# Программная инженерия Лекция 7. Методы генерации тест-кейсов

Составитель: Эверстов В.В.

Дата составления: 17/04/2014

Дата модификации: 17/04/2014

# Стратегии

- Стратегия «Черного ящика»
- Стратегия «Белого ящика»
- Но на самом деле это не стратегии и даже не методы, а классы методов, которые объединяют много разных методов тестирования.

# Ключевой вопрос тестирования

- *Какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?*
- Изучение методологий проектирования тестов дает ответ на этот вопрос.

# Методы тестирования

- Стратегия «Черного ящика»
  - эквивалентное разбиение;
  - анализ граничных значений;
  - применение функциональных диаграмм;
  - предположение об ошибке;
- Стратегия «Белого ящика»
  - покрытие операторов;
  - покрытие решений;
  - покрытие условий;
  - покрытие решений/условий.

# Рекомендации

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

# Стратегия «Черного ящика»

- Как мы уже говорили при использовании стратегии «Черного ящика» полный перебор всех всевозможных входных данных невозможно.
- Если ограничивать множество тестов, которым необходимо подвергнуть программный продукт, то мы должны выбрать только такие тесты, которые имеют наибольшую вероятность обнаружения ошибок.

# «Правильный» тест

- Правильно выбранный тест должен обладать следующими 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.

# Метод эквивалентного разбиения

- Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:
  - выделение классов эквивалентности;
  - построение тестов.

# Выделение классов эквивалентности

- Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп.
- Чтобы произвести данную операцию вам необходимо заполнить таблицу:

# Таблица

| Входные условия | Правильные классы эквивалентности | Неправильные классы эквивалентности |
|---|---|---|
| | | |

# Правила

- 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).
- 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).

# Правила

- 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»).
- 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).

# Правила

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

# Построение тестов

- Назначение каждому классу эквивалентности уникального номера.
- 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.
- 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.

# Пример

- Предположим, что при разработке интерпретатора для подмножества языка Бейсик требуется протестировать синтаксическую проверку оператора DIM
- Оператор DIM используется для определения массивов, форма оператора:

  DIM *ad*[,*ad*]…,
- где *ad* есть описатель массива в форме

  *n*(*d*[,*d*]...),
- *n* – символическое имя массива, а *d* – индекс массива.

# Пример

- 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.

# 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) | |

# Построение тестов

- Сперва, строим тест, покрывающий 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.

# Построение тестов

- 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)

# Метод анализа граничных условий

- 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:
  - Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
  - При разработке тестов рассматривают не только входные условия (пространство входов), но и *пространство результатов* (т. е. выходные классы эквивалентности).

# Метод анализа граничных условий

- Достаточно трудно описать принимаемые решения при анализе граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее существует несколько общих правил этого метода.

# Правила

- 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.
- 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.

# Правила

- 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.

# Правила

- 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.
- Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.
- Попробовать свои силы в поиске других граничных условий.

# Метод анализа граничных условий

- **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*.

# Недостатки

- Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют *комбинаций* входных условий.
- 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.

# Предположение об ошибке

- Некоторые люди обладают умением «выискивать» ошибки и без привлечения какой-либо методологии тестирования.
- 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.

# Предположение об ошибке

- 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.

# Покрытие Логики

- 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.

