Введение

В 1979 году, когда эта книга была впервые опубликована, действовало известное эмпирическое правило: примерно 50% времени и более 50% общей стоимости типичного программного проекта тратится на тестирование разрабатываемой системы или программы.

С тех пор прошло более тридцати лет, вышло два новых издания книги, но указанное правило действует и поныне. Появились новые системы разработки программного обеспечения, языки со встроенным инструментарием тестирования и программисты, владеющие быстрыми методиками разработки. Однако тестирование по-прежнему остается важной частью любого проекта по разработке ПО.

Учитывая вышесказанное, можно было бы рассчитывать, что к настоящему времени тестирование программного обеспечения поднимется до уровня точной науки. Увы, но это далеко не так. В действительности данный аспект разработки ПО является, вероятно, наименее изученным по сравнению с другими. Более того, тестирование считается непопулярным предметом. Так было на момент выхода первого издания данной книги, и, к сожалению, аналогичная ситуация наблюдается до сих пор. Правда, сегодня книг и статей по тестированию программного обеспечения выпускается гораздо больше. Таким образом, хотя бы делаются попытки осветить данную тему лучше, чем тридцать лет назад, но тестирование по-прежнему остается одной из областей разработки ПО, требующих применения “черной магии”.

Уже одно это могло стать серьезным поводом для обновления книги, однако у нас имелась и другая мотивация. От профессорского и преподавательского состава нам не раз приходилось слышать: “Впервые попадая в производственную среду, наши выпускники даже не знают толком, как приступить к тестированию программного продукта, а наши вводные курсы почти не содержат практических рекомендаций, которыми студенты могли бы воспользоваться при тестировании и отладке своих учебных программ”.

Таким образом, обновленное издание книги преследует те же цели, что и издания 1979 и 2004 годов: восполнить пугающий пробел в знаниях профессиональных программистов и студентов, специализирующихся в области вычислительной техники. Как следует из названия самой книги, она ориентирована не на теоретическую, а на практическую сторону дела и дополнена обсуждением новых языков и технологий.

Несмотря на принципиальную возможность рассматривать вопросы тестирования ПО в теоретической плоскости, данная книга нацелена на рассмотрение сугубо практических, прикладных аспектов. Поэтому многие темы, связанные с тестированием программного обеспечения, такие, например, как попытка математического доказательства корректности программы, нами были сознательно опущены.

В главе 1 предлагается небольшой тест для самопроверки, который рекомендуется пройти каждому, кто собирается читать книгу дальше. Оказывается, наиболее важную практическую информацию о тестировании программ, которой вы должны владеть, можно представить в виде нескольких философских и экономических принципов; их рассмотрению посвящена глава 2. Глава 3 ознакомит читателей с важными концепциями сквозного просмотра и инспекции кода — процедур проверки, не требующих использования компьютера. Вместо того чтобы делать акцент на собственно процедурных или управленческих аспектах этих процессов, что характерно для большинства книг по данной теме, в этой главе основное внимание уделяется рассмотрению технических деталей с точки зрения того, как искать ошибки.

Внимательный читатель сразу поймет, что самое важное для тестировщика — знать, как создаются эффективные тесты. Эта тема рассматривается в главе 4. В главе 5 обсуждаются методы тестирования отдельных модулей и подпрограмм, а в главе 6 изучается тестирование более крупных программных блоков. В главе 7 описывается тестирование удобства использования, или пользовательского тестирования. Эта часть процесса тестирования всегда была важна, но в наши дни ее роль существенно возросла, поскольку количество приложений, ориентированных на массового пользователя, значительно увеличилось. В главе 8 даются практические рекомендации, касающиеся отладки программ, а в главе 9 описываются основные концепции тестирования в контексте гибкой разработки и экстремального программирования. В главе 10 демонстрируется применение средств автоматизированного тестирования ПО при разработке веб-приложений, в том числе систем электронной коммерции. В главе 11 объясняется, как тестировать программное обеспечение для мобильных устройств.

Книга рассчитана на три основные категории читателей. Во-первых, это профессиональные программисты. Пусть многое здесь будет для них знакомым, но мы все же считаем, что книга поможет профессионалам расширить свои знания в области тестирования. Даже если почерпнутые программистом сведения помогут ему дополнительно обнаружить всего одну ошибку в какой-либо программе, то деньги, потраченные на покупку книги, окупятся сторицей.

Ко второй категории читателей относятся менеджеры проектов, которым будет полезна приведенная в книге информация практического характера, касающаяся управления процессом тестирования. Наконец, третья категория — это студенты, изучающие программирование и вычислительную технику. В их отношении мы преследуем двоякую цель: продемонстрировать, с какими проблемами им придется столкнуться при тестировании программного обеспечения, и представить набор эффективных методик тестирования. Читателям третьей категории мы рекомендуем использовать данную книгу как дополнение к изучаемым ими курсам программирования, чтобы они имели возможность ознакомиться с дисциплиной тестирования программного обеспечения на ранней стадии обучения.

Предисловие

Книга Гленфорда Майерса *Искусство тестирования программ*, впервые выпущенная в 1979 году, давно стала классикой. Она выдержала испытание временем, в течение 25 лет не покидая прайс-лист издательства Wiley. Уже одно это говорит о том, насколько основателен, востребован и ценен предлагаемый вашему вниманию труд.

С тех пор авторы настоящего (третьего) издания выпустили в общей сложности свыше 200 книг, бо́льшая часть из которых посвящена программному обеспечению. Некоторые из них пользовались широкой популярностью и, как и данная книга, выдержали несколько переизданий. Например, книга Кори Сандлера *Ремонт персонального компьютера* издавалась восемь раз; несколько раз переиздавались также книги Тома Баджетта, посвященные PowerPoint и другим приложениям Microsoft Office. Однако ни одной из этих книг, в отличие от книги Майерса, не удавалось сохранять свою актуальность в течение столь долгого времени.

Чем это объяснить? Новые книги охватывали темы, сильнее подверженные устареванию: операционные системы, прикладные программы, средства безопасности, телекоммуникационные и аппаратные технологии. Темпы обновления компьютерного оборудования и программных технологий на протяжении 1980–1990-х годов были настолько высоки, что информация по этим темам устаревала слишком быстро.

За аналогичный период были опубликованы также сотни книг по тестированию программного обеспечения. Однако все они в большей степени касались частных вопросов, актуальность которых со временем ослабевала. И лишь книга *Искусство тестирования программ* до сих пор остается актуальным фундаментальным руководством по тестированию ПО, дающим ответ на один из самых главных вопросов компьютерной индустрии: как добиться того, чтобы создаваемое программное обеспечение делало то, что должно делать, и при этом, что не менее важно, не делало того, чего делать не должно?

Издание, которое вы сейчас читаете, по-прежнему базируется на основополагающих философских концепциях, выдвинутых Гленфордом Майерсом более трех десятилетий лет тому назад. Вместе с тем мы обновили все примеры, переписав их с использованием более распространенных в настоящее время языков программирования, и дополнили книгу рассмотрением вопросов, которые при подготовке первого издания вообще не стояли на повестке дня: веб-программирование, электронная коммерция, экстремальное (гибкое) программирование и тестирование, а также тестирование приложений для мобильных устройств.

Работая над новым изданием книги, мы старались следовать духу предыдущих изданий и описывать процесс тестирования, в равной степени пригодный как для существующих программных и аппаратных платформ, так и для тех, которые могут появиться в ближайшем будущем. Надеемся, третье издание, как и предыдущие, станет хорошим подспорьем для целого поколения проектировщиков и разработчиков программного обеспечения.

5 Модульное (блочное) тестирование

До сих пор мы в основном игнорировали техническую сторону тестирования и не делали никаких оговорок в отношении размера тестируемых программ. Однако тестирование крупных программ (содержащих, например, свыше 500 инструкций или более 50 классов) требует особого подхода, и поэтому в данной главе мы рассмотрим начальный этап структурированного процесса такого тестирования — *модульное (блочное) тестирование* (module testing, unit testing). Последующие этапы рассматриваются в главах 6 и 7.

Модульное тестирование — это процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу. Другими словами, прежде чем тестировать программу в целом, необходимо сосредоточить внимание на ее меньших по размеру компонентах. На то есть три причины. Во-первых, при таком подходе повышается эффективность тестирования сложных объектов, поскольку на первом этапе внимание фокусируется на небольших программных блоках, которые легче тестировать. Во-вторых, при таком подходе облегчается отладка программ (точная локализация и исправление обнаруженной ошибки), поскольку, обнаружив ошибку, мы всегда точно знаем, в каком именно модуле она содержится. Наконец, модульное тестирование позволяет распараллеливать процесс тестирования, что обеспечивает возможность одновременного тестирования нескольких модулей.

Цель модульного тестирования — сравнение функций, реализуемых модулем, со спецификациями, описывающими его функциональные или интерфейсные характеристики. Вновь подчеркнем, что под этим подразумевается не доказательство соответствия модуля требованиям спецификации,

а демонстрация того, что поведение модуля отличается от того, которое специфицировано. В данной главе рассматриваются три аспекта модульного тестирования:

- методики проектирования тестов;
- порядок тестирования и интеграции модулей;
- рекомендуемые правила выполнения тестов.

