И. В. Степанченко

МЕТОДЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
КАМЫШИНСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ (ФИЛИАЛ)
ВОЛГОГРАДСКОГО ГОСУДАРСТВЕННОГО ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА

И. В. Степанченко

МЕТОДЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

РПК «Политехник»

Волгоград
2006

УДК 681. 3. 06 (075)
С 79

Рецензенты: кафедра «Информатика и вычислительная техника» Волгоградского государственного архитектурно-строительного университета, зав. кафедрой, д. т. н. профессор А. Н. Богомолов; заведующий кафедрой «Системы автоматизированного проектирования и поискового конструирования» Волгоградского государственного технического университета, д. т. н., профессор В. А. Камаев

Степанченко И. В. Методы тестирования программного обеспечения: Учеб. пособие / ВолгГТУ, Волгоград, 2006. – 74 с.

ISBN 5-230-04563-9

Излагаются подходы и методы тестирования, начиная с проблемы определения термина «тестирование» и общих принципов и заканчивая описанием методов и стратегий тестирования. В частности, рассмотрены методы белого и черного ящика (покрытие операторов, условий, решений, комбинаторного покрытия условий и решений, эквивалентного разбиения, анализа граничных значений и др.).

Каждый метод, рассматриваемый в пособии, сопровождается примерами и пояснениями, а в конце каждой главы приводятся контрольные вопросы и задания.

Ориентировано на студентов, обучающихся по направлению 552800 «Информатика и вычислительная техника» и специальности 220200 «Автоматизированные системы обработки информации и управления» очной и очно-заочной форм обучения по основной и сокращенной программам обучения. Может быть полезно студентам других специальностей при создании программного обеспечения.

Ил. 15. Табл. 2. Библиогр.: 18 назв.

Печатается по решению редакционно-издательского совета
Волгоградского государственного технического университета

ISBN 5-230-04563-9

Илья Викторович Степанченко

МЕТОДЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

.
Редакторы: Попова Л. В., Пчелинцева М. А.
Компьютерная верстка Степанченко И. В.
Темплан 2006 г., поз. № 21.
Подписано в печать 2006 г. Формат 60×84 $^{1}/_{16}$.
Бумага потребительская. Гарнитура ”Times“.
Усл. печ. л. 4, 63. Усл. авт. л. 4,44.
Тираж 100 экз. Заказ

Волгоградский государственный технический университет
400131 Волгоград, просп. им. В. И. Ленина, 28.
РПК «Политехник»
Волгоградского государственного технического университета
400131 Волгоград, ул. Советская, 35.

ВВЕДЕНИЕ

Известно, что при создании типичного программного проекта около 50 % общего времени и более 50 % общей стоимости расходуется на тестирование разрабатываемой программы или системы. Эти цифры могут вызвать целую дискуссию, но, оставив вопрос точности оценки в стороне и основываясь на том, что тестирование является важным этапом в создании программного продукта, можно было бы предположить, что к настоящему времени тестирование программ поднялось до уровня точной науки. Увы, это не так. На самом деле тестирование программ освещено, пожалуй, меньше, чем любой другой аспект разработки программного обеспечения. К тому же тестирование является до сих пор «немодным» предметом, если иметь в виду спорность публикаций по этому вопросу.

Любой программист может похвастать «хорошо» написанным кодом, модулем, классом, но, как правило, он практически ничего не может сказать, насколько полно оттестирован этот код. Многие готовы ругать других разработчиков, указывая на их ошибки, сбои их программного обеспечения, забывая о своих.

Так что же делать разработчику, менеджеру проекта, руководителю фирмы? Как сократить расходы и повысить качество программного обеспечения? Сколько нужно тестировать программное обеспечение? Как построить эффективный процесс тестирования? Какие инструментальные средства использовать? Вопросов в области тестирования настолько много, что охватить их в одном пособии практически невозможно. Было решение создать пособие, с которого можно было бы начать изучение тестирования программного обеспечения. В качестве базы выбрана замечательная книга [1].

Материал книги значительно переработан, расширен современными публикациями и дополнен собственным опытом. Весь публикуемый материал неоднократно излагался автором на лекциях по дисциплинам «Технология программирования» и «Технологии программирования».

1. ФИЛОСОФИЯ ТЕСТИРОВАНИЯ

1.1. Тест для самооценки

Хотите испытать себя в тестировании? Задача состоит в том, чтобы проверить некоторый метод.

Данный метод получает в качестве параметров три целых числа, которые интерпретируются как длины сторон треугольника. Выходом метода является сообщение о том, является ли треугольник неравносторонним, равнобедренным или равносторонним [1].

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

Следующий шаг состоит в оценке эффективности вашей проверки. Оказывается, что метод труднее написать, чем это могло показаться вначале. Были изучены различные версии данного метода 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 перестановкой 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 неравностороннего 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. Описали ли вы заранее в каждом тесте не только входные значения, но и выходные данные метода?

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 данного метода, адекватный тест для него должен их обнаруживать. Для сравнения отметим, что опытные профессиональные программисты и тестировщики набирают в среднем только 7–8 очков из 14 возможных. Выполненное упражнение показывает нам, что тестирование даже тривиальных программ, подобных приведенной, – непростая задача.

1.2. Определение термина «тестирование»

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 до сих пор тщательно не исследованы. Однако, необходимо разобраться в общих моментах экономики и 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 преду-

смотренные 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 (так как тестирование стоит денег и нам желательно возвратить затраченную сумму, а это можно сделать только путем увеличения стоимости программы). Увеличение стоимости означает повышение качества или возрастание надежности программы, в противном случае пользователь будет недоволен платой за качество. Повышение качества или надежности программы 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 обнаруживается их максимально возможное число. Таким образом, сформулируем наиболее приемлемое и простое 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. Например, одно из них состоит в том, что **тестирование – процесс деструктивный** (т. е. обратный созидательному, конструктивному). Именно этим и объясняется, почему многие программисты и тестировщики 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 руково-

дителями проектов при оценке результатов тестирования. Некоторые руководители программных 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. Естественно, заранее неизвестно, будет ли тест удачным или неудачным, но построение удачных тестов – отдельная тема.

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 на 500 рублей, 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» (например, в [5] и [6]) порождают еще одну проблему: они ставят цель, которая не может быть достигнута ни для одной программы, даже весьма тривиальной. Результаты психологических исследований показывают, что если перед человеком ставится невыполнимая задача, то он работает хуже. Например, если 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 [1]. 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, позво-

ляющий 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. Вспомните тест для самооценки, метод может допустить ошибку, если будет делать то, что он не должен делать (например, сообщать, что тройка 1, 2, 3 представляет неравносторонний треугольник, а тройка 0, 0, 0 – равносторонний). Ошибки этого класса можно обнаружить скорее, если рассматривать тестирование как процесс поиска ошибок, а не демонстрацию корректности работы.

Подводя итог вопросу определения термина «тестирование», 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.

1.3. 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 [2].

1.3.1. 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, метод может представить треугольник со сторонами 3, 4, 5 неразносторонним, 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 языка Java. Для построения исчерпывающего теста нужно использовать все множество правильных программ на Java (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 распределенных вычислений), то дело обстоит еще хуже. В таких программах исполнение команды (например, задание, запрос в базу данных, выполнение расчета) 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).

1.3.2. 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 исполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь в детали, укажем лишь, что исчерпывающему входному тестированию может быть поставлено в соответствие исчерпывающее тестирование маршрутов. Подразумевается, что программа проверена полностью, если с помощью тестов удается 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 повторяющих друг друга маршрутов в программе – астрономическое. Чтобы убедиться в этом, рассмотрим представленный на рис. 1 граф передач управления в простейшей программе. 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 **WHILE** (или DO WHILE), который исполняется не менее 20 раз (на рисунке показан темным цветом). Внутри цикла имеется несколько операторов **IF** (на рисунке соответствующие узлы графа изображены пустыми кружками). Для того чтобы определить число неповторяющихся маршрутов при исполнении программы, подсчитаем число неповторяющихся маршрутов из точки **A** в **B** в предположении, что все приказы взаимно независимы. Это число вычис-

ляется как сумма $5^{20} + 5^{19} + ... + 5^1 = 10^{14}$, или 100 триллионам, где 5 – число путей внутри цикла. Поскольку большинству людей 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.

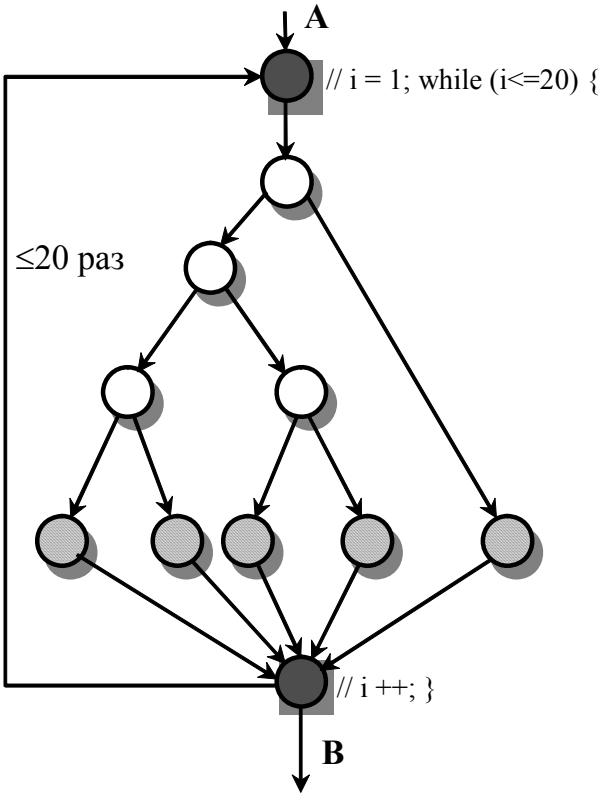


Рис. 1. Граф передач управления небольшой программы

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. 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 программа 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. Вопрос выбора методов тестирования и их описание будет рассмотрен в дальнейшем.

