	(95,96)
36. DISPLAY	7000 — END	(95,96)
37. DISPLAY	4—14	(95,96)
38. DISPLAY	500.11	(95,96)

Заметим, что в том случае, когда двум или более различным тестам соответствует один и тот же набор причин, следует стараться выбирать различные значения причин с тем, чтобы хотя бы незначительно улучшить результативность тестов. Заметим также, что из-за ограниченного размера памяти тест 22 является нереализуемым (при его использовании будет получено следствие 95 вместо 94, как отмечено в тесте 33). Следовательно, реализуемы только 37 тестов.

### З а м е ч а н и я

Применение функциональных диаграмм — систематический метод генерации тестов, представляющих комбинации условий. Альтернативой является специальный выбор комбинаций, но при этом существует вероятность пропуска многих «интересных» тестов, определенных с помощью функциональной диаграммы.

При использовании функциональных диаграмм требуется трансляция спецификации в булевскую логическую сеть. Следовательно, этот метод открывает перспективы ее применения и дополнительные возможности спецификаций. Действительно, разработка функциональных диаграмм есть хороший способ обнаружения неполноты и неоднозначности в исходных спецификациях. Например, читатель может заметить, что предложенный процесс обнаруживает неполноту в спецификации команды DISPLAY. Спецификация устанавливает, что все выход-

ные строки содержат четыре слова. Это справедливо не во всех случаях; так, для тестов 18 и 26 это неверно, поскольку для них начальный адрес отображаемой памяти отличается от конечного адреса памяти машины менее чем на 16 байт.

Метод функциональных диаграмм позволяет построить набор полезных тестов, однако его применение обычно не обеспечивает построение *всех* полезных тестов, которые могут быть определены. Так, в нашем примере мы ничего не сказали о проверке идентичности отображаемых на экран терминала значений данных данным в памяти и об установлении случая, когда программа может отображать на экран любое возможное значение, хранящееся в ячейке памяти. Кроме того, функциональная диаграмма неадекватно исследует граничные условия. Конечно, в процессе работы с функциональными диаграммами можно попробовать покрыть граничные условия. Например, вместо определения единственной причины

$$\text{адрес } 2 \geq \text{адрес } 1$$

можно определить две причины

$$\begin{aligned}\text{адрес } 2 &= \text{адрес } 1 \\ \text{адрес } 2 &> \text{адрес } 1\end{aligned}$$

Однако при этом граф существенно усложняется, и число тестов становится чрезвычайно большим. Поэтому лучше отделить анализ граничных значений от метода функциональных диаграмм. Например, для спецификации команды DISPLAY могут быть определены следующие граничные условия:

1. Адрес 1 длиной в одну цифру.
2. Адрес 1 длиной в шесть цифр.
3. Адрес 1 длиной в семь цифр.
4. Адрес 1=0.
5. Адрес 1=7FFF.
6. Адрес 1=8000.
7. Адрес 2 длиной в одну цифру.
8. Адрес 2 длиной в шесть цифр.
9. Адрес 2 длиной в семь цифр.
10. Адрес 2=0.
11. Адрес 2=7FFF.
12. Адрес 2=8000.
13. Адрес 2=адрес 1.
14. Адрес 2=адрес 1+1.
15. Адрес 2=адрес 1-1.

16. *Счетчик байтов* длиной в одну цифру.
17. *Счетчик байтов* длиной в шесть цифр.
18. *Счетчик байтов* длиной в семь цифр.
19. *Счетчик байтов* = 1.
20. *Адрес 1 + счетчик байтов* = 8000.
21. *Адрес 1 + счетчик байтов* = 8001.
22. Отображение шестнадцати байт (одна строка).
23. Отображение семнадцати байт (две строки).

Это вовсе не означает, что следует писать 60 (37 + 23) тестов. Поскольку функциональная диаграмма дает только направление в выборе определенных значений операндов, граничные условия могут входить в полученные из нее тесты. В нашем примере, переписывая некоторые из первоначальных 37 тестов, можно покрыть все 23 граничных условия без дополнительных тестов. Таким образом, мы получаем небольшой, но убедительный набор тестов, удовлетворяющий поставленным целям.

Заметим, что метод функциональных диаграмм согласуется с некоторыми принципами тестирования, изложенными в гл. 2. Его неотъемлемой частью является определение ожидаемого выхода каждого теста (все столбцы в таблице решений обозначают ожидаемые следствия). Заметим также, что данный метод помогает выявить ошибочные побочные следствия. Например, столбец (тест) 1 устанавливает, что должно присутствовать следствие 91 и что следствия 92—97 должны отсутствовать.

Наиболее трудным при реализации метода является преобразование диаграммы в таблицу решений. Это преобразование представляет собой алгоритмический процесс. Следовательно, его можно автоматизировать посредством написания соответствующей программы. Фирма IBM имеет ряд таких программ, но не предоставляет их.

Еще один пример применения функциональных диаграмм можно найти в [2].

## ПРЕДПОЛОЖЕНИЕ ОБ ОШИБКЕ

Замечено, что некоторые люди по своим качествам оказываются прекрасными специалистами по тестированию программ. Они обладают умением «выискивать» ошибки и без привлечения какой-либо методологии тестирования (такой, как анализ граничных значений или применение функциональных диаграмм).

Объясняется это тем, что человек, обладающий прак-

тическим опытом, часто подсознательно применяет метод проектирования тестов, называемый *предположением об ошибке*. При наличии определенной программы он интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения.

Процедуру для метода предположения об ошибке описать трудно, так как он в значительной степени является интуитивным. Основная идея его заключается в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка написать тесты. Например, такая ситуация возникает при значении 0 на входе и выходе программы. Следовательно, можно построить тесты, для которых определенные входные данные имеют нулевые значения и для которых определенные выходные данные устанавливаются в 0. При переменном числе входов или выходов (например, число искомых входных записей при поиске в списке) ошибки возможны в ситуациях типа «никакой» и «один» (например, пустой список, список, содержащий только одну искомую запись). Другая идея состоит в том, чтобы определить тесты, связанные с предположениями, которые программист может сделать во время чтения спецификации (т. е. моменты, которые были опущены из спецификации либо случайно, либо из-за того, что автор спецификации считал их очевидными).

Поскольку данная процедура не может быть четко определена, лучшим способом обсуждения смысла предположения об ошибке представляется разбор примеров. Если в качестве примера рассмотреть тестирование подпрограммы сортировки, то нужно исследовать следующие ситуации:

1. Сортируемый список пуст.
2. Сортируемый список содержит только одно значение.
3. Все записи в сортируемом списке имеют одно и то же значение.
4. Список уже отсортирован.

Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании программы. Если пример заключается в тестировании подпрограммы двоичного поиска, то можно проверить следующие ситуации: 1) существует только один вход в таблицу, в которой ведется поиск; 2) размер

таблицы есть степень двух (например, 16); 3) размер таблицы меньше или больше степени двух (например, 15, 17).

Рассмотрим программу MTEST, приведенную в разделе, посвященном анализу граничных значений. При тестировании этой программы методом предположения об ошибке целесообразно учесть следующие дополнительные тесты:

1. Допускает ли программа «пробел» в качестве ответа?
2. Запись типа 2 (ответ) появляется в наборе записей типа 3 (студент).
3. Запись без 2 или 3 в последней колонке появляется не как начальная запись (название).
4. Два студента имеют одно и то же имя или номер.
5. Поскольку медиана вычисляется по-разному в зависимости от того, четно или нечетно число элементов, необходимо протестировать программу как для четного, так и для нечетного числа студентов.
6. Поле числа вопросов имеет отрицательное значение.

Для команды DISPLAY из предыдущего раздела целесообразно рассмотреть следующие тесты метода предположения об ошибке:

1. DISPLAY 100 — (неполный второй операнд).
2. DISPLAY 100. (неполный второй операнд).
3. DISPLAY 100—10A42 (слишком большое значение операнда).
4. DISPLAY 000—0000FF (нули слева).

## СТРАТЕГИЯ

Методологии проектирования тестов, обсуждавшиеся в этой главе, могут быть объединены в общую стратегию. Причина объединения их теперь становится очевидной: каждый метод обеспечивает создание определенного набора используемых тестов, но ни один из них сам по себе не может дать полный набор тестов. Приемлемая стратегия состоит в следующем:

1. Если спецификация содержит комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм.
2. В любом случае необходимо использовать анализ граничных значений. Напомним, что этот метод

включает анализ граничных значений входных и выходных переменных. Анализ граничных значений дает набор дополнительных тестовых условий, но, как замечено в разделе, посвященном функциональным диаграммам, многие из них (если не все) могут быть включены в тесты метода функциональных диаграмм.

3. Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.
4. Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.
5. Проверить логику программы на полученном наборе тестов. Для этого нужно воспользоваться критерием покрытия решений, покрытия условий, покрытия решений/условий либо комбинаторного покрытия условий (последний критерий является более полным). Если необходимость выполнения критерия покрытия приводит к построению тестов, не встречающихся среди построенных на предыдущих четырех шагах, и если этот критерий не является нереализуемым (т. е. определенные комбинации условий невозможно создать вследствие природы программы), то следует дополнить уже построенный набор тестов тестами, число которых достаточно для удовлетворения критерию покрытия.

Эта стратегия опять-таки не гарантирует, что все ошибки будут найдены, но вместе с тем ее применение обеспечивает приемлемый компромисс. Реализация подобной стратегии весьма трудоемка, но ведь никто и никогда не утверждал, что тестирование программы — легкое дело.

#### ЛИТЕРАТУРА

1. El mendorf W. R. Cause-Effect Graphs in Functional Testing. TR—00. 2487, IBM Systems Development Division, Poughkeepsie, N. Y., 1973.
2. Myers G. J. Software Reliability: Principles and Practices. New York, Wiley-Interscience, 1976. Русский перевод: Майс Г. Надежность программного обеспечения. М., Мир, 1980.

## ГЛАВА 5

### ТЕСТИРОВАНИЕ МОДУЛЕЙ

Рассматривая проблемы тестирования, мы до сих пор в основном не учитывали такие факторы, как организация процедуры тестирования и размер тестируемых программ. Однако переход к большим программам (500 операторов и более) требует специальных способов структурирования процесса тестирования. В этой главе мы рассмотрим начальный шаг структурирования — тестирование модулей. Последующие шаги рассматриваются в гл. 6.

Тестирование модулей (или блоков) представляет собой процесс тестирования отдельных подпрограмм или процедур программы. Здесь подразумевается, что, прежде чем начинать тестирование программы в целом, следует протестировать отдельные небольшие модули, образующие эту программу. Такой подход мотивируется тремя причинами. Во-первых, появляется возможность управлять комбинаторикой тестирования, поскольку первоначально внимание концентрируется на небольших модулях программы. Во-вторых, облегчается задача отладки программы, т. е. обнаружение места ошибки и исправление текста программы. Наконец, в-третьих, допускается параллелизм, что позволяет одновременно тестировать несколько модулей.

Цель тестирования модулей — сравнение функций, реализуемых модулем, со спецификациями его функций или интерфейса. Подчеркнем, что эта же цель выдвигается и при общей постановке задачи тестирования, причем проблема состоит не в том, чтобы установить соответствие модуля спецификации, а в том, чтобы показать противоречие между ними. Процесс тестирования модулей рассматривается в трех аспектах: способы построения наборов тестов, порядок, в котором модули тестиру-

ются и собираются в программу, и некоторые практические рекомендации по реализации тестирования.

## ПРОЕКТИРОВАНИЕ ТЕСТОВ

При проектировании тестов для тестирования модулей должны быть доступны два источника информации: спецификация модуля и его текст. Спецификация обычно содержит описание входных и выходных параметров модуля и его функций.

Тестирование модулей в основном ориентировано на

Имя	Код работы	Отдел	Оклад

Отдел	Сумма продаж

Рис. 5.1. Таблица входов модуля BONUS

принцип белого ящика. Это объясняется прежде всего тем, что принцип белого ящика труднее реализовать при переходе в последующем к тестированию более крупных единиц, например программ в целом. Кроме того, последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, т. е. ошибок, не обязательно связанных с логикой программы, а возникающих, например, из-за несоответствия программы требованиям пользователя. Следовательно, процедура создания набора тестов для тестирования модулей такова: анализируется логика отдельного модуля с помощью одного или нескольких методов белого ящика, а затем этот набор те-

стов применяется при тестировании методами черного ящика по спецификации модуля.

Проиллюстрируем использование методов построения тестов, описанных в гл. 4, при тестировании модулей. Допустим, нужно тестировать модуль BONUS. Функция этого модуля состоит в увеличении на 200 дол. размера

### MODULE TESTING

```

BONUS: PROCEDURE (EMPTAB, DEPTTAB, ESIZE, DSIZE, ERRCODE);
DECLARE 1 EMPTAB (*),
        2 NAME CHAR (6),
        2 CODE CHAR (1),
        2 DEPT CHAR (3),
        2 SALARY FIXED DECIMAL (7, 2);
DECLARE 1 DEPTTAB (*),
        2 DEPT CHAR (3),
        2 SALES FIXED DECIMAL (8, 2);
DECLARE (ESIZE, DSIZE) FIXED BINARY;
DECLARE ERRCODE FIXED DECIMAL (1);
DECLARE MAXSALES FIXED DECIMAL (8, 2) INIT (0); /*MAX. SALES IN DEPTTAB*/
DECLARE (I, J, K) FIXED BINARY; /*COUNTERS*/
DECLARE FOUND BIT (1); /*TRUE IF ELIGIBLE DEPT. HAS EMPLOYEES*/
DECLARE SINC FIXED DECIMAL (7, 2) INIT (200.00); /*STANDARD INCREMENT*/
DECLARE LINC FIXED DECIMAL (7, 2) INIT (100.00); /*LOWER INCREMENT*/
DECLARE LSALARY FIXED DECIMAL (7, 2) INIT (15000.00); /*SALARY BOUNDARY*/
DECLARE MGR CHAR (1) INIT ('M');

1  ERRCODE=0;
2  IF (ESIZE<=0) | (DSIZE<=0)
3  THEN ERRCODE=1; /*EMPTAB OR DEPTTAB ARE EMPTY*/
4  ELSE DO;
5      DO I = 1 TO DSIZE; /*FIND MAXSALES AND MAXDEPTS*/
6          IF (SALES (I) >= MAXSALES) THEN MAXSALES=SALES (I);
7      END;
8      DO J = 1 TO DSIZE;
9          IF (SALES (J) = MAXSALES) /*ELIGIBLE DEPARTMENT*/
10         THEN DO;
11             FOUND='D'B;
12             DO K = 1 TO ESIZE;
13                 IF (EMPTAB.DEPT (K) = DEPTTAB.DEPT (J))
14                 THEN DO;
15                     FOUND='1'B;
16                     IF (SALARY (K) >= LSALARY) | (CODE (K) = 'MGR')
17                     THEN SALARY (K) = SALARY (K) + LINC;
18                     ELSE SALARY (K) = SALARY (K) + SINC;
19                 END;
20             END;
21             IF (~FOUND) THEN ERRCODE=2;
22         END;
23     END;
24     END;
25 END;

```

Рис. 5.2. Модуль BONUS

заработной платы всем служащим отдела или отделов, обеспечивших реализацию товаров на наибольшую сумму. Однако служащим, зарплата которых превышает 15000 дол., и администраторам надбавка составляет 100 дол.

На рис. 5.1 показаны таблицы входных величин. По результатам работы модуль формирует код ошибки. Если

ошибок нет, то код ошибки равен 0. Если в обеих таблицах нет входных величин, то код ошибки равен 1. Если же список служащих какого-то отдела пуст, то будет выработан код ошибки, равный 2.

На рис. 5.2 представлен текст модуля. Входные параметры ESIZE и DSIZE содержат номера входов в таблицу служащих и таблицу отделов соответственно. Модуль написан на языке PL/1, но последующее обсуждение не зависит от выбранного языка программирования; одна и та же технология может быть использована для программ, написанных на различных языках. Логика модуля достаточно проста и должна быть понятна читателям, которые незнакомы с PL/1.

Вне зависимости от используемого метода покрытия логики на первом шаге нужно составить список всех принятых решений в программе. Целесообразно анализировать все операторы IF и DO. Анализируя текст программы, можно убедиться в том, что операторы DO представляют собой простые итерации, в которых конечное значение параметра цикла больше или равно начальному значению (это означает, что каждый цикл выполняется хотя бы один раз), и единственный способ выхода из цикла возможен через оператор DO. Следовательно, операторы DO не требуют к себе специального внимания, поскольку любой тест, который приводит к исполнению цикла DO, выполнит ветвление в обоих направлениях (т. е. при входе и при выходе из цикла). По этой причине необходимо проанализировать следующие операторы:

```
2 IF (EIZE <= 0) | (DSIZE <= 0)
6 IF (SALES(I) >= MAXSALES)
9 IF (SALES(J) = MAXSALES)
13 IF (EMPTAB.DEPT(K) = DEPTTAB.DEPT(J))
16 IF (SALARY(K) >= LSALARY) | (CODE(K) = MGR)
21 IF ( ¬ FOUND)
```

Число решений невелико, поэтому скорее всего следует выбрать покрытие многих условий, но необходимо испытать все критерии покрытия логики (за исключением покрытия операторов, использование которого нецелесообразно), чтобы рассмотреть их результативность.

Для того чтобы удовлетворить критерию покрытия решений, необходимо построить тесты, которые приводили бы к исполнению каждого из шести операторов в обоих направлениях. Требуемые для этого входные ситуации приведены в табл. 5.1. Существует 10 ситуаций, которые

Ситуации, соответствующие исходам решений

Решение	Истинный исход	Ложный исход
2 6	ESIZE или DSIZE $\leq 0$ Случается хотя бы один раз	ESIZE и DSIZE $> 0$ Порядок в DEPTTAB таков, что отдел с меньшим уровнем продажи следует в списке отделов следом за отделом с более высоким уровнем продажи
9	Случается хотя бы один раз	У всех отделов разные уровни продажи
13	В соответствующем отделе есть служащий	Существует служащий, который числится не в соответствующем отделе
16	Соответствующий служащий — управляющий либо зарабатывает I.SALARY или больше	Соответствующий служащий — не управляющий и зарабатывает меньше I.SALARY
21	Соответствующий отдел не имеет сотрудников	В соответствующем отделе есть хотя бы один сотрудник

Тест	Вход	Предполагаемый выход																														
1	ESIZE=0 Значения всех остальных базовых переменных произвольны	ERRCODE=1 ESIZE, DSIZE, EMPTAB и DEPTTAB не изменяются																														
2	ESIZE=DSIZE=3 EMPTAB <table><tr><td>Джонс</td><td>E</td><td>D42</td><td>21 000,00</td></tr><tr><td>Смит</td><td>E</td><td>D32</td><td>14 000,00</td></tr><tr><td>Лорин</td><td>E</td><td>D42</td><td>10 000,00</td></tr></table> DEPTTAB <table><tr><td>D42</td><td>10 000,00</td></tr><tr><td>D32</td><td>8 000,00</td></tr><tr><td>D95</td><td>10 000,00</td></tr></table>	Джонс	E	D42	21 000,00	Смит	E	D32	14 000,00	Лорин	E	D42	10 000,00	D42	10 000,00	D32	8 000,00	D95	10 000,00	ERRCODE=2 ESIZE, DSIZE и DEPTTAB не изменяются EMPTAB <table><tr><td>Джонс</td><td>F</td><td>D42</td><td>21 100,00</td></tr><tr><td>Смит</td><td>F</td><td>D32</td><td>14 000,00</td></tr><tr><td>Лорин</td><td>E</td><td>D42</td><td>10 200,00</td></tr></table>	Джонс	F	D42	21 100,00	Смит	F	D32	14 000,00	Лорин	E	D42	10 200,00
Джонс	E	D42	21 000,00																													
Смит	E	D32	14 000,00																													
Лорин	E	D42	10 000,00																													
D42	10 000,00																															
D32	8 000,00																															
D95	10 000,00																															
Джонс	F	D42	21 100,00																													
Смит	F	D32	14 000,00																													
Лорин	E	D42	10 200,00																													

Рис. 5.3. Тесты, удовлетворяющие критерию покрытия решений

нужно проверить с помощью тестов, так как два направления ветвления будут иметь место всегда. Заметим, что при конструировании табл. 5.1 направления выходов из операторов принятия решения должны быть прослежены по логике программы; это даст возможность оценить необходимые входные условия. Например, решению 16 удовлетворяет не любой из служащих, встречавшихся в условиях, а только служащий соответствующего отдела.

Десять интересующих нас ситуаций из табл. 5.1 мо-

гут быть проверены с помощью двух тестов, приведенных на рис. 5.3. Каждый тест в соответствии с изложенным в гл. 2, включает предполагаемые значения выходных величин.

Очевидно, что, хотя тесты и удовлетворяют критерию покрытия решений, в модуле, возможно, остается много не обнаруженных ими типов ошибок. Например, эти тесты не анализируют ситуацию, когда код ошибки равен 0, сотрудник является управляющим или пуста таблица отделов ( $DSIZE \leq 0$ ).

