Лекция

Тема: Методы и средства организации тестирования

«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.

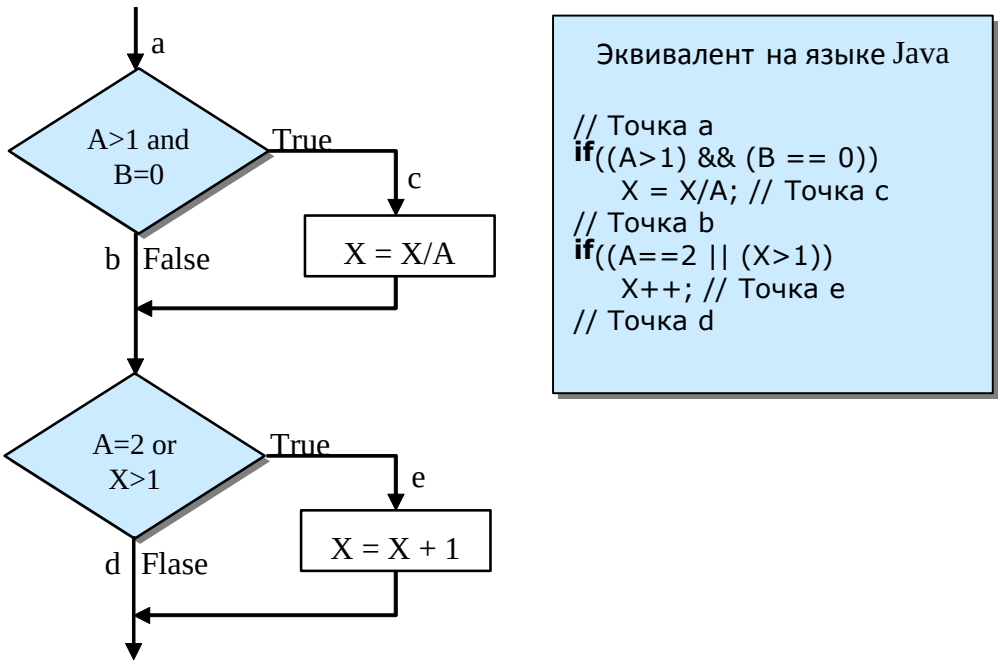


Рис. 3. Блок-схема небольшого участка программы, который должен быть протестирован

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.

Впрограмме, представленной на рис. 3, покрытиерешенийможет 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 изчетырехрезультатов 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.

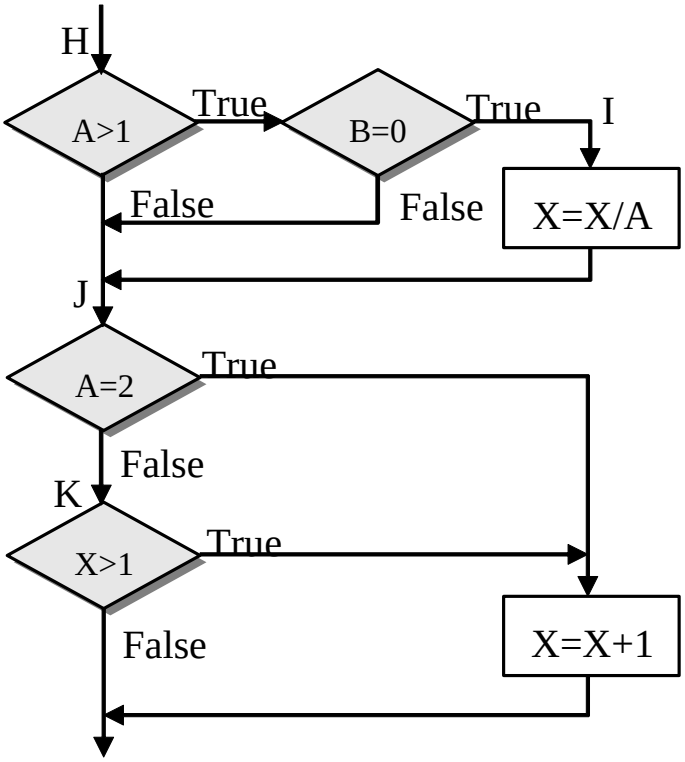


Рис. 4. Блок-схема машинного кода программы, изображенной на рис. 3

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 ≠ 0. 3. A ≤ 1, B = 0. 4. A ≤ l, B ≠ 0.