1.4. Принципы тестирования

Сформулируем основные принципы тестирования, используя главную предпосылку настоящей главы о том, что наиболее важными в тестировании программ являются вопросы психологии [4]. 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 компо-

ненты: **описание входных данных** и **описание точного и корректного результата**, соответствующего набору входных данных.

Необходимость этого подчеркивал логик Копи в работе [3]: «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* подвергается 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.

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.

Этот принцип следует из того факта, что тестирование – это деструктивный процесс. После выполнения конструктивной части при проектировании и написании программы программисту трудно быстро (в течение одного дня) перестроиться на деструктивный образ мышления.

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 вчера их наклеивали. И вам не придет в голову срывать их, если где-то они чуть-чуть неровно легли на стену. Вот так же и большинство программистов не могут эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки.

Это действительно сильный психологический фактор при коллективной разработке. Программист, тщательно отлаживающий программу, невольно 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.

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 разработчиком. Тогда существует вероятность, что к тестированию разработчик 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.

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.

В некоторых организациях подобная практика существует, но только на этапах комплексной отладки. Подобный способ тестирования чрезвычайно сложно реализовать из-за организационных трудностей.

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 испытуемые 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 недостаточно тщательного анализа результатов первого тестового прогона.

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, 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.

Эту проблему почти полностью решают современные инструментальные средства тестирования, однако, она перешла в область организации труда разработчика.

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

Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле A больше, чем в модуле B. Справедли-

вость этого принципа подтверждается еще и тем, что для ошибок свойственно располагаться в программе в виде неких скоплений. 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.

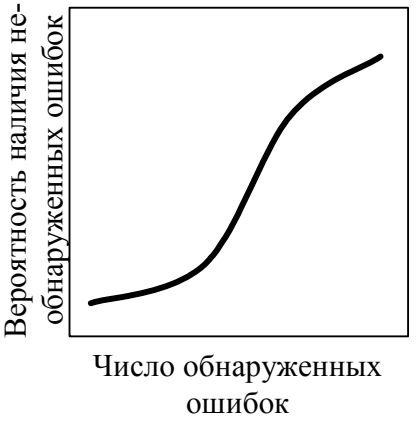


Рис. 2. Неожиданное соотношение числа оставшихся и числа обнаруженных ошибок

Интуитивно понятно, что ошибки могут группироваться в частях программы (модулях), разрабатываемых программистами низкой квалификации, или в модулях, в которых слабо проработана общая идея. Раннее выявление таких модулей – залог эффективного процесса тестирования.

Преимущество рассматриваемого принципа заключается в том, что он позволяет ввести обратную связь в 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 будут обсуждаться методы построения хороших наборов тестов, но применение этих методов должно быть творческим.

1.5. Контрольные вопросы и задания

1. Сформулируйте характеристики хорошего теста.
2. Придумайте, каким образом два программиста, создающие одну программу, могут протестировать ее, не нарушая принципов тестирования.
3. Покажите и докажите, что абсолютное тестирование невозможно на конкретной программе.

4. Дайте определение понятию «тестирование».
5. Перечислите принципы тестирования.
6. Поясните, почему тестирование является деструктивным процессом.
7. Повторите эксперимент с кроссвордом, описанный в пункте 1.2.
8. Почему при тестировании необходимо желать, чтобы программа дала сбой?
9. Почему в участке кода, где было обнаружено больше всего ошибок, может содержаться еще большее их количество?
10. К чему ведет планирование теста в предположении отсутствия ошибок?
11. Приведите пример, когда отсутствие тестов, проверяющих, не делает ли программа лишних действий (что она не должна делать), может повлечь ошибки в работе?
12. Что дает проверка программы на неправильных входных данных?
13. Зачем необходимо хранить тесты, если программа уже выпущена?
14. В чем заключается принцип тестирования программы как белого ящика?
15. В чем заключается принцип тестирования программы как черного ящика?

2. ТИПЫ ОШИБОК И РУЧНЫЕ МЕТОДЫ ТЕСТИРОВАНИЯ

«Отчего у нас никогда нет времени сделать что-либо хорошо,
но всегда находится время на переделку?»

Бытует мнение, что первая программная ошибка была обнаружена на заре развития ЭВМ, когда в Массачусетском технологическом институте окончилась неудачей попытка запуска машины Whirlwind I («Вихрь I»). Неистовая проверка монтажа, соединений и оборудования не выявила никаких неисправностей. Наконец, уже отчаявшись, решили проверить программу, представляющую собой маленькую полоску бумажной ленты. И ошибка была обнаружена именно в ней – в этом программистском ящике Пандоры[1], из которого на будущие поколения программистов обрушились беды, связанные с ошибками программ.

Задача любого тестировщика заключается в нахождении наибольшего количества ошибок, поэтому он должен хорошо знать наиболее часто допускаемые ошибки и уметь находить их за минимально короткий период времени. Остальные ошибки, которые не являются типовыми, обнаруживаются только тщательно созданными наборами тестов. Однако, из этого не следует, что для типовых ошибок не нужно составлять тесты.

Далее будет дана классификация ошибок, что поможет сосредоточить наши усилия в правильном направлении.

2. 1. Классификация ошибок

Для классификации ошибок мы должны определить термин «ошибка».

> Ошибка – это расхождение между вычисленным, наблюдаемым и истинным, заданным или теоретически правильным значением [7].

Такое определение понятия «ошибка» не является универсальным, так как оно больше подходит для понятия «программная ошибка». В технологии программирования существуют не только программные ошибки, но и ошибки, связанные с созданием программного продукта, например, ошибки в документации программы. Отличие программы и программного продукта достаточно четко определены в [8]. Но нас пока будут интересовать программные ошибки.

Итак, *по времени появления* ошибки можно разделить на три вида:

- **структурные ошибки набора**;
- **ошибки компиляции**;
- **ошибки периода выполнения**.

---

[1] Пандора – в древнегреческой мифологии девушка, созданная из земли и воды богом огня и кузнечного ремесла Гефестом. Она получила от верховного бога Зевса ящик со всеми человеческими несчастьями, которые случайно выпустила, приоткрыв из любопытства крышку; отсюда «ящик Пандоры» – источник всяческих бедствий.

Структурные ошибки возникают непосредственно при наборе программы. Что это за ошибки? Если кто-то работал в среде разработки Microsoft Visual Basic, то он знает, что если набрать оператор **If**, затем сравнение и нажать на клавишу **Enter**, не набрав слова **Then**, то Visual Basic укажет, что возникла ошибка компиляции. Это не совсем верно, так как компиляция в Visual Basic происходит только непосредственно при выполнении команды программы. В данном случае мы имеем дело именно со структурной ошибкой набора.

Данный тип ошибок определяется либо при наборе программы (самой IDE (**I**ntegrated **D**evelopment **E**nvironment) – интегрированной средой разработки) или при ее компиляции, если среда не разделяет первые два типа ошибок.

К данному типу ошибок относятся такие как: несоответствие числа открывающих скобок числу закрывающих, отсутствие парного оператора (например, **try** без **catch**), неправильное употребление синтаксических знаков и т. п.

Во многих средах разработки программного обеспечения данный тип ошибок объединяется со следующим типом, так как раннее определение ошибок вызывает некоторое неудобство при наборе программ (скажем, я задумал что-то написать, а потом вспомнил, что в начале пропустил оператор, тогда среда разработки может выдать мне ошибку при попытке перейти в другую строку).

Еще раз нужно отметить, что данный тип ошибок достаточно уникален и выделяется в отдельный тип только некоторыми средами разработки программного обеспечения.

Ошибки компиляции возникают из-за ошибок в тексте кода. Они включают ошибки в синтаксисе, неверное использование конструкций языка (оператор **else** в операторе **for** и т. п.), использование несуществующих объектов или свойств, методов у объектов.

Среда разработки (компилятор) обнаружит эти ошибки при общей компиляции приложения и сообщит о **последствиях** этих ошибок. Необходимо подчеркнуть слово «последствия» – это очень важно. Дело в том, что часто, говоря об ошибках, мы не разделяем проявление ошибки и саму ошибку, хотя это и не одно и то же. Например, ошибка «неопределенный класс» не означает, что класс не определен. Он может быть неподключенным, так как не подключен пакет классов.

Ошибки периода выполнения возникают, когда программа выполняется и компилятор (или операционная система, виртуальная машина) обнаруживает, что оператор делает попытку выполнить недопустимое или невозможное действие. Например, деление на ноль. Предположим, имеется такое выражение:

```
ratio = firstValue / sum.
```

Если переменная sum содержит ноль, то деление – недопустимая операция, хотя сам оператор синтаксически правилен. Прежде, чем программа обнаружит эту ошибку, ее необходимо запустить на выполнение.

Хотя данный тип ошибок называется «ошибками периода выполнения», это не означает, что ошибки находятся только после запуска программы. Вы можете выполнять программу в уме и обнаружить ошибки данного типа, однако, понятно, что это крайне неэффективно.

Если проанализировать все типы ошибок согласно первой классификации, то можно прийти к заключению, что при тестировании приходится иметь дело с ошибками периода выполнения, так как первые два типа ошибок определяются на этапе кодирования.