Проектирование тестов

При проектировании модульных тестов используются два источника информации: спецификация модуля и его исходный код. Типичная спецификация описывает назначение модуля, а также его входные и выходные параметры.

Модульное тестирование в основном ориентировано на использование метода "белого ящика". Объясняется это прежде всего тем, что при последующем переходе к тестированию более крупных программных единиц, например программ в целом, применимость метода "белого ящика" снижается. Кроме того, последующие этапы процесса тестирования ориентированы на обнаружение ошибок другого типа (не обязательно связанных с программной логикой, а обусловленных, например, несоответствием программы ожиданиям пользователей). Таким образом, процедура проектирования тестов для модульного тестирования состоит в следующем.

> Используя одну или несколько методик тестирования по принципу "белого ящика", анализируем логику модуля, а затем проектируем дополнительные тесты путем применения методик тестирования по принципу "черного ящика" к спецификации модуля.

Методы проектирования тестов, которые мы будем использовать, были описаны в главе 4. Их применение к модульным тестам будет проиллюстрировано на конкретном примере.

Предположим, требуется протестировать модуль `BONUS`, функцией которого является увеличение на $200 размера заработной платы всем сотрудникам отдела или отделов, обеспечивших реализацию товара на наибольшую сумму. Впрочем, если текущая заработная плата такого сотрудника превышает $15 000 или он является менеджером, то надбавка к его зарплате уменьшается до $100.

Входные данные модуля представлены на рис. 5.1. При корректной работе модуль возвращает код ошибки, равный 0. В случае отсутствия записей в

таблице сотрудников или таблице отделов возвращается код ошибки, равный 1. Если среди сотрудников не удается найти тех, кто работает в лидирующем отделе, то возвращается код ошибки, равный 2.

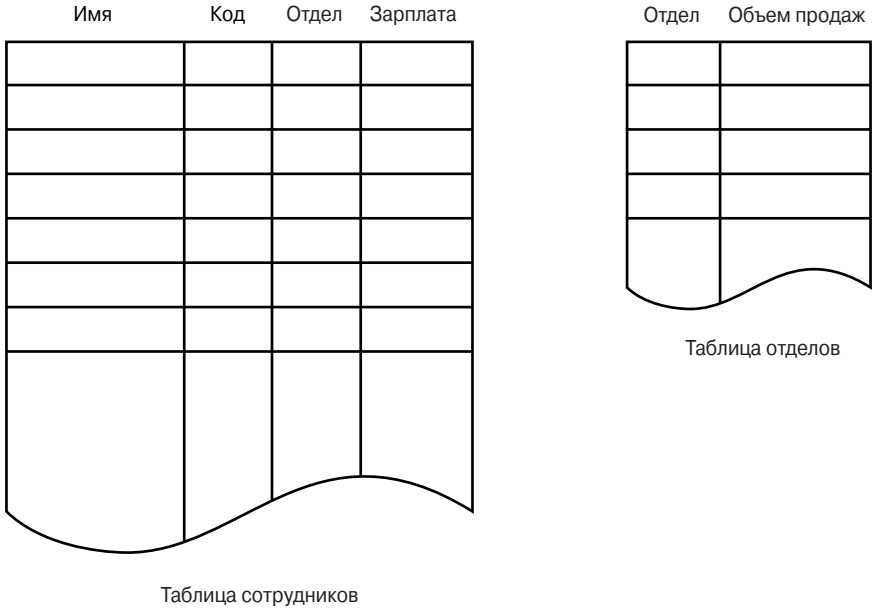

Рис. 5.1. Таблицы входных данных для модуля BONUS

Исходный код модуля BONUS приведен в листинге 5.1. Входные параметры ESIZE и DSIZE задают количество записей в таблицах сотрудников и отделов соответственно. В данном случае для написания модуля использован язык PL/1, но последующее обсуждение в значительной мере не зависит от выбранного языка, и описанные методики в равной степени применимы к программам, написанным на других языках. Логика модуля довольно проста и будет понятна даже тем читателям, которые не знакомы с языком PL/1.

Листинг 5.1. Исходный код модуля BONUS

```
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); /*МАКСИМАЛЬНАЯ СУММА
                 ПРОДАЖ, ОПРЕДЕЛЯЕМАЯ ПО ДАННЫМ ТАБЛИЦЫ DEPTTAB*/
DECLARE (I,J,K) FIXED BINARY;                          /*СЧЕТЧИКИ*/
DECLARE FOUND BIT(1);    /*TRUE, ЕСЛИ В ЛИДИРУЮЩЕМ ОТДЕЛЕ ЧИСЛЯТСЯ
                                                       СОТРУДНИКИ*/
DECLARE SINC FIXED DECIMAL(7,2) INIT(200.00);       /*СТАНДАРТНАЯ
                                                         НАДБАВКА*/
DECLARE LINC FIXED DECIMAL(7,2) INIT(100.00);       /*УМЕНЬШЕННАЯ
                                                         НАДБАВКА*/
DECLARE LSALARY FIXED DECIMAL(7,2) INIT(15000.00);    /*ГРАНИЧНАЯ
                                                         ЗАРПЛАТА*/
DECLARE MGR CHAR(1) INIT('M');
1 ERRCODE=0;
2 IF(ESIZE<=0)|(DSIZE<=0)
3    THEN ERRCODE=1;         /*ПУСТАЯ ТАБЛИЦА EMPTAB ИЛИ DEPTTAB*/
4    ELSE DO;
5       DO I = 1 TO DSIZE;           /*НАЙТИ MAXSALES И MAXDEPTS*/
6          IF(SALES(I)>=MAXSALES) THEN MAXSALES=SALES(I);
7       END;
8       DO J = 1 TO DSIZE;
9          IF(SALES(J)=MAXSALES)              /*ЛИДИРУЮЩИЙ ОТДЕЛ*/
10            THEN DO;
11               FOUND='0'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;
```

ВРЕЗКА 5.1. ОБЩИЕ СВЕДЕНИЯ О PL/1

Возможно, читатели, не имеющие большого опыта в разработке программного обеспечения, не знакомы с PL/1 и считают его "мертвым" языком программирования. Действительно, новые продукты, в которых используется язык PL/1, в настоящее время являются редкостью, однако сопровождение существующих систем продолжается, а языковые конструкции PL/1 весьма удобны для изучения программных процедур.

Язык программирования PL/1 (Programming Language One — язык программирования номер один) был создан в 1960-х годах фирмой IBM как часть среды разработки, ориентированной на использование элементов естественного английского языка и предназначенной для машин класса мэйнфреймов, начиная с IBM System/360. В тот период компьютерной истории многие программисты переходили к использованию специализированных языков, таких как Кобол, который был предназначен для разработки финансовых приложений, и Фортран, который был ориентирован на выполнение научных расчетов. (Краткая информация об этих языках программирования приводилась в главе 3.)

Одной из задач, стоявших перед проектировщиками PL/1, была разработка языка, который мог бы успешно конкурировать с Коболом и Фортраном и в то же время предоставлял среду разработки, изучение которой было бы упрощено за счет использования языковых конструкций, максимально приближенных к естественному языку. Похоже, ни одна из первоначальных целей так и не была достигнута, однако не вызывает сомнений, что первые проектировщики проделали неплохую работу, поскольку с годами язык PL/1 совершенствовался и обновлялся и по-прежнему используется в некоторых современных вычислительных средах.

К середине 1990-х годов PL/1 был перенесен на другие компьютерные платформы, включая OS/2, Linux, UNIX и Windows. Поддержка со стороны новых операционных системам обеспечила гибкость расширений PL/1 и обогатила их новыми функциональными возможностями.

Независимо от того, какую методику покрытия логики вы будете использовать, первый шаг заключается в составлении списка всех точек ветвления в программе. Кандидатами на эту роль в данной программе являются все инструкции `IF` и `DO`. Анализируя текст программы, можно убедиться в том, что все инструкции `DO` представляют простые итерации, в которых конечное значение параметра цикла превышает начальное значение или равно ему (откуда следует, что тело цикла будет выполняться не менее одного раза). Кроме того, единственный способ выхода из каждого цикла обеспечивается самой инструкцией `DO`. Таким образом, в данной программе инструкции `DO` не требуют особого внимания, поскольку любой тест, приводящий к выполнению такой инструкции, в конечном счете выполнит ветвление в обоих направлениях (вход в тело цикла и пропуск тела цикла). Следовательно, мы должны проанализировать лишь следующие инструкции.

```
2 IF (ESIZE<=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) THEN ERRCODE=2
```

Учитывая, что количество точек ветвления невелико, достаточно было бы ограничиться критерием комбинаторного покрытия условий, однако ради иллюстрации различных возможностей мы исследуем все критерии покрытия логики (кроме критерия покрытия операторов, который, в силу его ограниченности, редко приносит ощутимую пользу).

Чтобы удовлетворить критерию покрытия ветвлений, необходимо построить тесты, охватывающие каждый из двух возможных булевых результатов (истинный и ложный) вычисления логических выражений в шести условных операторах. Ситуации, которые для этого требуется смоделировать

с помощью входных данных, перечислены в табл. 5.1. Поскольку две из двенадцати ветвей в любом случае будут выполнены хотя бы по одному разу, остается 10 ситуаций, которые требуется принудительно создать тестовыми вариантами. Заметим, что для определения соответствующих входных условий при создании табл. 5.1 потребовалось отслеживать логику программы от точек ветвления в обратном направлении. Например, условный оператор 16 активизируется не любыми сотрудниками, удовлетворяющими условиям, а только теми, кто числится в лидирующем отделе.