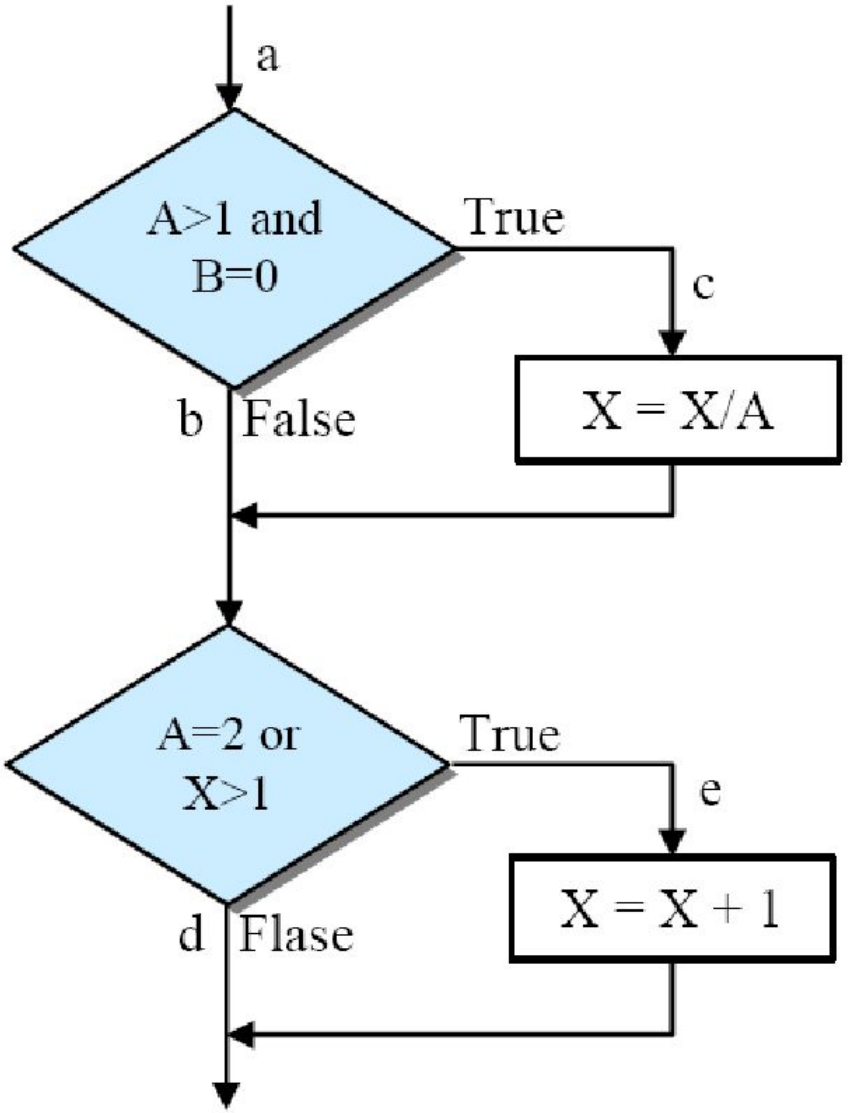
# Покрытие Логики

- Покрытие логики программы реализуется несколькими методами:
  - Покрытие операторов,
  - Покрытие решений,
  - Покрытие условий,
  - Покрытие условий/решений,
  - Комбинаторное покрытие условий.

# 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

# Покрытие операторов




Эквивалент на языке Java

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

# 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**.

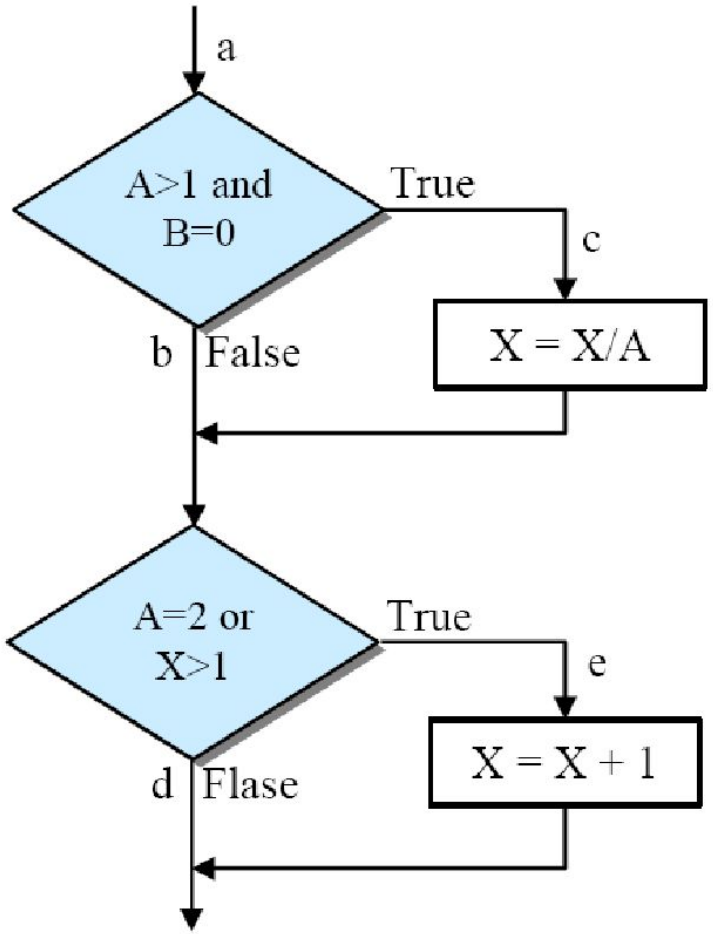
# Покрытие решений

- 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**.

# Покрытие решений

- 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.