В теоретической информатике программные ошибки классифицируют *по степени нарушения логики на*:

- **синтаксические**;
- **семантические**;
- **прагматические**.

Синтаксические ошибки заключаются в нарушении правописания или пунктуации в записи выражений, операторов и т. п., т. е. в нарушении грамматических правил языка. В качестве примеров синтаксических ошибок можно назвать:

- пропуск необходимого знака пунктуации;
- несогласованность скобок;
- пропуск нужных скобок;
- неверное написание зарезервированных слов;
- отсутствие описания массива.

Все ошибки данного типа обнаруживаются компилятором.

Семантические ошибки заключаются в нарушении порядка операторов, параметров функций и употреблении выражений. Например, параметры у функции **add** (на языке Java) в следующем выражении указаны в неправильном порядке:

```
GregorianCalendar.add(1, Calendar.MONTH).
```

Параметр, указывающий изменяемое поле (в примере – месяц), должен идти первым. Семантические ошибки также обнаруживаются компилятором.

Надо отметить, что некоторые исследователи относят семантические ошибки к следующей группе ошибок.

Прагматические ошибки (или логические) заключаются в неправильной логике алгоритма, нарушении смысла вычислений и т. п. Они являются самыми сложными и крайне трудно обнаруживаются. Компилятор может выявить только следствие прагматической ошибки (см. выше пример с делением на ноль, компилятор обнаружит деление на ноль, но когда и почему переменная sum стала равна нулю – должен найти программист).

Таким образом, после рассмотрения двух классификаций ошибок можно прийти к выводу, что на этапе тестирования ищутся прагматические ошибки периода выполнения, так как остальные выявляются в процессе программирования.

На этом можно было бы закончить рассмотрение классификаций, но с течением времени накапливался опыт обнаружения ошибок и сами ошибки, некоторые из которых образуют характерные группы, которые могут тоже служить характерной классификацией.

Ошибка адресации – ошибка, состоящая в неправильной адресации данных (например, выход за пределы участка памяти).

Ошибка ввода-вывода – ошибка, возникающая в процессе обмена данными между устройствами памяти, внешними устройствами.

Ошибка вычисления – ошибка, возникающая при выполнении арифметических операций (например, разнотипные данные, деление на нуль и др.).

Ошибка интерфейса – программная ошибка, вызванная несовпадением характеристик фактических и формальных параметров (как правило, семантическая ошибка периода компиляции, но может быть и логической ошибкой периода выполнения).

Ошибка обращения к данным – ошибка, возникающая при обращении программы к данным (например, выход индекса за пределы массива, не инициализированные значения переменных и др.).

Ошибка описания данных – ошибка, допущенная в ходе описания данных.

2.2. Первичное выявление ошибок

В течение многих лет большинство программистов убеждено 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» [9]. 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 («ручное тестирование»), являющийся по сути первичным обнаружением ошибок. 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 ме-

тоды 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). Некоторые из таких методов приводятся в работах [10] и [11].

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.

Кроме того, скептицизм связан с тем, что это «первобытный метод». Да, сейчас стоимость машинного времени очень низка, а стоимость труда программиста, тестировщика высока и ряд руководителей пойдут на все, чтобы сократить расходы. Однако, есть другая сторона ручного тестирования – при тестировании за компьютером причины ошибок выявляются только в программе, а самая глубокая их причина – мышление программиста, как правило, не претерпевает изменений, при ручном же тестировании, программист глубоко анализирует свой код, попутно выявляя возможные пути его оптимизации, и изменяет собственный стиль мышления, повышая квалификацию. Таким образом, можно прийти к выводу, что ручное тестирование можно и нужно проводить на первичном этапе, особенно, если нет прессинга времени и бюджета.

2.3. 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.

Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Эти методы развиты из идей Вейнберга [9]. 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. 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 ранее принципами (в разделе 1), обычно не эффективными при тестировании собственной программы. Фактически «инспекция» и «сквозной 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 является, обычно, точное определение природы ошибок. 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.

Ранее, более двух десятков лет, проводились широкие эксперименты по применению этих методов, которые 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, было экспериментально установлено, что при проведении инспекций и сквозных просмотров определяются в среднем 38 % общего числа ошибок в учебных программах [12]. При использовании инспекций исходного текста в фирме IBM эффективность обнаружения ошибок составляла 80 % [13] (в данном случае имеется в виду не 80 % общего числа ошибок, поскольку, как отмечалось ранее, 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 [14]. Кроме того, можно было бы утверждать, что ручное тестирование «морально устаре-

ло», но если обратить внимание на список типовых ошибок, то они до сих пор остались прежними и увеличит ли скорость тестирования ЭВМ не всегда очевидно. Но то, что эти методы стали совсем непопулярными – это факт. Бесспорно, что каждый метод хорош для своих типов ошибок и сочетание методов ручного тестирования и тестирования с применением ЭВМ для конкретной команды разработчиков представляется наиболее эффективным подходом; эффективность обнаружения ошибок уменьшится, если тот или иной из этих подходов не будет использован.

Наконец, хотя методы ручного тестирования 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.

2.3.1. Инспекции исходного текста

Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении (чтении) текста группой специалистов [15]. При рассмотрении инспекций исходного текста внимание будет сосредоточено в основном на методах, процедурах, формах выполнения и т. д.

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 до заседания. 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 чле-

ны группы. Этот феномен известен давно и часто его применяют для решения проблем. Когда решение неочевидно, то объяснение проблемы другому человеку заставляет разработчика «разложить все по полочкам» и решение «само приходит» к разработчику.

2. Программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования.

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. Можно даже вести учет типов ошибок, на основании которого следует проводить дополнительную стажировку программиста в слабых областях.

В большинстве примеров описания процесса инспектирования утверждается, что во время инспекционного заседания ошибки не должны корректироваться. Однако существует и другая точка зрения [16]: «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... 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 строк 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 [9]; 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 [1]).

2.3.2. 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 ин-

спектирования, 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 эта процедура 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 адресованы 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.

2.3.3. 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 [1], 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.

2.4. 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 [17]) 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 различными авторами. За основу взят список Майерса [18] и дополнен автором после многолетнего изучения ошибок программного обеспечения, разработанного как лично, так и другими специалистами, а также учебных программ. В значительной мере он является независимым от

языка; это означает, что большинство ошибок встречается в любом языке программирования. Любой специалист может дополнить этот список вопросами, позволяющими выявить ошибки, специфичные для того языка программирования, который он использует, и обнаруженные им в результате выполнения процесса инспектирования.

2.4.1. Ошибки обращения к данным

Сводный список вопросов таков:

1. Используются ли переменные с неустановленными значениями?

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, атрибуту в классе) попытайтесь неформально «доказать», что ей присвоено значение в проверяемой точке.

2. Лежат ли индексы вне заданных границ?

Не выходит ли значение каждого из индексов за границы, определенные для соответствующего измерения при всех обращениях к массиву, вектору, списку и т. п.?

3. Есть ли нецелые индексы?

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

4. Есть ли «подвешенные» обращения?

Создан ли объект (выделена ли память) для всех обращений с помощью указателей или переменных-ссылок на объект (или память)? Наличие, переменных-ссылок представляет собой ошибку типа «подвешенного обращения». Она возникает в ситуациях, когда время жизни указателя больше, чем время жизни объекта/памяти, к которому/ой производится обращение. Например, к такому результату приводит ситуация, когда указатель задает локальную переменную в теле метода, значение указателя присваивается выходному параметру или глобальной переменной, метод 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/объект существует.

Этот тип ошибок характерен для языка Си или C++, где широко используются ссылки и указатели. Язык Java в этом отношении более развит, например, при потере всех ссылок на объект, объект переходит в «кучу мусора», где автоматически освобождается память из-под объекта

(объект удаляется) специальным сборщиком мусора. Последние изменения в языке C++, выполненные командой разработчиков Microsoft, которые преобразовали этот язык в C#, реализуют похожий механизм.

5. Соответствуют ли друг другу определения структуры и ее использование в различных методах?

Если к структуре данных обращаются из нескольких методов или процедур, то определена ли эта структура одинаково в каждой процедуре и используется ли она корректным способом?

6. Превышены ли границы строки?

Не превышены ли границы строки при индексации в ней? Существуют ли какие-нибудь другие ошибки в операциях с индексацией или при обращении к массивам по индексу?

2.4.2. Ошибки описания данных

Сводный список вопросов таков:

1. Все ли переменные описаны?

   Все ли переменные описаны явно? Отсутствие явного описания не обязательно является ошибкой (например, Visual Basic допускает отсутствие описания), но служит потенциальным источником беспокойства. Так, если в подпрограмме на Visual Basic используется элемент массива и отсутствует его описание (например, в операторе DIM), то обращение к массиву может вызвать ошибку (например, X = A(12)), так как по умолчанию, массив определен только на 10 элементов. 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? При разработке больших программных изделий неявное описание данных (описание данных по умолчанию) зачастую запрещают методически (если это не запрещено языком), чтобы упростить поиск ошибок при комплексной отладке.

2. Правильно ли инициализированы объекты, массивы и строки?

   Если начальные значения присваиваются переменным в операторах описания, то правильно ли инициализируются эти значения? Правильно ли создаются объекты, используется ли соответствующий конструктор?

3. Понятны ли имена переменных?

   Наличие переменных с бессмысленными именами (например, i и j) не является ошибкой, но является объектом пристального внимания. Классически i и j являются цикловыми переменными, а вот названий типа t125 следует избегать, так как возможна путаница имен.

4. Нельзя ли обойтись без переменных со сходными именами?

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

5. Корректно ли произведено описание класса?