Таблица 5.1. Ситуации, соответствующие двум исходам решений

Решение	Результат *true*	Результат *false*
2	`ESIZE` или `DSIZE<=0`	`ESIZE` или `DSIZE>0`
6	Случится не менее одного раза	Записи в таблице `DEPTTAB` упорядочиваются таким образом, чтобы отдел с меньшей суммой продаж следовал за отделом с большей суммой продаж
9	Случится не менее одного раза	У всех отделов разные суммы продаж
13	Сотрудник числится в лидирующем отделе	Сотрудник не числится в лидирующем отделе
16	Сотрудник лидирующего отдела является менеджером или имеет заработную плату не ниже `LSALARY`	Сотрудник лидирующего отдела не является менеджером и имеет заработную плату ниже `LSALARY`
21	Не найдены сотрудники, числящиеся в лидирующих разделах	Найден хотя бы один сотрудник, числящийся в одном из лидирующих разделов

Все десять интересующих нас ситуаций, приведенных в табл. 5.1, могут быть активизированы двумя тестами, представленными на рис. 5.2. Обратите внимание на то, что каждый тест включает в себя определение ожидаемых выходных значений, что соответствует принципам, рассмотренным в главе 2.

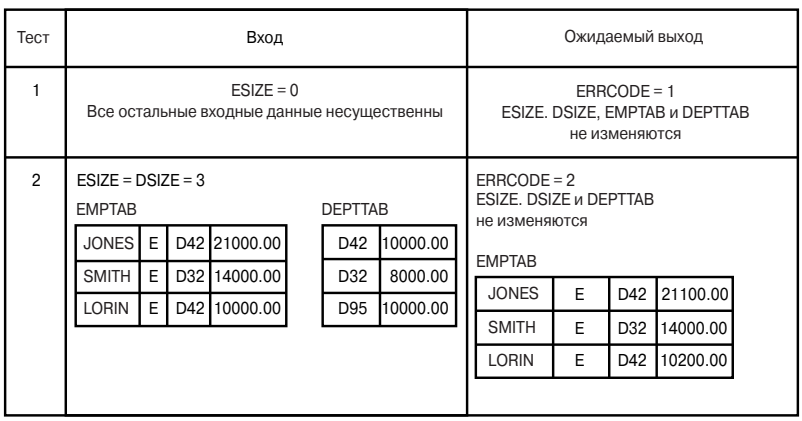

Рис. 5.2. Тесты, удовлетворяющие критерию покрытия решений

Оба теста удовлетворяют критерию покрытия решений. Однако очевидно, что в модуле могут оставаться ошибки многих типов, которые эти тесты не в состоянии обнаружить. Например, они не исследуют ситуации, в которых код ошибки равен 0, сотрудник является менеджером или таблица отделов является пустой (`DSIZE<=0`).

Использование критерия покрытия условий позволяет получить более приемлемый тест. В этом случае нам потребуется такое количество тестов, которое обеспечит активизацию обеих ветвей в каждой точке ветвления. Необходимые для этого условия и ситуации на входе перечислены в табл. 5.2. Поскольку две ветви будут всегда выполняться, необходимо создать тесты, обеспечивающие принудительное создание 14 ситуаций. Опять-таки для активизации этих ситуаций нам потребуются всего лишь два теста, показанные на рис. 5.3.

Таблица 5.2. Ситуации, соответствующие результатам вычисления условий

Решение	Условие	Результат *true*	Результат *false*
2	`ESIZE<=0`	`ESIZE<=0`	`ESIZE>0`
2	`DSIZE<=0`	`DSIZE<=0`	`ESIZE>0`
6	`SALES(I)>=MAXSALES`	Случится не менее одного раза	Записи в таблице `DEPTTAB` упорядочиваются таким образом, чтобы отдел с меньшей суммой продаж следовал за отделом с большей суммой продаж
9	`SALES(J)=MAXSALES`	Случится не менее одного раза	У всех отделов разные суммы продаж
13	`EMPTAB.DEPT(K)=DEPTTAB.DEPT(J)`	Сотрудник числится в лидирующем отделе	Сотрудник не числится в лидирующем отделе
16	`SALARY(K)>=LSALARY`	Сотрудник лидирующего отдела имеет заработную плату не ниже `LSALARY`	Сотрудник лидирующего отдела имеет заработную плату ниже `LSALARY`
16	`CODE(K)=MGR`	Сотрудник лидирующего отдела является менеджером	Сотрудник лидирующего отдела не является менеджером
21	`-FOUND`	Не найдены сотрудники, числящиеся в лидирующих разделах	Найден хотя бы один сотрудник, числящийся в одном из лидирующих разделов

Эти тесты были спроектированы так, чтобы проиллюстрировать одну проблему. Поскольку они активизируют все возможные результаты принятия решений, перечисленные в табл. 5.2, то удовлетворяют критерию покрытия

условий, однако, по-видимому, уступают набору тестов, представленных на рис. 5.2, в плане удовлетворения критерию покрытия решений. Причина состоит в том, что данные тесты не обеспечивают выполнения всех без исключения инструкций. Например, инструкция 18 в них никогда не выполняется. Более того, они не в состоянии справиться с гораздо бо́льшим количеством случаев, чем тесты, представленные на рис. 5.2. Они не приводят к созданию на выходе ситуации, в которой `ERRCODE=0`. Если предположить, что в результате неправильной записи инструкции 2 будут ошибочно устанавливаться значения переменных `ESIZE=0` и `DSIZE=0`, то эта ошибка не будет обнаружена. Разумеется, указанные проблемы можно было бы решить, используя другой возможный тестовый набор, но от того факта, что тесты, приведенные на рис. 5.3, удовлетворяют критерию покрытия условий, никуда не денешься.

<table>
<tr><th>Тест</th><th>Вход</th><th>Ожидаемый выход</th></tr>
<tr><td>1</td><td>ESIZE = DSIZE = 0
Все остальные входные данные несущественны</td><td>ERRCODE = 1
ESIZE. DSIZE, EMPTAB и DEPTTAB
не изменяются</td></tr>
<tr><td>2</td><td>ESIZE = DSIZE = 3
EMPTAB
<table>
<tr><td>JONES</td><td>E</td><td>D42</td><td>21000.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14000.00</td></tr>
<tr><td>LORIN</td><td>M</td><td>D42</td><td>10000.00</td></tr>
</table>
DEPTTAB
<table>
<tr><td>D42</td><td>10000.00</td></tr>
<tr><td>D32</td><td>8000.00</td></tr>
<tr><td>D95</td><td>10000.00</td></tr>
</table></td><td>ERRCODE = 2
ESIZE. DSIZE и DEPTTAB
не изменяются
EMPTAB
<table>
<tr><td>JONES</td><td>E</td><td>D42</td><td>21100.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14000.00</td></tr>
<tr><td>LORIN</td><td>M</td><td>D42</td><td>10100.00</td></tr>
</table></td></tr>
</table>

Рис. 5.3. Тесты, удовлетворяющие критерию покрытия условий

Использование критерия покрытия решений и условий позволило бы устранить слабые места тестов, приведенных на рис. 5.3. В этом случае нам следовало бы составить тесты в таком количестве, чтобы каждая ветвь условного оператора и каждое значение условия активизировались не менее одного раза. Этого можно добиться, присвоив Джонсу статус менеджера, а Лорин — статус обычного сотрудника. В результате были бы активизированы оба возможных исхода решения 16, что привело бы к выполнению инструкции 18.

И все же этот тестовый набор, по сути, ничем не лучше набора тестов, представленного на рис. 5.2. Если используемый компилятор прекращает оценку выражения, содержащего операцию `or`, сразу же после того, как выясняется, что один из операндов имеет значение `true`, то выражение

`CODE(K)=MGR` в инструкции 16 никогда не примет значение `true`. Следовательно, если это выражение закодировано неверно, то данный набор тестов не сможет обнаружить ошибку.

Сейчас нам осталось исследовать лишь критерий комбинаторного покрытия условий. В соответствии с этим критерием количество тестов должно быть таким, чтобы каждая из возможных комбинаций условий в каждом решении активизировалась не менее одного раза. Для составления нужного набора тестов воспользуемся табл. 5.2. Решения 6, 9, 13 и 21 содержат по две комбинации условий каждое, а решения 2 и 16 — по четыре. Согласно рассматриваемой методологии построения тестов сначала выбирается один тест, который покрывает как можно большее количество комбинаций, затем выбирается другой тест, покрывающий максимально возможное количество оставшихся комбинаций, и т.д. Набор тестов, удовлетворяющий критерию комбинаторного покрытия условий, представлен на рис. 5.4. Этот набор отличается наибольшей полнотой по сравнению со всеми ранее рассмотренными тестами, откуда следует, что именно данный критерий и нужно было выбрать с самого начала.

