Лекция 5 Тестирование программы при стратегии белого ящика

При тестировании системы как белого - стеклянного ящика тестировщик имеет доступ не только к требованиям к системе, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и определять тем самым, на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, не покрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы - часто одна проблема нейтрализует другую, и они никогда не возникают одновременно.

Тестирование по стратегии белого ящика (англ. White-box testing) — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась

Тестирование по стратегии белого ящика, также называемое техникой тестирования, управляемой логикой программы, позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии, тестировщик получает тестовые данные путём анализа логики работы программы.

Техника Белого ящика включает в себя следующие методы тестирования:
Покрытие операторов
покрытие решений
покрытие условий
покрытие решений и условий
комбинаторное покрытие условий

1. ПОКРЫТИЕ ОПЕРАТОРОВ.

Критерий покрытия операторов подразумевает выполнение каждого оператора программы хотя бы один раз. Это необходимое ,но не достаточное условие для приемлемого тестирования.
Рассмотрим пример:

```
Void m (const float a, const float b, float x){
If( a>1) && (b==0)
X=X/a;
If (a==2)||(x>1)
X=X++;
}
```

Для приведённого фрагмента можно было бы выполнить каждый оператор 1-н раз, задав в качестве входных данных a=2,b=0,x=1, но при этом из 2-го условия следует, что X может принимать любое значение и оно не проверяется. Кроме того:
1 если при написании программы в 1-м условии написать a>1 || b=0, то ошибка обнаружена не будет.
2 если во 2-м условии вместо x>1 записано x>0 ,то ошибка тоже не будет обнаружена.
3 существует путь a в d ,в котором x вообще не меняется и ,если здесь есть ошибка, то она не будет обнаружена.

ПОКРЫТИЕ РЕШЕНИЙ.

Решением будем считать логическую функцию, предшествующую оператору.(a>1 && b=0 – решение). Для реализации этого критерия необходимо достаточное число тестов такое ,

что каждое решение на этих тестах принимает значение «истина» или «ложь» по крайней мере 1-н раз.

Не трудно показать , что критерий покрытия решений удовлетворяет критерию покрытия операторов, но является более сильным.

Приведенная программа может быть протестирована по методу покрытия решений двумя тестами, покрывающими , либо путь ace abc , либо acd и abe.

Если мы выбираем 2-е покрытие, то входами 2-ч тестов являются:

a=3 ,b=0, x=3

a=2 ,b=1, x=1

если во 2-м условии вместо x>1 записать x<1 , то ошибка не будет обнаружена двумя тестами.

ПОКРЫТИЕ УСЛОВИЙ.

Критерий покрытия условий более сильный по сравнению с предыдущим. В этом случае записывается число тестов достаточное для того , чтобы все возможные результаты каждого условия в решении были выполнены по крайней мере 1-н раз.

Однако как и случай покрытия решений это покрытие не всегда приводит к выполнению каждого оператора по крайней мере 1-н раз. К этому критерию требуется дополнение, заключающееся в том, что каждой точке входа управление должно быть передано по крайней мере 1-н раз.

Приведенная программа имеет 4-е условия:

a>1, b==0, a==2, x>1

необходимо реализовать ситуацию, где a>1, a<=1;b=0, b!=0; a==2, a!=2; x>1 ,x<=1;

тесты , удовлетворяющие этому условию.

1) *a=2, b=0, x=4 (ace)*
2) *a=1, b=1, x=1 (abd)*

когда количество тестов такое же как в случае покрытия решений, этот критерий может вызвать выполнение решения в условиях, не реализуемых при покрытии решений. То есть этот критерий в основном удовлетворяет критерию покрытия решений, но не всегда тесты покрытия условия для ранее рассмотренных примеров покрывает результаты всех решений, но это случайное совпадение.

Например тесты:

1) a=1, b=0, x=3
2) a=2, b=1, x=1

покрывают результаты всех условий , но только 2-а из 4-х результатов решений (не выполняется результат «истина» 1-го решения и результат «ложь» 2-го.).

Покрытие решений/условий.

Этот метод требует составить тесты так, чтобы все возможные результаты каждого условия выполнялись, по крайней мере, один раз и каждой точке управления передается, по крайней мере, один раз.