Если воспользоваться методом покрытия условий, то можно получить более приемлемый тест. Этот метод предполагает, что тесты должны приводить к выполнению обоих выходов каждого условия в операторе принятия решения. В табл. 5.2 представлены условия, требуе-

Таблица 5.2

Ситуации, соответствующие выходам из условий

Решение	Условия	Истинное выходное значение	Ложное выходное значение
2	$ESIZE \leq 0$	$ESIZE \leq 0$	$ESIZE > 0$
2	$DSIZE \leq 0$	$DSIZE \leq 0$	$DSIZE > 0$
6	$SALES(I) \geq MAXSALES$	Случается хотя бы один раз	Порядок в DEPTTAB таков, что отдел с меньшей суммой продажи встречается позже, чем отдел с большей суммой
9	$SALES(J) = MAXSALES$	Случается хотя бы один раз	У всех отделов разные уровни продажи
13	$EMPTAB.$ $DEPT(K) = DEPTTAB.$ $DEPT(J)$	В данном отделе есть служащий	Существует служащий, который числится не в соответствующем отделе
16	$SALARY(K) \geq LSALARY$	Данный служащий имеет оклад LSALARY или больше	Данный служащий имеет оклад меньше LSALARY
16	$CODE(K) = MGR$	Данный служащий — управляющий	Данный служащий — не управляющий
21	$\neg FOUND$	В данном отделе нет служащих	В данном отделе есть хотя бы один сотрудник

мые входные ситуации и соответствующие им выходные результаты. Для любого условия следует рассмотреть две входные и две выходные ситуации, поэтому тесты должны проверять всего 14 ситуаций. И вновь эти си-

туации могут быть проверены двумя тестами, которые показаны на рис. 5.4.

Для иллюстрации на рис. 5.4 приведены тесты, которые удовлетворяют методу покрытия условий, поскольку они приводят к исполнению всех выходных результатов, указанных в табл. 5.2. Однако эти тесты, вероятно, сла-

Тест	Вход	Предполагаемый выход																																																								
1	$ESIZE = DSIZE = 0$ Значения всех остальных входных переменных произвольны	$ERRCODE = 1$ $ESIZE, DSIZE, EMPTAB$ и $DEPTTAB$ не изменяются																																																								
2	$ESIZE = DSIZE = 3$ <table><tr><th colspan="4">EMPTAB</th><th colspan="4">DEPTTAB</th></tr><tr><td>Джонс</td><td>E</td><td>D42</td><td>21 000,00</td><td>D42</td><td>10 000,00</td><td></td><td></td></tr><tr><td>Смит</td><td>E</td><td>D32</td><td>14 000,00</td><td>D32</td><td>8 000,00</td><td></td><td></td></tr><tr><td>Лорин</td><td>M</td><td>D42</td><td>10 000,00</td><td>D95</td><td>10 000,00</td><td></td><td></td></tr></table>	EMPTAB				DEPTTAB				Джонс	E	D42	21 000,00	D42	10 000,00			Смит	E	D32	14 000,00	D32	8 000,00			Лорин	M	D42	10 000,00	D95	10 000,00			$ERRCODE = 2$ $ESIZE, DSIZE$ и $DEPTTAB$ не изменяются  $EMPTAB$ <table><tr><td>Джонс</td><td>E</td><td>D42</td><td>21 100,00</td><td></td><td></td><td></td><td></td></tr><tr><td>Смит</td><td>E</td><td>D32</td><td>14 000,00</td><td></td><td></td><td></td><td></td></tr><tr><td>Лорин</td><td>M</td><td>D42</td><td>10 100,00</td><td></td><td></td><td></td><td></td></tr></table>	Джонс	E	D42	21 100,00					Смит	E	D32	14 000,00					Лорин	M	D42	10 100,00				
EMPTAB				DEPTTAB																																																						
Джонс	E	D42	21 000,00	D42	10 000,00																																																					
Смит	E	D32	14 000,00	D32	8 000,00																																																					
Лорин	M	D42	10 000,00	D95	10 000,00																																																					
Джонс	E	D42	21 100,00																																																							
Смит	E	D32	14 000,00																																																							
Лорин	M	D42	10 100,00																																																							

Рис. 5.4. Тесты, удовлетворяющие критерию покрытия условий

бее, чем тесты рис. 5.3, удовлетворяющие критерию покрытия решений. Суть в том, что они не приводят к исполнению каждого оператора; оператор 18 не исполняется никогда. Их результативность не выше, чем у тестов рис. 5.3. Здесь невозможна выходная ситуация  $ERRCODE = 0$ . Если, допустим, в операторе 2 запись условия  $(ESIZE = 0) \& (DSIZE = 0)$  ошибочна, то эта ошибка не будет обнаружена. Разумеется, данную проблему можно решить применением дополнительных тестовых наборов, но факт остается фактом — тесты, приведенные на рис. 5.4, удовлетворяют критерию покрытия условий.

Используя критерий покрытия условий, можно продемонстрировать недостатки набора тестов рис. 5.4. Построим тесты, которые обеспечат выполнение хотя бы один раз каждого выхода всех условий и всех решений. Этого можно достигнуть, если в тесте сделать Джонса управляющим, а Лорина — не управляющим. В результате будут выполнены обе ветви решения 16 и оператор 18.

И все же этот тестовый набор, по существу, не лучше тестового набора на рис. 5.3. Если используемый компилятор прекращает оценку выражения *или* сразу же, как только один из операторов принимает значение «истина», то результат выражения  $CODE(K) = MGR$  в операторе

ре 16 никогда не примет значения *истина*. Следовательно, если это выражение закодировано неверно, то данная ошибка не будет обнаружена набором тестов.

Последним исследуем критерий комбинаторного покрытия условий. Этот критерий требует, чтобы соответствующий ему набор тестов приводил к исполнению всех возможных комбинаций условий в каждом решении.

Тест	Вход	Предполагаемый выход																																																
1	ESIZE=0 DSIZE=0 Значения всех остальных входных переменных произвольны	ERRCODE=1 ESIZE, DSIZE, EMPTAB и DEPTAB не изменяются																																																
2	ESIZE=0 DSIZE>0 Значения всех остальных входных переменных произвольны	То же																																																
3	ESIZE>0 DSIZE=0 Значения всех остальных входных переменных произвольны	То же																																																
4	ESIZE=5 DSIZE=4 <div><div>EMPTAB</div><table><tr><td>Джонс</td><td>М</td><td>D42</td><td>21 000,00</td></tr><tr><td>Уорнс</td><td>М</td><td>D95</td><td>12 000,00</td></tr><tr><td>Лорин</td><td>Е</td><td>D42</td><td>10 000,00</td></tr><tr><td>Тоу</td><td>Е</td><td>D95</td><td>16 000,00</td></tr><tr><td>Смит</td><td>Е</td><td>D32</td><td>14 000,00</td></tr></table></div> <div><div>DEPTAB</div><table><tr><td>D42</td><td>10 000,00</td></tr><tr><td>D32</td><td>8 000,00</td></tr><tr><td>D95</td><td>10 000,00</td></tr><tr><td>D44</td><td>10 000,00</td></tr></table></div>	Джонс	М	D42	21 000,00	Уорнс	М	D95	12 000,00	Лорин	Е	D42	10 000,00	Тоу	Е	D95	16 000,00	Смит	Е	D32	14 000,00	D42	10 000,00	D32	8 000,00	D95	10 000,00	D44	10 000,00	ERRCODE=2 ESIZE, DSIZE и DEPTAB не изменяются  <div><div>EMPTAB</div><table><tr><td>Джонс</td><td>М</td><td>D42</td><td>21 100,00</td></tr><tr><td>Уорнс</td><td>М</td><td>D95</td><td>12 100,00</td></tr><tr><td>Лорин</td><td>Е</td><td>D42</td><td>10 200,00</td></tr><tr><td>Тоу</td><td>Е</td><td>D95</td><td>16 100,00</td></tr><tr><td>Смит</td><td>Е</td><td>D32</td><td>14 000,00</td></tr></table></div>	Джонс	М	D42	21 100,00	Уорнс	М	D95	12 100,00	Лорин	Е	D42	10 200,00	Тоу	Е	D95	16 100,00	Смит	Е	D32	14 000,00
Джонс	М	D42	21 000,00																																															
Уорнс	М	D95	12 000,00																																															
Лорин	Е	D42	10 000,00																																															
Тоу	Е	D95	16 000,00																																															
Смит	Е	D32	14 000,00																																															
D42	10 000,00																																																	
D32	8 000,00																																																	
D95	10 000,00																																																	
D44	10 000,00																																																	
Джонс	М	D42	21 100,00																																															
Уорнс	М	D95	12 100,00																																															
Лорин	Е	D42	10 200,00																																															
Тоу	Е	D95	16 100,00																																															
Смит	Е	D32	14 000,00																																															

Рис. 5.5. Тесты, удовлетворяющие критерию комбинаторного покрытия условий

Воспользуемся для рассмотрения табл. 5.2. Любое из решений 6, 9, 13 и 21 имеет по две комбинации условий, а любое из решений 2 и 16 — по четыре комбинации. Согласно данной методологии, используемой для построения тестов, вначале следует выбрать тест, который покрывает максимальное число комбинаций, затем тест, покрывающий максимальное число комбинаций из оставшихся, и т. д. На рис. 5.5 показан набор тестов, удовлетворяющий критерию комбинаторного покрытия условий. Этот набор более понятен, чем предыдущие, и нам представляется, что именно его и нужно было выбрать с самого начала.

Важно отметить, что модуль BONUS может содержать значительное число ошибок, которые нельзя обнаружить даже тестом, удовлетворяющим критерию комбинаторного покрытия условий. Например, ни один тест

не может воспроизвести ситуацию, когда выходная величина ERRCODE возвращается со значением, равным нулю. Таким образом, если пропущен оператор 1, то ошибка не будет обнаружена. Ошибка также не будет обнаружена, если константе LSALARY при инициализации присвоено значение \$ 15000,01 или если в операторе 16 записано SALARY(K) > LSALARY вместо SALARY(K) ≥ LSALARY. Как правило, ошибки, допускаемые человеком (например, неверно выполнена последняя запись в DEPTTAB или EMPTAB), могут быть обнаружены только случайно.

Из изложенного следует, что, во-первых, критерий комбинаторного покрытия условий превосходит все остальные критерии и, во-вторых, ни один из критериев логических покрытий не может быть использован в качестве единственного средства, на основании которого строятся тесты модуля. Поэтому необходимо сделать следующий шаг и совместить тесты, представленные на рис. 5.5, с набором тестов, построенных методом черного ящика. С этой целью рассмотрим спецификацию интерфейса модуля BONUS, приведенную ниже.

BONUS — модуль, написанный на PL/I, он получает пять параметров, их символические имена — EMPTAB, DEPTTAB, ESIZE, DSIZE, ERRCODE; эти параметры имеют следующие атрибуты:

```
DECLARE1 EMPTAB (*), /*входной и выходной*/
          2NAME CHARACTER (6),
          2CODE CHARACTER (1),
          2DEPT CHARACTER (3),
          2SALARY FIXED DECIMAL (7,2);
DECLARE1 DEPTTAB (*), /*входной*/
          2DEPT CHARACTER (3),
          2SALES FIXED DECIMAL (8,2);
DECLARE (ESIZE, DSIZE) FIXED BINARY; /*входной*/
DECLARE ERRCODE FIXED DECIMAL (1); /*входной*/
```

Передаваемые модулем аргументы имеют атрибуты, перечисленные выше в спецификации. ESIZE и DSIZE задают число входных значений в EMPTAB и DEPTTAB соответственно. Порядок входных значений в EMPTAB и DEPTTAB не имеет значения. Функция модуля состоит в увеличении заработной платы (EMPTAB, SALARY) служащим отдела или отделов, имеющим наибольшую сумму продаж (DEPTTAB, SALES). Если служащий получает заработную плату 15000 дол. и более или если он является управляющим (EMPTAB, CODE=«M»), то прибавка к заработной плате составляет 100 дол.; во всех остальных случаях — 200 дол. Увеличение заработной платы записывается в поле EMPTAB, SALARY. При ESIZE и DSIZE, меньших нуля, вырабатывается код ошибки ERRCODE, равный 1, и никакие другие действия не производятся. Во всех иных ситуациях функции модуля исполняются полностью. Однако если в отделе,

имеющем максимальную сумму продаж, число служащих равно нулю, то переменной ERRCODE присваивается значение, равное 2; во всех остальных случаях — значение, равное 0.

Эту спецификацию нельзя использовать для построения функциональных диаграмм (в ней нет явно выраженного набора входных условий, комбинацию которых нужно исследовать), поэтому рекомендуется метод анализа граничных значений. Определены следующие входные граничные значения:

1. ЕМРТАВ имеет одно входное значение.
2. ЕМРТАВ имеет максимальное число входных значений (65535).
3. ЕМРТАВ имеет 0 входных значений.
4. ЕМРТАВ имеет одно входное значение.
5. ДЕРТТАВ имеет максимальное число входных значений (65535).
6. ДЕРТТАВ имеет 0 входных значений.
7. Отдел с максимальной суммой продаж имеет одного служащего.
8. Отдел с максимальной суммой продаж имеет 65535 служащих.
9. Отдел с максимальной суммой продаж не имеет служащих.
10. Все отделы в ДЕРТТАВ имеют равную сумму продаж.
11. Отдел с максимальной суммой продаж занимает первую позицию в ДЕРТТАВ.
12. Отдел с максимальной суммой продаж занимает последнюю позицию в ДЕРТТАВ.
13. Очередной служащий занимает первую позицию в ЕМРТАВ.
14. Очередной служащий занимает последнюю позицию в ЕМРТАВ.
15. Очередной служащий — управляющий.
16. Очередной служащий — не управляющий.
17. Очередной служащий — не управляющий и имеет заработную плату 14 999,99 дол.
18. Очередной служащий — не управляющий и имеет заработную плату 15 000,00 дол.
19. Очередной служащий — не управляющий и имеет заработную плату 15 000,01 дол.

Выходные граничные значения:

20. ERRCODE=0.
21. ERRCODE=1.

22. `ERRCODE=2`.

23. Увеличиваемая заработная плата составляет 99 999,00 дол., что представляет собой максимум.

Последующее тестовое условие основано на предположении об ошибке.

24. Вслед за отделом, имеющим максимальную сумму продаж и не имеющим служащих, следует другой отдел с максимальной суммой продаж, в списке которого имеются служащие.

Это сделано затем, чтобы определить, действительно ли обработка входных данных завершилась ошибочно в случае если выработан код ошибки `ERRCODE`.

Из анализа перечисленных выше 24 условий видно, что для построения теста условия 2, 5 и 8 не имеют практического значения. Эти условия представляют ситуации, возникновение которых невозможно. В нашем варианте такое предположение вполне допустимо, хотя в общем случае оно небезопасно. На следующем шаге построения тестового набора необходимо сравнить оставшееся 21 условие с уже имеющимся набором тестов (см. рис. 5.5) и определить, какие граничные условия им еще не покрыты. Сравнение показывает, что условия 1, 4, 7, 10, 14, 17, 18, 19, 20, 23 и 24 требуют расширения уже имеющегося тестового набора, представленного на рис. 5.5.

Далее необходимо спроектировать дополнительные тесты, покрывающие эти 11 граничных условий. Один из возможных подходов состоит в том, чтобы объединить все условия с существующими тестами (например, путем модификации некоторым образом набора 4 на рис. 5.5). Но поступать таким образом не рекомендуется, так как при этом можно по небрежности разрушить полноту комбинаторного покрытия условий уже существующего теста. Следовательно, наиболее безопасным способом построения теста является создание новых тестов в дополнение к приведенным на рис. 5.5, причем число тестов, необходимых для покрытия граничных условий, должно быть минимальным. На рис. 5.6 представлены три теста, соответствующие этому требованию. Тест 5 покрывает условия 7, 10, 14, 17, 18, 19 и 20, тест 6 — условия 1, 4, 23, и тест 7 — условие 24.

Итак, при построении приемлемого теста для модуля `BONUS` использовался метод покрытия логики, или принцип белого ящика, на основе которого был разработан тест, представленный на рис. 5.5, а затем по прин-

ципу черного ящика были созданы дополнительные тесты рис. 5.6.

Тест	Вход	Предполагаемый выход																												
5	<p>ESIZE=3 DSIZE=2</p> <p>EMPTAB</p> <table><tr><td>Олли</td><td>E</td><td>D36</td><td>14 999,99</td></tr><tr><td>Бест</td><td>E</td><td>D33</td><td>15 000,00</td></tr><tr><td>Селтоу</td><td>E</td><td>D33</td><td>15 000,01</td></tr></table> <p>DEPTTAB</p> <table><tr><td>D33</td><td>55 400,01</td></tr><tr><td>D36</td><td>55 400,01</td></tr></table>	Олли	E	D36	14 999,99	Бест	E	D33	15 000,00	Селтоу	E	D33	15 000,01	D33	55 400,01	D36	55 400,01	<p>ERRCODE=0</p> <p>ESIZE, DSIZE и DEPTTAB не изменяются</p> <p>EMPTAB</p> <table><tr><td>Олли</td><td>E</td><td>D36</td><td>15 199,99</td></tr><tr><td>Бест</td><td>E</td><td>D33</td><td>15 100,00</td></tr><tr><td>Селтоу</td><td>E</td><td>D33</td><td>15 100,01</td></tr></table>	Олли	E	D36	15 199,99	Бест	E	D33	15 100,00	Селтоу	E	D33	15 100,01
Олли	E	D36	14 999,99																											
Бест	E	D33	15 000,00																											
Селтоу	E	D33	15 000,01																											
D33	55 400,01																													
D36	55 400,01																													
Олли	E	D36	15 199,99																											
Бест	E	D33	15 100,00																											
Селтоу	E	D33	15 100,01																											
6	<p>ESIZE=1 DSIZE=1</p> <p>EMPTAB</p> <table><tr><td>Чуф</td><td>M</td><td>D99</td><td>99 999,99</td></tr></table> <p>DEPTTAB</p> <table><tr><td>D99</td><td>99 000,00</td></tr></table>	Чуф	M	D99	99 999,99	D99	99 000,00	<p>ERRCODE=0</p> <p>ESIZE, DSIZE и DEPTTAB не изменяются</p> <p>EMPTAB</p> <table><tr><td>Чуф</td><td>M</td><td>D99</td><td>99 999,99</td></tr></table>	Чуф	M	D99	99 999,99																		
Чуф	M	D99	99 999,99																											
D99	99 000,00																													
Чуф	M	D99	99 999,99																											
7	<p>ESIZE=2 DSIZE=2</p> <p>EMPTAB</p> <table><tr><td>Доул</td><td>E</td><td>D67</td><td>10 000,00</td></tr><tr><td>Форд</td><td>E</td><td>D22</td><td>33 333,33</td></tr></table> <p>DEPTTAB</p> <table><tr><td>D66</td><td>20 000,00</td></tr><tr><td>D67</td><td>20 000,00</td></tr></table>	Доул	E	D67	10 000,00	Форд	E	D22	33 333,33	D66	20 000,00	D67	20 000,00	<p>ERRCODE=2</p> <p>ESIZE, DSIZE и DEPTTAB не изменяются</p> <p>EMPTAB</p> <table><tr><td>Доул</td><td>E</td><td>D67</td><td>10 200,00</td></tr><tr><td>Форд</td><td>E</td><td>D22</td><td>33 333,33</td></tr></table>	Доул	E	D67	10 200,00	Форд	E	D22	33 333,33								
Доул	E	D67	10 000,00																											
Форд	E	D22	33 333,33																											
D66	20 000,00																													
D67	20 000,00																													
Доул	E	D67	10 200,00																											
Форд	E	D22	33 333,33																											

Рис. 5.6. Дополнительные тесты анализа граничных значений для модуля

## ПОШАГОВОЕ ТЕСТИРОВАНИЕ

Реализация процесса тестирования модулей опирается на два ключевых положения: построение эффективного набора тестов (этот вопрос рассмотрен в предыдущей главе) и выбор способа, посредством которого модули комбинируются при построении из них рабочей программы. Второе положение является весьма важным, так как оно задает форму написания тестов модуля, типы средств, используемых при тестировании, порядок кодирования и тестирования модулей, стоимость генерации тестов и стоимость отладки (т. е. локализации и исправления ошибок). В этом разделе мы рассмотрим два подхода к комбинированию модулей: пошаговое и монолитное тестирование, а в следующем — два вариан-

та пошагового подхода: тестирование снизу вверх (восходящее) и тестирование сверху вниз (нисходящее).

Возникает вопрос: что лучше — выполнить по отдельности тестирование каждого модуля, а затем, комбинируя их, сформировать рабочую программу или же каж-

дый модуль для тестирования подключать к набору ранее оттестированных модулей? Первый подход обычно называют *монолитным методом* тестирования, или методом «большого удара» при тестировании и сборке программы; второй подход известен как *пошаговый метод* тестирования или сборки.

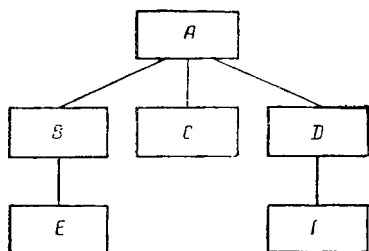


Рис. 5.7. Пример программы, состоящей из шести модулей

В качестве примера рассмотрим программу на рис. 5.7. Прямоугольниками обозначены шесть модулей программы (подпрограммы или процедуры). Линиями показана иерархия управления: модуль А вызывает модули В, С и D, модуль В вызывает модуль Е и т. д. При традиционном монолитном подходе тестирование выполняется следующим образом. Сначала тестируются шесть модулей, входящих в программу, причем тестирование каждого осуществляется независимо от других. При различных условиях (пакетная обработка или интерактивный режим) и числе исполнителей модули могут тестироваться последовательно или параллельно. Затем модули комбинируются или собираются в программу (например, путем редактирования связей).