<table>
<tr><th>Тест</th><th>Вход</th><th>Ожидаемый выход</th></tr>
<tr><td>1</td><td>ESIZE = DSIZE = 0
Все остальные входные данные несущественны</td><td>ERRCODE = 1
ESIZE. DSIZE, EMPTAB и DEPTTAB
не изменяются</td></tr>
<tr><td>2</td><td>ESIZE = 0 DSIZE > 0
Все остальные входные данные несущественны</td><td>То же</td></tr>
<tr><td>3</td><td>ESIZE > 0 DSIZE = 0
Все остальные входные данные несущественны</td><td>То же</td></tr>
<tr><td>4</td><td>ESIZE = 5 DSIZE = 4
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21000.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12000.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10000.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16000.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14000.00</td></tr>
</table>
DEPTTAB
<table>
<tr><td>D42</td><td>10000.00</td></tr>
<tr><td>D32</td><td>8000.00</td></tr>
<tr><td>D95</td><td>10000.00</td></tr>
<tr><td>D44</td><td>10000.00</td></tr>
</table></td><td>ERRCODE = 2
ESIZE. DSIZE и DEPTTAB
не изменяются
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21100.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12100.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10200.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16100.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14000.00</td></tr>
</table></td></tr>
</table>

Рис. 5.4. Тесты, удовлетворяющие критерию комбинаторного покрытия условий

Важно понимать, что модуль BONUS может содержать так много ошибок, что обнаружить их все не удастся даже тестами, удовлетворяющими критерию комбинаторного покрытия условий. Например, ни один из тестов не сможет воспроизвести ситуацию, когда параметр ERRCODE возвращается со значением 0. Таким образом, если пропущена инструкция 1, то эта ошибка останется необнаруженной. Если бы константа LSALARY была ошибочно инициализирована значением $15 000.01, то эта ошибка также осталась бы незамеченной. Точно так же, если бы оператор 16 был записан как SALARY(K)>LSALARY вместо SALARY(K)>=LSALARY, то эта ошибка не была бы обнаружена. Кроме того, обнаружение одной из возможных ошибок "смещение на единицу", т.е. ошибок диапазона (например, некорректная обработка последней записи таблицы DEPTTAB или EMPTAB), фактически оставалось бы делом случая.

Все это отчетливо демонстрирует два важных момента. Во-первых, критерий комбинированного покрытия условий по своей эффективности значительно превосходит остальные критерии, а во-вторых, ни один из критериев покрытия логики не может использоваться в качестве единственного метода при написании модульных тестов. Поэтому следующее, что мы должны сделать, — это дополнить тесты, представленные на рис. 5.4, набором тестов, построенных по принципу "черного ящика". С этой целью обратимся к приведенной ниже спецификации интерфейса модуля BONUS.

Модуль BONUS написан на языке PL/1 и принимает пять аргументов с символическими именами EMPTAB, DEPTTAB, ESIZE, DSIZE и ERRCODE, которые имеют следующие атрибуты.

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

Предполагается, что передаваемые модулю аргументы имеют указанные атрибуты. Аргументы ESIZE и DSIZE задают количество записей в таблицах EMPTAB и DEPTTAB соответственно. Никаких предположений относительно порядка следования записей в этих таблицах не делается. Функциональным

назначением модуля является увеличение заработной платы (EMPTAB.SALARY) сотрудникам отделов, обеспечивших наибольшую выручку от продажи товаров (DEPTTAB.SALES). Если текущая заработная плата сотрудника лидирующего отдела составляет $15 000 и выше или сотрудник является менеджером (EMPTAB.CODE='M'), размер надбавки составляет $100, в противном случае — $200. Увеличенная заработная плата записывается в поле EMPTAB.SALARY. Если значение ESIZE или DSIZE не превышает 0, то коду ошибки ERRCODE присваивается значение 1 и никакие дальнейшие действия не предпринимаются. Во всех остальных случаях функция модуля полностью выполняется. Но если оказывается, что ни один из представленных сотрудников не числится в лидирующем отделе, обработка данных продолжается и переменной ERRCODE присваивается значение 2. Во всех остальных случаях значение ERRCODE устанавливается равным 0.

Эту спецификацию нельзя использовать для построения причинно-следственных диаграмм (в ней нет явно выраженного набора входных условий, комбинации которых следовало бы изучить), поэтому будем применять анализ граничных значений. Определим входные граничные значения.

1. EMPTAB содержит одну запись.
2. EMPTAB содержит максимально возможное количество записей (65535).
3. EMPTAB содержит 0 записей.
4. DEPTTAB содержит одну запись.
5. DEPTTAB содержит 65535 записей.
6. DEPTTAB содержит 0 записей.
7. В лидирующем отделе числится 1 сотрудник.
8. В лидирующем отделе числится 65535 сотрудников.
9. В лидирующем отделе не числится ни одного сотрудника.
10. Для всех отделов в таблице DEPTTAB указана одна и та же сумма выручки.
11. Лидирующему отделу соответствует первая запись в таблице DEPTTAB.
12. Лидирующему отделу соответствует последняя запись в таблице DEPTTAB.
13. Сотруднику лидирующего отдела соответствует первая запись в таблице EMPTAB.
14. Сотруднику лидирующего отдела соответствует последняя запись в таблице EMPTAB.

15. Сотрудник лидирующего отдела является менеджером.

16. Сотрудник лидирующего отдела не является менеджером.

17. Заработная плата сотрудника лидирующего отдела, который не является менеджером, составляет $14 999.99.

18. Заработная плата сотрудника лидирующего отдела, который не является менеджером, составляет $15 000.00.

19. Заработная плата сотрудника лидирующего отдела, который не является менеджером, составляет $15 000.01.

Определим следующие выходные граничные условия.

20. `ERRCODE=0`.

21. `ERRCODE=1`.

22. `ERRCODE=2`.

23. Увеличенная зарплата сотрудника лидирующего отдела составляет $29 999.99.

Ниже приведено дополнительное тестовое условие, основанное на предположении об ошибке.

24. Вслед за лидирующим отделом, в котором не числится ни одного сотрудника, в таблице `DEPTTAB` указан еще один лидирующий отдел, в котором числятся сотрудники.

Это условие используется для того, чтобы определить, не прерывается ли ошибочно обработка входных данных, если возникает ситуация, в которой `ERRCODE=2`.

Проанализировав все 24 условия, можно сделать вывод, что тесты, соответствующие условиям 2, 5 и 8, вряд ли будут представлять какой-либо практический интерес. Если при этом также учесть, что вероятность их наступления крайне мала (обычно делать такого рода предположения при тестировании весьма рискованно, но в данном случае, по-видимому, это безопасно), данные условия можно вообще исключить из рассмотрения. Следующий шаг состоит в том, чтобы сравнить оставшиеся условия с текущим тестовым набором (см. рис. 5.4) и определить, какие граничные условия еще не покрыты. Как показывает такое сравнение, дополнительные тесты требуются для условий 1, 4, 7, 10, 14, 17, 18, 19, 20, 23 и 24.

Далее необходимо спроектировать дополнительные тесты, покрывающие перечисленные 11 граничных условий. Один из возможных подходов заключается во встраивании данных условий в существующие тесты

(например, путем видоизменения теста 4 на рис. 5.4). Впрочем, поступать таким образом не рекомендуется, поскольку это может привести к нарушению полного комбинаторного покрытия условий уже имеющимися тестами. Безопаснее всего создать новые тесты в дополнение к тем, которые представлены на рис. 5.4. При этом нашей целью является проектирование минимального количества тестов, необходимых для покрытия граничных условий. Это обеспечивается тремя тестами, приведенными на рис. 5.5. Тест 5 покрывает условия 7, 10, 14, 17, 18, 19 и 20, тест 6 — условия 1, 4 и 23, а тест 7 — условие 24.

<table>
<tr><th>Тест</th><th>Вход</th><th>Ожидаемый выход</th></tr>
<tr><td>5</td><td>ESIZE = 3 DSIZE = 2
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>14999.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15000.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15000.01</td></tr></table>DEPTTAB
<table><tr><td>D33</td><td>55400.01</td></tr><tr><td>D36</td><td>55400.01</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE и DEPTTAB не изменяются
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>15199.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15100.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15100.01</td></tr></table></td></tr>
<tr><td>6</td><td>ESIZE = 1 DSIZE = 1
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>99899.99</td></tr></table>DEPTTAB
<table><tr><td>D99</td><td>99000.00</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE и DEPTTAB не изменяются
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>99999.99</td></tr></table></td></tr>
<tr><td>7</td><td>ESIZE = 2 DSIZE = 2
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10000.00</td></tr><tr><td>FORD</td><td>E</td><td>D22</td><td>33333.33</td></tr></table>DEPTTAB
<table><tr><td>D66</td><td>20000.00</td></tr><tr><td>D67</td><td>20000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE и DEPTTAB не изменяются
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10200.00</td></tr><tr><td>FORD</td><td>E</td><td>D22</td><td>33333.33</td></tr></table></td></tr>
</table>

Рис. 5.5. Дополнительные тесты анализа граничных условий для модуля `BONUS`

Предпосылкой для применения описанного подхода послужило то, что приведенные на рис. 5.5 тесты, спроектированные с использованием покрытия логики, т.е. по принципу "белого ящика", образуют приемлемый набор для модульного тестирования процедуры `BONUS`.

