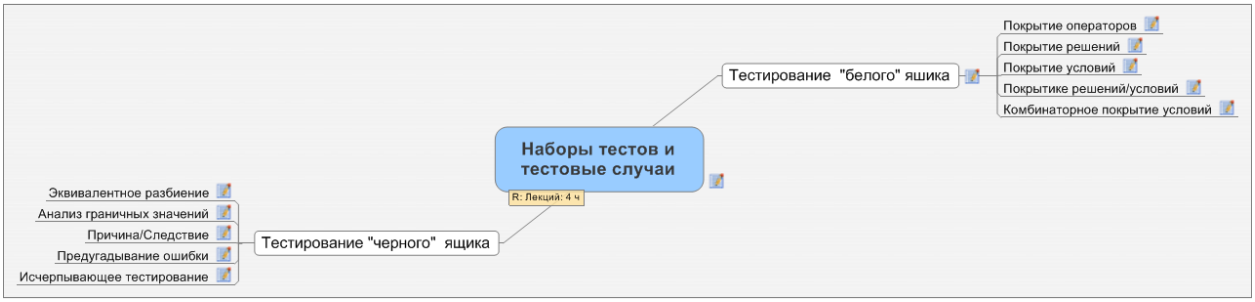
Лекции 14-15. Наборы тестов и тестовые случаи



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 тестовимеет малую вероятность быть оптимальным или близким к оптимальному подмножеству.

Мы рассмотрим несколько подходов, которые позволяют более разумно выбирать тестовые данные. Ранее отмечалось, что исчерпывающее тестирование по принципу "черного" или "белого" ящика в общем случае невозможно. Однако при этом подчеркивалось, что приемлемая стратегия тестирования может обладать элементами обоих подходов. Таковой является стратегия, описываемая ниже. 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 ниже, можно разделить на следующие:

стратегии "белого" ящика:

- покрытие операторов;
- покрытие решений;
- покрытие условий;
- покрытие решений/условий;
- комбинаторное покрытие условий

стратегии "черного" ящиика:

- эквивалентное разбиение;
- анализ граничных значений;
- анализ причин и следствий;
- предугадывание ошибки.
- исчерпывающее тестирование.

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).

Поэтому рекомендуемая процедура заключается в том, чтобы разрабатывать тесты, используя стратегию "черного" ящика, а затем как необходимое условие – дополнительные тесты, используя методы "белого" ящика.

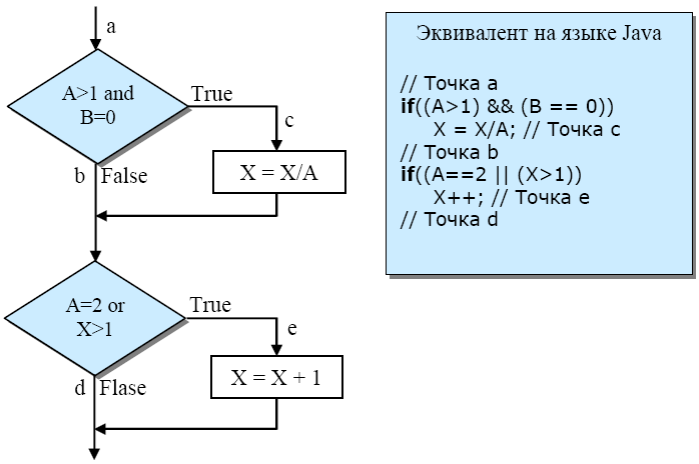
1 Тестирование "белого" яшика

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

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.

На рисунке представлена небольшая программа, котораядолжна быть протестирована.



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 быодин 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.

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; выполнениекаждого 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 рисунке, покрытие решений может бытьвыполнено 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.

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 тес- тами 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 рисунке 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.

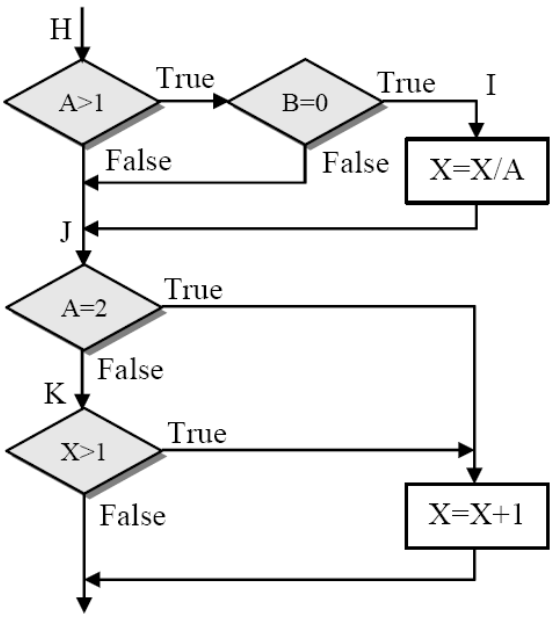
2. A = 2, B = 1, X = 1,

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).

1.4 Покрытике 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 край- ней 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 риунке схему передач управления в коде, генерируемым компилятором языка, рассматриваемой программы.



Блок-схема машинного кода программы, изображенной на предыдущем рисунке

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 рисунке, 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.

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 рассматриваеиой программы 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. $A \leq$ l, $B \neq 0$.

5. $A = 2$, $X > 1$. 6. $A = 2$, $X \leq$ l.

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

2 Тестирование "черного" ящика

2.1 Эквивалентное разбиение

Эквивалентное Разбиение (Equivalence Partitioning - EP). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.

2.2 Анализ граничных значений

Анализ Граничных Значений (Boundary Value Analysis - BVA). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям имеющим ограничения.

2.3 Причина/Следствие

Причина / Следствие (Cause/Effect - CE). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".

2.4 Предугадывание ошибки

Предугадывание ошибки (Error Guessing - EG). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.

2.5 Исчерпывающее тестирование

Исчерпывающее тестирование (Exhaustive Testing - ET) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.