Для тестирования любого модуля требуются специальный *модуль-драйвер* и один или несколько *модулей-заглушек*<sup>1</sup>. Например, если тестируется модуль В, то первоначально нужно разработать тесты, а затем написать небольшую программу, которая передаст модулю В входные тестовые данные, необходимые для его исполнения (прогона на тесте). (Для этой цели можно исполь-

<sup>1</sup> В практике программирования драйверы называют также отлаживающими модулями, а заглушки — имитаторами. — *Примеч. ред.*

зовать и отладочные средства.) Драйвер должен также отобразить программисту некоторые сведения о результатах работы модуля В. Кроме того, поскольку в модуле В есть вызов модуля Е, рекомендуется сделать модуль-заглушку, которому будет передано управление при исполнении вызова модуля Е. Модулю-заглушке, используемому вместо модуля Е во время тестирования, присваивается тоже имя «Е», и он должен имитировать функции этого модуля. После завершения тестирования всех шести модулей они собираются в единую программу.

Метод пошагового тестирования предполагает, что модули тестируются не изолированно друг от друга, а подключаются поочередно для выполнения теста к набору уже ранее оттестированных модулей.

Число возможных реализаций пошагового метода велико, поэтому рассмотрение конкретной пошаговой процедуры для программы, изображенной на рис. 5.7, пока преждевременно. Ключевым является вопрос о том, как следует начинать тестирование программы — сверху или снизу. Однако оставим этот вопрос до следующего раздела и допустим, что тестирование начинается снизу. Первоначально можно последовательно или параллельно (например, это могут сделать три человека) выполнить тестирование модулей Е, С и F. При этом для каждого модуля придется сделать драйвер; заглушки же здесь не нужны. Следующий шаг — тестирование модулей В и D, но не независимо, а совместно с модулями Е и F соответственно. Другими словами, для тестирования модуля В разрабатывается драйвер, включающий тесты, и выполняется тестирование пары В—Е. Пошаговый процесс продолжается до тех пор, пока к набору оттестированных модулей не будет подключен последний модуль, в данном случае модуль А. Заметим, что эта процедура может быть развита и сверху вниз.

Уже на этом этапе мы можем сделать некоторые обобщения:

1. Монолитное тестирование требует больших затрат труда. Для программы рис. 5.7 необходимо создать пять драйверов и пять заглушек с учетом того, что для верхнего модуля драйвер не нужен. При пошаговом же тестировании снизу вверх потребуется только пять драйверов, а сверху вниз — только пять заглушек. Сокращение затрат труда объясняется тем, что при тестировании сверху вниз тестируемые модули выполняют функции

заглушек, а при тестировании снизу вверх — функции драйверов.

2. При пошаговом тестировании раньше обнаруживаются ошибки в интерфейсах между модулями, поскольку раньше начинается сборка программы. В противоположность этому при монолитном тестировании модули «не видят друг друга» до последней фазы процесса тестирования.

3. Отладка программ при пошаговом тестировании легче. Если есть ошибки в межмодульных интерфейсах, а обычно так и бывает, то при монолитном тестировании они могут быть обнаружены лишь тогда, когда собрана вся программа. В этот момент локализовать ошибку довольно трудно, поскольку она может находиться в любом месте программы. Напротив, при пошаговом тестировании ошибки такого типа в основном связаны с тем модулем, который подключается последним.

4. Результаты пошагового тестирования более совершенны. Например, при тестировании модуля В одновременно с ним исполняется или модуль А, или модуль Е (в зависимости от того, сверху или снизу начинается тестирование). Хотя эти модули ранее уже были тщательно оттестированы, совместное их использование с модулем В создаст новые условия, и, возможно, одно из них совпадет с условием, которое не было учтено при их автономном тестировании.

С другой стороны, при монолитном тестировании модуля результаты ограничены только этим модулем. Иными словами, из-за того что в случае пошагового тестирования модули, оттестированные ранее, затем используются в качестве драйверов или заглушек, они подвергаются дополнительной проверке при исполнении теста отлаживаемого модуля.

5. Расход машинного времени при монолитном тестировании меньше. Если с использованием пошагового подхода снизу вверх тестируется модуль А, то одновременно с ним исполняются модули В, С, D, Е и, возможно, еще модуль F, а в случае монолитного тестирования исполняется только сам модуль А и заглушки вместо модулей В, С и D. То же самое происходит, если используется пошаговый подход сверху вниз: если тестируется модуль F, то одновременно с ним могут исполняться модули А, В, С, D и Е, в то же время как при монолитном тестировании исполняется только сам модуль F и соот-

ветствующий ему драйвер. Следовательно, число исполняемых машинных команд во время прогона теста при пошаговом тестировании явно больше, чем при монолитном. Однако указанное превышение компенсируется тем, что монолитное тестирование требует больше заглушек и драйверов, чем пошаговое, и в результате расходуется машинное время на разработку этих драйверов и заглушек<sup>1</sup>.

6. Использование монолитного метода предоставляет большие возможности для параллельной организации работы на начальной фазе тестирования (тестирования всех модулей одновременно). Это положение может иметь важное значение при выполнении больших проектов, в которых много модулей и много исполнителей, поскольку численность персонала, участвующего в проекте, максимальна на начальной фазе.

В заключение отметим, что п. 1—4 демонстрируют преимущества пошагового тестирования, а п. 5 и 6 — его недостатки. Поскольку для современного этапа развития вычислительной техники характерны тенденции к уменьшению стоимости аппаратуры и увеличению стоимости труда, последствия ошибок в математическом обеспечении весьма серьезны, а стоимость устранения ошибки тем меньше, чем раньше она обнаружена; преимущества, указанные в п. 1—4, выступают на первый план. В то же время ущерб, наносимый недостатками (п. 5 и 6), невелик. Все это позволяет нам сделать вывод, что пошаговое тестирование является предпочтительным.

## НИСХОДЯЩЕЕ И ВОСХОДЯЩЕЕ ТЕСТИРОВАНИЕ

Убедившись в преимуществах пошагового тестирования перед монолитным, исследуем две возможные стратегии тестирования: *нисходящее* и *восходящее*. Прежде

---

<sup>1</sup> Это логически непротиворечивое утверждение в конкретных условиях может оказаться неправильным. Например, если время счета каждого реального отлаживаемого модуля составляет единицы минут или более, а машинное время на разработку драйвера или заглушки — десятки секунд, то наблюдается обратное соотношение. При выборе метода тестирования для конкретной разработки следует учитывать особенности программного обеспечения, наличие ресурсов и требования к степени отлаженности готового продукта; в этих условиях машинное время не всегда является доминирующим фактором даже для большой разработки. — *Примеч. ред.*

всего внесем ясность в терминологию. Во-первых, термины «нисходящее тестирование», «нисходящая разработка», «нисходящее проектирование» часто используются как синонимы. Действительно, термины «нисходящее тестирование» и «нисходящая разработка» являются синонимами (в том смысле, что они подразумевают определенную стратегию при тестировании и создании текстов модулей), но нисходящее проектирование — это совершенно иной и независимый процесс. Программа, спроектированная нисходящим методом, может тестироваться и нисходящим, и восходящим методами.

Во-вторых, восходящая разработка или тестирование часто отождествляется с монолитным тестированием. Это недоразумение возникает из-за того, что начало восходящего тестирования идентично монолитному при тестировании нижних или терминальных модулей. Но в предыдущем разделе показано, что восходящее тестирование на самом деле представляет собой пошаговую стратегию. Теперь, когда очевидно, что обе стратегии — пошаговые, не будем обсуждать преимуществ пошагового подхода; рассмотрим различие между нисходящей и восходящей стратегиями.

### Нисходящее тестирование

Нисходящее тестирование начинается с верхнего, головного модуля программы. Строгой, корректной процедуры подключения очередного последовательно тестируемого модуля не существует. Единственное правило, которым следует руководствоваться при выборе очередного модуля, состоит в том, что им должен быть один из модулей, вызываемых модулем, предварительно прошедшим тестирование.

Для иллюстрации этой стратегии рассмотрим рис. 5.8. Изображенная на нем программа состоит из двенадцати модулей A—L. Допустим, что модуль J содержит операции чтения из внешней памяти, а модуль I — операции записи.

Первый шаг — тестирование модуля A. Для его выполнения необходимо написать модули-заглушки, замещающие модули B, C и D. К сожалению, часто неверно понимают функции, выполняемые модулями-заглушками. Так, порой можно услышать, что «заглушка должна только выполнять запись сообщения, устанавливающего: «модуль подключился» или «достаточно, чтобы заг-

лушка существовала, не выполняя никакой работы вообще». В большинстве случаев эти утверждения ошибочны. Когда модуль А вызывает модуль В, А предполагает, что В выполняет некую работу, т. е. модуль А получает результаты работы модуля В (например, в форме значений выходных переменных). Когда же модуль В

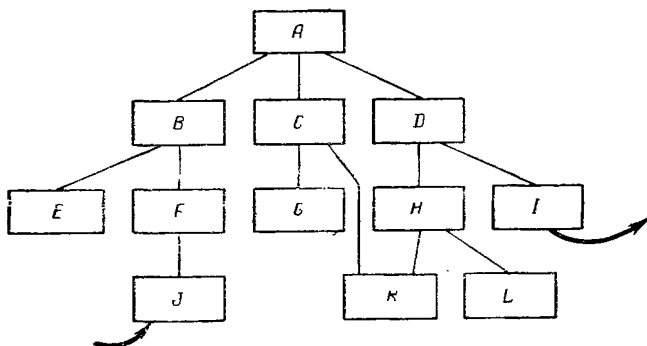


Рис. 5.8. Пример программы, состоящей из двенадцати модулей

просто возвращает управление или выдает сообщение об ошибке без передачи в А определенных осмысленных результатов, модуль А работает неверно не вследствие ошибок в самом модуле, а из-за несоответствия ему модуля-заглушки. Более того, результат может оказаться неудовлетворительным, если ответ модуля-заглушки не меняется в зависимости от условий теста. Например, допустим, что нужно написать заглушку, замещающую программу вычисления квадратного корня, программу поиска в таблице или программу чтения соответствующей записи. Если заглушка всегда возвращает один и тот же фиксированный результат вместо конкретного значения, предполагаемого вызывающим модулем именно в этом вызове, то вызывающий модуль сработает как ошибочный (например, заикнется) или выдаст неверное выходное значение. Следовательно, создание модулей-заглушек — задача нетривиальная.

При обсуждении метода нисходящего тестирования часто упускают еще одно положение, а именно форму представления тестов в программе. В нашем примере вопрос состоит в том, как тесты должны быть переданы

модулю А? Ответ на этот вопрос не является совершенно очевидным, поскольку верхний модуль в типичной программе сам не получает входных данных и не выполняет операций ввода-вывода. Верхнему модулю (в нашем случае модулю А) данные передаются через одну или несколько заглушек. Для иллюстрации допустим, что модули В, С и D выполняют следующие функции:

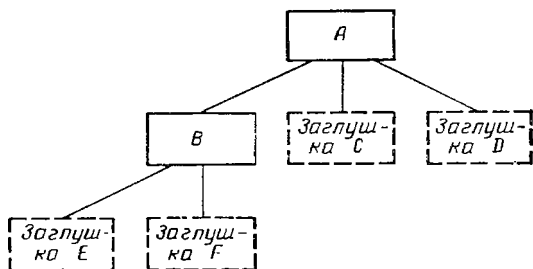


Рис. 5.9. Второй шаг при нисходящем тестировании

В — получает сводку о вспомогательном файле;

С — определяет, соответствует ли недельное положение дел установленному уровню;

D — формирует итоговый отчет за неделю.

В таком случае тестом для А является сводка о вспомогательном файле, получаемая от заглушки В. Заглушка D содержит операторы, выдающие ее входные данные на печатающее устройство или терминал, чтобы сделать возможным анализ результатов прохождения каждого теста.

С этой программой связана еще одна проблема. Поскольку модуль А вызывает модуль В, вероятно, один раз, нужно решить, каким образом передать в А несколько тестов. Одно из решений состоит в том, чтобы вместо В сделать несколько версий заглушки, каждая из которых имеет один фиксированный набор тестовых данных. Тогда для использования любого тестового набора можно несколько раз исполнить программу, причем всякий раз с новой версией модуля-заглушки, замещающего В. Другой вариант решения — записать наборы тестов во внешнюю память, заглушкой читать их и передавать в модуль А. В общем случае создание заглушки может

быть более сложной задачей, чем в разобранный выше примере. Кроме того, часто из-за характеристик программы оказывается необходимым сообщать тестируемому модулю данные от нескольких заглушек, замещающих модули нижнего уровня; например, модуль может получать данные от нескольких вызываемых им модулей.

После завершения тестирования модуля А одна из заглушек замещается реальным модулем и добавляются заглушки, необходимые уже этому модулю. Например, на рис. 5.9 представлена следующая версия программы.

После тестирования верхнего (головного) модуля тестирование выполняется в различных последовательностях. Так, если последовательно тестируются все модули, то возможны следующие варианты:

```

A B C D E F G H I J K L
A B E F J C G K D H L I
A D H I K L C G B F J E
A B F J D I E C G K H L

```

При параллельном выполнении тестирования могут встречаться иные последовательности. Например, после тестирования модуля А одним программистом может тестироваться последовательность А—В, другим — А—С, третьим — А—D. В принципе нет такой последовательности, которой бы отдавалось предпочтение, но рекомендуется придерживаться двух основных правил:

1. Если в программе есть критические в каком-либо смысле части (возможно, модуль G), то целесообразно выбирать последовательность, которая включала бы эти части как можно раньше. Критическими могут быть сложный модуль, модуль с новым алгоритмом или модуль со значительным числом предполагаемых ошибок (модуль, склонный к ошибкам).

2. Модули, включающие операции ввода-вывода, также необходимо подключать в последовательность тестирования как можно раньше.

Целесообразность первого правила очевидна, второе же следует обсудить дополнительно. Напомним, что при проектировании заглушек возникает проблема, заключающаяся в том, что одни из них должны содержать тесты, а другие — организовывать выдачу результатов на печать или на терминал. Если к программе подключается реальный модуль, содержащий операции

ввода, то представление тестов значительно упрощается. Форма их представления становится идентичной той, которая используется в реальной программе для ввода данных (например, из вспомогательного файла или ввод с терминала). Точно так же, если подключаемый модуль содержит выходные функции программы, то отпадает необходимость в заглушках, записывающих ре-

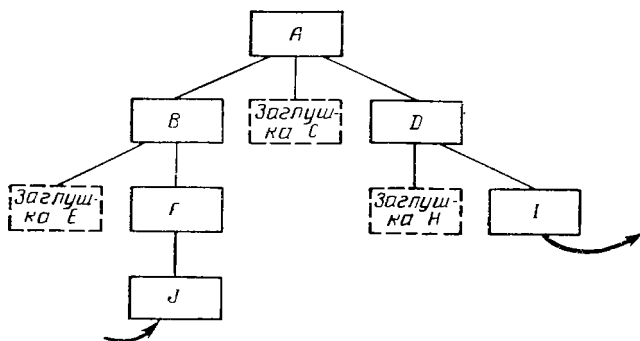


Рис. 5.10. Промежуточное состояние при нисходящем тестировании

зультаты тестирования. Пусть, например, модули J и I выполняют функции входа-выхода, а G — некоторая критическая функция; тогда пошаговая последовательность может быть следующей:

A B F J D I C G E K H L

и после шестого шага становится такой, как показано на рис. 5.10.

По достижении стадии, отражаемой рис. 5.10, представление тестов и анализ результатов тестирования существенно упрощаются. Появляются дополнительные преимущества. В этот момент уже имеется рабочая версия структуры программы, выполняющая реальные операции ввода-вывода, в то время как часть внутренних функций имитируется заглушками. Эта рабочая версия позволяет выявить ошибки и проблемы, связанные с организацией взаимодействия с человеком; она дает возможность продемонстрировать программу пользователю, вносит ясность в то, что производится испытание всего проекта в целом, а для некоторых является и положительным моральным стимулом. Все это, безуслов-

но, достоинства стратегии нисходящего тестирования.

Однако нисходящее тестирование имеет ряд серьезных недостатков. Пусть состояние проверяемой программы соответствует показанному на рис. 5.10. На следующем шаге нужно заменить заглушку самим модулем Н. Для тестирования этого модуля требуется спроектировать (или они спроектированы ранее) тесты по правилам, приведенным ранее в данной главе. Заметим, что все тесты должны быть представлены в виде реальных данных, вводимых через модуль J. При этом создаются две проблемы. Во-первых, между модулями Н и J имеются промежуточные модули (F, B, A и D), поэтому может оказаться *невозможным* передать модулю такой текст, который бы соответствовал каждой предварительно описанной ситуации на входе модуля Н. Например, если Н — это модуль BONUS, показанный на рис. 5.2, то может случиться, что из-за промежуточного модуля D нельзя подать на вход модуля Н все семь тестов, приведенных на рис. 5.5 и 5.6. Во-вторых, даже если есть возможность передать все тесты, то из-за «дистанции» между модулем Н и точкой ввода в программу возникает довольно трудная интеллектуальная задача — оценить, какими должны быть данные на входе модуля J, чтобы они соответствовали требуемым тестам модуля Н.

Третья проблема состоит в том, что результаты выполнения теста демонстрируются модулем, расположенным довольно далеко от модуля, тестируемого в данный момент. Следовательно, установление соответствия между тем, что демонстрируется, и тем, что происходит в модуле на самом деле, достаточно сложно, а иногда просто невозможно. Допустим, мы добавляем к схеме рис. 5.10 модуль Е. Результаты каждого теста определяются путем анализа выходных результатов модуля I, но из-за стоящих между модулями Е и I промежуточных модулей трудно восстановить действительные выходные результаты модуля Е (т. е. те результаты, которые он передает в модуль В).

В нисходящем тестировании в связи с организацией его проведения могут возникнуть еще две проблемы. Некоторые программисты считают, что тестирование может быть совмещено с проектированием программ. Например, если проектируется программа, изображенная на рис. 5.8, то может сложиться впечатление, что после проектирования двух верхних уровней следует перейти к ко-

дированию и тестированию модулей А и В, С и D и к разработке модулей нижнего уровня. Как отмечается в работе [1], такое решение не является разумным. Проектирование программ — процесс итеративный, т. е. при создании модулей, занимающих нижний уровень в структуре программы, может оказаться необходимым произвести изменения в модулях верхнего уровня. Если же модули верхнего уровня уже закодированы и оттестированы, то скорее всего эти изменения внесены не будут, и принятое раньше не лучшее решение получит долгую жизнь.

Последняя проблема заключается в том, что на практике часто переходят к тестированию следующего модуля до завершения тестирования предыдущего. Это объясняется двумя причинами: во-первых, трудно вставлять тестовые данные в модули-заглушки и, во-вторых, модули верхнего уровня используют ресурсы модулей нижнего уровня. Из рис. 5.8 видно, что тестирование модуля А может потребовать нескольких версий заглушки модуля В. Программист, тестирующий программу, как правило, решает так: «Я сразу не буду полностью тестировать модуль А — сейчас это трудная задача. Когда подключу модуль J, станет легче представлять тесты, и уж тогда я вернусь к тестированию модуля А». Конечно, здесь важно только не забыть проверить оставшуюся часть модуля тогда, когда это предполагалось сделать. Аналогичная проблема возникает в связи с тем, что модули верхнего уровня также запрашивают ресурсы для использования их модулями нижнего уровня (например, открывают файлы). Иногда трудно определить, корректно ли эти ресурсы были запрошены (например, верны ли атрибуты открытия файлов) до того момента, пока не начнется тестирование использующих их модулей нижнего уровня.

## Восходящее тестирование

Рассмотрим восходящую стратегию пошагового тестирования. Во многих отношениях восходящее тестирование противоположно нисходящему; преимущества нисходящего тестирования становятся недостатками восходящего тестирования и, наоборот, недостатки нисходящего тестирования становятся преимуществами восходящего. Имея это в виду, обсудим кратко стратегию восходящего тестирования.

Данная стратегия предполагает начало тестирования с терминальных модулей (т. е. модулей, не вызывающих другие модули). Как и ранее, здесь нет такой процедуры для выбора модуля, тестируемого на следующем шаге, которой бы отдавалось предпочтение. Единственное правило состоит в том, чтобы очередной модуль вызывал уже оттестированные модули.

Если вернуться к рис. 5.8, то первым шагом должно быть тестирование нескольких или всех модулей Е, J, G, К, L и I последовательно или параллельно. Для каждого из них требуется

свой модуль-драйвер, т. е. модуль, который содержит фиксированные тестовые данные, вызывает тестируемый модуль и отображает выходные результаты (или сравнивает реальные выходные результаты с ожидаемыми). В отличие от заглушек, драйвер не должен иметь несколько версий, поэтому

он может последовательно вызывать тестируемый модуль несколько раз. В большинстве случаев драйверы проще разработать, чем заглушки.

Как и в предыдущем случае, на последовательность тестирования влияет критичность природы модуля. Если мы решаем, что наиболее критичны модули D и F, то промежуточное состояние будет соответствовать рис. 5.11. Следующими шагами могут быть тестирование модуля Е, затем модуля В и комбинирование В с предварительно оттестированными модулями Е, F, J.