Инкрементное тестирование

Процесс модульного тестирования зависит от двух важных аспектов: проектирование эффективного набора тестов (это обсуждалось в предыдущем разделе) и то, каким образом модули объединяются в работающую программу. Второй аспект очень важен, поскольку от него зависят:

- форма записи модульных тестов;
- типы инструментов тестирования, которые могут быть использованы;
- очередность кодирования и тестирования модулей;
- стоимость генерации тестов;
- стоимость отладки (т.е. локализации и устранения ошибок).

В этом разделе мы обсудим два подхода к тестированию — *инкрементный* и *неинкрементный*, а в следующем — две стратегии инкрементного подхода, используемые как при разработке, так и при тестировании программного обеспечения: стратегию тестирования *сверху вниз*, или *нисходящую стратегию*, и стратегию тестирования *снизу вверх*, или *восходящую стратегию*.

Вопрос, на который мы сейчас попытаемся ответить, заключается в следующем: какой способ организации процесса тестирования более целесообразен — независимое тестирование каждого из модулей по отдельности с последующим их объединением в единую программу или объединение очередного модуля с набором ранее протестированных модулей и последующее тестирование результирующей сборки? Первый из двух подходов, применяемых при тестировании или интеграции программного обеспечения, называется неинкрементным (его жаргонное название — *большой взрыв*), а второй — инкрементным, или пошаговым.

В качестве примера рассмотрим программу, схематически представленную на рис. 5.6. Прямоугольники представляют шесть модулей (функций или процедур), образующих программу. Линии, соединяющие модули, представляют иерархию управления в программе: модуль `A` вызывает модули `B`, `C` и `D`, модуль `B` вызывает модуль `E` и т.д. При традиционном неинкрементном подходе тестирование выполняется следующим образом. Сначала выполняется модульное тестирование всех шести модулей, причем каждый из них

тестируется как независимая сущность. Модули могут тестироваться одновременно или поочередно, в зависимости от среды (например, пакетная обработка или интерактивный режим) и количества тестировщиков. Наконец, модули объединяются или интегрируются в программу (например, путем редактирования связей).

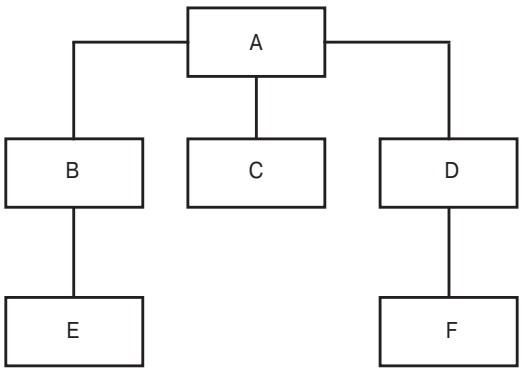

Рис. 5.6. Пример программы, состоящей из шести модулей

Для тестирования каждого модуля требуется специальный *модуль-драйвер* (driver module) и один или несколько *модулей-заглушек* (stub modules). Например, чтобы протестировать модуль `B`, сначала необходимо спроектировать тесты, а затем написать небольшую программу-драйвер, которая обеспечит передачу тестируемому модулю `B` входных параметров, необходимых для прогона тестов. (Для этой цели можно также воспользоваться инструментальными средствами тестирования.) Драйвер должен показывать тестировщику результаты, выдаваемые модулем `B`. Кроме того, поскольку модуль `B` вызывает модуль `E`, управление при таком вызове должно куда-то передаваться. Это достигается за счет использования заглушки — специального модуля, которому присваивается имя `"E"` и который имитирует выполнение функций модуля `E`.

После завершения модульного тестирования всех шести модулей их объединяют в единую программу.

Альтернативный подход предполагает инкрементное тестирование. При таком подходе модули не тестируются изолированно друг от друга, а поочередно подключаются к постепенно наращиваемому набору уже проверенных модулей и лишь после этого подвергаются тестированию.

Ввиду того, что число возможных реализаций инкрементного подхода слишком велико, пока что преждевременно рассматривать применение

какой-либо конкретной методики к программе, показанной на рис. 5.7. Здесь ключевым является вопрос о том, с какого программного уровня должно начинаться тестирование: верхнего или нижнего. Но поскольку этот вопрос будет обсуждаться в следующем разделе, мы сейчас просто примем, что тестирование начинается с нижнего уровня.

Первый шаг заключается в тестировании модулей `E`, `C` и `F`, что можно делать либо параллельно (силами трех человек), либо последовательно. При этом для каждого модуля придется создать драйвер; заглушки в данном случае не нужны. Следующий шаг — тестирование модулей `B` и `D`, но не изолированных, а объединенных соответственно с модулями `E` и `F`. Иными словами, чтобы протестировать модуль `B`, следует написать драйвер, объединяющий тесты, и протестировать пару `B-E`. Инкрементный процесс добавления очередного модуля к множеству или подмножеству ранее протестированных модулей продолжается до тех пор, пока не будет протестирован последний модуль (в данном случае `A`). Заметим, что с тем же успехом данную процедуру можно было бы выполнить и в нисходящем порядке.

Уже на данном этапе можно сделать несколько очевидных наблюдений.

1. Неинкрементное тестирование более трудоемко. Для программы, показанной на рис. 5.6, в этом случае необходимо создать пять драйверов и пять заглушек (предполагается, что для верхнего модуля драйвер не нужен). В то же время для инкрементного тестирования снизу вверх потребовалось бы лишь пять драйверов, а сверху вниз — всего пять заглушек. Уменьшение объема работы объясняется тем, что вместо драйверов (при нисходящем тестировании) или заглушек (при восходящем тестировании), которые требуются при неинкрементном подходе, используются ранее протестированные модули.
2. Программные ошибки, связанные с несоответствием межмодульных интерфейсов или с некорректными предположениями относительно взаимодействия модулей, при использовании инкрементного подхода будут обнаруживаться раньше, поскольку тестирование комбинаций модулей в этом случае начинается на более раннем этапе. В то же время при неинкрементном тестировании модули “не видят” друг друга до самого конца процесса.
3. При инкрементном тестировании отладка программ упрощается. Если предположить, что ошибки, связанные с несогласованностью межмодульных интерфейсов или некорректностью допущений о характере взаимодействия модулей, действительно существуют (а такое предположение, как показывает опыт, зачастую оправдывается), то в случае

неинкрементного тестирования ошибки нельзя будет выявить до тех пор, пока все модули не будут собраны в единую программу. К тому времени точная локализация ошибок будет уже затруднена, так как ошибки могут оказаться рассредоточенными по всей программе. Вместе с тем в случае инкрементного тестирования локализовать ошибки будет легче, поскольку они, вероятнее всего, будут связаны с тем модулем, который добавлялся последним.

4. Результаты инкрементного тестирования могут оказаться более полными. Например, если тестируется модуль `B`, то в результате будет выполняться либо модуль `E`, либо модуль `A` (в зависимости от того, начинается ли тестирование снизу или сверху). И хотя эти модули ранее уже были тщательно протестированы, не исключено, что их совместное выполнение с модулем `B` при его тестировании активизирует некие новые условия, которые, возможно, позволят обнаружить недостаточность первоначального тестирования модулей `E` или `A`. С другой стороны, результаты неинкрементного тестирования модуля `B` будут касаться только этого модуля. Иначе говоря, при инкрементном тестировании ранее протестированные модули заменяют собой заглушки или драйверы, которые требуются в случае неинкрементного тестирования. В результате реальные модули дополнительно тестируются на протяжении всего процесса вплоть до завершения тестирования последнего модуля.
5. По всей видимости, неинкрементный подход характеризуется меньшим расходом машинного времени. Если изображенный на рис. 5.6 модуль `A` тестируется с использованием подхода “снизу вверх”, то одновременно с ним будут, скорее всего, выполняться также модули `B`, `C`, `D`, `E` и `F`. При неинкрементном тестировании модуля `A` выполняются лишь заглушки для модулей `B`, `C` и `E`. То же самое относится и к инкрементному тестированию “сверху вниз”. Если тестируется модуль `F`, то в процессе этого могут выполняться модули `A`, `B`, `C`, `D` и `E`; при неинкрементном тестировании модуля `F` выполняется лишь драйвер этого модуля плюс сам модуль. Таким образом, количество машинных команд, выполняемых во время тестового прогона, при инкрементном подходе явно больше, чем при неинкрементном. Такое превышение компенсируется тем, что неинкрементное тестирование требует большего количества драйверов и заглушек, чем инкрементное, а на их разработку также тратится определенное количество машинного времени.
6. Использование неинкрементного подхода предоставляет больше возможностей для параллельной организации работы на начальном этапе

тестирования (одновременное тестирование всех модулей). Этот фактор может сыграть определенную роль при работе над большими проектами (включающими множество модулей и многочисленный штат разработчиков), поскольку численность персонала, участвующего в проекте, обычно достигает максимума в начале фазы модульного тестирования.