5. A = 2, X > 1. 6. A = 2, X ≤ l.

7. A ≠ 2, X > 1. 8. A ≠ 2, X ≤ l.

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 какимлибо 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.

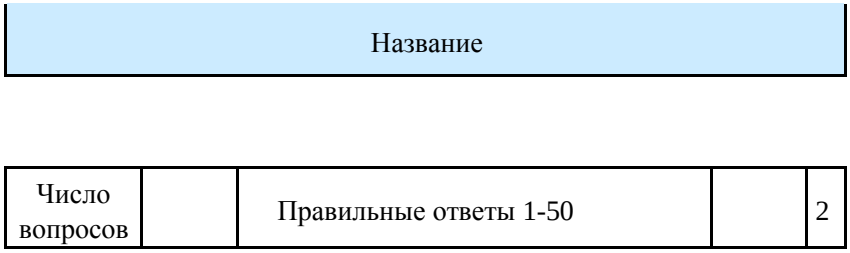
Чтобы проиллюстрировать необходимость анализа граничных значений, можно использовать программу анализа треугольника, приведенную в первой главе. Для задания треугольника входные значения должны быть целыми положительными числами, и сумма любых двух из них должна быть больше третьего. Если определены 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)

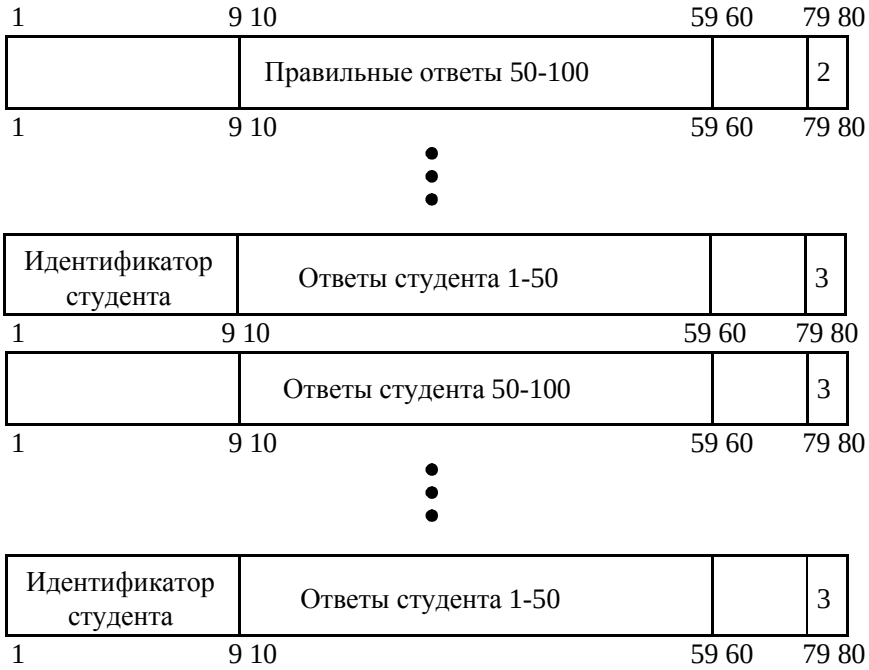


Рис. 6. Структуры входных записей для программы

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 чтолибо специфическое (т. е. необходимо много записей). Приемлемое разбиение вопросов на классы эквивалентности представляет разбиение на два подмножества: 1–50 и 51–999. Следовательно, необходимы тесты, где поле числа вопросов принимает значения 0, 1, 50, 51 и 999. Эти тесты покрывают большинство граничных условий для записей о правильных ответах; однако существуют три более интересные ситуации – отсутствие записей об ответах, наличие записей об ответах типа «много ответов на один вопрос» и наличие записей об ответах типа «мало ответов на один вопрос» (например, число вопросов – 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 причинноследственных 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 определяются 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.