Правильно ли происходит описание атрибутов и методов класса? Имеются ли методы или атрибуты, которые по смыслу не подходят к данному классу? Не является ли класс громоздким? Наличие положительных ответов на эти вопросы указывает на возможные ошибки в анализе и проектировании системы.

2.4.3. Ошибки вычислений

Сводный список вопросов таков:

1. Производятся ли вычисления с использованием данных разного типа?

Существуют ли вычисления, использующие данные разного типа? 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 важно как для языков с сильной типизацией (например, Ada, Java), так и для языков со слабой типизацией (например, C++, хотя он тяготеет к сильной типизации). Например, для языка Java код

```
byte a, b, c;
…
c = a + b;
```

может вызвать ошибку, так как операция «сложение» преобразует данные к типу int, и результат может превысить максимально возможное значение для типа byte. Таким образом, важным для вычислений с использованием различных типов данных является явное или неявное преобразование типов. Ошибки, связанные с использованием данных разных типов являются одними из самых распространенных.

2. Производятся ли вычисления неарифметических переменных?
3. Возможно ли переполнение или потеря промежуточного результата при вычислении?

Это означает, что конечный результат может казаться правильным, но промежуточный результат может быть слишком большим или слишком малым для машинного представления данных. Ошибки могут возникнуть даже если существует преобразование типов данных.

4. Есть ли деление на ноль?

Классическая ошибка. Требует проверки всех делителей на неравенство нулю. Следствием данной ошибки является либо сообщение «деление на ноль», либо «переполнение», если делитель очень

близок к нулю, а результат не может быть сохранен в типе частного (превышает его).

5. Существуют ли неточности при работе с двоичными числами?
6. Не выходит ли значение переменной за пределы установленного диапазона?

   Может ли значение переменной выходить за пределы установленного для нее логического диапазона? Например, для операторов, присваивающих значение переменной probability (вероятность), может быть произведена проверка, будет ли полученное значение всегда положительным и не превышающим единицу. Другие диапазоны могут зависеть от области решаемых задач.
7. Правильно ли осуществляется деление целых чисел?

   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, какое действие – умножение или деление – выполняется первым.

2.4.4. Ошибки при сравнениях

Сводный список вопросов таков:

1. Сравниваются ли величины несовместимых типов? Например, число со строкой?
2. Сравниваются ли величины различных типов?

   Например, переменная типа int с переменной типа long? Каждый язык ведет себя в этих случаях по-своему, проверьте это по его описанию. Как выполняются преобразования типов в этих случаях?
3. Корректны ли отношения сравнения?

   Иногда возникает путаница понятий «наибольший», «наименьший», «больше чем», «меньше чем».
4. Корректны ли булевские выражения?

   Если выражения очень сложные, имеет смысл преобразовать их или проверять обратное утверждение.
5. Понятен ли порядок следования операторов?

   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?
6. Понятна ли процедура разбора компилятором булевских выражений?

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

```
if ((x != 0) && ((y/x) > z))
```

является приемлемым для Java (т. е. компилятор заканчивает проверку, как только одно из выражений в операции И окажется ложным), но это выражение может привести к делению на ноль при использовании компиляторов других языков.

2.4.5. Ошибки в передачах управления

Сводный список вопросов таков:

1. Может ли значение индекса в переключателе превысить число переходов? Например, значение переключателя для оператора select case.
2. Будет ли завершен каждый цикл?

   Будет ли каждый цикл, в конце концов, завершен? Придумайте неформальное доказательство или аргументы, подтверждающие их завершение. Хотя иногда бесконечные циклы не являются ошибкой, но лучше их избегать.
3. Будет ли завершена программа? Будет ли программа, метод, модуль или подпрограмма в конечном счете завершена?
4. Существует ли какой-нибудь цикл, который не выполняется из-за входных условий?

   Возможно ли, что из-за входных условий цикл никогда не сможет выполняться? Если это так, то является ли это оплошностью?
5. Есть ли ошибки отклонения числа итераций от нормы?

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

2.4.6. Ошибки интерфейса

Сводный список вопросов таков:

1. Равно ли число входных параметров числу аргументов?

   Равно ли число параметров, получаемых рассматриваемым методом, числу аргументов, ему передаваемых каждым вызывающим методом? Правилен ли порядок их следования? Первый тип ошибок может обнаруживаться компилятором (но не для каждого языка), а вот правильность следования (особенно, если параметры одинакового типа) является важным моментом.
2. Соответствуют ли единицы измерения параметров и аргументов?

   Например, нет ли случаев, когда значение параметра выражено в градусах, а аргумента – в радианах? Или ошибки связанные с размерностью параметра/аргумента (например, вместо тонн передаются килограммы).
3. Не изменяет ли метод аргументы, являющиеся только входными?
4. Согласуются ли определения глобальных переменных во всех использующих их методах?

2.4.7. Ошибки ввода-вывода

Сводный список вопросов таков:

1. Правильны ли атрибуты файлов? Не происходит ли запись в файлы read-only?
2. Соответствует ли формат спецификации операторам ввода-вывода? Не читаются ли строки вместо байт?
3. Соответствует ли размер буфера размеру записи?
4. Открыты ли файлы перед их использованием?
5. Обнаруживаются ли признаки конца файла?
6. Обнаруживаются ли ошибки ввода-вывода? Правильно ли трактуются ошибочные состояния ввода-вывода?
7. Существуют ли какие-нибудь текстовые ошибки в выходной информации?

   Существуют ли смысловые или грамматические ошибки в тексте, выводимом программой на печать или экран дисплея? Все сообщения программы должны быть тщательно проверены.

2.5. Контрольные вопросы и задания

1. Дайте определение термина «ошибка».
2. Приведите классификацию ошибок по времени их появления.
3. Приведите классификацию ошибок по степени нарушения логики.
4. Какие ошибки (в разных классификациях) бывают в программах на языке C++ и когда они появляются?
5. Какие языки обнаруживают ошибки структурного набора?
6. Определите вид ошибки: if((x>3) && (x<2)) …
7. Какие типовые ошибки встречаются в программах?
8. В чем заключается сущность инспекции?
9. Какие этапы включает метод сквозного просмотра программы?
10. Приведите пример ошибки обращения к данным.
11. Приведите пример ошибки описания данных.
12. Приведите пример ошибки интерфейса.
13. Приведите пример ошибки передачи управления.
14. Приведите пример ошибки при сравнениях.
15. Приведите пример ошибки вычисления.
16. Приведите пример ошибки ввода-вывода.

3. СТРАТЕГИИ ТЕСТИРОВАНИЯ БЕЛОГО И ЧЕРНОГО ЯЩИКА

«Отлаженная программа – это программа, для которой пока еще не найдены такие условия, в которых она окажется неработоспособной»
Огден
Из неопубликованных заметок

Автор так и не смог найти первоисточник идей методов «белого» и «черного» ящика (black-box, white-box). Но каждый, кто сталкивается с тестированием, первое что слышит – это метод черного и метод белого ящика. И хотя их общая идея проста как все гениальное, но то, что на самом деле это не два метода, а классы методов или стратегии, удивляет даже специалистов.

В данной главе будут рассмотрены классические методы, которые относятся к этим двум стратегиям. Это методы, которые предназначены для тестирования не программного комплекса в целом, а для тестирования, прежде всего, программного кода. Понимание данных методов позволит вам оценивать остальные методы с точки зрения полноты тестирования и подхода к тестированию.

Наверное, вы помните из гл. 1 результаты психологических исследований, которые 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). Поэтому **главной целью** любой стратегии проектирования является уменьшение этой «неполноты» тестирования настолько, насколько 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 подмножеству.

В данной главе рассматриваются несколько подходов, которые позволяют более разумно выбирать тестовые данные. В первой главе 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 является стратегия, излагаемая в этой главе. Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе черного ящика, а затем дополнить его проверкой логики программы (т. е. с привлечением методов стратегии белого ящика).

Все методологии, обсуждаемые в настоящей главе можно разделить на следующие [1]:

стратегии черного ящика:

- эквивалентное разбиение;
- анализ граничных значений;
- применение функциональных диаграмм;
- предположение об ошибке;

стратегии белого ящика:

- покрытие операторов;
- покрытие решений;
- покрытие условий;
- покрытие решений/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 приведу 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 стратегию черного ящика, а затем как необходимое условие – дополнительные тесты, используя методы белого ящика.

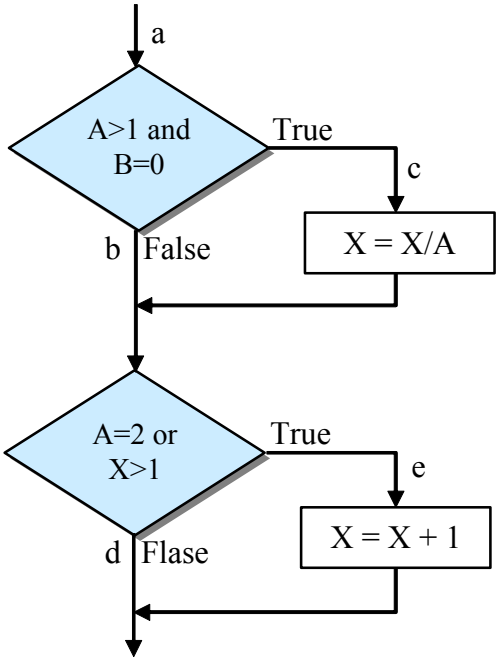
3.1. Тестирование путем покрытия логики программы

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

3.1.1. Покрытие операторов

Если отказаться полностью от тестирования всех путей, то можно показать, что критерием покрытия является **выполнение каждого оператора программы, по крайней мере, один раз**. Это *метод покрытия операторов*. 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. 3). Предположим, что на рис. 3 представлена небольшая программа, которая должна быть протестирована.