# Покрытие решений




Эквивалент на языке Java

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

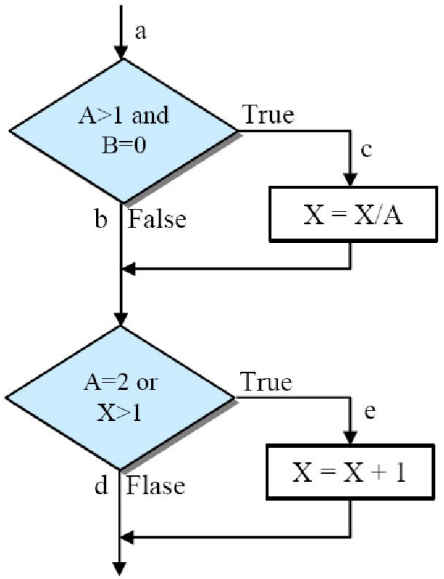
A = 3, B = 0, X = 3 и A = 2, B = 1, X = 1

# 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**.
- 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.

# Покрытие условий

- Программа 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:
  - A = 2, B = 0, X = 4 *ace*.
  - A = 1, B = 1, X = 1 *abd*.




Эквивалент на языке Java

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

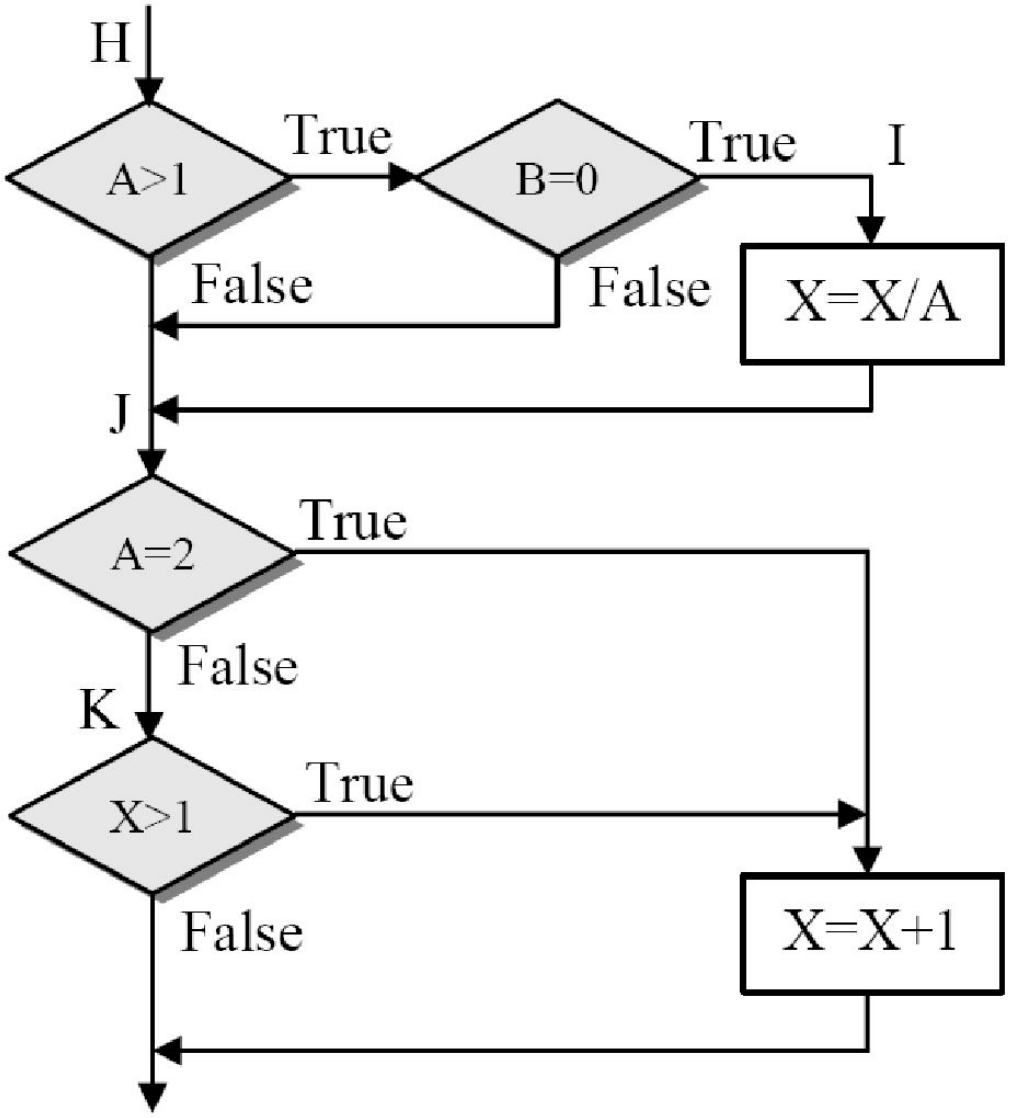
# Покрытие решений/условий

- 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**.