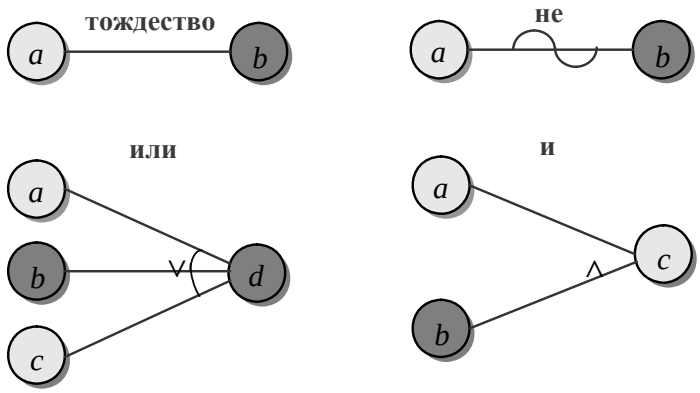


Рис. 7. Базовые логические отношения функциональных диаграмм

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).

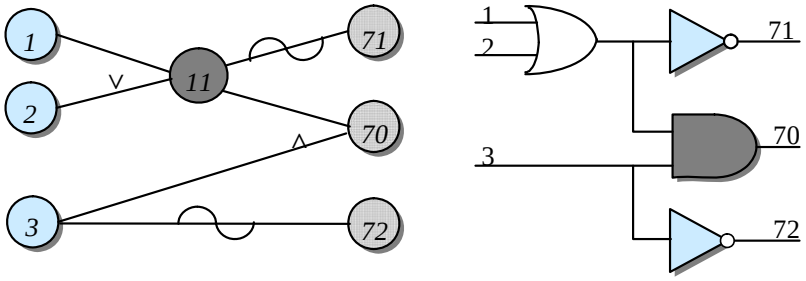
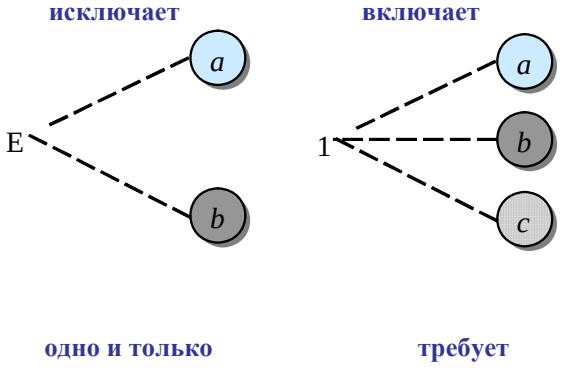


Рис. 8. Пример функциональной диаграммы и эквивалентной логической схемы

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



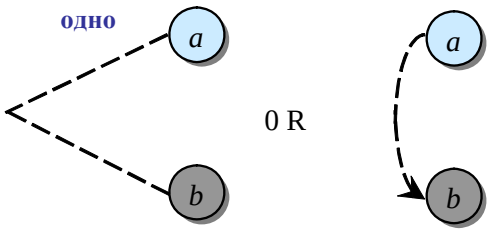


Рис. 9. Символы ограничений

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).

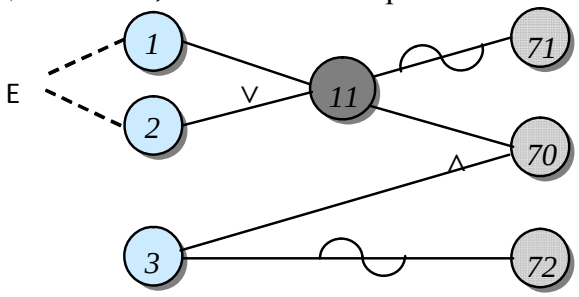


Рис. 11. Пример функциональной диаграммы с ограничением «исключает»

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.