Эквивалент на языке Java

```
// Точка a
if((A>1) && (B == 0))
    X = X/A; // Точка c
// Точка b
if((A==2 || (X>1))
    X++; // Точка e
// Точка d
```

Рис. 3. Блок-схема небольшого участка программы, который должен быть протестирован

Можно выполнить каждый оператор, записав один-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*» (в первом условии вместо “&&” стоит “||”). Тогда при тестировании с помощью данного критерия эта ошибка не будет обнаружена. Пусть второе решение записано в программе как X > 0 (во втором операторе условия); 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.

3.1.2. Покрытие решений

Более сильный критерий покрытия логики программы (и метод тестирования) 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 **while** или **if**.

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 (например, в программах на языке Ассемблера); данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Третье исключение – операторы внутри **switch**-конструкций; выполнение каждого направления перехода не обязательно будет вызывать выполнение всех **case**-единнц. 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 каждую **case**-единицу) должно 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 (как для **case**-единиц). 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. 3, 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.

3.1.3. Покрытие условий

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 (и методом) 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 **switch**-единицам должно быть передано управление при вызове, по крайней мере, один раз.

Программа на рис. 3 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 ≤ l, B = 0 и B ≠ 0 в точке *a* и A = 2, A ≠ 2, X > 1 и X ≤ 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.

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 тело условия. 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).

3.1.4. Покрытие решений/условий

Очевидным следствием из этой дилеммы является критерий, названный *покрытием решений/условий.* Он требует такого достаточного набора тестов, чтобы **все возможные результаты каждого условия в решении, 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 схему передач управления в коде, генерируемым компилятором языка, программы рис. 3.

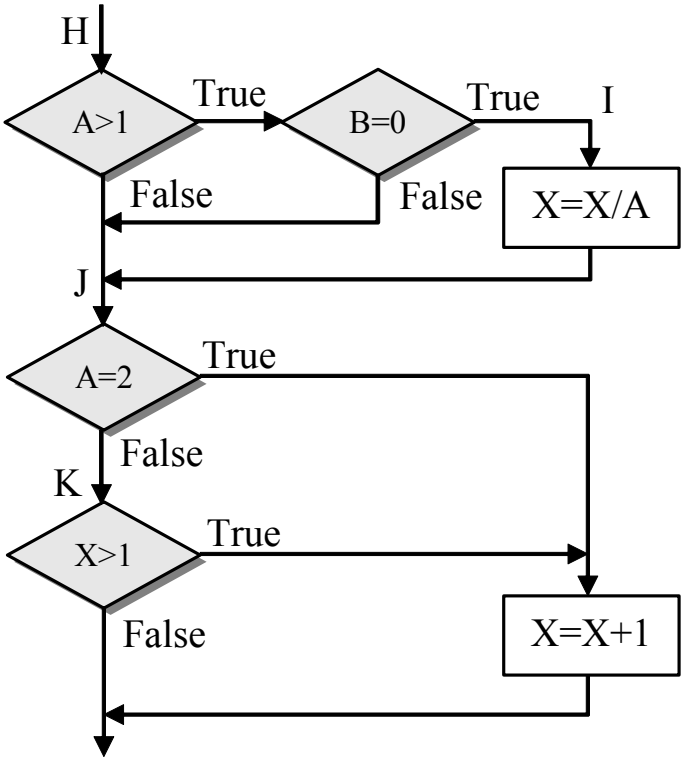


Рис. 4. Блок-схема машинного кода программы, изображенной на рис. 3

Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство компьютеров не

имеет команд, реализующих решения со 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 выполнения результата *ложь* решения H и результата *истина* решения K. 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 K.

Причина этого заключается в том, что, как показано на рис. 4, 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.

3.1.5. Комбинаторное покрытие условий

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**.

По этому критерию для программы на рис. 3 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, необязательно 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. 3, 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) && (z == 0) && end)
    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;
2) передает управление каждой точке входа (например, точке входа, **case**-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.

3.2. Стратегии черного ящика

3.2.1. Эквивалентное разбиение

В главе 1 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:

- уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
- 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. § 1.1) является набор «трех равных чисел, имеющих целые значения, большие нуля». Определяя этот набор как класс эквивалентности, устанавливают, 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;
2) построение тестов.

3.2.1.1. 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. 5.

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности

Рис. 5. Форма таблицы для перечисления классов эквивалентности

Заметим, что различают два типа классов эквивалентности: *правильные классы эквивалентности*, представляющие правильные входные данные программы, и *неправильные классы эквивалентности*, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). Таким образом, придерживаются одного из принципов тестирования 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 99»), то определяются один правильный класс эквивалентности (1 ≤ значение целого данного ≤ 99) и два неправильных (значение целого данного <1 и значение целого данного >99).
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 должности ИНЖЕНЕР, ТЕХНИК, НАЧАЛЬНИК ЦЕХА, ДИРЕКТОР»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «БУХГАЛТЕР»).
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.

3.2.1.2. 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 (1-999)». 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.

3.2.1.3. Пример

Предположим, что при разработке интерпретатора для подмножества языка Бейсик требуется протестировать синтаксическую проверку оператора DIM [1]. Спецификация приведена ниже. (Этот оператор не является полным оператором DIM Бейсика; спецификация была значительно сокращена, что позволило сделать ее «учебным примером». Не следует думать, что тестирование реальных программ так же легко, как в этом примере.) 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 DIM используется для определения массивов, форма оператора DIM:

DIM *ad*[,*ad*]…,

где *ad* есть описатель массива в форме

n(*d*[,*d*]...),

n – символическое имя массива, а *d* – индекс массива. Символические 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 DIM может быть продолжен на нескольких строках. (Конец спецификации.)

Первый шаг заключается в том, чтобы идентифицировать входные условия и по ним определить классы эквивалентности (табл. 1). Классы эквивалентности в таблице обозначены числами в круглых скобках.

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

DIM A(2)

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

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

Эти классы эквивалентности покрываются 18 тестами. Можно, 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.

Таблица 1

Классы эквивалентности

Входные условия	**Правильные классы эквивалентности**	**Неправильные классы эквивалентности**
Число описателей массивов	Один (1), больше одного (2)	Ни одного (3)
Длина имени массива	1–6(4)	0(5), больше 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), больше 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	От -65534 до 65535 (24)	Меньше -65534 (25), больше 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), больше 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)	

3.2.2. 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).

Достаточно трудно описать принимаемые решения при анализе 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 существует 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 от –1.0 до +1.0, то нужно 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 первое правило для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет $0.00, а максимум – $1165.25, то построить тесты, которые вызывают расходы с $0.00 и $1165.25. 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 второе правило для каждого выходного условия. Например, если система информационного поиска отображает на экране наиболее релевантные статьи 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 первой главе. 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 ≥ 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 [1].

Пусть имеется программа или модуль, которая сортирует различную информацию об экзаменах. Входом программы является файл, названный results.txt, 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 студента; 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. 6.

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, математическое ожидание (медиану) и дисперсию (среднеквадратическое отклонение) качества ответов;

4) отчет, упорядоченный по номерам вопросов и показывающий процент студентов, отвечающих правильно на каждый вопрос.

(Конец спецификации)

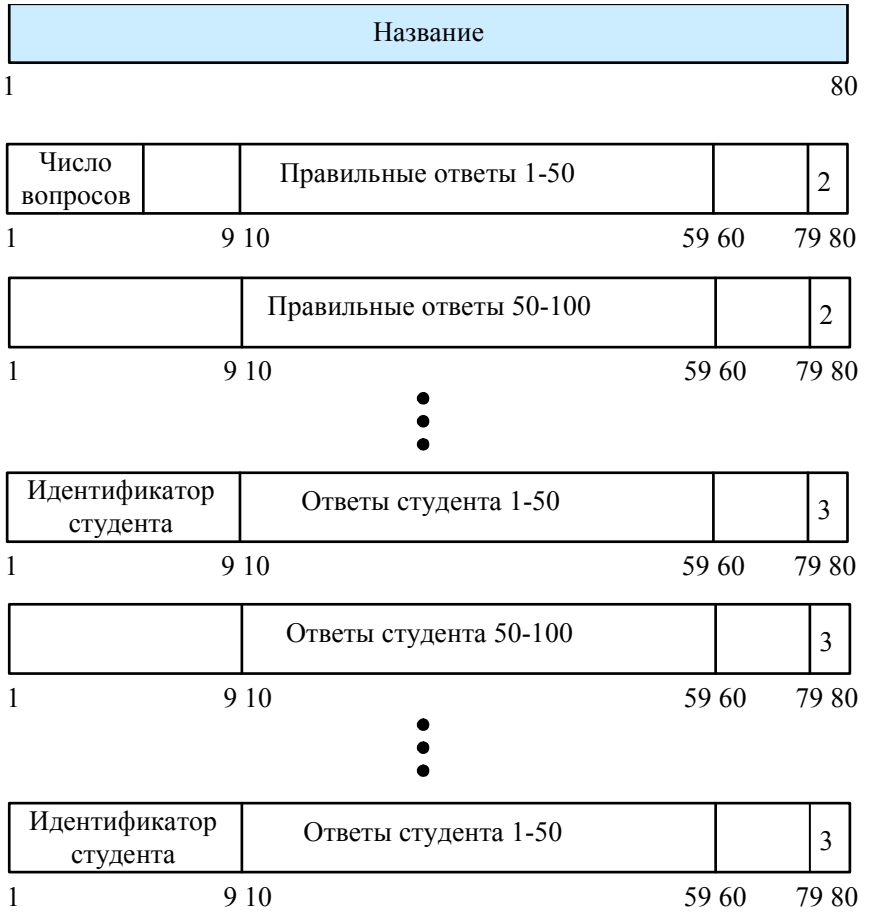


Рис. 6. Структуры входных записей для программы

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

ветах; 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 – 50, 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.
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, среднеквадратическое отклонение).