# Покрытие решений/условий

- 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.

# Покрытие условий

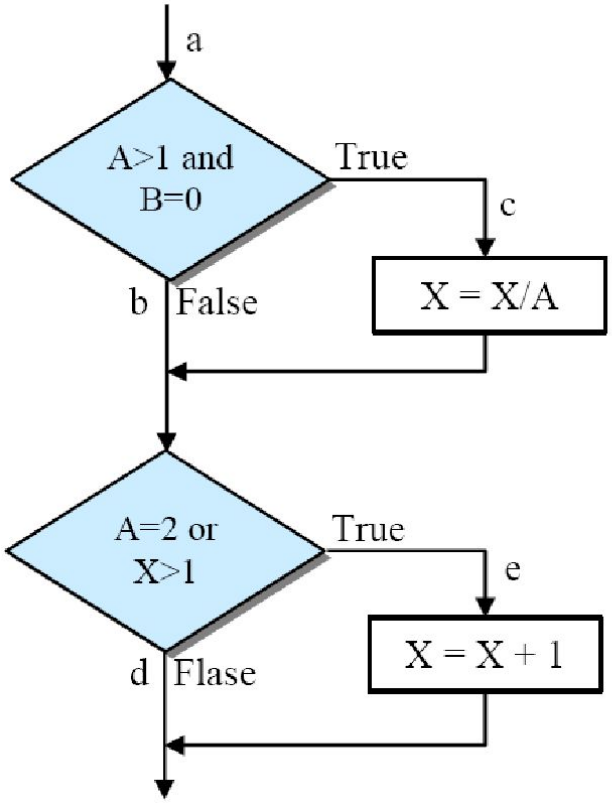


# Комбинаторное покрытие условий

- Требуется создания такого числа тестов, чтобы **все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз**.

# Комбинаторное покрытие условий

- A > 1, B = 0.
- A > 1, B ≠ 0.
- A ≤ 1, B = 0.
- A ≤ l, B ≠ 0.
- A = 2, X > 1.
- A = 2, X ≤ l.
- A ≠ 2, X > 1.
- A ≠ 2, X ≤ l.




Эквивалент на языке Java

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

# Комбинаторное покрытие условий

- 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.

# Комбинаторное покрытие условий

- Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого:
  - вызывает выполнение всех результатов каждого решения, по крайней мере, один раз;
  - 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.

# Стратегия Тестирования

- Если спецификация содержит комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм. Однако, данный метод достаточно трудоемок.
- В любом случае необходимо использовать анализ граничных значений. Этот метод включает анализ граничных значений входных и выходных переменных. Анализ граничных значений дает набор дополнительных тестовых условий.