Недостаток рассматриваемой стратегии заключается в том, что концепция построения структуры рабочей программы на ранней стадии тестирования отсутствует. Действительно, рабочая программа не существует до тех пор, пока не добавлен последний модуль (в примере модуль А), и это уже готовая программа. Хотя функции ввода-вывода могут быть проверены прежде, чем собрана вся программа (использовавшиеся модули ввода-вывода

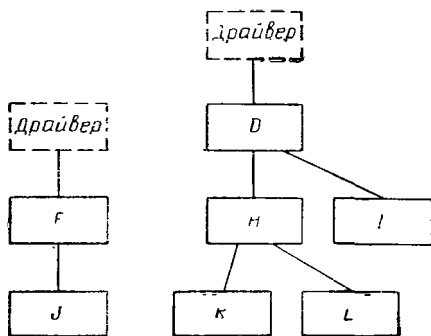


Рис. 5.11. Промежуточное состояние при восходящем тестировании

показаны на рис. 5.11), преимущества раннего формирования структуры программы снижаются.

Здесь отсутствуют проблемы, связанные с невозможностью или трудностью создания всех тестовых ситуаций, характерные для нисходящего тестирования. Драйвер как средство тестирования применяется непосредственно к тому модулю, который тестируется, нет промежуточных модулей, которые следует принимать во внимание. Анализируя другие проблемы, возникающие при нисходящем тестировании, можно заметить, что при восходящем тестировании невозможно принять неразумное решение о совмещении тестирования с проектированием программы, поскольку нельзя начать тестирование до тех пор, пока не спроектированы модули нижнего уровня. Не существует также и трудностей с незавершенностью тестирования одного модуля при переходе к тестированию другого, потому что при восходящем тестировании с применением нескольких версий заглушки нет сложностей с представлением тестовых данных.

### С р а в н е н и е

К сожалению, при сравнении стратегий нисходящего и восходящего тестирования нельзя противопоставить их друг другу, как это делалось при сравнении пошагового и монолитного подходов. В табл. 5.3 показаны их относительные недостатки и преимущества (за исключением их общих преимуществ как методов пошагового тестирования). Первое преимущество каждого из методов могло бы явиться решающим фактором, однако трудно сказать, где больше недостатков: в модулях верхнего уровня или модулях нижних уровней типичной программы. Поэтому при выборе стратегии целесообразно взвесить все пункты из табл. 5.3 с учетом характеристик конкретной программы. Для программы, рассматриваемой в качестве примера, большое значение имеет четвертый из недостатков нисходящего тестирования. Учитывая этот недостаток, а также то, что отладочные средства сокращают потребность в драйверах, но не в заглушках, предпочтение следует отдать стратегии восходящего тестирования.

В заключение отметим, что рассмотренные стратегии нисходящего и восходящего тестирования не являются единственными возможными при пошаговом подходе. В работе [2] рассматриваются еще три варианта стратегии тестирования.

## Сравнение нисходящего и восходящего тестирования

Преимущества	Недостатки
<i>Нисходящее тестирование</i>	
<ol style="list-style-type: none"> <li>1. Имеет преимущества, если ошибки главным образом в верхней части программы.</li> <li>2. Представление теста облегчается после подключения функций ввода-вывода.</li> <li>3. Раннее формирование структуры программы позволяет провести ее демонстрацию пользователю и служит моральным стимулом.</li> </ol>	<ol style="list-style-type: none"> <li>1. Необходимо разрабатывать модули-заглушки.</li> <li>2. Модули-заглушки часто оказываются сложнее, чем кажется вначале.</li> <li>3. До применения функций ввода-вывода может быть сложно представлять тестовые данные в заглушки.</li> <li>4. Может оказаться трудным или невозможным создать тестовые условия.</li> <li>5. Сложнее оценка результатов тестирования.</li> <li>6. Допускается возможность формирования представления о совмещении тестирования и проектирования.</li> <li>7. Стимулируется незавершение тестирования некоторых модулей.</li> </ol>
<i>Восходящее тестирование</i>	
<ol style="list-style-type: none"> <li>1. Имеет преимущества, если ошибки главным образом в модуле нижнего уровня.</li> <li>2. Легче создавать тестовые условия</li> <li>3. Проще оценка результатов.</li> </ol>	<ol style="list-style-type: none"> <li>1. Необходимо разрабатывать модули-драйверы.</li> <li>2. Программа как единое целое не существует до тех пор, пока не добавлен последний модуль.</li> </ol>

## ИСПОЛНЕНИЕ ТЕСТА

Исполнение теста — завершающий этап тестирования модулей. Ниже описан определенный набор рекомендаций и положений, которых следует придерживаться на этом этапе.

Если исполнение теста приносит результаты, не соответствующие предполагаемым, то это означает, что либо модуль имеет ошибку, либо неверны предполагаемые результаты (ошибка в тесте). Для устранения такого рода недоразумений нужно тщательно проверять набор тестов («тестировать» тесты).

Применение автоматизированных средств позволяет снизить трудоемкость процесса тестирования. Например, существуют средства, которые позволяют избавиться от потребности в драйверах. Средства анализа потоков дают возможность пронумеровать маршруты в программе, определить неисполняемые операторы, обнаружить места, где переменные используются до присвоения им значения.

При подготовке к тестированию модулей целесообразно еще раз пересмотреть психологические и экономические принципы, обсуждавшиеся в гл. 2. Вспомните, что, как было показано ранее, необходимой частью тестового набора является описание ожидаемых результатов. При исполнении теста следует обращать внимание на побочные эффекты, например, если модуль делает то, чего он делать не должен. В общем случае такую ситуацию обнаружить трудно, но иногда побочные эффекты можно выявить, если проверить не только предполагаемые выходные переменные, но и другие, состояние которых в процессе тестирования измениться не должно. Например, тест 7 рис. 5.6 не должен изменить значений переменных `ESIZE`, `DSIZE` и `DEPTTAB`. Поэтому при его исполнении наряду с предполагаемыми результатами необходимо проверить и эти переменные.

Во время тестирования модулей возникают и психологические проблемы, связанные с личностью тестирующего. Программистам полезно поменяться модулями, чтобы не тестировать свои собственные. Так, программист, сделавший вызываемый модуль, является хорошим кандидатом для тестирования вызываемого модуля. Заметим, что это относится только к тестированию, а не к отладке, которую всегда должен выполнять автор модуля. Не сле-

дует выбрасывать результаты тестов; представляйте их в такой форме, чтобы можно было повторно воспользоваться ими в будущем. Учитывайте возможность ошибочного интуитивного представления (см. рис. 2.2). Если в некотором подмножестве модулей обнаружено большое число ошибок, то эти модули, по-видимому, содержат еще большее число необнаруженных ошибок. Такие модули должны стать объектом дальнейшего тестирования; желательно даже дополнительно произвести контроль или просмотр их текста. Наконец, следует помнить, что задача тестирования заключается не в демонстрации корректной работы модулей, а в выявлении ошибок.

#### ЛИТЕРАТУРА

1. Myers G. J. Composite/Structured Design. New York, Van Nostrand Reinhold, 1978.
2. Myers G. J. Software Reliability: Principles and Practices. New York, Wiley-Interscience, 1976. Русский перевод: Майерс Г. Надежность программного обеспечения. М., Мир, 1980.

### ТЕСТИРОВАНИЕ КОМПЛЕКСОВ ПРОГРАММ

Окончание тестирования модуля вовсе не означает, что тестирование программы завершено. На самом деле процесс тестирования еще только начинается, особенно если программа большая или представляет собой программный продукт. Эта мысль находит свое подтверждение в следующем определении ошибки программного обеспечения [1]:

*в программе имеется ошибка, если ее выполнение не оправдывает ожиданий пользователя.*

Ясно, что проведение даже абсолютно полного тестирования модуля никоим образом не гарантировало бы обнаружение всех ошибок программного обеспечения (с учетом приведенного выше определения). В этом и заключается одна из причин необходимости разработки какой-либо формы дальнейшего тестирования.

Другая причина связана с предварительным описанием возникновения ошибок программного обеспечения. Дело в том, что разработка программного обеспечения в значительной степени есть процесс передачи информации о конечной программе и перевода этой информации из одной формы в другую [1]. Кроме того, подавляющее большинство ошибок программного обеспечения определяется дефектами организации работ, недостаточным взаимопониманием и искажениями в процессе передачи и перевода информации. Эта точка зрения на разработку программного обеспечения иллюстрируется рис. 6.1, на котором представлена модель цикла разработки программного продукта<sup>1</sup>. Процесс разработки можно разбить на семь шагов:

---

<sup>1</sup> В советской государственной системе стандартизации установлена несколько другая совокупность стадий и этапов разработки. Поэтому и терминологию, и содержание этапов разработки, приводимые в этой книге, не следует рассматривать как общепринятые

1. Потребности будущего пользователя разрабатываемой программы воплощаются в документ, описывающий набор требований к разрабатываемому продукту.

2. Путем оценки осуществимости и стоимости, разрешения противоречивых требований, установления очередности разработки и поставки эти требования переводятся в точно сформулированные цели.

3. На основании указанных целей создается подробная и точная спецификация конечного программного продукта, причем сам продукт рассматривается как черный ящик и учитываются только его интерфейсы и взаимодействия с внешним миром (например, с конечным пользователем). Эта спецификация называется внешней.

4. Осуществляется проектирование системы, если программный продукт представляет собой системы<sup>1</sup> (например, операционную систему, систему управления воздушным движением, систему управления базой данных, систему учета кадров), а не *программу* (например, компилятор, программу расчета зарплаты, программу форматирования текста). На этом шаге система разбивается на отдельные программы, компоненты или подсистемы, и определяются их интерфейсы.

5. Проектируется структура программы путем специфицирования (установления) функции каждого модуля, иерархической структуры всех модулей и всех интерфейсов между модулями.

6. Разрабатывается подробная и точная спецификация путем определения интерфейса и функции каждого модуля.

7. За один или несколько подшагов спецификация интерфейса модуля переводится в алгоритм каждого модуля, записанный на исходном языке.

Иными словами, требования обосновывают, *почему*

---

и применять безоговорочно. Вместе с тем опыт и практика разработки программного обеспечения в США, отраженные здесь, представляют несомненный интерес. В настоящее время проводится большая работа по приведению действующих стандартов в соответствие с современной практикой создания программных изделий как по содержанию, так и по терминологии. — *Примеч. ред.*

<sup>1</sup> Для всех приведенных примеров, кроме первого, правильнее было бы сказать «является составной частью системы», так как в соответствующие системы входят также аппаратура и персонал. В отечественной практике в таких случаях применяется термин «изделие» или «программное изделие», в отличие от термина «программа». — *Примеч. ред.*

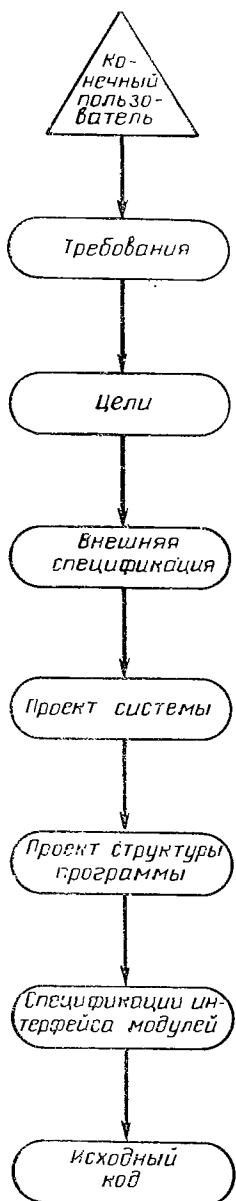


Рис. 6.1. Процесс разработки программного обеспечения

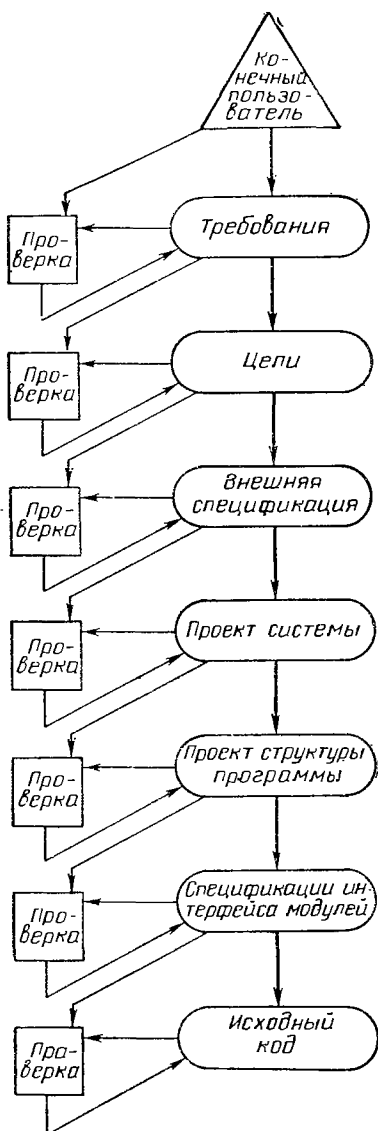


Рис. 6.2. Процесс разработки с промежуточными проверками

нужна данная программа, цели определяют, *что* должна делать данная программа и *как хорошо* она должна это делать, внешние спецификации дают точное *представление* о программе ее пользователей, а документация, связанная с описанным выше последовательным процессом, показывает на всех уровнях детализации, *как* построена эта программа<sup>1</sup>.

Приведенное предварительное описание процесса разработки включает в себя передачу, осмысление и перевод информации от шага к шагу, а также предположение о том, что наибольшее число ошибок программного обеспечения происходит из-за пошагового преобразования информации. Существуют три дополнительных метода предупреждения и (или) выявления этих ошибок. Первый из них основан на повышении четкости самого процесса разработки, что позволит избежать появления многих ошибок. Второй метод заключается во введении в конце каждого процесса (шага) отдельного шага проверки, имеющего целью локализовать наибольшее число ошибок до перехода к следующему процессу (шагу). Этот метод иллюстрируется рис. 6.2. Например, внешняя спецификация проверяется путем сравнения ее с выходными результатами предыдущей стадии (отчетом, содержащим описание целей), а каждая обнаруженная ошибка возвращается для устранения в процесс (на шаг) разработки внешних спецификаций. Способы проверки исходного теста и его сквозного просмотра, обсуждавшиеся в гл. 3, используются на шаге проверки в конце седьмого процесса.

Третий метод ориентирует конкретные процессы тест-

---

<sup>1</sup> В отечественной практике требования к программному обеспечению (ПО) формулируют в техническом задании на изделие, в которое ПО входит как составная часть; цели указывают в техническом задании на программное обеспечение. Внешняя спецификация требований как отдельный документ не составляется, но в состав эскизного и технического проекта предусматриваются соответствующие документы или разделы. Сведения, которые автор предлагает включать в проект системы, представляют основное содержание технического проекта. Проект структуры программы и спецификации интерфейса модулей являются промежуточной документацией рабочего проекта.

Вместе с тем автор справедливо указывает на *настоятельную* необходимость фиксировать все эти сведения в виде согласовываемых разработчиками и утверждаемых Главным конструктором (а для технического задания, эскизного и технического проектов — и заказчиком) *документов*. — *Примеч. ред.*

тирования на конкретные процессы разработки, т. е. сосредоточивает каждый процесс тестирования на каком-либо шаге перевода, в результате чего фиксируется опре-

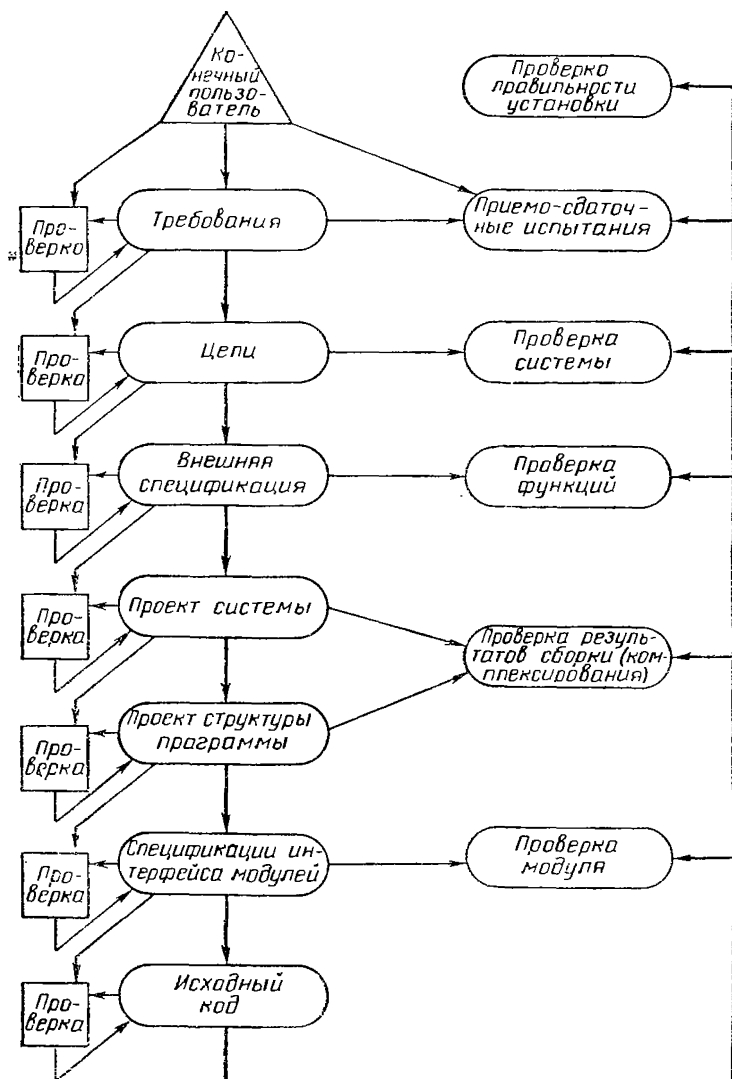


Рис. 6.3. Соотношение между процессами разработки и тестирования

деленный класс ошибок. Этот метод иллюстрируется рис. 6.3. Цикл тестирования здесь структурирован в модель приведенного на рисунке цикла разработки. Таким образом, необходимо уметь установить однозначное соответствие между процессами разработки и тестирования. Например, назначение проверки модулей — найти противоречия между модулями программы и спецификациями их интерфейса, назначение тестирования функций — показать несоответствие программы своим внешним спецификациям, назначение тестирования системы — продемонстрировать противоречивость программного продукта сформулированным для него исходным целям. Преимущества описанной структуры состоят в том, что она позволяет избежать непродуктивного излишнего тестирования и исключает возможность появления целого класса ошибок. Например, при этом методе тестирование системы ориентировано на определенный класс ошибок (сделанных при переводе целей во внешнюю спецификацию), а его характеристики измеряются по отношению к конкретному типу документации, создававшейся в процессе разработки. Такой подход существенно отличается от бытующего излишне упрощенного «тестирования всей системы», при котором иногда ограничиваются просто повторением тестов, созданных на более ранних стадиях.

Как уже отмечалось, формы тестирования комплексов программ (рис. 6.3) наиболее применимы к *программным продуктам* (программам, написанным в соответствии с контрактом, или программам, предназначенным для широкого использования; в противоположность экспериментальным программам или программам, эксплуатируемым их автором). Программы, не являющиеся продуктом, обычно не имеют формальных требований и целей; для них проверка функций часто оказывается единственной проверкой более высокого уровня. К тому же необходимость в тестировании более высокого уровня возрастает по мере увеличения размера программы. Причина заключается, видимо, в том, что отношение числа ошибок проектирования (ошибок, сделанных на ранних стадиях процесса разработки) к числу ошибок кодирования значительно выше в больших программах, чем в программах малого размера.

Заметим, что последовательность выполнения процессов тестирования (проверки), показанная на рис. 6.3, не обязательно совпадает с их последовательностью по вре-

мени. Например, поскольку проверка системы определяется не как «вид тестирования, который следует после проверки функций», а как особый тип тестирования, направленный на выявление конкретного класса ошибок, она вполне может частично перекрываться во времени с другими процессами тестирования.

В настоящей главе рассматриваются процессы тестирования функций и системы, прямо-сдаточные испытания, а также проверка правильности установки систем программного обеспечения. Проверка результатов сборки (комплексирования), как правило, не считается отдельным шагом и при использовании пошагового тестирования модулей неявно входит в проверку модуля. Все вопросы обсуждаются в общем виде, кратко и (большей частью) без примеров. Это связано с тем, что специальные методы, применяемые при проверках комплексов программ, существенно зависят от специфики программы, подлежащей проверке. Так, характеристики теста системы (конкретные типы тестовых примеров, способ их проектирования, используемые средства проверки) для операционной системы заметно отличаются от теста системы для компилятора, программы управления атомным реактором или прикладной программы базы данных. Последние разделы данной главы посвящены планированию и организации тестирования. Здесь же решается и важный вопрос об определении момента его окончания.

## ТЕСТИРОВАНИЕ ФУНКЦИЙ

Как показано на рис. 6.3, тестирование функций заключается в поиске различий между программой и ее внешней спецификацией. Внешняя спецификация представляет собой точное описание поведения программы с точки зрения «внешнего мира» (например, пользователя).