33. Среднее значение максимально (качество ответов всех студентов наивысшее).
34. Среднее значение равно 0 (качество ответов всех студентов равно 0).
35. Среднеквадратическое отклонение равно своему максимуму (один студент получает оценку 0, а другой – 100).
36. Среднеквадратическое 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 тестировщик, 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. Необходимо понимать, что граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.

3.2.3. Применение функциональных диаграмм

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют *комбинаций* входных условий. Например, пусть программа из приведенного выше примера 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 булевской 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 программы приводит к обновлению основного файла, то изменение в нем и является преобразованием системы; подтверждающее сообщение было бы выходным условием. Причины и следствия определяют-

ся путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждым причине и следствию приписывается 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. 7. 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 «A» или «B», а в колонке 2 – цифрой. В этом случае файл обновляется. Если первый символ неправильный, то выдается сообщение X12, а если второй символ неправильный – сообщение X13.

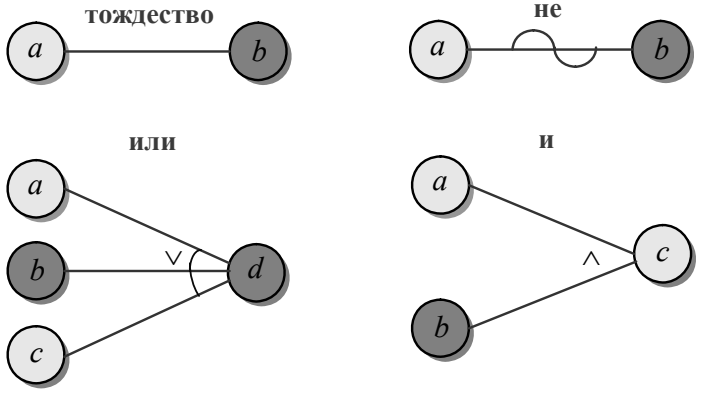


Рис. 7. Базовые логические отношения функциональных диаграмм

Причинами являются: 1 – символ «A» в колонке 1; 2 – символ «B» 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. 8. Отметим, что здесь создан промежуточный узел 11. Следует убедиться в том, что диаграмма действительно отображает данную спецификацию, задавая причинам все возможные значения и проверяя, принимают ли при этом следствия правильные значения. Рядом показана эквивалентная логическая схема.

Хотя диаграмма, показанная на рис. 8, 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 «A» и «B» одновременно).

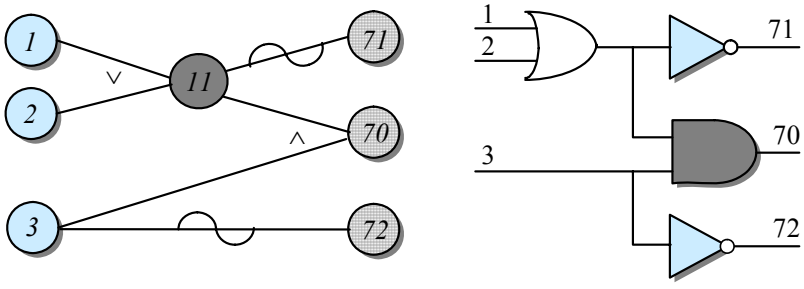


Рис. 8. Пример функциональной диаграммы и эквивалентной логической схемы

В этом случае используются дополнительные логические ограничения, изображенные на рис. 9.

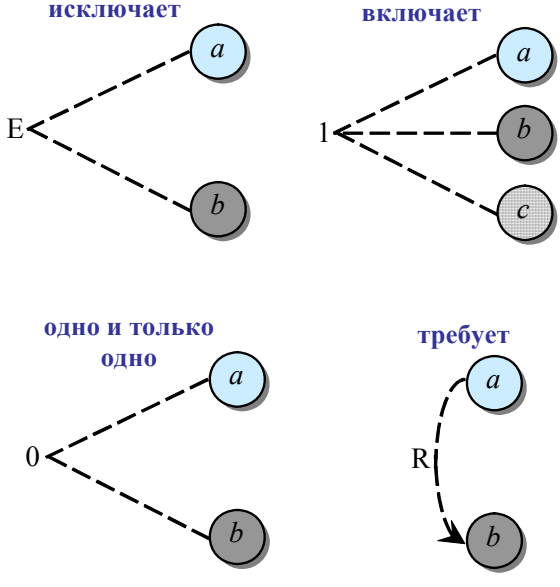


Рис. 9. Символы ограничений

Ограничение *E* устанавливает, что *E* должно быть истинным, если хотя бы одна из величин – *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 *0* 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* была равна 1, а *b – 0*).

Часто возникает необходимость в ограничениях для следствий. Ограничение *M* на рис. 10 устанавливает, что если следствие *a* имеет значение 1, то следствие *b* должно принять значение 0.

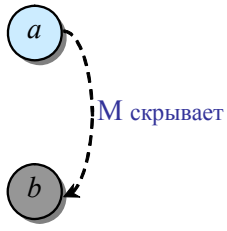


Рис. 10. Символ для «скрытого» ограничения

Как видно из рассмотренного выше примера, физически невозможно, чтобы причины 1 и 2 присутствовали одновременно, но возможно, чтобы присутствовала одна из них. Следовательно, они связаны ограничением *E* (рис. 11).

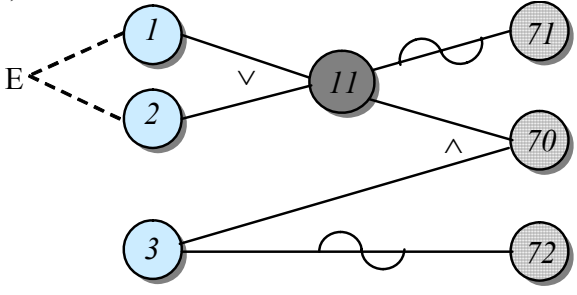


Рис. 11. Пример функциональной диаграммы с ограничением «исключает»

3.2.3.1. Замечания

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.

Метод функциональных диаграмм позволяет построить набор полезных тестов, однако его применение обычно не обеспечивает построение *всех* полезных тестов, которые могут быть определены. Кроме того, функциональная диаграмма неадекватно исследует граничные условия. Конечно, в процессе работы с функциональными диаграммами можно попробовать покрыть граничные условия. Однако при этом граф существенно усложняется, и число тестов становится чрезвычайно большим. Поэтому лучше отделить анализ граничных значений от метода функциональных диаграмм.

Поскольку функциональная диаграмма дает только направление в выборе определенных значений операндов, граничные условия могут входить в полученные из нее тесты.

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 поставляет их.

3.2.4. 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 практическим 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 спецификаций (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 из раздела 2.2, посвященного 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.

Надо отметить, что применение метода предположения об ошибке не является совсем неформальным и не поддающимся совершенствованию. С течением времени каждый программист, тестировщик увеличивает собственный опыт, который позволяет все больше и больше применять данный метод, кроме того, имеются методы совершенствования интуиции (математической, программистской) и догадки [6].

3.3. Стратегия

Методологии проектирования тестов, обсуждавшиеся в этой статье, 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. Однако, данный метод достаточно трудоемок.
2. В любом случае необходимо использовать анализ граничных значений. Напомню, 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.

3.4. Нисходящее и восходящее тестирование

Исследуем две возможные стратегии тестирования: *нисходящее* и *восходящее.* Прежде всего внесем ясность в терминологию. Во-первых, термины «нисходящее тестирование», «нисходящая разработка», «нисходящее проектирование» часто используются как синонимы. Действительно, два первых термина являются синонимами (в том смысле, что они

подразумевают определенную стратегию при тестировании и создании классов/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 классов/модулей. Рассмотрим различие между нисходящей и восходящей стратегиями.

3.4.1. Нисходящее тестирование

Нисходящее тестирование начинается с верхнего, головного класса (или модуля) программы. Строгой, корректной процедуры подключения очередного последовательно тестируемого класса не существует. Единственное правило, которым следует руководствоваться при выборе очередного класса, состоит в том, что им должен быть один из классов, методы которого вызываются классом, предварительно прошедшим тестирование.

Для иллюстрации этой стратегии рассмотрим рис. 12. Изображенная на нем программа состоит из двенадцати классов A-L. Допустим, что класс J содержит операции чтения из внешней памяти, а класс I – операции записи.