В заключение отметим, что пп. 1–4 демонстрируют преимущества инкрементного тестирования, а пп. 5–6 — его недостатки. Для современного этапа развития компьютерной индустрии характерна тенденция к уменьшению стоимости оборудовании (и эта тенденция, по-видимому, будет сохраняться) и увеличению его производительности при одновременном росте стоимости труда и величины экономических рисков, связанных с ошибками в программном обеспечении. Таким образом, со временем преимущества, указанные в пп. 1–4, приобретают все большее значение, а недостатки, отмеченные в п. 5, теряют свою значимость. Указанный в п. 6 недостаток инкрементного подхода представляется несущественным. Все это приводит нас к выводу, что предпочтение следует отдавать инкрементному тестированию.

Нисходящее и восходящее тестирование

Убедившись в преимуществах инкрементного тестирования по сравнению с неинкрементным, исследуем две его возможные стратегии: тестирование *сверху вниз*, или *нисходящее тестирование*, и тестирование *снизу вверх*, или *восходящее тестирование*. Однако прежде внесем ясность в терминологию. Во-первых, термины *нисходящее тестирование*, *нисходящая разработка* и *нисходящее проектирование* часто употребляются как синонимы. Но если первые два из них действительно являются синонимами (они представляют стратегию очередности кодирования и тестирования модулей), то термин *нисходящее проектирование* относится к независимому процессу совершенно иной природы. К программе, которая проектировалась с использованием нисходящей стратегии, может применяться как нисходящее, так и восходящее инкрементное тестирование.

Во-вторых, восходящее тестирование (или восходящая разработка) часто отождествляется с неинкрементным тестированием. Это недоразумение возникает из-за того, что начальные стадии восходящего и неинкрементного тестирования совпадают (в обоих случаях первыми тестируются модули нижнего уровня). Однако, как было показано в предыдущем разделе, восходящее тестирование представляет собой инкрементную стратегию. Наконец,

поскольку обе стратегии — инкрементные, мы не будем повторно рассматривать преимущества инкрементного подхода и обсудим лишь различия между нисходящей и восходящей стратегиями тестирования.

Нисходящее тестирование

Нисходящее тестирование начинается с верхнего в иерархической структуре, или головного, модуля программы. Какой-либо единственно "правильной" процедуры, регламентирующей последующий выбор каждого очередного модуля, подлежащего инкрементному тестированию, не существует. Руководствоваться следует лишь тем, что подходящим для этих целей будет такой модуль, для которого найдется по крайней мере один вызывающий его модуль, уже прошедший тестирование.

Эту стратегию иллюстрирует рис. 5.7. Изображенная на нем программа состоит из 12 модулей с именами от `A` до `L`. Предположим, модуль `J` содержит операции чтения, а модуль `I` — операции записи.

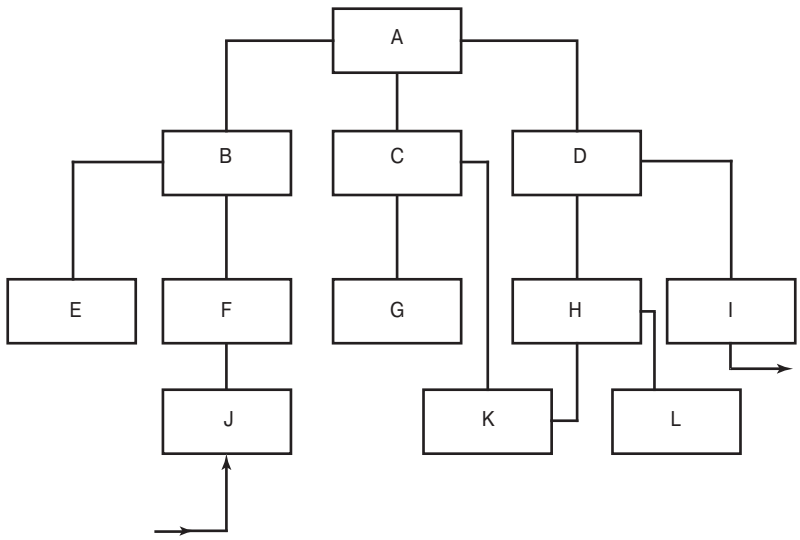

Рис. 5.7. Пример программы, состоящей из 12 модулей

Первый шаг — тестирование модуля `A`. Для его выполнения необходимо написать заглушки, представляющие модули `B`, `C` и `D`. К сожалению, задачи, которые должны решать модули-заглушки, часто понимают неверно. Нередко можно встретить высказывания, в которых утверждается, что роль заглушек сводится лишь к выводу сообщений о достижении определенной точки программы или что они вообще не выполняют никаких полезных действий и должны просто присутствовать, подменяя собой еще не разработанные

модули. В большинстве случаев подобные утверждения ошибочны. Коль скоро модуль B вызывается модулем A, то ожидается, что он выполнит некоторую полезную работу, которая, скорее всего, должна завершаться определенным результатом (выходными данным), возвращаемым модулем A. Если заглушка просто возвращает управление или выводит сообщение об ошибке, не возвращая никакого значимого результата, то модуль A аварийно завершит свою работу, причем это произойдет не из-за того, что в нем содержится ошибка, а из-за того, что заглушка не обеспечивает надлежащей имитации недостающего модуля. Более того, одного лишь возврата жестко закодированного в заглушке вывода, не меняющегося в зависимости от условий теста, часто будет недостаточно. Пусть, например, требуется запрограммировать заглушку, которая имитировала бы подпрограмму, предназначенную для извлечения квадратного корня из числа, выполнения поиска в базе данных, считывания заданной записи из основного файла или чего-нибудь в этом роде. Если вместо конкретных выходных значений, ожидаемых вызывающим модулем при каждом вызове заглушки, она будет возвращать некий фиксированный вывод, то модуль либо выдаст неопределенный результат, либо аварийно завершит свою работу. Поэтому создание заглушек — задача вовсе не тривиальная.

С реализацией нисходящего тестирования связан еще один аспект, касающийся того, в какой форме тестовые данные передаются в программу, причем, несмотря на всю важность этого аспекта, в большинстве случаев о нем даже не упоминают. Применительно к нашему примеру этот вопрос можно сформулировать так: каким образом тестовые данные поступают в модуль A? В типичных программах головной (верхний в иерархии) модуль не получает никаких входных аргументов и не выполняет никаких операций ввода-вывода, так что ответ на этот вопрос далеко не очевиден. В нашем случае можно ответить, что тестовые данные поступают в головной модуль (модуль A) из одной или нескольких связанных с ним заглушек. Для примера предположим, что модули B, C и D выполняют следующие функции:

- B — получает сводные данные из файла транзакций;
- C — определяет, соответствуют ли недельные показатели установленным критериям;
- D — формирует итоговый отчет за неделю.

В таком случае тестом для A является сводка данных о транзакциях, возвращаемая заглушкой B. Заглушка D может содержать инструкции, осуществляющие вывод данных на печать, что позволит наблюдать результаты каждого теста.

С этой программой связана еще одна проблема. Поскольку вполне вероятно, что модуль A вызывает модуль B всего один раз, нужно решить, каким образом передать в модуль A несколько тестов. Одно из возможных решений состоит в том, чтобы разработать несколько версий заглушки B, каждая из которых содержит один фиксированный набор тестовых данных для возврата в модуль A. Тогда для выполнения всех тестов достаточно запустить программу несколько раз, используя при каждом запуске разные версии заглушки B. Возможен и другой вариант решения, когда тестовые данные помещаются во внешние файлы, откуда заглушка B сможет прочитать их, а затем возвратить в модуль A. В любом случае, если вспомнить предыдущее обсуждение, становится совершенно очевидным, что разработка модулей-заглушек — не такое простое занятие, каким его часто представляют. Кроме того, в силу специфики программ часто требуется, чтобы тестовые данные передавались в тестируемый модуль из нескольких заглушек, имитирующих модули нижнего уровня (например, когда модуль получает необходимые для обработки данные путем вызова нескольких других модулей).

После того как модуль A будет протестирован, одну из заглушек заменяют реальным модулем с последующим добавлением требуемых им дополнительных заглушек. В качестве примера на рис. 5.8 представлена возможная очередная версия рассматриваемой программы.

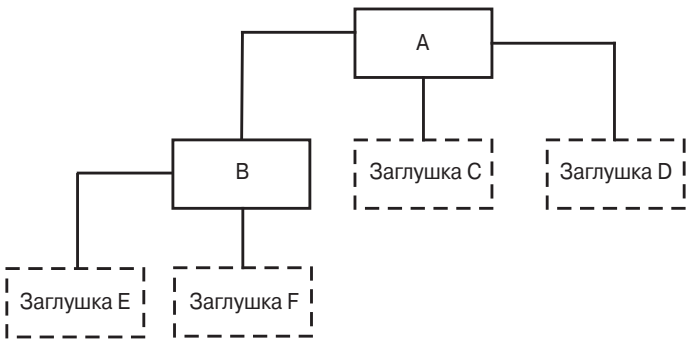

Рис. 5.8. Второй шаг процесса нисходящего тестирования

По завершении тестирования верхнего (головного) модуля возможны самые разные варианты последовательности тестирования оставшихся модулей и их слияния для образования единой программы. Из всего возможного многообразия таких последовательностей ниже в качестве примера представлено четыре возможных варианта.