Тестирование функций обычно выполняется по методу черного ящика, за исключением случаев проверки небольших программ. При этом предполагается, что на более раннем этапе тестирования модулей требуемый критерий покрытия логики, свойственный методу белого ящика, удовлетворяется.

Для того чтобы создать тест функции, спецификацию анализируют с целью получения набора тестов. Наиболее подходящими для тестирования функций являются описанные в гл. 4 методы эквивалентного разбиения, анализа

граничных значений, функциональных диаграмм, предположений об ошибках. Примеры, приведенные в гл. 4, могут служить также примерами тестов функций. Описания фортрановского оператора DIMENSION, программы сортировки данных об экзаменах и команды DISPLAY представляют собой реальные примеры внешних спецификаций. (Заметим, однако, что реальны они все-таки не в полной мере; так, программа сортировки должна еще содержать точное описание форматов сообщений.) Поэтому здесь не приводятся примеры тестов функций.

К тестированию функций применимы многие из правил, сформулированных в гл. 2. Проследите за тем, какие функции выявили наибольшее число ошибок. Эта информация полезна, так как свидетельствует о том, что эти функции, вероятно, содержат большое число невыявленных ошибок. Следует уделять значительное внимание неправильным и непредвиденным входным условиям. Напомним, что определение ожидаемого результата — это важнейшая часть теста. При выполнении теста функции не забывайте о том, что его целью является выявление ошибок, а не демонстрация соответствия программы своей внешней спецификации.

## ТЕСТИРОВАНИЕ СИСТЕМЫ

Тестирование системы — процесс, наиболее трудный для восприятия и исполнения. Его не следует отождествлять с тестированием полной системы или программы. Как показано на рис. 6.3, назначение тестирования системы — сопоставить результат с исходными целями. Отсюда следуют два вывода:

1. Тестирование системы не ограничивается только «системами». Если продукт является программой, то тестирование системы представляет собой попытку продемонстрировать, в какой мере эта программа соответствует поставленным перед ней целям.

2. Тестирование системы по определению невозможно, если при проектировании не был составлен документ, отражающий цели, поставленные перед продуктом (системой).

При рассмотрении различий между полученным результатом и исходными целями программы наибольшее внимание уделяется выявлению ошибок перевода, возникающих в процессе разработки внешней спецификации.

Это делает проверку системы жизненно важной, так как именно на данном этапе возникает больше всего ошибок, причем довольно серьезных. В отличие от теста функции, внешнюю спецификацию нельзя использовать как базис для получения тестов системы, поскольку это противоречило бы цели теста системы. С другой стороны, документ, отражающий цели системы как таковой, нельзя использовать для формулирования ее тестов: он по определению не содержит точных описаний внешних интерфейсов про-

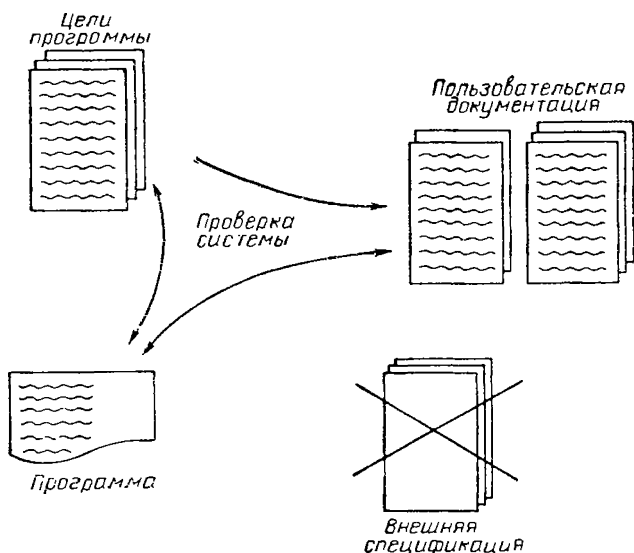


Рис. 6.4. Проверка системы

граммы. Дилемма разрешается применением эксплуатационной пользовательской документации и других изданий (публикаций). Тесты системы проектируются на основе анализа ее целей и затем формулируются по результатам изучения пользовательской документации. Такая практика весьма полезна потому, что позволяет сравнить не только программы с исходным документом, описывающим цели ее создания, но и результаты ее функционирования с пользовательской документацией, а также пользовательскую документацию с исходным документом, как показано на рис. 6.4.

Тестирование системы является наиболее трудной частью процесса тестирования. Его основная задача — сравнить программу с исходным документом, описывающим цели ее создания (самая левая стрелка на рис. 6.4). Однако общепризнанных методологий проектирования таких тестов не существует. Исходный документ устанавливает, что именно и насколько хорошо должна делать программа, но не определяет способа представления ее функций. Например, цели команды `DISPLAY`, специфицированной в гл. 4, следовало бы описывать таким образом:

Команда при выдаче ее с терминала должна обеспечивать отображение содержания ячеек основной памяти. Ее синтаксис должен быть совместим с синтаксисом всех других команд системы. Пользователю нужно предоставить возможность указывать область ячеек путем задания либо начального и конечного адресов, либо начального адреса и числа ячеек. Для операндов команды необходимо обеспечить разумное умолчание.

Результаты исполнения команды требуется отображать на нескольких строках в виде нескольких слов (шестнадцатеричных) с промежутками между ними. Каждая строка должна содержать адрес первого отображаемого слова. Эта команда — «тривиальная», означающая, что при ее выдаче разумно загруженная система начнет отображение выходных результатов через две секунды без заметной задержки между отображением выходных строк. Программная ошибка в интерпретаторе команды в худшем случае должна вызывать ее неудачное выполнение, но не приводить к срыву системного и пользовательского сеансов. В интерпретаторе команды после сдачи системы в производство допускается не более одной ошибки.

Итак, хотя цели и сформулированы, но тождественной методологии, определенной, полезной, устанавливающей правила написания тестов для демонстрации того, что данная программа противоречит каждому предложению в исходном документе, нет. Поэтому здесь приводятся различные подходы к проектированию тестов и обсуждаются отдельные категории тестов системы, а не описывается методология. В связи с ее отсутствием в тестировании системы следует внести элемент творчества. Как известно, проектирование хорошего набора тестов требует большего умения и опыта, чем разработка той же системы или программы.

Ниже рассматриваются 15 категорий тестов. Конечно, мы не утверждаем, что все они применимы к любой программе, но при проектировании тестов их целесообразно исследовать.

## Тестирование удобства использования

Наиболее очевидным видом тестирования системы является проверка выполнения каждого пункта исходного документа (или функции, но слово «функция» здесь не используется, чтобы не спутать тестирование системы с тестированием функций). Процедура проверки состоит в последовательном просмотре исходного документа (предложение за предложением). Если предложение указывает конкретную задачу (например, «синтаксис должен быть совместим...», «пользователь должен быть в состоянии указать область адресов...»), то определяют, выполняет ли программа эту задачу. Таковую проверку часто можно осуществить без вычислительной машины, мысленно сравнив цели с содержанием пользовательской документации.

## Тестирование на предельных объемах

Второй вид тестирования системы — выполнение программы на больших объемах данных. Например, на вход компилятора подают нелепо большую исходную программу, на вход редактора связей — программу, содержащую тысячи модулей, а на вход программы моделирования электронных схем — схему, содержащую тысячи компонент. Очередь заданий операционной системы заполняют до предельной емкости. Если предположить, что программа должна управлять файлами, расположенными на нескольких томах (допустим, на нескольких бобинах ленты), то для проверки будет достаточно такого объема данных, который заставит ее переключаться с одного тома на другой. Таким образом, тестирование на предельных объемах должно показать, что программа не может управлять объемом данных, специфицированным в исходных целях.

Тестирование на предельных объемах, конечно, требует больших затрат машинного времени. Однако им не следует пренебрегать. Наоборот, каждую программу необходимо подвергнуть проверке по меньшей мере на нескольких тестах объема.

## Тестирование на предельных нагрузках

Тестирование на предельных нагрузках представляет собой проверку выполнения программы с применением тяжелых нагрузок, или стрессов. Его не следует путать

с тестированием на предельных объемах: тяжелый стресс означает поступление пикового объема данных *в течение короткого интервала времени*. Здесь можно провести аналогию с оценкой работы машинистки. При тестировании на предельных объемах следовало бы определить, сможет ли машинистка справиться с черновиком большого отчета, а при тестировании на предельных нагрузках — сможет ли машинистка печатать со скоростью 50 слов в минуту.

Поскольку тестирование на предельных нагрузках включает в себя фактор времени, оно неприменимо к многим программам, например к компилятору или к групповой обработке в программе расчета платежной ведомости. Однако его можно использовать для программ, которые функционируют при переменных нагрузках, программ интерактивных или работающих в реальном времени и программ управления процессами. Так, если предполагается, что система управления воздушным движением должна отслеживать до 200 трасс самолетов в своем секторе, то ее следует тестировать на предельной нагрузке путем моделирования наличия 200 самолетов<sup>1</sup>. Поскольку нет никаких оснований для появления в секторе 201-го самолета, то дальнейшее увеличение нагрузки приведет к реакции системы на самолет, который никогда не появится. Дополнительное тестирование на предельных нагрузках может представлять собой тест, моделирующий *одновременное* вхождение в сектор большого числа самолетов.

Допустим, операционная система поддерживает выполнение максимум до 15 заданий в мультипрограммном режиме, тогда ее нагружают путем запуска 15 заданий одновременно. Если система деления времени обеспечивает работу до 64 терминалов, нагрузите ее предельной нагрузкой: подключите к ней 64 пользователя, пытающихся осуществить одновременное обращение. (Это не «ситуация, которой никогда не будет»; она имеет место на практике, когда система выходит из строя и оператор немедленно восстанавливает ее работоспособность). Для определения реакции системы — тренажера имитируйте на ней действия пилота, проходящего тренировку, такие, как поворот руля влево, включение тор-

---

<sup>1</sup> Здесь автор несколько непоследователен: в контексте двух последних разделов это скорее тестирование на предельных объемах. — *Примеч. ред.*

мозов, опускание закрылков, подъем передней части самолета, выпуск шасси, включение посадочных огней, крен влево, — и все это в одно и то же время. (Такой тест может потребовать присутствия в кабине «четырехрукого» пилота или двух специалистов по тестированию). Систему управления процессами можно проверять на предельных нагрузках, если имитировать одновременную выдачу сигналов от всех управляемых процессов. Систему коммутации телефонной связи испытывают тестами предельной нагрузки, вводя в нее одновременно большое число вызовов.

Большинство тестов предельных нагрузок представляет собой условия, подобные тем, которые действительно имеют место на входе программы при ее эксплуатации. Однако некоторые тесты могут моделировать «ситуации, которых никогда не будет». Тем не менее они небесполезны. Если с их помощью выявляются ошибки, значит, они также нужны, поскольку вполне вероятно, что аналогичные ошибки могут проявляться и в реальных непредельных ситуациях.

## Тестирование удобства эксплуатации

Не менее важным видом тестирования системы является попытка выявления психологических (пользовательских) проблем, или проблем удобства эксплуатации. Анализ психологических факторов является, к сожалению, до сих пор весьма субъективным, так как при производстве вычислительной техники уделялось недостаточное внимание изучению и определению психологических аспектов программных систем. Ниже приводится тест, позволяющий проверить некоторые свойства системы.

1. Можно ли приспособить разработанный интерфейс для информирования и обучения конечного пользователя, а также для обеспечения его работы в реальных условиях?

2. Значимы ли, не бранны, вняты ли и т. д. выходные сообщения программы?

3. Понятна ли диагностика ошибок (например, сообщения об ошибках) или пользователь для ее уяснения должен обладать квалификацией доктора наук в области вычислительной техники? Например, выдаст ли программа сообщения вида «IEKØ22A ОШИБКА ПРИ ОТ-

## КРЫТИИ ФАЙЛА 'SYSIN', КОД АВАРИЙНОГО ЗАВЕРШЕНИЯ = 1Ø2»?

4. Проявляет ли все множество пользовательских интерфейсов важное свойство «концептуальной целостности» [2], постоянство и единообразие синтаксиса, соглашений, семантик, формата, стиля и сокращений?

5. Есть ли соответствующая избыточность в представлении входных данных (например, порядковый номер и имя клиента) в тех системах, где точность существенна (например, в диалоговых банковских системах)?

6. Содержит ли система опции, число которых чрезмерно или использование которых маловероятно?

7. Выдает ли система какие-либо подтверждения на все входные сообщения?

8. Легко ли использовать программу? Например, требуется ли при вводе команды в систему разделения времени набирать символы переключения регистров перед символами, расположенными на верхнем и нижнем регистрах?

## Тестирование защиты

В обществе возрастает стремление к сохранению секретности, поэтому перед многими программами ставится цель — обеспечить защиту информации от несанкционированного доступа. Для тестирования защиты важно построить такие тесты, которые нарушат программные средства защиты, например механизм защиты памяти операционной системы или механизмы защиты данных системы управления базой данных. Одним из путей разработки подобных тестов является изучение известных проблем защиты в этих системах и генерация тестов, которые позволяют проверить, как решаются аналогичные проблемы в тестируемой системе. Например, существуют описания [3, 4] некоторых способов обеспечения защиты в операционных системах.

## Тестирование производительности

При разработке многих программ ставится задача — обеспечить их производительность, или эффективность. Определяются такие характеристики, как время отклика и уровень пропускной способности при определенных нагрузке и конфигурации оборудования. Проверка системы в этих случаях сводится к демонстрации того, что данная программа не удовлетворяет поставленным це-

лям. Поэтому необходимо разработать тесты, с помощью которых можно попытаться показать, что она не обладает требуемой производительностью.

### Тестирование требований к памяти

При проектировании некоторых систем определяются предельные объемы основной и вторичной памяти, используемой данной программой, и размеры временных или буферных файлов. Необходимо разработать тесты, с помощью которых можно показать, что поставленные цели не достигаются.

### Тестирование конфигураций оборудования

Операционные системы, системы управления базами данных и системы коммутации сообщений поддерживают множество конфигураций оборудования (например, типы и число устройств ввода-вывода и линий связи, различные объемы памяти). Часто число возможных конфигураций слишком велико, чтобы попытаться проверить программу с каждой из них. Однако программу следует тестировать по меньшей мере с каждым типом оборудования при минимальной и максимальной конфигурациях. Если можно изменять конфигурацию самой программы (например, не включать в нее некоторые компоненты или размещать часть из них на отдельном процессоре), то необходимо тестировать все возможные конфигурации этой программы.

### Тестирование совместимости (конверсии)

Большинство разрабатываемых программ не являются полностью новыми; часто они заменяют несовершенные, устаревшие системы обработки данных или неавтоматизированные процессы. Поэтому при разработке программ необходимо обеспечить совместимость с существующей системой и создать процедуры конверсии<sup>1</sup>. В этом случае, как и при других формах тестирования, тесты должны быть ориентированы на проверку обеспечения совместимости и работы процедур конверсии.

---

<sup>1</sup> Под конверсией понимают процесс перехода от одного метода обработки данных к другому и от одной системы обработки данных к другой. — *Примеч. ред.*

## Тестирование удобства установки

Процедуры установки (настройки) некоторых типов систем программного обеспечения весьма сложны (например, процесс генерации системы, или «sysgen» в операционных системах фирмы IBM). Тестирование подобных процедур является частью процесса тестирования системы.

## Тестирование надежности

Конечно, назначение всех видов тестирования — повысить надежность тестируемой программы, но если в исходном документе, отражающем цели программы, есть особые указания, касающиеся надежности, можно разработать специальные тесты для ее проверки. Тестирование надежности по формулировке целей может оказаться затруднительным. Например, в системе коммутации сообщений TSPS фирмы Bell System время простоя должно составлять не более 2 часов за 40 лет функционирования. Проверить выполнение этого свойства при времени испытаний в несколько месяцев или даже лет не представляется возможным. Однако если к программе предъявлено требование обеспечить определенное время наработки на отказ (например, среднее время наработки на отказ 20 часов) или задано допустимое число ошибок (например, программа должна содержать не более 12 ошибок после того, как она передана в производство), то существует ряд математических моделей (см. [1], гл. 18), позволяющих оценить обоснованность подобных требований.

## Тестирование восстановления

Для операционной системы, системы управления базой данных и средств телеработки часто определяется, как система должна восстанавливаться после программных ошибок, неисправностей аппаратуры и ошибок в данных. При тестировании системы мы должны показать, что эти функции восстановления не выполняются. Можно намеренно ввести в операционную систему программные ошибки, чтобы проверить, восстановится ли она после их устранения. Неисправности аппаратуры, например ошибки устройств ввода-вывода и контроля на четность ячеек памяти, можно промоделировать. Ошибки в данных (помехи в линиях связи и неправиль-

ные значения указателей в базе данных) можно намеренно создать или промоделировать для анализа реакции на них системы.

## Тестирование удобства обслуживания

В исходном документе иногда формулируются специальные цели удобства обслуживания или сопровождения программы. Они могут определять средства обслуживания, которыми должна снабжаться система (например, программы выдачи дампов памяти, программы диагностики), среднее время на отладку простой задачи, процедуры сопровождения и качество документации о внутренней логике программы. Все эти цели должны быть протестированы.

## Тестирование документации

Как показано на рис. 6.4, в проверку системы входит и проверка точности пользовательской документации. Ее проверяют при определении правильности представления предшествующих тестов системы (например, при проектировании теста предельной нагрузки с помощью пользовательской документации его записывают в реальной, закодированной форме). Пользовательская документация должна быть и предметом инспекции при проверке ее на точность и ясность (аналогично инспекции исходного текста, рассмотренной в гл. 3). Любые примеры, приведенные в документации, следует оформить как тест и подать на вход программы.

## Тестирование процедур

Многие программы входят составной частью в большие, не полностью автоматизированные системы, включающие в себя процедуры, выполняемые человеком. Все эти процедуры (процедуры оператора системы, администратора базы данных, пользователя за терминалом и др.) должны быть проверены при тестировании системы.

## Выполнение проверки системы

При проверке системы важно определить, кто именно должен ее выполнять. Выясним этот вопрос спосо-

бом «от противного»: 1) ее не должны выполнять программисты; 2) ее не должна выполнять организация, ответственная за разработку программы.

Первое положение основано на том, что тестирующий должен мыслить теми же категориями, что и конечный пользователь программы, т. е. понимать все взаимоотношения и особенности окружающей обстановки и хорошо представлять себе, как будет использоваться программа. Очевидно поэтому наиболее подходящей кандидатурой для этой цели является конечный пользователь (один или несколько). Однако, поскольку типичный конечный пользователь обычно не обладает умением выполнять тестирование или анализировать результаты применения многих типов тестов, описанных ранее, целесообразно создать бригаду из нескольких профессиональных экспертов по тестированию систем (которые «всю жизнь» занимаются проверкой систем), одного-двух представителей конечного пользователя, специалиста по инженерной психологии и кого-нибудь из основных аналитиков — авторов проекта или проектировщиков программы. Включение последних не нарушает принципа, запрещающего автору тестировать собственную программу, так как эта программа, вероятно, разрабатывалась коллективом проектировщиков, а не одним этим человеком.

Следовательно, в данном случае основные проектировщики не имеют столь сильных психологических связей с программой, как при тестировании модулей, а ведь именно прочность связей и явилась поводом для формулирования этого принципа.

Что касается второго положения, то здесь следует отметить, что разрабатывающая организация имеет психологические связи с программой. Большинство разрабатывающих организаций крайне заинтересовано в том, чтобы тестирование системы протекало «гладко» и по графику. У них нет причин демонстрировать, что созданная ими программа не удовлетворяет исходным целям. Проверка системы должна выполняться по крайней мере независимой группой лиц, почти не связанной организационно с разрабатывающей организацией. Возможно, наиболее экономически приемлемым путем осуществления проверки системы (в смысле нахождения большинства ошибок или нахождения такого же числа ошибок при меньших финансовых затратах) является заключе-

ние договора на ее проведение с какой-либо фирмой. Этот вопрос обсуждается подробно в последнем разделе настоящей главы.

## ПРИЕМО-СДАТОЧНЫЕ ИСПЫТАНИЯ

Как показывает модель процесса разработки, изображенная на рис 6.3, приемочные испытания<sup>1</sup> есть процесс сравнения разработанной программы с исходным документом, отражающим начальные требования к ней и текущие нужды ее конечных пользователей. Это довольно необычный вид проверки в том смысле, что он выполняется покупателем или конечным пользователем программы и не входит в обязанности разрабатывающей организации. Организация-заказчик (пользователь) выполняет приемочную проверку программы, спроектированной по договору, путем сопоставления характеристик системы с требованиями контракта. Такую проверку целесообразно проводить, применяя тесты, с помощью которых можно попытаться показать, что данная программа не удовлетворяет требованиям контракта. Если тестирование окажется неудачным, то разработанная программа принимается. Разумный покупатель всегда выполняет приемочную проверку, чтобы убедиться в том, что приобретаемый им программный продукт (например, операционная система или компилятор, разработанный изготовителем ЭВМ, система управления базой данных, разработанная компанией по производству программного обеспечения) удовлетворяет его запросам.