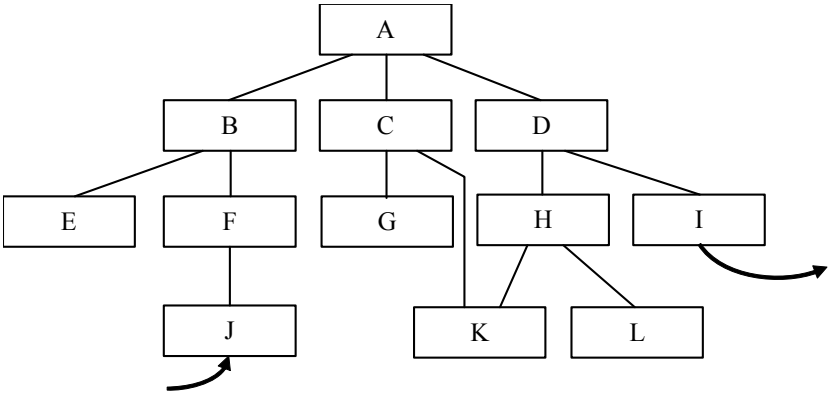


Рис. 12. Пример программы, состоящей из двенадцати классов

Первый шаг – тестирование класса A. Для его выполнения необходимо написать заглушки, замещающие классы B, C и D. К сожалению, часто неверно понимают функции, выполняемые заглушками. Так, порой можно услышать, что «заглушка» должна только выполнять запись сообщения, устанавливающего: «класс подключен» или «достаточно, чтобы заглушка существовала, не выполняя никакой работы вообще». В большинстве случаев эти утверждения ошибочны. Когда класс A вызывает метод класса B, A предполагает, что B выполняет некую работу, т. е. класс A получает результаты работы метода класса B (например, в форме значений выходных переменных). Когда же метод класса B просто воз-

вращает управление или выдает сообщение об ошибке без передачи в класс A определенных осмысленных результатов, класс A работает неверно не вследствие ошибок в самом классе, а из-за несоответствия ему заглушки. Более того, результат может оказаться неудовлетворительным, если ответ 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 методом класса именно в этом вызове, то вызывающий метод сработает как ошибочный (например, зациклится) или выдаст неверное выходное значение. Следовательно, создание 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 классу A? Ответ на этот вопрос не является совершенно очевидным, поскольку верхний класс в типичной программе сам не получает входных данных и не выполняет операций ввода-вывода. В верхний класс (в нашем случае, A) данные передаются через одну или несколько заглушек. Для иллюстрации допустим, что классы B, C и D выполняют следующие функции:
B – получает сводку о вспомогательном файле;
C – определяет, соответствует ли недельное положение дел установленному уровню;
D – формирует итоговый отчет за неделю.

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

С этой программой связана еще одна проблема. Поскольку метод класса A вызывает класс B, вероятно, один раз, нужно решить, каким образом передать в A несколько тестов. Одно из решений состоит в том, чтобы вместо B сделать несколько версий заглушки, каждая из которых имеет один фиксированный набор тестовых данных. Тогда для использования любого тестового набора нужно несколько раз исполнить программу, причем всякий раз с новой версией заглушки, замещающей B. Другой вариант решения – записать наборы тестов в файл, заглушкой читать их и передавать в класс A. В общем случае создание заглушки может быть более сложной задачей, чем в разобранном выше примере. Кроме того, часто из-за характеристик программы оказывается необходимым сообщать тестируемому классу данные от нескольких заглушек, заме-

щающих классы нижнего уровня; например, класс может получать данные от нескольких вызываемых им методов других классов.

После завершения тестирования класса A одна из заглушек заменяется реальным классом и добавляются заглушки, необходимые уже этому классу. Например, на рис. 13 представлена следующая версия программы.

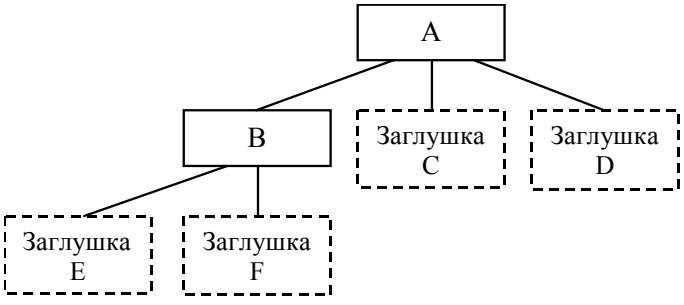


Рис. 13. Второй шаг при нисходящем тестировании

После тестирования верхнего (головного) класса тестирование выполняется в различных последовательностях. Так, если последовательно тестируются все классы, 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 класса A одним программистом может тестироваться последовательность A–B, другим – A–C, третьим – A–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, класс G), то целесообразно выбирать последовательность, которая включала бы эти части как можно раньше. Критическими могут быть сложный класс, класс с новым алгоритмом или класс со значительным числом предполагаемых ошибок (класс, склонный к ошибкам).
2. Классы, 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 дисплей. Если к программе подключается реальный класс, содержащий методы ввода, то представление тестов значительно упрощается. Форма их представления становится идентичной той, которая использу-

ется в реальной программе для ввода данных (например, из вспомогательного файла или ввод с клавиатуры). Точно так же, если подключаемый класс содержит выходные функции программы, то отпадает необходимость в заглушках, записывающих результаты тестирования. Пусть, например, классы J и I выполняют функции входа-выхода, а G содержит некоторую критическую функцию; 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. 14.

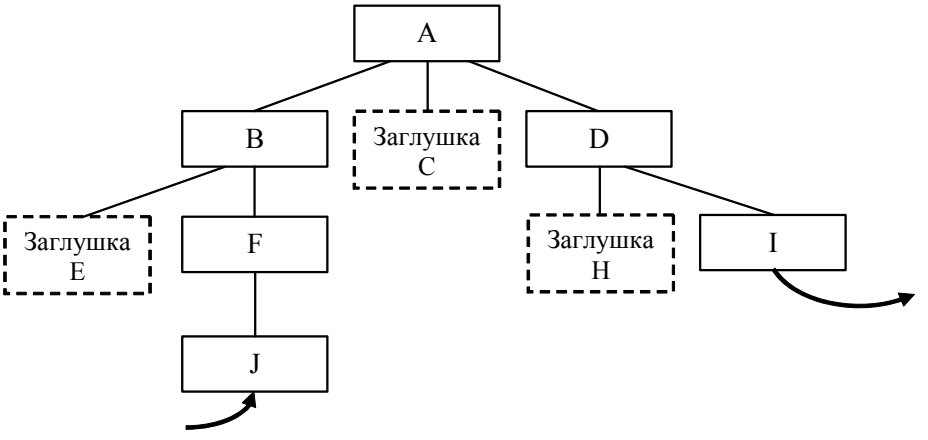


Рис. 14. Промежуточное состояние при нисходящем тестировании

По достижении стадии, отражаемой рис. 14, 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, безусловно, достоинства стратегии нисходящего тестирования.

Однако нисходящее тестирование имеет ряд серьезных недостатков. Пусть состояние проверяемой программы соответствует показанному на рис. 14. На следующем шаге нужно заменить заглушку самим классом H. Для тестирования этого класса требуется спроектировать (или они спроектированы ранее) тесты. Заметим, что все тесты должны быть представлены в виде реальных данных, вводимых через класс J. При этом создаются две проблемы. Во-первых, между классами H и J имеются промежуточные классы (F, B, A и D), поэтому может оказаться *невозможным* пе-

редать методу класса такой текст, который бы соответствовал каждой предварительно описанной ситуации на входе класса H. Во-вторых, даже если есть возможность передать все тесты, то из-за «дистанции» между классом H и точкой ввода в программу возникает довольно трудная интеллектуальная задача – оценить, какими должны быть данные на входе метода класса J, чтобы они соответствовали требуемым тестам класса H.

Третья проблема состоит в том, что результаты выполнения теста демонстрируются классом, расположенным довольно далеко от класса, тестируемого в данный момент. Следовательно, установление соответствия между тем, что демонстрируется, и тем, что происходит в классе на самом деле, достаточно сложно, а иногда просто невозможно. Допустим, мы добавляем к схеме рис. 14 класс E. Результаты каждого теста определяются путем анализа выходных результатов методов класса I, но из-за стоящих между классами E и I промежуточных классов трудно восстановить действительные выходные результаты методов класса E (т. е. те результаты, которые он передает в методы класса B).

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. 12, то может сложиться впечатление, что после проектирования двух верхних уровней следует перейти к кодированию и тестированию классов A и B, C и D и к разработке классов нижнего уровня. Как отмечается в работе [1], такое решение не является разумным. Проектирование программ — процесс итеративный, т. е. при создании классов, занимающих нижний уровень в архитектуре программы, может оказаться необходимым произвести изменения в классах верхнего уровня. Если же классы 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 класса до завершения тестирования предыдущего. Это объясняется двумя причинами: во-первых, трудно вставлять тестовые данные в заглушки и, во-вторых, классы верхнего уровня используют ресурсы классов нижнего уровня. Из рис. 12 видно, что тестирование класса A может потребовать несколько версий заглушки класса B. Программист, тестирующий программу, как правило, решает так: «Я сразу не буду полностью тестировать A – сейчас это трудная задача. Когда подключу класс J, станет легче представлять тесты, и уж тогда я вернусь к тестированию класса A». Конечно, здесь важно только не забыть проверить оставшуюся часть класса тогда, когда это предполагалось сделать. Аналогичная проблема возникает в связи с тем, что классы верхнего уровня также запрашивают ресурсы для использования их классами нижнего уровня (например, открывают файлы). Иногда трудно оп-

ределить, корректно ли эти ресурсы были запрошены (например, верны ли атрибуты открытия файлов) до того момента, пока не начнется тестирование использующих их классов нижнего уровня.

3.4.2. 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 классов (т. е. классов, не использующих методы других классов). Как и ранее, здесь нет такой процедуры для выбора класса, тестируемого на следующем шаге, которой бы отдавалось предпочтение. Единственное правило состоит в том, чтобы очередной класс использовал уже оттестированные классы.

Если вернуться к рис. 12, то первым шагом должно быть тестирование нескольких или всех классов E, J, G, K, L и I последовательно или параллельно. Для каждого из них требуется свой драйвер, т. е. программа, которая содержит фиксированные тестовые данные, вызывает тестируемый класс 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 класс несколько раз. В большинстве случаев драйверы проще разработать, чем заглушки.