1. A B C D E F G H I J K L
2. A B E F J C G K D H L I
3. A D H I K L C G B F J E
4. A B F J D I E C G K H L

При параллельном тестировании нескольких модулей возможны альтернативные варианты. Например, после тестирования модуля A один программист может тестировать комбинацию A-B, второй — A-C, третий — A-D. В общем случае сказать, какая последовательность будет лучшей, нельзя, но по этому поводу можно дать две рекомендации.

1. Если в программе имеются критически важные разделы (возможно, таковым в данном случае является модуль G), то целесообразно выбирать последовательность так, чтобы эти разделы включались в цепочку как можно раньше. В качестве критически важного раздела может выступать сложный модуль, модуль, реализующий новый алгоритм, или модуль, относительно которого есть опасения, что он может содержать ошибки.
2. Модули, содержащие операции ввода-вывода, следует включать в цепочку тестирования как можно раньше.

Целесообразность первой из этих рекомендаций должна быть очевидной, вторая же требует дополнительных пояснений. Напомним: суть проблемы заглушек состоит в том, что одни из них должны содержать тесты, тогда как другие — выводить данные на печать или на экран. Но как только в цепочку тестирования включается реальный модуль, принимающий входные данные программы, представление тестов значительно упрощается. Форма их представления становится идентичной той, которая используется в готовой программе для ввода данных (например, данные поступают из файла транзакций или с терминала). Точно так же после подключения реального модуля, реализующего функции вывода данных, добавление в модули-заглушки кода, записывающего результаты тестов, может больше не потребоваться. Так, если модули J и I реализуют функции ввода-вывода, а модуль G выполняет некоторую критически важную функцию, то цепочка инкрементного тестирования может выглядеть следующим образом:

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

При этом сама программа после выполнения шестого шага инкрементного тестирования будет выглядеть так, как показано на рис. 5.9.

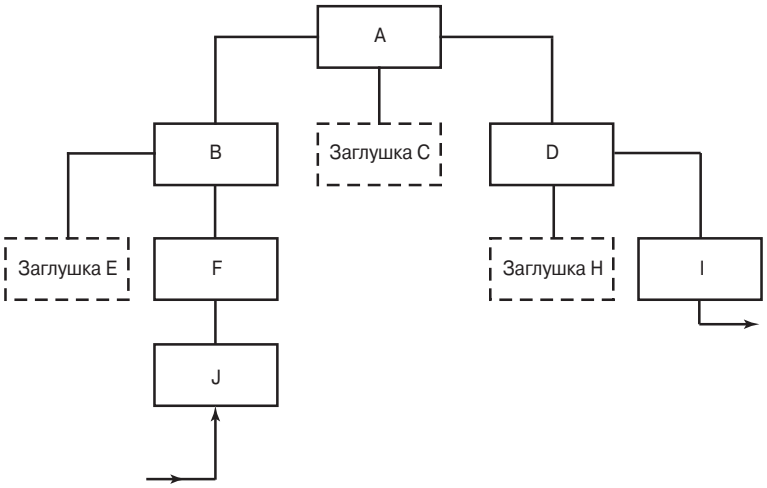

Рис. 5.9. Промежуточное состояние программы в процессе нисходящего тестирования

По достижении промежуточного состояния программы, изображенного на рис. 5.9, представление тестов и анализ результатов тестирования существенно упрощаются. Это состояние обладает еще одним преимуществом, которое заключается в том, что вы получаете в свое распоряжение рабочую каркасную версию, выполняющую реальные операции ввода-вывода, в то время как часть остальных "внутренностей" программы по-прежнему имитируется заглушками. Эта ранняя каркасная версия программы предоставляет следующие возможности:

- позволяет обнаруживать ошибки и проблемы, обусловленные человеческим фактором;
- дает возможность продемонстрировать работу программы конечному пользователю;
- служит доказательством правильности подхода, лежащего в основе разработки программы;
- служит вдохновляющим моральным стимулом.

Таковы главные преимущества нисходящей стратегии.

В то же время у нисходящего подхода имеется ряд серьезных недостатков. Предположим, текущее состояние проверяемой программы соответствует рис. 5.9 и на следующем шаге нам предстоит заменить заглушку `H` модулем `H`. Для этого мы должны разработать набор тестов для модуля `H` (при условии, что это еще не было сделано) с помощью описанных ранее методов.

Заметим, однако, что теперь тесты должны быть представлены как обычные входные данные реальной программы, передаваемые в модуль J. Это влечет за собой целый ряд проблем. Во-первых, поскольку связь между модулями J и H осуществляется через целый ряд других модулей (F, B, A и D), то может оказаться так, что предоставить модулю J все тесты, необходимые для тестирования любой заранее заданной ситуации в модуле H, будет невозможно. Например, если H — это модуль BONUS (см. листинг 5.1), то может случиться так, что в силу природы промежуточного модуля D вам не удастся создать некоторые из семи тестов, представленных на рис. 5.4 и 5.5.

Во-вторых, даже если бы возможность протестировать любую ситуацию в модуле H и существовала, то в силу "удаленности" модуля H от точки, в которой тестовые данные вводятся в программу, определение того, какие именно данные должны быть предоставлены модулю J для тестирования этих ситуаций, часто будет требовать больших интеллектуальных усилий.

В-третьих, поскольку отображаемые выходные данные теста могут поступать из модуля, находящегося на большом "расстоянии" от тестируемого модуля, установление соответствия между наблюдаемыми результатами и тем, что на самом деле происходит в модуле, может представлять собой трудную или вообще неразрешимую задачу. Допустим, мы добавляем в схеме на рис. 5.9 модуль E. Результаты каждого теста определяются путем анализа выходных данных, записываемых модулем I, но из-за наличия промежуточных модулей, вклинившихся между E и I, делать какие-либо заключения относительно фактического выхода модуля E (т.е. данных, возвращаемых в модуль B) может быть затруднительно.

В зависимости от способа применения нисходящей стратегии могут возникнуть еще две проблемы. Иногда может показаться, что данная стратегия хорошо совмещается с этапом проектирования программы. Например, если проектируется программа, приведенная на рис. 5.7, то можно было бы подумать, что после того, как будут спроектированы первые два уровня, кодирование и тестирование модулей от A до B можно начинать одновременно с проектированием нижних уровней. Однако, как в свое время нами уже отмечалось, обычно такие действия не могут признаваться разумными. Проектирование программного обеспечения — процесс итеративный, а это означает, что при проектировании модулей, относящихся к нижним иерархическим уровням программы, может обнаружиться, что модули верхних уровней нуждаются в изменении или улучшении. И если эти модули уже закодированы и прошли тестирование, то, вероятнее всего, желаемые изменения в них внесены не будут, что в долгосрочной перспективе является не лучшим выходом.

И наконец, последняя проблема, с которой часто приходится сталкиваться на практике, заключается в том, что к тестированию следующего модуля приступают еще тогда, когда тестирование предыдущего полностью не завершено. Это происходит по двум причинам: из-за трудностей внедрения тестовых данных в модули-заглушки, а также из-за того, что ресурсы модулей нижних уровней обеспечиваются, как правило, модулями верхних уровней. На рис. 5.7 видно, что для тестирования модуля `A` может потребоваться несколько версий заглушки для модуля `B`. В подобных ситуациях многие тестировщики склонны рассуждать примерно следующим образом: "На выполнение всех тестов для модуля `A` уйдет масса времени. Лучше я отложу часть тестов до тех пор, пока не будет готов модуль `J`, с которым предоставлять тестовые данные будет проще, а затем вернусь к модулю `A` и завершу его тестирование". Конечно же, проблема заключается в том, что о необходимости вернуться к тестированию модуля `A` впоследствии можно даже и не вспомнить. К тому же, поскольку ответственность за предоставление ресурсов (например, открытие файлов) модулям нижнего уровня обычно возлагается на модули более высоких уровней, определить без затруднений, были ли эти ресурсы предоставлены корректно (например, был ли файл открыт с указанием надлежащих атрибутов), иногда можно лишь после того, как будут протестированы модули нижнего уровня, использующие эти ресурсы.

Восходящее тестирование

Перейдем к рассмотрению восходящей инкрементной стратегии тестирования. Во многих отношениях восходящее тестирование противоположно нисходящему; таким образом, преимущества первого становятся недостатками второго, а недостатки — преимуществами. С учетом этого обсуждение восходящего тестирования будет более кратким.

В соответствии с данной стратегией тестирование начинают с терминальных модулей программы (т.е. тех, которые не вызывают другие модули). Аналогично предыдущему случаю, наилучшего универсального рецепта выбора следующего модуля для инкрементного тестирования просто не существует. Единственное, что можно сказать по этому поводу, — выбираемый модуль должен быть таким, чтобы все его подчиненные (вызываемые им) модули предварительно были протестированы.

Если вновь обратиться к рис. 5.7, то первым шагом должно быть параллельное или последовательное тестирование всех или только части модулей `E`, `J`, `G`, `K`, `L` и `I`. С этой целью для каждого модуля потребуется написать специальный драйвер — модуль, который содержит "встроенные" тестовые

данные, вызывает тестируемый модуль и отображает выходные результаты тестирования (или сравнивает их с ожидаемым выходом). В отличие от заглушек, нет никакой необходимости создавать несколько версий каждого драйвера, поскольку драйвер может итеративно вызывать тестируемый модуль несколько раз, передавая с каждым вызовом очередной набор тестовых данных. В большинстве случаев создавать драйверы проще, чем заглушки.