Недостатки метода.

1. Не всегда можно проверить все условия.
2. Невозможно проверить условия, которые скрыты другими условиями.
3. Недостаточная чувствительность к ошибкам в логических выражениях.

Комбинаторное покрытие условий.

Этот критерий требует создания такого числа тестов, чтобы в каждом решении все точки входа выполнялись, по крайней мере, один раз.

Для приведённого примера покрыть тестами 8 следующих комбинаций:

1. A>1, B=0; 2. A>1, B!=0; 3. A<=1, B=0; 4. A<=1, B!=0; 5. A=2, X>1; 6. A=2, X<=1; 7. A!=2, X>1; 8. A!=2, X<=1.

Эти комбинации можно проверить четырьмя тестами:

1. A=2, B=0, X=4 K: 1, 5 2. A=2, B=1, X=1 K: 2, 6
3. A=1, B=0, X=2 K: 3, 7 4. A=1, B=1, X=1 K: 4, 8

В данном случае то, что четырём тестам соответствуют четыре пути, является совпадением. Представленные тесты не покрывают всех путей (например, acd). Т.о. 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. Передаёт управление каждой точке входа хотя бы один раз (чтобы обеспечить выполнение каждого оператора, по крайней мере, один раз).

Для программ, содержащих решение, 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 точки входа, по крайней мере, один раз. Термин “возможных” употреблён здесь потому, что некоторые комбинации условий могут быть не реализуемы. Например, для комбинаций K<0, K>40 задать K невозможно.

Хотя метод и является достаточно мощным и позволяет находить достаточно большое количество ошибок, он имеет и недостатки:

не всегда можно проверить все условия

невозможно проверить условия, которые скрыты другими условиям

метод обладает недостаточной чувствительностью к ошибкам в логических выражениях

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий набора тестов которого:

вызывает выполнение всех результатов каждого решения, по крайней мере, один раз

выполняет каждый оператор по крайней мере один раз.

Для программ содержащих более одного условия минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий и выполняющий каждый оператор минимум один раз.

Термин "белый ящик" означает, что при разработке тестовых случаев тестировщики используют любые доступные сведения о внутренней структуре или коде. Технологии, применяемые во время тестирования "белого ящика", обычно называют технологиями статического тестирования.

Этот метод не ставит цели выявление синтаксических ошибок, так как дефекты такого рода обычно обнаруживает компилятор. Методы белого ящика направлены на локализацию ошибок, которые сложнее выявить, найти и зафиксировать. С их помощью можно обнаружить логические ошибки и проверить степень покрытия тестами.

Тестовые процедуры, связанные с использованием стратегии белого ящика, используют управляющую логику процедур. Они предоставляют ряд услуг, в том числе:

Дают гарантию того, что все независимые пути в модуле проверены по крайней мере один раз.

Проверяют все логические решения на предмет того, истины они или ложны.

Выполняют все циклы внутри операционных границ и с использованием граничных значений.

Исследуют структуры внутренних данных с целыо проверки их достоверности.

Тестирование посредством белого ящика, как правило, включает в себя стратегию модульного тестирования, при котором тестирование ведется на модульном или функциональном уровне и работы по тестированию направлены на исследование внутреннего устройства модуля. Данный тип тестирования называют также модульным тестированием, тестированием прозрачного ящика (clear box) или прозрачным (translucent)

тестированием, поскольку сотрудники, проводящие тестирование, имеют доступ к программному коду и могут видеть работу программы изнутри. Данный подход к тестированию известен также как структурный подход.

На этом уровне тестирования проверяется управляющая логика, проявляющаяся на модульном уровне. Тестовые драйверы используются для того, чтобы все пути в данном модуле были проверены хотя бы один раз, все логические решения рассмотрены во всевозможных условиях, циклы были выполнены с использованием верхних и нижних границ и роконтролированы структуры внутренних данных.