Как и в предыдущем случае, на последовательность тестирования влияет критичность природы класса. Если мы решаем, что наиболее критичны классы D и F, то промежуточное состояние будет соответствовать рис. 15. Следующими шагами могут быть тестирование класса E, затем класса B и комбинирование B с предварительно оттестированными классами E, F, J.

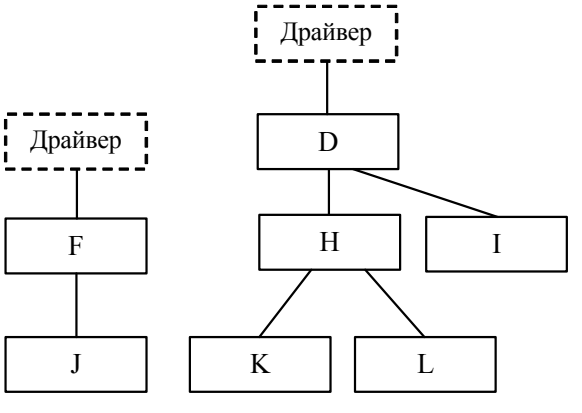


Рис. 15. Промежуточное состояние при восходящем тестировании

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 класс (в примере класс A), и это уже готовая программа. Хотя функции ввода-вывода могут быть проверены прежде, чем собрана вся программа (использовавшиеся классы ввода-вывода показаны на рис. 15), 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 классу, который тестируется, нет промежуточных классов, 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 классы нижнего уровня. Не существует также и трудностей с незавершенностью тестирования одного класса при переходе к тестированию другого, потому что при восходящем тестировании с применением нескольких версий заглушки нет сложностей с представлением тестовых данных.

3.4.3. Сравнение

В табл. 2 показаны относительные недостатки и преимущества нисходящего и восходящего тестирования (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 классах верхнего уровня или классах нижних уровней типичной программы. Поэтому при выборе стратегии целесообразно взвесить все пункты из табл. 2 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 единственно возможными при пошаговом подходе. В работе [10] рассматриваются еще три варианта стратегии тестирования.

Таблица 2

Сравнение нисходящего и восходящего тестирования

Преимущества	Недостатки
Нисходящее тестирование	
1. Имеет преимущества, если ошибки главным образом в верхней части программы. 2. Представление теста облегчается после подключения функции ввода-вывода. 3. Раннее формирование структуры программы позволяет провести ее демонстрацию пользователю и служит моральным стимулом.	1. Необходимо разрабатывать заглушки. 2. Заглушки часто оказываются сложнее, чем кажется вначале. 3. До применения функций ввода-вывода может быть сложно представлять тестовые данные в заглушки. 4. Может оказаться трудным или невозможным создать тестовые условия. 5. Сложнее оценка результатов тестирования. 6. Допускается возможность формирования представления о совмещении тестирования и проектирования. 7. Стимулируется незавершение тестирования некоторых классов/модулей.
Восходящее тестирование	
1. Имеет преимущества, если ошибки главным образом в классе/модуле нижнего уровня. 2. Легче создавать тестовые условия. 3. Проще оценка результатов.	1. Необходимо разрабатывать драйверы. 2. Программа как единое целое не существует до тех пор, пока не добавлен последний класс/модуль.

3.5. Проектирование и исполнение теста

Проектирование теста, как можно понять из вышеизложенного материала, может быть достаточно трудоемким процессом. Оно включает в себя следующие этапы:

1) задаться целью теста;
2) написать входные значения;
3) написать предполагаемые выходные значения;
4) выполнить тест и зафиксировать результат;
5) проанализировать результат.

От правильного подхода к каждому этапу зависит качество тестирования в целом. О проблеме неверной поставки цели говорилось в первой главе. Необходимость второго этапа не вызывает сомнений.

Третий этап позволит избежать неоднозначности на пятом этапе. Очень часто, при отсутствии описания, что должно получиться, пытаются «подогнать» логику рассуждений в анализе результатов. Кроме того, очень часто этот пункт требует формирования либо независимой оценки (критерия), либо альтернативного просчета по алгоритму. В первом случае очень легко контролировать общий результат, во втором – более де-

тально понять работу алгоритма. Бывают случаи, когда при ручном просчете предполагаемых выходных значений находят ошибки в логике работы программы.

Четвертый этап является практически механическим. На этом этапе не нужно думать, а только строго следовать предписанию и аккуратно фиксировать полученные значения.

Если исполнение теста приносит результаты, не соответствующие предполагаемым, то это означает, что либо имеется ошибка, 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. Также существуют программы позволяющие выполнять функции с набором параметров, которые варьируются в заданных пределах, что в общем случае, позволяет методом перебора проверить работу функции или метода.

При подготовке к тестированию модулей целесообразно еще раз пересмотреть психологические и экономические принципы, обсуждавшиеся в гл. 1. При исполнении теста следует обращать внимание на побочные эффекты, например, если метод 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. Поэтому при его исполнении наряду с предполагаемыми результатами необходимо проверить и эти переменные.

Во время тестирования возникают и психологические проблемы, связанные с личностью тестирующего. Программистам полезно поменяться кодом, чтобы не тестировать свой собственный. Так, программист, сделавший функцию вызова метода, является хорошим кандидатом для тестирования вызываемого метода. Заметим, что это относится только к тестированию, а не к отладке, которую всегда должен выполнять автор класса или модуля.

Не следует выбрасывать результаты тестов; представляйте их в такой форме, чтобы можно было повторно воспользоваться ими в будущем. Если в некотором подмножестве классов обнаружено большое число ошибок, то эти классы, по-видимому, содержат еще большее число необнаруженных ошибок. Такие классы 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, а в выявлении ошибок.

3.6. Контрольные вопросы и задания

1. Какие бывают стратегии тестирования?
2. Опишите процесс тестирования методом анализа граничных значений.
3. Опишите процесс тестирования методом эквивалентного разбиения.
4. Опишите процесс тестирования методом функциональных диаграмм.
5. Опишите процесс тестирования методом предположения об ошибке.
6. Опишите процесс тестирования методом покрытия операторов.
7. Опишите процесс тестирования методом покрытия условий.
8. Опишите процесс тестирования методом покрытия решений.
9. Опишите процесс тестирования методом покрытия решений/условий.
10. Опишите процесс тестирования методом комбинаторного покрытия условий.
11. В чем заключается метод восходящего тестирования?
12. В чем заключается метод нисходящего тестирования?
13. Сравните методы восходящего и нисходящего тестирования.
14. Составьте тесты методом покрытия операторов к участку программы
```
if ((C == 3) && (X > 0)) M = M/C;
if ((X > 2) && (M == 1)) M++;
```
15. Составьте тесты методом покрытия решений к участку программы
```
if ((C == 1) && (X < 0)) M = M/C;
if ((X > 2) && (M == 1)) M++;
```
16. Составьте тесты методом комбинаторного покрытия условий к участку программы
```
if ((C == 2) && (X > 1)) M = M/C;
if ((X > 5) || (M == 1)) M++;
```
17. Составьте тесты методом покрытия решений к участку программы
```
if ((C == 1) || (X < 0)) M = M/C;
if ((X > 2) || (M == 1)) M++;
```
18. Какие этапы входят в проектирование теста?

Список использованной литературы

1. Майерс Г. Искусство тестирования программ / Пер. с англ. под ред. Б. А. Позина. – М.: Финансы и статистика, 1982. – 176 с.
2. B.W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1980.
3. Copi I. M. Introduction to Logic. New York, Macmillan, 1968.
4. Weinberg G. M. The Psychology of Computer Programming. New York, Van Nostrand Reinhold, 1971.
5. Yourdon E. Techniques of program structure and design. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975. Русский перевод: Йодан Э. Структурное проектирование и конструирование программ. – М.: Мир, 1979. – 416 с.
6. Hetzel W.C. (ed.). Program test methods, Prentice-Hall, Inc., 1973.
7. Першиков В. И., Савинков В. М. Толковый словарь по информатике. – М.: Финансы и статистика, 1991. – 543 с.
8. Brooks F.B., The mythical man-month and other essays on software engineering. Anniversary edition. University of North Carolina at Chapel Hill, 1995. Русский перевод: Брукс Ф. Мифический человеко-месяц или как создаются программные системы: Пер. с англ. – СПб.: Символ-Плюс, 1999. – 304 с.
9. Weinberg G. M. The Psychology of Computer Programming. New York, Van Nostrand Reinhold, 1971.
10. Myers G. J. Software Reliability: Principles and Practices. New York, Wiley-Interscience, 1976. Русский перевод: Майерс Г. Надежность программного обеспечения. – М.: Мир, 1980.
11. Myers G. J. Composite/Structured Design. New York, Van Nostrand Reinhold, 1978.
12. Myers G. J. A Controlled Experiment in Programm Testing and Code Walkthroughs/Inspections. – Commun. ACM, 1978, 21(9), p. 760–768.
13. 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).
14. 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.
15. Fagan M. E. Design and Code Inspections to Reduce Errors in Program Development. – IBM Systems J., 1976, 15(3), p. 182–211.
16. Freeman R. D. An Experiment in Software Development. – The Bell System Technical Journal, Special Safeguard Supplement, 1975, p. 199–209.
17. Ascoly J. et al. Code Inspection Specification.– TR–21. 630, IBM System Communication Division, Kingston, N. Y., 1976.
18. IBM Developer Works [Электронный ресурс] / Diagnosing Java Code: Designing "testable" applications. / Eric E. Allen – Электрон. дан. 2001. – http://www-106.ibm.com/developerworks/java/library/j-diag0911.html, свободный. – Загл. с экрана. – Яз. англ.

СОДЕРЖАНИЕ