Как и при нисходящем тестировании, последовательность тестирования определяется прежде всего критичностью того или иного модуля. Если мы решаем, что наиболее критичны модули D и F, то промежуточное состояние восходящего инкрементного теста будет соответствовать рис. 5.10. Следующими действиями могут быть тестирование модуля E, затем — B и комбинирование B с предварительно протестированными модулями E, F и J.

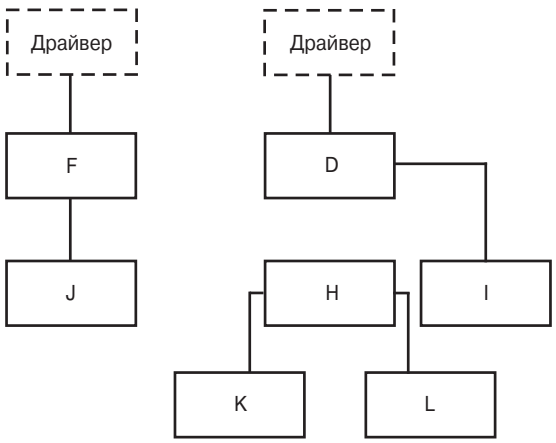

Рис. 5.10. Промежуточное состояние при восходящем тестировании

Недостатком описанной стратегии является то, что в ее рамках создание каркаса программы на ранних стадиях тестирования принципиально невозможно. По сути, работающий вариант программы может оказаться в ваших руках лишь после того, как завершится тестирование последнего модуля (A), и этот вариант будет представлять собой программу с полностью сопряженными модулями. И хотя функции ввода-вывода могут тестироваться еще до объединения программы в единое целое (модули ввода-вывода присутствуют на рис. 5.10), преимуществ раннего формирования полной структуры программы в данном случае нет.

Проблемы, связанные с невозможностью или трудностью создания всех тестовых ситуаций, присущие нисходящему подходу, при восходящем

тестировании не возникают. Ввиду отсутствия промежуточных модулей, создающих помехи, драйвер как средство тестирования применяется непосредственно к тому модулю, который тестируется. 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 будут спроектированы модули нижнего уровня. Проблем с незавершенностью тестирования одного модуля при переходе к тестированию другого, обусловленных трудностью кодирования тестовых данных в различных версиях заглушки, при восходящем тестировании также не существует.

Сравнение нисходящего и восходящего тестирования

Было бы здорово, если бы между нисходящей и восходящей стратегиями тестирования существовала столь же четкая грань, как между инкрементным и неинкрементным подходами, но, к сожалению, это не так. В табл. 5.3 приведены относительные достоинства и недостатки обеих стратегий (за исключением общих преимуществ, обеспечиваемых применением инкрементного подхода). Преимущества, указанные в качестве первых для каждого из подходов, можно было бы считать определяющими, однако не существует никаких доказательств того, что в типичных программах основные дефекты встречаются чаще всего в модулях верхнего или нижнего уровня. Самый безопасный способ принятия решений — это взвешивать значение факторов, указанных в табл. 5.3, сообразуясь с конкретными особенностями тестируемой программы. Если же отвлечься от какой-либо конкретной специфики программ, то, принимая во внимание серьезность последствий четвертого недостатка нисходящего тестирования, а также учитывая доступность инструментов тестирования, устраняющих потребность в драйверах, но не заглушках, можно, пожалуй, сделать вывод о том, что предпочтение следует отдавать стратегии восходящего тестирования.

Таблица 5.3. Сравнение нисходящего и восходящего тестирования

Нисходящее тестирование	
Преимущества	**Недостатки**
1. Имеет преимущества, если основные ошибки встречаются главным образом на верхних иерархических уровнях программы	1. Необходимо разрабатывать модули-заглушки
2. Представление тестов упрощается после подключения функций ввода-вывода	2. Модули-заглушки часто оказываются сложнее, чем предполагалось первоначально

Окончание табл. 5.3

Нисходящее тестирование	
Преимущества	**Недостатки**
3. Раннее формирование каркаса программы обеспечивает возможность демонстрации ее работы и служит моральным стимулом для тестировщиков	3. До подключения функций ввода-вывода представление тестовых данных в заглушках может вызывать затруднения
	4. Невозможность или значительная сложность создания тестовых условий
	5. Труднее обеспечить наблюдение за результатами тестирования
	6. Наталкивает на мысль о возможности совмещения стадий проектирования и тестирования
	7. Поощряет отсрочку окончательного тестирования некоторых модулей
Восходящее тестирование	
Преимущества	**Недостатки**
1. Имеет преимущества, если основные ошибки встречаются главным образом на нижних иерархических уровнях программы	1. Необходимо разрабатывать модули-драйверы
2. Легче создавать тестовые условия	2. Программа как единое целое не существует до тех пор, пока не добавлен последний модуль
3. Проще обеспечить наблюдение за результатами тестирования	

В заключение необходимо отметить, что рассмотренные стратегии нисходящего и восходящего тестирования не являются единственно возможными при инкрементном подходе.

Выполнение теста

Завершающий этап модульного тестирования — это фактическое выполнение тестов. Дадим некоторые советы и рекомендации относительно того, как это лучше всего делать.

Если выполнение теста приводит к ситуации, в которой фактические результаты не совпадают с ожидаемыми, то это может свидетельствовать либо о наличии ошибок в модуле, либо о некорректности ожидаемых результатов (некорректности самого теста). Чтобы свести к минимуму подобную неопределенность, необходимо тщательно проверять и анализировать наборы тестов перед их выполнением (т.е. тестировать сами тесты).

Применение автоматизированных инструментальных средств позволяет снизить трудоемкость тестирования. Например, существуют инструменты тестирования, которые избавляют от необходимости создавать модули-драйверы. Средства анализа потоков перечисляют пути выполнения в программе, находят инструкции, которые не выполняются ни разу ("недостижимый код"), и обнаруживают места в программе, в которых переменные используются еще до того, как им присвоено какое-либо значение.

Ранее мы уже говорили о том, что необходимой частью тестового набора является описание ожидаемых результатов. При выполнении тестов обращайте внимание на возможные побочные эффекты (случаи, когда модуль делает не то, что должен делать). В общем случае обнаружить такую ситуацию трудно, но иногда побочные эффекты можно выявить, если проверить переменные, значения которых в процессе тестирования изменяться не должны. Например, тест 7 (см. рис. 5.5) не должен изменять значения переменных `ESIZE`, `DSIZE` и `DEPTTAB`, являющихся частью ожидаемых результатов. При выполнении этого теста следует проверять не только корректность выходных результатов, но и значения указанных переменных, чтобы выявлять возможные случаи их ошибочного изменения.

Проблемы психологического характера, возникающие в тех случаях, когда программист пытается тестировать собственную программу, проявляются и при модульном тестировании. Программистам полезно обмениваться модулями и тестировать модули своих коллег, а не собственные. Например, может оказаться целесообразным, чтобы тестирование вызываемого модуля выполнял программист, написавший вызывающий модуль. Заметим, что сказанное относится только к тестированию; отладкой же модуля всегда должен заниматься его автор.

Не выбрасывайте тесты и всегда представляйте их в такой форме, чтобы ими можно было повторно воспользоваться в будущем. Вспомните о, казалось бы, противоречащем здравой логике явлении, графическая иллюстрация которого приводилась на рис. 2.2: чем больше в некотором подмножестве модулей обнаружено ошибок, тем выше вероятность того, что в этих модулях содержатся ошибки, оставшиеся необнаруженными. Такие модули должны проходить дополнительное модульное тестирование, а их код, возможно, должен подвергаться дополнительному сквозному просмотру или инспекции. Наконец, следует помнить, что целью модульного тестирования является не демонстрация корректной работы модулей, а демонстрация наличия в них ошибок.

Резюме

В этой главе вы ознакомились с основами техники тестирования, особенно больших программ, когда тестированию подвергаются отдельные компоненты — подпрограммы, классы и процедуры. При модульном тестировании функциональность программного обеспечения сравнивается со спецификациями, описывающими назначение функций. Модульное, или блочное, тестирование может служить важной частью инструментария разработчика, позволяя создавать надежные приложения, особенно при работе с такими объектно-ориентированными языками, как Java и C#. Перед модульным тестированием стоит та же цель, что и перед любым другим типом тестирования программного обеспечения: демонстрация несоответствия программы требованиям ее спецификации. Для выполнения модульного теста требуется не только спецификация, но и исходный код каждого модуля.

В значительной степени модульное тестирование относится к стратегии "белого ящика". (Более подробно об этом говорилось главе 4.) Тщательное выполнение модульного тестирования предполагает использование инкрементных стратегий, таких как нисходящее или восходящее тестирование.

Прежде чем приступить к модульному тестированию, целесообразно прочитать материал, посвященный психологическим и экономическим аспектам тестирования (см. главу 2).

В заключение необходимо сделать одно важное замечание. Модульное тестирование — это лишь начало процедуры исчерпывающего тестирования. На последующих этапах вы должны будете перейти к высокоуровневому тестированию, о котором говорится в главе 6, а затем — к пользовательскому тестированию, которому посвящена глава 7.