---

<sup>1</sup> Автор, пожалуй, излишне лаконичен при описании этого вида проверки программного изделия. Приемочные испытания (ПСИ), во время которых исследуются внешние характеристики изделия, представляют собой, как правило, многоэтапный процесс, регламентированный в отечественной практике различными нормативно-техническими документами. Он особенно трудоемок, сложен и важен при приемке программного обеспечения систем реального времени. Например, программа ПСИ систем реального времени включает обычно все виды тестирования, которые автор относит к тестированию системы и тестированию правильности установки, а также ряд дополнительных испытаний. Стоимость ПСИ может быть весьма значительной. Более подробные сведения о подходе к проведению ПСИ в нашей стране можно найти в книге В. В. Липаева «Проектирование математического обеспечения АСУ. Системотехника. Архитектура. Технология» (М., Советское радио, 1977). — *Примеч. ред.*

## ТЕСТИРОВАНИЕ ПРАВИЛЬНОСТИ УСТАНОВКИ

Последний процесс, изображенный на рис. 6.3, — тестирование правильности установки. Этот процесс не связан с какими-либо фазами процесса проектирования, как все другие процессы тестирования. Он является необычным, поскольку его цель состоит в том, чтобы найти ошибки установки, а не программного обеспечения.

Во время установки систем программного обеспечения пользователь должен выбрать множество опций, разместить и загрузить файлы и библиотеки, зафиксировать правильную конфигурацию аппаратуры; при этом данная программа должна быть связана с другими программами. Назначение проверки — локализация всех ошибок, сделанных в процессе установки.

## ПЛАНИРОВАНИЕ ТЕСТИРОВАНИЯ И КОНТРОЛЬ

. Если представить себе, что тестирование большой системы связано с написанием, выполнением и верификацией десятков тысяч тестов, поддержкой тысяч модулей, исправлением тысяч ошибок и работой, возможно, сотен людей в разные периоды времени в течение года и более, то становится ясным, сколь велика проблема управления проектированием, в частности планирование, управление и контроль процесса тестирования. Управлению тестированием программного обеспечения можно было бы посвятить целую книгу. Поэтому в данном разделе лишь обобщаются некоторые соображения по этому вопросу.

Основной просчет в планировании процесса тестирования (см. гл. 2) состоит в том, что при составлении графика работ предполагается отсутствие ошибок в программном обеспечении. Результатом такого просчета может оказаться недооценка планируемых ресурсов (людей, календарного и машинного времени). Это известная проблема в вычислительной технике. Она усложняется еще и тем, что процесс тестирования приходится на завершающую часть цикла разработки, когда уже трудно маневрировать ресурсами. Вторая и, вероятно, более значительная проблема связана с неправильным определением тестирования. Поскольку опыт использования правильного определения тестирования (цель тестирования — нахождение ошибок) пока недостаточен, про-

верка планируется в предположении, что ошибки не будут обнаружены<sup>1</sup>.

Планирование проверки является главной частью процесса тестирования. План тестирования включает следующие компоненты:

1. *Цели.* Здесь определяются цели каждой фазы тестирования.

2. *Критерий завершения.* Указывается критерий, по которому фиксируется завершение каждой фазы тестирования. Этот вопрос обсуждается в следующем разделе.

3. *Графики работ.* Календарные графики работ составляются для каждой фазы тестирования. Они должны отражать сроки проектирования, кодирования и выполнения тестов.

4. *Ответственные.* Здесь указываются фамилии лиц, которые будут на каждой фазе проектировать, кодировать, выполнять и проверять результаты прогона тестов, а также тех, кто будет исправлять найденные ошибки. Здесь же называется арбитр, поскольку в больших проектах при обсуждении результатов прогона теста, к сожалению, часто возникают споры (например, из-за двусмысленностей или неверных определений в спецификациях).

5. *Библиотеки тестов и стандарты.* В больших проектах необходимы систематические методы идентификации, написания и хранения тестов.

6. *Средства.* Указываются необходимые средства тестирования, включая план, в котором определяются их разработчик или будущий владелец, а также способ и место их использования.

7. *Машинное время.* Разрабатывается план предоставления машинного времени для каждой фазы тестирования.

8. *Конфигурация аппаратуры.* Если нужна специальная конфигурация аппаратуры, то план тестирования

---

<sup>1</sup> При планировании тестирования необходимо иметь обширный и достоверный статистический материал о среднем числе ошибок, приходящихся, например, на тысячу команд программного продукта, среднем времени поиска и устранения ошибки на разных этапах тестирования и отладки, стоимости этих работ. Недостаточный статистический опыт и отсутствие соответствующих нормативов в настоящее время, пожалуй, создаст более серьезные трудности при составлении обоснованных графиков работ, чем просто понимание или непонимание сущности процессов тестирования. — *Примеч. ред.*

описывает этапы, на которых в них возникает необходимость, и требования, которым они должны удовлетворять.

9. *Комплексирование.* Здесь определяется, как будет собрана данная программа (например, пошаговым нисходящим тестированием). Систему, состоящую из подсистем или программ, можно собирать пошаговым способом (используя нисходящий или восходящий подходы, если в качестве компонент рассматриваются не модули, а программы или подсистемы). В таком случае план комплексирования тем более необходим. Он указывает порядок комплексирования, функциональную мощность каждой версии этой системы и обязанности по производству «строительных лесов», или «подпорок» (кодов, моделирующих функции несуществующих компонент).

10. *Процедуры отслеживания.* Указываются средства отслеживания различных аспектов прогресса в тестировании (включая определение модулей, потенциально содержащих ошибки) и его оценки по отношению к графику, ресурсам и критерию завершения.

11. *Процедуры отладки.* Определяются механизмы выдачи сообщений о выявленных ошибках, отслеживания состава и порядка внесения дополнений и изменений в систему. Частью плана отладки должны быть также графики работ, распределение ответственности, средств и машинного времени.

12. *Регрессионное тестирование.* Регрессионное тестирование производится после усовершенствования функций программы или внесения в нее изменений. Его цель — выяснить, не повлияло ли такое изменение отрицательно (не привело ли к регрессу) на другие свойства программы. Обычно оно сводится к повторному выполнению некоторого подмножества тестов программы. Регрессионное тестирование имеет важное значение потому, что внесенные в программу коды изменений и исправлений ошибок, как правило, более подвержены ошибкам, чем исходные коды программы (можно провести аналогию с типографскими ошибками в газетах; большинство из них является следствием редакционных правок в последнюю минуту, а не низким качеством исходного материала). Необходимо составлять план проведения регрессионного тестирования (например, чтобы определить, кто, как и где должен проводить проверку).

## КРИТЕРИЙ ЗАВЕРШЕНИЯ ПРОВЕРКИ

Одним из наиболее трудных является вопрос о том, когда следует закончить тестирование программы, поскольку не представляется возможным определить, является ли выявленная ошибка последней. Действительно, даже для небольших программ было бы неразумно полагать, что в итоге будут найдены все ошибки. Тем не менее тестирование в конце концов придется завершать хотя бы по экономическим соображениям. Этот вопрос решают обычно либо чисто волевым способом, либо с помощью какого-нибудь критерия (часто бессмысленного и непродуктивного). На практике наибольшее распространение получили следующие критерии:

1. Время, отведенное по графику работ на тестирование, истекло.
2. Все тесты выполнены без выявления ошибок (т. е. они неудачны).

Оба эти критерия никуда не годятся; первому можно удовлетворить, ничего не делая (т. е. он не содержит оценки качества тестирования); второй же не имеет смысла, поскольку он не зависит от качества тестов. Кроме того, второй критерий непродуктивен, так как подсознательно побуждает к построению тестов с низкой вероятностью обнаружения ошибки. Как уже отмечалось в гл. 2, мышление человека ориентируется на поставленную цель. Если программист ставит перед собой цель — решить задачу при неудачных тестах, то он будет подсознательно разрабатывать тесты, которые ведут его к этой цели, избегая построения полезных конструктивных тестов с высокой обнаруживающей способностью.

Существуют три категории более или менее приемлемых критериев. К первой из них относятся критерии (далеко не наилучшие), основанные на использовании определенных методологий проектирования тестов. Например, можно определить условие завершения тестирования модуля с помощью тестов, получаемых двумя путями: 1) удовлетворением комбинаторному критерию покрытия и 2) методом анализа граничных значений по спецификации интерфейса модуля. Все получающиеся в результате тесты в конце концов должны стать неудачными.

Завершение тестирования функций можно определить при выполнении следующих условий:

тесты, получаемые методами 1) функциональных диаграмм, 2) анализа граничных значений, 3) предположения об ошибке, в конце концов должны стать неудачными.

Хотя эти критерии лучше двух определенных ранее, с ними связаны три проблемы. Во-первых, они бесполезны на той фазе тестирования, когда определенные методологии становятся непригодными, например, на фазе тестирования системы. Во-вторых, такое измерение субъективно, так как нет гарантии, что данный специалист использовал нужную методологию (например, анализ граничных значений) правильно и точно. В-третьих, вместо того чтобы поставить цель и дать возможность выбора наиболее подходящего пути ее достижения, рассмотренные критерии предписывают использование конкретных методологий проектирования тестов, но не ставят никакой цели. Таким образом, критерии первой категории полезны для некоторых фаз тестирования, но их следует применять лишь в тех случаях, если у вас уже есть опыт работы с этими методологиями.

Критерий второй категории, видимо, представляет наибольшую ценность, так как четко формулирует условие завершения тестирования. Поскольку цель тестирования — поиск ошибок, почему бы в качестве критерия не выбрать некоторое заранее установленное число ошибок? Например, можно установить, что проверка модуля завершается после обнаружения трех ошибок. Возможно, условием завершения проверки системы следует считать обнаружение и устранение 70 ошибок или общее время проверки 3 месяца, смотря по тому, какое из событий наступит раньше.

Заметим, что такой критерий подтверждает определение тестирования. С ним связаны две в принципе решаемые проблемы. Одна из них заключается в установлении способа получения числа выявленных ошибок. Для этого требуется:

1. Оценить общее число ошибок в программе.
2. Выяснить, какой процент этих ошибок можно обнаружить с помощью тестирования.
3. Определить, какая именно часть ошибок возникла в процессе проектирования и во время каких фаз тестирования целесообразно их выявлять.

Грубую оценку общего числа ошибок можно получить несколькими методами. Один из них основан на анализе данных о разработке других программ. Существует целый ряд моделей предсказания (например, [1], гл. 18), часть которых требуют тестировать программу в течение какого-то периода времени, фиксировать интервалы времени между обнаружением последовательных ошибок и затем подставлять эти интервалы как параметры в формулу. Некоторые модели предполагают, что в программу искусственно вносятся ошибки, но об их наличии не объявляется. Программа тестируется определенное время, после чего проверяется соотношение обнаруженных и необнаруженных ошибок из числа тех, которые были в нее внесены. Существуют модели, предполагающие, что в течение определенного времени программа отлаживается двумя независимыми тестовыми бригадами, а затем анализируются ошибки, найденные каждой бригадой, и ошибки, обнаруженные обеими бригадами вместе. Эти параметры используются для оценки общего числа ошибок. Другой «оптовый» метод получения такой оценки основан на статистических средних величинах, широко применяемых в промышленности. Например, число ошибок, которое существует в типичных программах ко времени завершения кодирования (до проведения сквозного просмотра или инспекции), составляет примерно 4—8 на 100 операторов программы.

Выяснение процента ошибок, которые можно найти методом тестирования, должно учитывать природу программы и последствия невыявленных ошибок и может быть в некоторой степени произвольным.

Определение ошибок, возникающих в процессе проектирования, связано с большими трудностями, поскольку в настоящее время информация о них практически отсутствует. Имеющиеся данные свидетельствуют о том, что в больших программах примерно 40% всех ошибок составляют ошибки проектирования логики и кодирования, а остальные порождены на более ранних этапах проектирования.

Поясним изложенное на простом примере, хотя читатель, желающий воспользоваться этим критерием, должен выработать свои собственные оценки, подходящие к его программе. Допустим, что тестируется программа размером в 10000 операторов; число ошибок, остающихся после инспекций исходного текста, оценивается в пять

на 100 операторов. Цель тестирования — обнаружить 98% ошибок кодирования и логики и 95% ошибок проектирования. Таким образом, общее число ошибок равно 500. Предполагается, что 200 из них составляют ошибки кодирования и логики, а 300 — ошибки проектирования. Следовательно, требуется найти 196 ошибок кодирования и логики и 285 ошибок проектирования. В табл. 6.1 по-

Таблица 6.1

Гипотетическая оценка распределения ошибок по этапам проверки

	Ошибки кодирования и логики	Ошибки проектирования
Проверка модуля	65%	0%
Проверка функций	30%	60%
Проверка системы	3%	35%
Итого	98%	95%

казано, на каком этапе тестирования они должны быть обнаружены.

Если по графику четыре месяца отводятся на тестирование функций и три месяца — на тестирование системы, то можно установить следующие критерии завершения:

1. Найденны и исправлены 130 ошибок (65% от оцененных 200 ошибок кодирования и логики).

2. Найденны и исправлены 240 ошибок (30% от 200 плюс 60% от 300) или истекли четыре месяца, отведенные на тестирование функций, смотря по тому, какое событие наступит позже. (Дело в том, что если 240 ошибок будут обнаружены быстро, то это может означать, что недооценено их общее число, прекращать тестирование функций не следует.)

3. Найденны и исправлены 111 ошибок или истекли три месяца, отведенные на тестирование системы, смотря по тому, какое событие наступит позже.

Другая очевидная проблема, связанная с критерием второй категории, — это проблема переоценки. Что если в приведенном выше примере к моменту начала проверки функций остается менее 240 ошибок? Основываясь на данном критерии, завершить фазу проверки функций не удастся никогда. Возникает довольно странная ситуация: не хватает ошибок, программа оказывается слишком

ком «хорошей». Казалось бы, проблемы не должно быть, поскольку это именно то, к чему мы стремимся. Однако если она возникает, то для ее решения требуется лишь немного здравого смысла. Если за четыре месяца не удастся найти 240 ошибок, руководитель проекта может пригласить со стороны эксперта, который, проанализи-

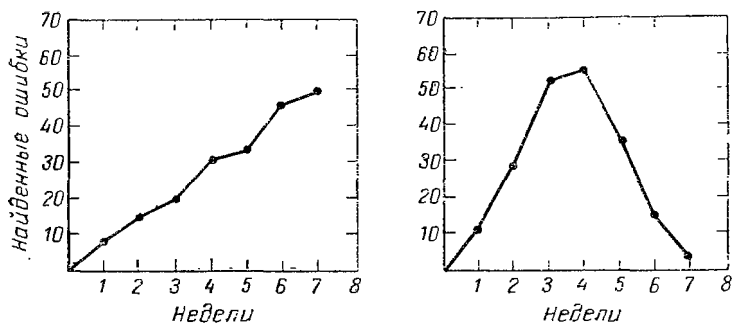


Рис. 6.5. Оценка завершения тестирования по графику ошибок, выявленных в единицу времени

ровав тесты, выскажет свое мнение о причинах возникновения проблемы: тесты могут быть неадекватны (1) или удачны (2), но в программе действительно мало ошибок.

Критерий третьей категории основан в значительной степени на здравом смысле и интуиции. Для его применения на фазе проверки требуется построить график числа ошибок, найденных в единицу времени. Анализируя форму кривой, часто можно определить, следует ли продолжать проверку на данной фазе или закончить ее и перейти к следующей.

Пусть некоторая программа проходит фазу тестирования функций. Строим график числа ошибок, найденных в ней за неделю. Даже если удовлетворяется критерий необходимого числа найденных ошибок, кривая за семь недель тестирования такова (рис. 6.5 слева), что неразумно прекратить тестирование функций этой программы. Поскольку скорость обнаружения ошибок велика (находят много ошибок), наиболее правильное решение (с учетом того, что нашей целью является их нахождение) — продолжить тестирование функций с проектированием при необходимости дополнительных тестов.

С другой стороны, предположим, что график имеет вид, представленный на рис. 6.5 справа. Эффективность обнаружения ошибок значительно уменьшилась, поэтому можно закончить тестирование функций и перейти к следующей фазе, например тестированию системы. (Конечно, необходимо также рассмотреть и другие факторы, которые могли привести к уменьшению эффективности обнаружения ошибок: недостаток машинного времени или невысокая обнаруживающая способность имеющихся тестов.)

На рис. 6.6 показано, что происходит, когда забывают

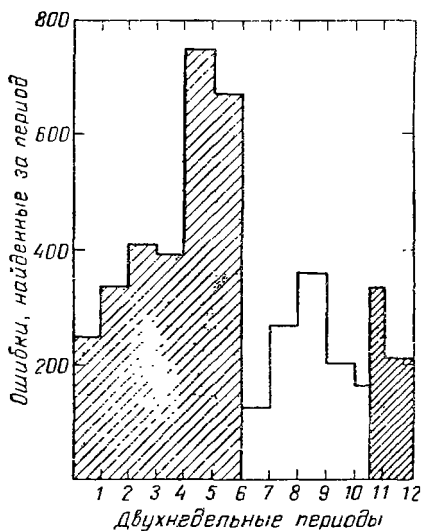


Рис. 6.6. Исследование процессов тестирования по завершении большого проекта

строить график числа обнаруженных ошибок. Диаграмма описывает три фазы тестирования очень большой системы программного обеспечения [5]; она построена как результат исследования этого проекта после его завершения. Вывод очевиден: после шестого периода не следовало переходить к другим фазам тестирования. На протяжении всего шестого периода скорость обнаружения ошибок была высокой (а для тестирования чем выше эта

скорость, тем лучше), переход же на следующую фазу привел к ее значительному снижению.

Наилучшим критерием завершения тестирования является, видимо, комбинация трех рассмотренных выше критериев. Для тестирования модулей оптимальным будет, очевидно, первый критерий завершения, потому что в большинстве проектов на этой фазе не следят за числом обнаруженных ошибок; здесь важно, чтобы использовался определенный набор методологий проектирования тестов. На фазах тестирования функций и системы кри-

терием завершения может служить останов по достижении заранее заданного числа обнаруженных ошибок или по достижении момента, предопределенного графиком работ, смотря по тому, такое событие наступит раньше (при условии, что анализ диаграммы зависимости числа ошибок от времени тестирования покажет снижение продуктивности этой проверки).

## НЕЗАВИСИМЫЕ АГЕНТСТВА ПО ТЕСТИРОВАНИЮ

Ранее в настоящей главе и гл. 2 подчеркивалось, что организация не должна тестировать разработанные ею программы на том основании, что она может испытывать трудности при попытках объективно оттестировать созданный ею продукт. В организационной структуре данной фирмы проверяющая (тестирующая) организация должна быть по возможности отдалена от разрабатывающей. Желательно даже, чтобы проверяющая организация вовсе не входила в эту фирму, так как, будучи ее составной частью, она находится в единой организационной системе с разрабатывающей организацией.

Так, ВВС США одно время заключали контракт на разработку программного обеспечения с одной организацией, а контракт на тестирование этого программного обеспечения — с другой [6]. Сейчас ВВС имеют свою собственную, административно независимую организацию по тестированию (Центр ВВС по проверке и оценке) и практикуют привлечение подрядчиков по тестированию, не зависящих от подрядчиков по разработке [7].

Эффективность этого шага очевидна [8]: «Одним из наиболее удачных подходов к повышению надежности программного обеспечения, который в настоящее время применяют ВВС, особенно по отношению к наиболее критичному по надежности программному обеспечению систем реального времени, является использование независимых агентств по проведению анализа, тестирования и общей оценки программного обеспечения».

Отмечаемые обычно преимущества усиливаются мотивацией процесса тестирования, здоровой конкуренцией с разрабатывающей организацией, разделенностью процесса тестирования и административного контроля разрабатывающей организации, а также большим накопленным опытом, на который при решении этой проблемы опираются независимые агентства по тестированию. Если та-

кая практика получит распространение, то это откроет дополнительные возможности для развития нового типа финансовой деятельности в области обработки данных.

#### ЛИТЕРАТУРА

1. Myers G. J. Software Reliability: Principles and Practices. New York, Wiley-Interscience, 1976. Русский перевод: Майерс Г. Надежность программного обеспечения. М., Мир, 1980.
2. Brooks F. P. Jr. The Mythical Man-Month: Essays on Software Engineering, Reading, Mass., Addison-Wesley, 1975. Русский перевод: Брукс Ф. П. Как проектируются и создаются программные комплексы. М., Наука, 1979.
3. McPhee W. S. Operating System Integrity in OS/VS2. — IBM Systems J., 1974, 12(3), p. 230—252.
4. Pinchuk P. L. TRW Evaluation of a Secure Operating System. — Data Security and Data Processing, V. 6, Evaluation and Installation Experiences: Resource Security System. White Plains, N. Y., IBM, 1974, p. G320—1376.
5. Craig C. R. et al. Software Reliability Study. — RADC—TR—74—250, TWR Corp., Redondo Beach, Cal., 1974.
6. Shelly M. Computer Software Reliability: Fact or Myth? — TR—MMER/RM—73—125, Hill Air Force Base, Utah, 1973.
7. Magill C. R. The Role of an Independent Software Validation Agency. — Report 76—1, Defense Systems Management School, Fort Belvoir, Va, 1976.
8. Thayer R. H. and Hinton E. S. Software Reliability — A Method that Works. — Proceedings of the 1975 National Computer Conference. Montvale, N. J., AFIPS Press, 1975, p. 877—883.

## ГЛАВА 7

### ОТЛАДКА

Основываясь на определениях, приведенных ранее в этой книге, отладку программы можно представить как процесс, осуществляемый после удачного выполнения теста. Процесс отладки начинается при обнаружении ошибки (например, при удачном завершении теста) и проводится в два этапа: 1) определяется природа и местонахождение подозреваемой ошибки в программе; 2) фиксируется или исправляется ошибка.