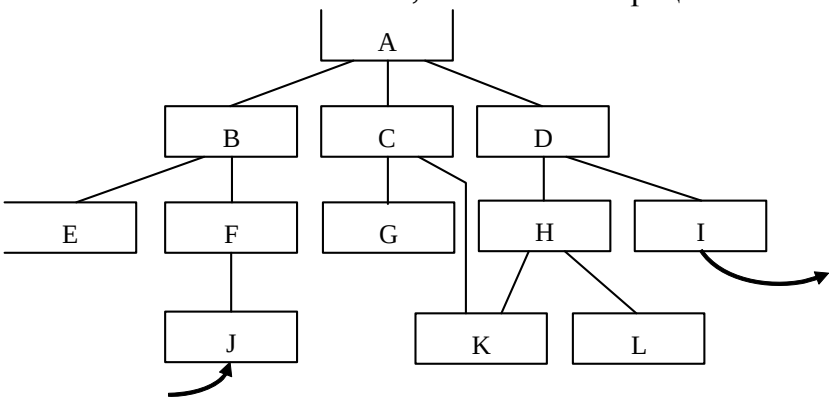


Рис. 12. Пример программы, состоящей из двенадцати классов

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 возвращает 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, замещающих 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.

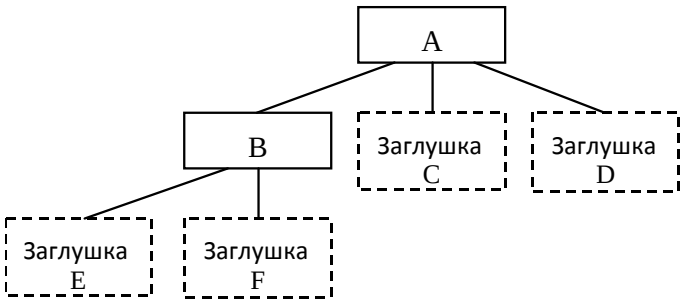


Рис. 13. Второй шаг при нисходящем тестировании

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 используется 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.

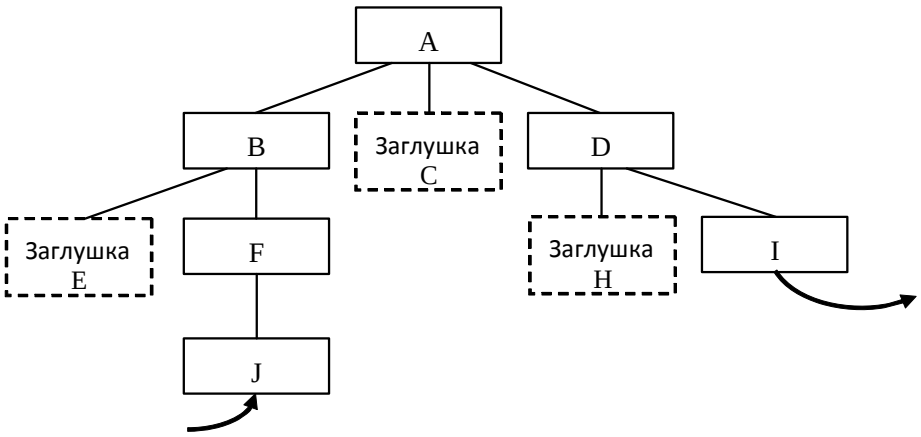


Рис. 14. Промежуточное состояние при нисходящем тестировании

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 вводавывода, 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* передать 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 определить, 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.

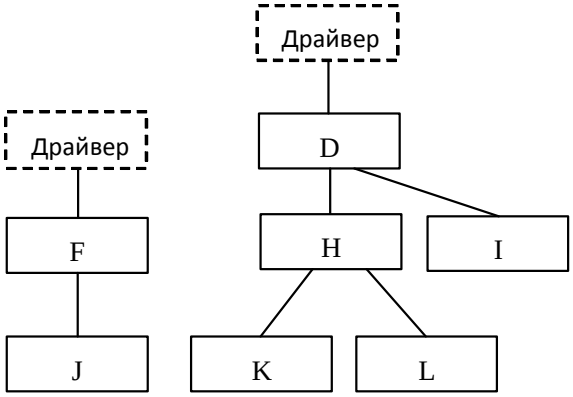


Рис. 15. Промежуточное состояние при восходящем тестировании

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 вводавывода 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 3. условия. 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 детально 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?