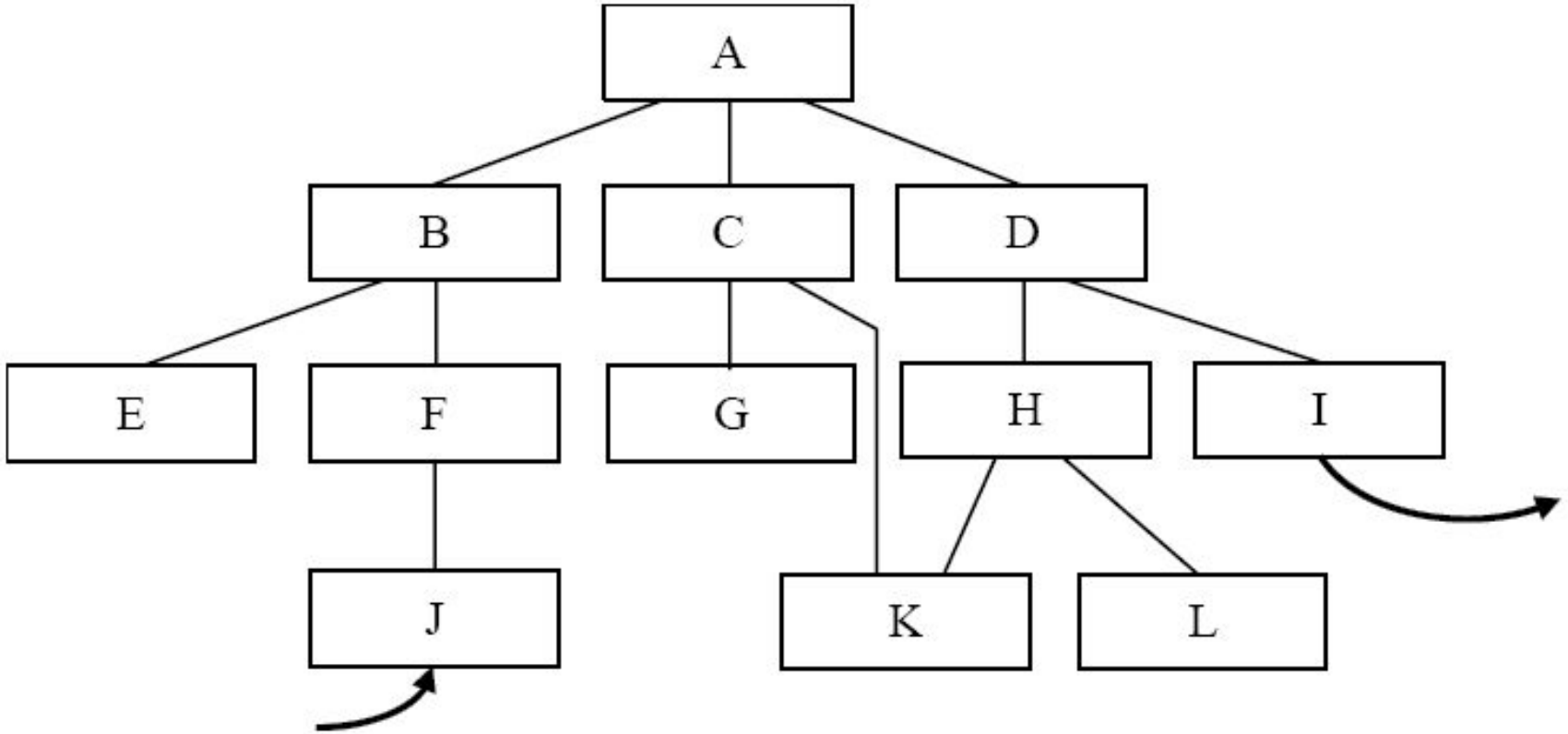
# Стратегия Тестирования

- Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.
- Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.
- 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).

# 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.

# Нисходящее тестирование



# Рекомендации

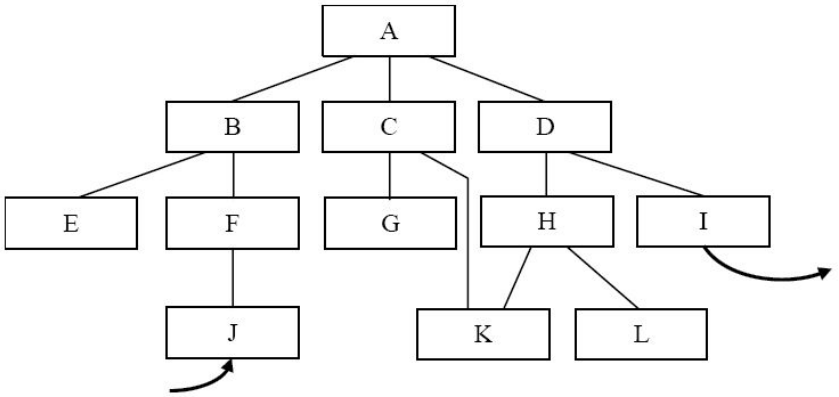
- В принципе нет такой последовательности, которой бы отдавалось предпочтение, но рекомендуется придерживаться двух основных правил:
  - Если в программе есть критические в каком-либо смысле части (возможно, класс G), то целесообразно выбирать последовательность, которая включала бы эти части как можно раньше. Критическими могут быть сложный класс, класс с новым алгоритмом или класс со значительным числом предполагаемых ошибок (класс, склонный к ошибкам).
  - Классы, включающие операции ввода-вывода, также необходимо подключать в последовательность тестирования как можно раньше.

# Восходящее тестирование

- 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.

# Восходящее тестирование

- Если вернуться к предыдущему примеру, 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).



# Сравнение

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

# ИТОГ

- Проектирование теста достаточно трудоемкий процесс. Оно включает в себя следующие этапы:
  - задаться целью теста;
  - написать входные значения;
  - написать предполагаемые выходные значения;
  - выполнить тест и зафиксировать результат;
  - проанализировать результат.