Следует отметить, что отладка является единственной частью процесса разработки программного обеспечения, к которой программисты питают некоторую неприязнь. Причины такого отношения, по-видимому, заключаются в следующем:

1. Программисты, которые не применяют «обезличенного программирования» [1], часто находят отладку психологически трудной, поскольку при ее выполнении выясняется, что они небезупречны (т. е. допускают ошибки при проектировании и кодировании программ).
2. Из всех видов деятельности при разработке программного обеспечения именно отладка требует наибольших интеллектуальных затрат. Более того, она обычно выполняется при громадном напряжении (организационном или внутреннем, которое вызвано стремлением установить подозреваемую ошибку как можно быстрее), что существенно усложняет проблему.
3. Существующие методы и природа большинства языков программирования таковы, что любая ошибка потенциально может возникнуть в любом операторе программы. Например, без первоначальной проверки теоретически нельзя исключить возможность того, что числовые ошибки в значениях оплаченных чеков, полученных программой расчета зарплаты, являются следствием ошибки, содержащейся в подпрограмме, которая просит оператора

системы указать форму выдачи результата на печатающее устройство. Это совершенно не похоже на наладку физической системы, такой, как автомобиль. При непредвиденной остановке автомобиля (симптом) мы немедленно и вполне обоснованно исключаем части системы, которые не могут быть причиной его неисправности (например, радиоприемник, спидометр, стержень замка). Все дело в двигателе, и, основываясь на общем знании автомобильного двигателя, можно исключить его определенные компоненты (водяной насос и масляный фильтр).

4. По сравнению с другими этапами разработки программного обеспечения процесс отладки мало исследован и слабо освещен в литературе. Недостаточно выпускается и формальных инструкций для его выполнения.

Хотя темой данной книги является тестирование, а не отладка, краткое обсуждение последней здесь вполне оправдано, так как оба эти процесса тесно связаны между собой. Из двух аспектов отладки (определения местонахождения ошибки и ее исправления) первый составляет до 95% решения проблемы. Поэтому в настоящей главе основное внимание уделяется поиску местонахождения ошибки при наличии фактов, свидетельствующих о том, что ошибка существует (т. е. на основании результатов прогона теста).

## МЕТОДЫ «ГРУБОЙ СИЛЫ»

Наиболее общими при отладке программы являются довольно неэффективные методы «грубой силы». Причина популярности этих методов, возможно, заключается в том, что они не требуют значительного внимания и больших умственных затрат.

Методы грубой силы можно разделить по крайней мере на три категории: 1) отладка с использованием дампа памяти; 2) отладка в соответствии с общим предложением «расставить операторы печати по всей программе»; 3) отладка с использованием автоматических средств. Наименее эффективна из них отладка посредством анализа дампа памяти (обычно необработанного отображения состояния всей памяти в восьмеричной или шестнадцатеричной форме). С ней связаны следующие проблемы:

1. Сложность установления соответствия между ячейками памяти и переменными в исходной программе.

2. Значительный объем выдаваемых данных, многие из которых не используются.

3. В действительности распечатка состояния памяти является *статическим* отображением программы (т. е. ее состояния только в какой-то момент времени), но для нахождения большинства ошибок должна быть изучена *динамика* выполнения этой программы (т. е. изменение ее состояния во времени).

4. Практически распечатка состояния памяти редко получается точно в том месте программы, где находится ошибка. Следовательно, дамп не отражает состояние программы в точке, содержащей ошибку; действия, осуществляемые в промежутке между моментом проявления ошибки и моментом получения дампа, могут помешать локализовать ошибку.

5. Отсутствие описанных в литературе методологий выявления причины ошибки посредством анализа дампа памяти (многие программисты просто смотрят на дампы, очевидно, ожидая, что ошибка волшебным образом сама себя обнаружит).

Методы второй категории, при которых операторы отображения значений переменных расставляются по всей программе, содержащей ошибку, несколько лучше. Эти методы имеют много недостатков, тем не менее их использование часто оказывается предпочтительнее использования дампов. Они позволяют отображать динамику выполнения программы и рассматривать информацию, которую легче поставить в соответствие исходному тексту. Перечислим некоторые из их недостатков:

1. Расстановка операторов печати в программе в значительной степени заставляет программиста работать методом проб и ошибок, вместо того чтобы поощрять его в процессе отладки думать о поставленной перед ним задаче.

2. В процессе отладки может оказаться необходимым проанализировать большое число данных.

3. Здесь требуется изменять программу при отладке; эти изменения могут скрыть ошибку, нарушить критические временные отношения или внести в программу новые ошибки.

Стоимость использования методов данной категории для больших программ или систем может быть огромной, но для небольших программ они вполне применимы. Кроме того, они не подходят для определенных типов

программ (например, операционных систем, программ управления процессами).

Методы третьей категории, использующие автоматические средства отладки, подобны методам второй категории, но отличаются от них в части внесения изменений в программу. Анализ динамики выполнения программы осуществляется с помощью отладочных средств языка программирования или специальных интерактивных средств отладки. Типичными языковыми средствами, которые могут быть использованы, являются средства, обеспечивающие получение распечатки трасс выполненных операторов, вызов подпрограмм и (или) изменений специфицированных переменных. Общая функция средств отладки — установление набора «точек прерывания», вызывающих приостанов процесса исполнения программы после выполнения определенного оператора или изменения значения определенной переменной и позволяющих программисту, работающему за терминалом, проверять текущее состояние программы. Однако эти методы в значительной степени относятся к методам проб и ошибок, и часто их результаты содержат чрезмерное число избыточных данных.

Общей проблемой методов «грубой силы» является то, что они игнорируют процесс *обдумывания*. Можно провести аналогию между отладкой программы и расследованием убийства. Фактически во всех детективных романах тайна раскрывается скорее посредством тщательного анализа улик и объединения незначительных на первый взгляд деталей, чем методами «грубой силы», такими, как блокировка улиц или осмотр имущества. Например, так был раскрыт ряд убийств из оружия 44-го калибра в Нью-Йорке в 1976—1977 гг. Подозреваемого удалось задержать не с помощью методов «грубой силы» (насыщением авиалиний и газет его портретами, выполненными художником, и увеличением числа полицейских на улицах), а в результате обнаружения нескольких, казалось бы, незначительных улик, одной из которых был билет на автостоянку.

Экспериментально доказано [2, 3] (причем это справедливо как для начинающих, так и для опытных программистов), что средства отладки не помогают процессу отладки и что программисты, предпочитающие обдумывание результатов прогона бесцельному многократному исполнению программы на одних и тех же тестах, быст-

рее и точнее находят ошибки. Следовательно, использование методов «грубой силы» рекомендуется только в том случае, если все остальные методы не дали желаемого эффекта или в дополнение (но не вместо) к описанным в последующих разделах процессам, основанным на обдумывании.

## МЕТОД ИНДУКЦИИ

Считается, что большинство ошибок может быть обнаружено посредством тщательного анализа, во многих случаях даже без выхода на машину. Одним из таких методов является индукция, в процессе которой осуществ-

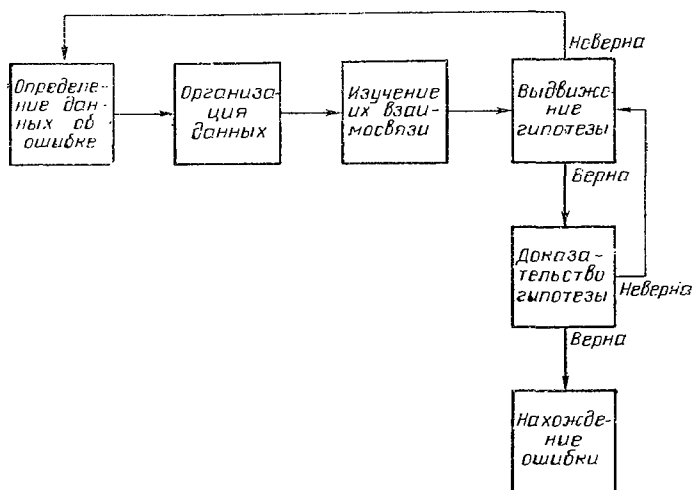


Рис. 7.1. Процесс индуктивной отладки

ляется анализ от частного к целому. При этом, просматривая детали (симптомы ошибки, возможно, установленные одним или несколькими тестами) и взаимосвязи между ними, часто можно прийти к ошибке.

Процесс индукции показан на рис. 7.1. Он разбивается на следующие шаги:

1. *Определение данных, имеющих отношение к ошибке.* Основная ошибка, встречающаяся при плохо организованной отладке программы, заключается в том, что рассматриваются все имеющиеся в распоряжении данные

или симптомы, относящиеся к задаче. Первым шагом должно быть перечисление всех действий, свидетельствующих о правильном выполнении программы и всех ее неправильных действий (т. е. симптомов, которые приводят к выводу о наличии ошибки). Дополнительные данные об ошибке получаются в результате выполнения подобных тестов, которые не вызывают появления указанных симптомов.

2. *Организация данных.* Индукция подразумевает анализ от частного к общему, поэтому второй шаг заключается в структурировании данных, имеющих отношение к ошибке, с целью выявления неких закономерностей. Особую важность представляет исследование на *противоречивость* (например, «ошибка встречается только тогда, когда счет клиента имеет дебетовое — положительное — сальдо»). Весьма полезной является организационная методика, известная как «Метод» [4]. Таблица, показанная на рис. 7.2, использовалась для структурирования

?	Есть	Нет
Что		
Где		
Когда		
Какова степень		

Рис. 7.2. Метод структурирования симптомов ошибки

имеющихся в распоряжении данных. Строка «Что» образует список общих симптомов ошибки, строка «Где» описывает место, в котором эти симптомы были обнаружены, строка «Когда» содержит все, что известно о времени проявления симптомов, а строка «Какова степень» определяет область распространения и степень их важности. Обратим внимание на графы «Есть» и «Нет» таблицы. Они описывают противоположные ситуации, которые могут в конечном счете привести к гипотезам об ошибке.

3. *Выдвижение гипотезы.* Следующими шагами являются изучение взаимосвязи между признаками и выдвижение одной или нескольких гипотез о причине ошибки с учетом

закономерностей, выявленных в структуре симптомов ошибки. Если нельзя выдвинуть гипотезы, то необходимы дополнительные данные, которые, возможно, будут получены путем построения и выполнения дополнительных тестов. В том случае, когда возможно несколько гипотез, первой выбирается наиболее вероятная из них.

4. *Доказательство гипотезы.* Пропуск этого шага и переход непосредственно к заключениям и попыткам решить проблему является серьезной ошибкой, обычно влияющей на результат проведения отладки. Необходимо доказать приемлемость гипотезы, прежде чем взять ее за основу. Отсутствие доказательства часто приводит к разрешению только части проблемы. Гипотеза доказывается путем сравнения ее с первоначальными симптомами ошибки или данными. Она должна *полностью* объяснить существование этих симптомов. Если такое объяснение получить не удастся, то, значит, гипотеза либо не обоснована, либо не полна, либо существуют многочисленные ошибки.

Например, предположим, что в программе ранжирования результатов экзаменов, описанной в гл. 4, обнаружена очевидная ошибка. Она заключается, видимо, в том, что медиана оценки качества ответа вычислялась неправильно в некоторых, но не во всех случаях. Специальный тест содержал записи об оценках качества ответов 51 студента. Среднее значение оказалось напечатанные правильно: 73.2, а значение медианы — 26 вместо ожидаемой величины 82. При рассмотрении результатов этого и нескольких других тестов симптомы были сгруппированы так, как показано на рис. 7.3.

Следующий шаг — выдвижение гипотезы об ошибке на основании просмотра результатов выполнения программы и противоречий. Одно из противоречий состоит в том, что, видимо, ошибка проявляется только при выполнении тестов с *нечетным* числом студентов. Это может быть случайностью, но имеет большое значение, так как от того, является ли число студентов четным или нечетным, зависит способ вычисления медианы. Кажется странным также и то, что в таких тестах вычисленная медиана всегда меньше или равна числу студентов ( $26 \leq 51$  и  $1 \leq 1$ ). В подобных ситуациях можно повторно запустить программу на тесте, содержащем данные о 51 студенте, с другими оценками качества ответов и посмотреть, как это повлияет на вычисление медианы.

?	Есть	Нет
Что	В третьем сообщении значение медианы напечатано неверно	Вычисление среднего значения или стандартного отклонения
Где	Только в третьем сообщении	В других сообщениях оценки студентов кажутся вычисленными правильно
Когда	При выполнении теста, содержащего данные о 51 студенте	При выполнении тестов для 2 и 200 студентов
Какова степень	Значение медианы напечатано равным 26. Невсрный результат также получен в тесте, содержащем данные об одном студенте. Значение медианы напечатано равным 11	

Рис. 7.3. Пример структурирования симптомов ошибки

Если и теперь медиана остается равной 26, то в строке «Какова степень» графы «Нет» будет записано следующее: «Значение медианы, кажется, не зависит от действительных значений оценок». Хотя этот результат представляет собой важный симптом, можно было предположить характер ошибки и без него. Из имеющихся данных видно, что вычисленное значение медианы равно половине числа студентов, округленной до ближайшего большего целого. Другими словами, если считать значения оценок такими, какими они были помещены в сортируемую таблицу, то можно утверждать, что программа печатает входной номер среднего студента, а не его оценку. Следовательно, гипотеза о природе ошибки достоверна. Эта гипотеза должна быть доказана посредством проверки текста программы или выполнения нескольких дополнительных тестов.

## МЕТОД ДЕДУКЦИИ

Процесс дедукции, показанный на рис. 7.4, позволяет на основании некоторых общих теорий или предпосылок, используя операции исключения и уточнения, прийти к определенному заключению (обнаружить место ошиб-

жи). Если вернуться к примеру с расследованием убийства, то в отличие от метода индукции, при котором подозреваемый выявляется по совокупности улик, процесс дедукции начинается с определения круга подозреваемых лиц, после чего посредством исключения (садовник имеет

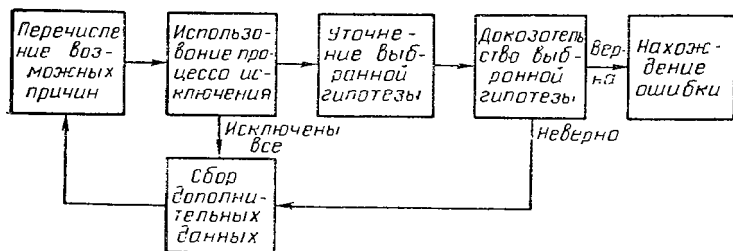


Рис. 7.4. Процесс дедуктивной отладки

обоснованное алиби) и уточнения (преступник — человек с рыжими волосами) приходят к выводу, что дворецкий совершил убийство. Процесс дедукции состоит в следующем:

1. *Перечисление возможных причин или гипотез.* Первый шаг заключается в разработке списка всех возможных причин ошибки. Эти причины не обязательно должны полностью объяснять ошибку; они являются только версиями, с помощью которых можно структурировать и анализировать имеющиеся в распоряжении данные.

2. *Использование данных для исключения возможных причин.* Путем тщательного анализа данных и (особенно) поиска противоречий (здесь можно использовать рис. 7.2) исключаются все возможные причины, кроме одной. Если исключить все причины, то потребуются дополнительные данные (получаемые, например, при построении дополнительных тестов) для выдвижения новых гипотез. Если же остается более чем одна причина, то первой выбирается наиболее вероятная из них — основная гипотеза.

3. *Уточнение выбранной гипотезы.* Возможная причина может быть определена верно, но маловероятно, чтобы она достаточно полно отражала специфику ошибки. Поэтому следующим шагом должно быть использование доступных данных для уточнения версии (например, «ошибка при обращении к последней записи файла») с

учетом некоторой специфики (например, «последняя запись в буфере затирает признак конца файла»).

4. *Доказательство выбранной гипотезы.* Этот шаг совпадает с шагом 4 в методе индукции.

В качестве примера рассмотрим тестирование функций команды DISPLAY (см. гл. 4). Начнем с выполнения четырех из 38 тестов, соответствующих процессу, отображенному на функциональной диаграмме. Инициализируем состояние памяти так, чтобы первое, пятое, девятое, ... слова содержали величину 0000; второе, шестое, ... слова — величину 4444; третье, седьмое, ... слова — величину 8888; четвертое, восьмое, ... слова — величину CCCC. Таким образом, каждому слову в памяти присвоено значение, равное младшей шестнадцатеричной цифре адреса первого байта этого слова (например, байты 23FC, 29FD, 23FE и 23FF содержат значение C).

Результаты тестирования (ожидаемые и действительные) показаны на рис. 7.5. Очевидно, при оценке этих ре-

Входные данные теста	Ожидаемые выходные данные	Действительные выходные данные
DISPLAY.E	000000=0000 4444 8888 CCCC	M1 нарушение синтаксиса команды
DISPLAY 21—29	000020=0000 4444 8888 CCCC	000020=4444 8888 CCCC 0000
DISPLAY.11	000000=0000 4444 8888 CCCC 000000=0000 4444 8888 CCCC	000000= =0000 4444 8888 CCCC
DISPLAY 8000—END	M2 запрашиваемая память выходит за пределы действительной памяти	008000= =0000 4444 8888 CCCC

Рис. 7.5. Результаты выполнения теста для проверки команды

зультатов возникают некоторые проблемы, так как ни один из тестов не дал ожидаемого эффекта (все тесты были удачными). Начнем с выявления ошибки, связанной с первым тестом. Команда показывает, что должны быть отображены E (14 десятичных) ячеек начиная с Ø (отсутствие значения до точки). (Напомним, что, как утверждается в спецификациях, все выходные результаты содержат четыре слова или 16 байт). Можно привести следующие возможные причины появления ошибочного сообщения:

1. Программа не воспринимает слово DISPLAY.
2. Программа не воспринимает число отображаемых байтов.
3. Программа не допускает пропуска в качестве первого операнда (т. е. она ожидает, что числу отображаемых байтов предшествует адрес).
4. Программа не допускает *E* в качестве правильного значения числа байтов.

Следующим шагом является попытка исключить причины. Если все причины исключены, следует пересмотреть и расширить данный список. Если же осталась более чем одна причина, можно рассмотреть дополнительные тесты для нахождения единственной гипотезы об ошибке или перейти к рассмотрению наиболее вероятной ее причины. Так как в нашем распоряжении имеются результаты выполнения других тестов, легко установить, что второй тест на рис. 7.5 исключает первую гипотезу, а третий, хотя и приводит к неправильному результату, по-видимому, исключает вторую и третью гипотезы.

Далее необходимо уточнить четвертую гипотезу. Эта гипотеза представляется достаточно специфической, но интуиция подсказывает, что она может дать больше, чем кажется на первый взгляд, поскольку проявляется как подобная, но более общая ошибка. Можно утверждать, что программа не распознает специальные шестнадцатеричные знаки (A — F). Отсутствие таких знаков в других тестах вполне объясняет эту ошибку.

Однако прежде чем сделать заключение, мы должны рассмотреть *всю* имеющуюся в нашем распоряжении информацию. Четвертый тест может отображать совершенно другую ошибку или предоставлять дополнительные данные о рассматриваемой ошибке. Наибольший допустимый адрес в рассматриваемой системе равен 7FFF, поэтому непонятно, как при выполнении четвертого теста отображается область, которая представляется несуществующей. Поскольку значения отображаемых величин задаются, а не являются внутренними константами программы, можно предположить, что эта команда некоторым образом отображает содержимое каких-то ячеек, имеющих адреса в диапазоне 7—7FFF. Например, такое предположение может быть справедливым, если программа воспринимает операнды в команде как *десятичные* величины (а не шестнадцатеричные, которые указаны в спецификации). Это подтверждается третьим те-

стом; вместо отображения 32 байт памяти, с последующим увеличением на 11 в шестнадцатеричной системе (17 в десятичной), при выполнении теста отображается 16 байт памяти. Последнее согласуется с нашей гипотезой, согласно которой «11» воспринимается как десятичная величина. Следовательно, по уточненной гипотезе программа воспринимает содержимое байта, операнды, содержащие адреса памяти, и адреса памяти для выходного листинга как десятичные величины.

Последний шаг заключается в доказательстве этой гипотезы. Рассмотрим четвертый тест. Если 8000 интерпретировать как десятичное число, то соответствующее ему шестнадцатеричное число будет 1 F40, что приведет к полученному результату. Для дальнейшего доказательства рассмотрим второй тест. Результат его выполнения является неправильным, но если 21 и 29 трактовать как десятичные числа, то должно быть отображено содержимое адресов 15—1D, что и происходит на самом деле. Следовательно, мы почти определенно установили ошибку: программа воспринимает операнды как десятичные величины и распечатывает адреса памяти так же, как десятичные величины в нарушение спецификации. Кроме того, эта ошибка представляется причиной неверных результатов выполнения всех четырех тестов. Таким образом, несложные рассуждения позволили нам выявить ошибку и тем самым решить три другие проблемы, которые на первый взгляд казались несвязанными.

Заметим, что ошибка, вероятно, проявляется в двух местах программы: в части, которая интерпретирует входную команду, и в части, которая печатает выходной список адресов памяти.

С другой стороны, эта ошибка, возможно, вызванная неправильным пониманием спецификации, укрепляет нас в мысли о том, что программист не должен пытаться тестировать свою программу. Если программист, допустивший ошибку, является также и разработчиком тестов, то вполне вероятно, что он сделает ту же самую ошибку и при написании тестов. Иными словами, ожидаемые программистом результаты окажутся не такими, как показано на рис. 7.5. Они будут вычислены в предположении, что операнды являются десятичными величинами. Следовательно, эта фундаментальная ошибка, очевидно, могла быть не обнаружена.

Эффективным методом локализации ошибки для небольших программ является прослеживание в обратном порядке логики выполнения программы с целью обнаружения точки, в которой нарушена логика. Другими словами, отладка начинается в точке программы, где был обнаружен (например, напечатан) некорректный результат. Для этой точки на основании полученного результата следует установить (логически вывести), какими должны быть значения переменных. Мысленно выполняя из данной точки программу в обратном порядке и опять рассуждая примерно так: «если в этой точке состояние программы (т. е. значения переменных) было таким, то в другой точке должно быть следующее состояние», можно достаточно быстро и точно локализовать ошибку (т. е. определить место в программе между точкой, где состояние программы соответствовало ожидаемому, и первой точкой, в которой состояние программы отличалось от ожидаемого).

### МЕТОД ТЕСТИРОВАНИЯ

Последний метод отладки, основанный на «обдумывании», заключается в использовании тестов. Этот метод может показаться несколько странным, так как в начале главы отмечались различия между отладкой и тестированием. Однако существуют два типа тестов: тесты для тестирования, целью которых является обнаружение заранее не определенной ошибки, и тесты для отладки, цель которых — обеспечить программиста информацией, полезной для выявления местонахождения подозреваемой ошибки. Тесты для тестирования имеют тенденцию быть «обильными» (небольшим числом тестов пытаются покрыть большое число условий), а тесты для отладки пытаются покрыть только одно условие или небольшое число условий.

Другими словами, после обнаружения симптома подозреваемой ошибки этот метод используют для написания вариантов первоначального теста, чтобы попытаться ее локализовать. На самом деле метод тестирования не является каким-то специальным, исключительным; он часто применяется совместно с методом индукции (для получения информации, необходимой при выдвижении

гипотез и (или) их доказательстве) либо с методом дедукции (для исключения возможных причин, уточнения отобранных гипотез и (или) их доказательства).

## ПРИНЦИПЫ ОТЛАДКИ

Как и в гл. 2, в данном разделе рассматривается ряд принципов отладки, многие из которых по своей природе являются психологическими. Ряд принципов интуитивно очевиден, но, несмотря на это, их часто забывают или игнорируют. Здесь представлены две группы принципов, поскольку процесс отладки складывается из двух этапов (определения местонахождения ошибки и последующего ее исправления).

### Принципы локализации ошибок.

*Думайте.*

В предыдущих разделах подразумевалось, что отладка представляет собой процесс решения задач. Наиболее эффективный метод отладки заключается в анализе информации, связанной с симптомами ошибок. Для ее эффективного проведения специалист должен обладать способностью точно определять большинство ошибок без использования ЭВМ.

*Если вы зашли в тупик, отложите рассмотрение программы.*

Наше подсознание является мощным механизмом решения проблем. То, что мы часто приписываем вдохновению, оказывается всего лишь выполненной подсознанием работой по решению задачи, тогда как наша сознательная деятельность в это время связана с чем-нибудь другим, например с едой, прогулкой или просмотром кинофильма. Если вы не можете локализовать ошибку в приемлемые сроки (предположительно за 30 минут для небольших программ и за несколько часов для больших), прекратите поиски и займитесь каким-нибудь другим делом, так как эффективность вашей работы, во всяком случае, значительно снизится. Проблему следует «забыть» до тех пор, пока вы либо подсознательно не найдете ее решения, либо отдохнете и будете готовы вновь рассмотреть симптомы ошибки.

*Если вы зашли в тупик, изложите задачу кому-нибудь еще.*

Сделав это, вы, вероятно, обнаружите что-то новое.

Часто случается так, что, просто пересказав задачу хорошему слушателю, вы вдруг найдете решение без какой-либо помощи с его стороны.

*Используйте средства отладки только как вспомогательные.*

Не применяйте эти средства вместо того, чтобы обдумывать задачу. Как отмечалось ранее в настоящей главе, такие средства, как дампы и трассы, отражают случайный подход к отладке. Эксперименты показали, что программисты, избегающие применения средств отладки, даже при отлаживании незнакомых им программ выполняют ее лучше, чем те, кто пользуется этими средствами [3].

*Избегайте экспериментирования. Пользуйтесь им только как последним средством.*

Наиболее общей ошибкой, которую допускают начинающие программисты, занимающиеся отладкой, является попытка решить задачу посредством внесения в программу экспериментальных изменений («Я не знаю, что неправильно, но я изменю этот оператор DO и посмотрю, что получится».) Этот абсолютно неверный подход не может даже рассматриваться как отладка; он основан на случайности. Экспериментирование не только уменьшает вероятность успеха, но часто и усложняет задачу, поскольку при этом в программу вносятся новые ошибки.

## Принципы исправления ошибок.

*Там, где есть одна ошибка, вероятно, есть и другие.*

Это положение повторяет принцип гл. 2, который утверждает, что если в части программы обнаружена ошибка, то велика вероятность существования в этой же части и другой ошибки. Другими словами, ошибки имеют тенденцию группироваться. При исправлении ошибки проверьте ее непосредственное окружение: нет ли здесь каких-нибудь подозрительных симптомов.

*Находите ошибку, а не ее симптом.*

Другим общим недостатком является устранение симптомов ошибки, а не ее самой. Если предполагаемое изменение устраняет не все симптомы ошибки, то она не может быть полностью выявлена.

*Вероятность правильного нахождения ошибки не равна 100%.*

С этим, безусловно, соглашаются, но в процессе исправления ошибки часто наблюдается иная реакция

(например, «да, в большинстве случаев это справедливо, но данная корректировка столь незначительна, что она правильна»). Никогда нельзя предполагать, что текст, который включен в программу для исправления ошибки, правилен. Можно утверждать, что корректировки более склонны к ошибкам, чем исходный текст программы. Подразумевается, что корректирующая программа должна тестироваться, возможно, даже более тщательно, чем исходная.

*Вероятность правильного нахождения ошибки уменьшается с увеличением объема программы.*

Это утверждение формулируется по-разному. Эксперименты показали, что отношение числа неправильно найденных ошибок к числу первоначально выявленных увеличивается для больших программ. В большой программе, рассчитанной на широкое применение, каждая шестая вновь обнаруженная ошибка может быть допущена при предшествующем внесении изменений в программу.

*Остерегайтесь внесения при корректировке новой ошибки.*

Необходимо рассматривать не только неверные корректировки, но и те, которые кажутся верными, однако имеют нежелательный побочный эффект и таким образом приводят к новым ошибкам. Другими словами, существует вероятность не только того, что ошибка будет обнаружена неверно, но и того, что ее исправление приведет к новой ошибке. Поэтому после проведения корректировки должно быть выполнено повторное регрессионное тестирование, позволяющее установить, не внесена ли новая ошибка.

*Процесс исправления ошибки должен временно возвращать разработчика на этап проектирования.*

Необходимо понимать, что исправление ошибок является одной из форм проектирования программы. Здравый смысл подсказывает нам, что все процедуры, методики и формализмы, использовавшиеся в процессе проектирования, должны применяться и для исправления ошибок, поскольку по своей природе корректировки склонны к ошибкам. Например, если при организации процесса проектирования предусматривались проверки исходного текста, то вдвойне важно, чтобы они использовались и после исправления ошибок.

*Изменяйте исходный текст, а не объектный код.*

При отладке больших систем, особенно написанных на языке Ассемблера, имеется тенденция исправлять ошибку путем внесения изменений непосредственно в объектный код (с помощью программы типа «*superczar*») с тем, чтобы изменить исходный текст программы в дальнейшем (т. е. «когда будет время»). Такой метод обычно является симптомом того, что применялась «отладка посредством экспериментирования». Кроме того, объектный код и исходный текст программы в этом случае не идентичны, следовательно, ошибка может появиться вновь при повторной компиляции или ассемблировании программы. Эта практика свидетельствует о непрофессиональном подходе к отладке.

## АНАЛИЗ ОШИБОК

Отладка программы дает возможность не только оценить стоимость устранения ошибки, но и получить такие сведения о природе ошибок в программном обеспечении, которыми мы до сих пор почти не располагали. Такая информация может служить обратной связью для дальнейшего улучшения процессов проектирования и тестирования.

Качество работы каждого отдельного программиста и всей организации, ведущей программирование, существенно повышается, если выполняется детальный анализ обнаруженных ошибок или по крайней мере их подмножества. Эта задача трудная и требующая больших временных затрат, поскольку она подразумевает нечто большее, чем просто поверхностная классификация, такая, как «*X%* ошибок являются ошибками логического проектирования» или «*Y%* ошибок встречается в операторах *IF*». Тщательный анализ может включать в себя рассмотрение следующих вопросов:

1. *Когда была сделана ошибка?* Данный вопрос является наиболее трудным, так как ответ на него требует исследования документации и истории проекта. Однако это и наиболее интересный вопрос. Необходимо точно определить первоначальную причину и время возникновения ошибки. Такой причиной может быть, например, неясная формулировка в спецификации, коррекция предшествующей ошибки или непонимание требований конечного пользователя.

2. *Кто сделал ошибку?* Небесполезно показать, что 60% ошибок проектирования допустил один из десяти аналитиков или что программист X сделал в три раза больше ошибок, чем другие программисты (не с целью наказания, а для разъяснения).

3. *Какова причина ошибки?* Недостаточно определить, когда и кем была сделана ошибка, нужно также выяснить, почему она произошла. Была ли она вызвана чьей-то неспособностью писать ясно, непониманием отдельных конструкций языка программирования, ошибкой при печатании на машинке или при набивке, неверным предположением, отсутствием рассмотрения недопустимых входных данных?

4. *Как ошибка могла быть предотвращена?* Что может быть сделано по-другому в следующем проекте, чтобы предотвратить этот тип ошибок? Ответ на этот вопрос наиболее ценен, так как позволяет осмыслить и количественно обосновать накапливаемый опыт проектирования.

5. *Почему ошибка не была обнаружена ранее?* Если ошибка была обнаружена на этапе тестирования, необходимо уточнить, почему ее не выявили на более ранних этапах: во время проверки исходного текста и при рассмотрении вопросов проектирования.

6. *Как ошибка могла быть определена ранее?* Ответ на этот вопрос является другим примером полезной обратной связи. Как могут быть улучшены процессы обзора и тестирования для более раннего нахождения этого типа ошибок в будущих проектах?

7. *Как была найдена ошибка?* При условии что мы не анализируем ошибки, найденные конечным пользователем (т. е. рассматриваем только те, которые обнаружены с помощью теста), необходимо выяснить, как был написан удачный тест. Почему этот тест был удачным? Можем ли мы что-нибудь почерпнуть из него для написания других удачных тестов с целью проверки данной программы или будущих программ?

Такой анализ, конечно, является сложным процессом, но его результаты могут оказаться полезными для дальнейшего улучшения работы программистов. Поэтому вызывает опасения тот факт, что подавляющее большинство программистов и организаций, занимающихся программированием, его не используют.

## ЛИТЕРАТУРА

1. Weinberg G. M. The Psychology of Computer Programming New York, Van Nostrand Reinhold, 1971.
2. Gould J. D. and Drongowski P. A Controlled Psychological Evidence on How People Debug Computer Programs. — Int. J. Man-Machine Stud., 1975, 7(2), p. 151—182.
3. Brown A. R. and Sampson W. A. Program Debugging. London, Macdonald, 1973.

## ЛИТЕРАТУРА, ДОБАВЛЕННАЯ ПРИ ПЕРЕВОДЕ

1. Липаев В. В. Проектирование математического обеспечения АСУ. Системотехника. Архитектура. Технология. М., Советское радио, 1977, с. 400.
2. Отладка систем управляющих алгоритмов ЦВМ реального времени/Под ред. В. В. Липаева. М., Советское радио, 1974, с. 328.
3. Липаев В. В. Надежность программного обеспечения АСУ. М., Энергоиздат, 1981, с. 240.
4. Ершов А. П. Введение в теоретическое программирование. М., Наука, 1977, с. 288.
5. Глушков В. М. Фундаментальные исследования и технология программирования. — Программирование, 1980, № 2, с. 3—13.
6. Головкии Б. А. Надежное программное обеспечение. — Зарубежная радиоэлектроника, 1978, № 12, с. 3—57.
7. Трахтенгерц Э. А. Программное обеспечение АСУ. М., Статистика, 1974, с. 288.
8. Пархоменко П. П., Правильщиков П. А. Диагностирование программного обеспечения. — Автоматика и телемеханика, 1980, № 1, с. 117—141.
9. Борзов Ю. В. Методы тестирования и отладки программ ЭВМ. Рига, ЛГУ им. П. Стучки, 1980, с. 88.
10. Тимофеев Б. Б., Козлик Г. А., Кулаков А. Ф., Мартынов А. И. Алгоритмизация в автоматизированных системах управления. Киев, Техника, 1972, с. 240.
11. Фуксман А. Л. Технологические аспекты создания программных систем. М., Статистика, 1979, с. 184.
12. Мамиконов А. Г., Цвиркун А. Д., Кульба В. В. Автоматизация проектирования АСУ. М., Энергоиздат, 1981, с. 328.
13. Айзенберг Я. Е., Кенорев Б. М. и др. Автоматизированная система производства программ СИНТЕРМ. — В кн.: Технология программирования. Киев, ИК АН УССР, 1977, с. 57—64.
14. Колганова Т. В. Критерии планирования тестирования структурной проверки управляющих программ. Технология программирования. Технология отладки программ. Киев, ИК АН УССР, 1979, с. 18—19.

15. Бичевский Я. Я. Автоматическое построение систем примеров. — Программирование, 1977, № 3, с. 60—70.
16. Позин Б. А. Метод структурного построения тестов для отладки управляющих программ. — Программирование, 1980, № 2, с. 62—69.
17. Единая система программной документации (ЕСПД). ГОСТ 19.101—77 — 19.506—79.
18. Майерс Г. Надежность программного обеспечения. М., Мир, 1980, с. 360.
19. Исдан Э. Структурное программирование и конструирование программ. М., Мир, 1979, с. 416.
20. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. М., Мир, 1980, с. 278.

## ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Анализ граничных значений	68	Модуль-драйвер	108
— ошибок	170	— -заглушка	108
Библиотеки тестов	144	Надежность	18
Внесение изменений в программы при тестировании	169	Независимые агентства по тестированию	152
Внешняя спецификация	125	Ненципированная переменная	38
Дамп памяти	156	Неправильное входное условие	131
Инспекции исходного текста	35	«Обезличенное» программирование	154
Информационные сообщения	47	Отладка методом тестирования	166
Класс эквивалентности	63	— прослеживанием логики в обратном порядке	166
Комбинаторное покрытие условий	61	Ошибки в бите или строке	39
Комплексирование	145	— ввода-вывода	46
Концептуальная целостность	137	— в передачах управления	44
Критерий покрытия	56	— интерфейса	45
— завершения проверки	146	— обращения к данным	38
Математическое доказательство корректности программ	32	— , проявление которых зависит от отрабатываемых данных	25
Метод «большого уда-ра»	108	— при сравнениях	43
— дедукции	161	— проектирования	129
— индукции	158	Оценка посредством просмотра	52
Методы «грубой силы»	155		
Модели надежности	139		

План отладки	145	— защиты	137
— тестирования	144	— конверсии	138
Планирование тести-		— конфигураций	
рования	143	оборудования	138
Покрывание операторов	56(57)	— модулей	96
— решений	57	— надежности	139
— — условий	58	— на предельных	
Пошаговое тестиро-		нагрузках	134
вание	107	— на предельных	
Предположение об		объемах	134
ошибке	92	— нисходящее	112
Председатель (при		— правильности	
инспекции)	35	установки	143
Предупреждения	47	— программы как	
Принципы локализа-		белого ящика	23
ции ошибок	167	— программы как	
— отладки	167	черного ящика	21
— тестирования	25	— производитель-	
Приемосдаточные		ности	137
испытания	128,142	— процедур	140
Причина	76	— путем покрытия	
Проверка за столом	51	логики программы	55
— результатов		— регрессионное	145
сборки (комплекс-		— системы	131
сирования)	128(130)	— требований к	
Проект структуры		памяти	138
программы	125	— со случайным	
Проектирование си-		входом (стохастиче-	
стемы	125	ское)	54
— тестов	54	— удобства ис-	
Психология тестиро-		пользования	134
вания	25	— удобства об-	
Распечатка состоя-		служивания	140
ния памяти	156	— удобства уста-	
Сквозные просмотры	49	новки	139
Склонные к ошибкам		— удобства эк-	
части программы	38	сплуатации	136
Следствие	76	— функций	130
Спецификация интер-		Точки прерывания	157
фейса модуля	104	Требования к про-	
Список вопросов для		граммному продук-	
выявления ошибок		ту	125
при инспекции	38	Фазы тестирования	129
Стандарты написания		Функциональная ди-	
тестов	144	аграмма	75
Таблица решений	76	Цели	125
Тест удачный	18	Чувствительность пу-	
— неудачный	18	ти	85
Тестирование (опре-		Эквивалентное раз-	
деление)	18	биение	64
— восстановления	139	Экономика тестиро-	
— восходящее	118	вания	21
— документации	140		

## ОГЛАВЛЕНИЕ

Предисловие к русскому изданию . . . . .	5
Предисловие . . . . .	13
Глава 1. ТЕСТ ДЛЯ САМООЦЕНКИ . . . . .	15
Глава 2. ПСИХОЛОГИЯ И ЭКОНОМИКА ТЕСТИРОВАНИЯ ПРОГРАММ . . . . .	17
Экономика тестирования . . . . .	21
Принципы тестирования . . . . .	25
Литература . . . . .	31
Глава 3. ИНСПЕКЦИИ, СКВОЗНЫЕ ПРОСМОТРЫ И ОБЗОРЫ ПРОГРАММЫ . . . . .	32
Инспекции и сквозные просмотры . . . . .	33
Инспекции исходного текста . . . . .	35
Список вопросов для выявления ошибок при инспекции . . . . .	38
Сквозные просмотры . . . . .	49
Проверка за столом . . . . .	51
Оценка посредством просмотра . . . . .	52
Литература . . . . .	53
Глава 4. ПРОЕКТИРОВАНИЕ ТЕСТА . . . . .	54
Тестирование путем покрытия логики программы . . . . .	55
Эквивалентное разбиение . . . . .	63
Анализ граничных значений . . . . .	68
Применение функциональных диаграмм . . . . .	75
Предположение об ошибке . . . . .	92
Стратегия . . . . .	94
Литература . . . . .	95
Глава 5. ТЕСТИРОВАНИЕ МОДУЛЕЙ . . . . .	96
Проектирование тестов . . . . .	97
Пошаговое тестирование . . . . .	107
Нисходящее и восходящее тестирование . . . . .	111
Исполнение теста . . . . .	122
Литература . . . . .	123
Глава 6. ТЕСТИРОВАНИЕ КОМПЛЕКСОВ ПРОГРАММ . . . . .	124
Тестирование функций . . . . .	130
Тестирование системы . . . . .	131
Приемо-сдаточные испытания . . . . .	142

Тестирование правильности установки . . . . .	143
Планирование тестирования и контроль . . . . .	143
Критерий завершения проверки . . . . .	146
Независимые агентства по тестированию . . . . .	152
Литература . . . . .	153
<b>Глава 7. ОТЛАДКА . . . . .</b>	<b>154</b>
Методы «грубой силы» . . . . .	155
Метод индукции . . . . .	158
Метод дедукции . . . . .	161
Прослеживание логики в обратном порядке . . . . .	166
Метод тестирования . . . . .	166
Принципы отладки . . . . .	167
Анализ ошибок . . . . .	170
Литература . . . . .	172
Литература, добавленная при переводе . . . . .	172
<b>Предметный указатель . . . . .</b>	<b>173</b>

**Г. Майерс**

## **ИСКУССТВО ТЕСТИРОВАНИЯ ПРОГРАММ**

Книга одобрена на заседании секции редсовета по электронной обработке данных в экономике 28/IV 1980 г.

Зав. редакцией *А. В. Павлюков*

Редактор *Н. К. Логинова*

Мл. редактор *О. Б. Степанченко*

Техн. редактор *Р. Н. Феоктистова*

Корректоры *Г. В. Хлопцева, Т. М. Иванова*

Худож. редактор *О. Н. Поленова*

Обложка художника *Л. Г. Прохорова*

ИБ № 1106

Сдано в набор 26.11.81.

Подписано в печать 17.05.82.

Формат 84×108<sup>1</sup>/<sub>32</sub>. Бум. тип. № 1. Гарнитура «Литературная».

Печать высокая. П. л. 5,5. Усл. п. л. 9,24. Усл. кр.-отт. 9,555.

Уч.-изд. л. 9,26. Тираж 30000 экз. Заказ 3848. Цена 60 коп.

Издательство «Финансы и статистика», Москва,  
ул. Чернышевского, 7

Великолукская городская типография управления издательств,  
полиграфии и книжной торговли Псковского облисполкома,  
г. Великие Луки, ул. Полиграфистов, 78/12

63 коп.

## ФИНАНСЫ И СТАТИСТИКА