Методы тестирования на основе стратегии белого ящика:

Ввод неверных значений. При вводе неверных значений тестировщик заставляет коды возврата показывать ошибки и смотрит на реакцию кода. Это хороший способ моделирования определенных событий, например переполнения диска, нехватки памяти и т.д. Популярным методом является замена alloc() функцией, которая возвращает значение NULL в 10% случаев с целью выяснения, сколько сбоев будет в результате. Такой подход еще называют тестированием ошибочных входных данных. При таком тестировании проверяется обработка как верных, так и неверных входных данных. Тестировщики могут выбрать значения, которые проверяют диапазон входных/выходных параметров, а также значения, выходящие за границу диапазона.

Модульное тестирование. При создании кода каждого модуля программного продукта проводится модульное тестирование для проверки того, что код работает верно и корректно реализует архитектуру. При модульном тестировании новый код проверяется на соответствие подробному описанию архитектуры; обследуются пути в коде, устанавливается, что экраны, ниспадающие меню и сообщения должным образом отформатированы; проверяются диапазон и тип вводимых данных, а также то, что каждый блок кода, когда нужно, генерирует исключения и возвращает ошибки (errог returns). Тестирование каждого модуля программного продукта проводится для того, чтобы проверить корректность алгоритмов и логики и то, что программный модуль удовлетворяет предъявляемым требованиям и обеспечивает необходимую функциональность. По итогам модульного тестирования фиксируются ошибки, относящиеся к логике программы, перегрузке и выходу из диапазона, времени работы и утечке памяти.

Тестирование обработки ошибок. При использовании этого метода признается, что нереально на практике проверить каждое возможное условие возникновения ошибки. По этой причине программа обработки ошибок может сгладить последствия при возникновении неожиданных ошибок. Тестировщик обязан убедиться в том, что приложение должным образом выдает сообщение об ошибке. Так, приложение, которое сообщает о системной ошибке, возникшей из-за промежуточного программного обеспечения представляет небольшую ценность, как для конечного пользователя, так и для тестировщика.

Утечка памяти. При тестировании утечки памяти приложение исследуется с целью обнаружения ситуаций, при которых приложение не освобождает выделенную память, в результате чего снижается производительность или возникает тупиковая ситуация. Данная технология применяется как для тестирования версии приложения, так и для тестирования готового программного продукта. Возможно применение инструментов тестирования. Они могут следить за использованием памяти приложения в течение нескольких часов или даже дней, чтобы проверить, будет ли расти объем используемой памяти. С их помощью можно также выявить те операторы программы, которые не освобождают выделенную память.

Комплексное тестирование. Целью комплексного тестирования является проверка того, что каждый модуль программного продукта корректно согласуется с остальными модулями продукта. При комплексном тестировании может использоваться технология

обработки сверху вниз и снизу вверх, при которой каждый модуль, являющийся листом в дереве системы, интегрируется со следующим модулем более низкого или более высокого уровня, пока не будет создано дерево программного продукта. Эта технология тестирования направлена на проверку не только тех параметров, которые передаются между двумя компонентами, но и на проверку глобальных параметров и, в случае объектно-ориентированного приложения, всех классов верхнего уровня.

Тестирование цепочек. Тестирование цепочек подразумевает проверку группы модулей, составляющих функцию программного продукта. Эти действия известны еще как модульное тестирование, с его помощью обеспечивается адекватное тестирование компонентов системы. Данное тестирование выявляет, достаточно ли надежно работают модули для того, чтобы образовать единый модуль, и выдает ли модуль программного продукта точные и согласующиеся результаты.

Исследование покрытия. При выборе инструмента для исследования покрытия важно, чтобы группа тестирования проанализировала тип покрытия, необходимый для приложения. Исследование покрытия можно провести с помощью различных технологий. Метод покрытия операторов часто называют C1, что также означает покрытие узлов. Эти измерения показывают, был ли проверен каждый исполняемый оператор. Данный метод тестирования обычно использует программу протоколирования (profiler) производительности.

Покрытие решений. Метод покрытия решений направлен на определение (в процентном соотношении) всех возможных исходов решений, которые были проверены с помощью комплекта тестовых процедур. Метод покрытия решений иногда относят к покрытию ветвей и называют C2. Он требует: чтобы каждая точка входа и выхода в программе была достигнута хотя бы единожды, чтобы все возможные условия для решений в программе были проверены не менее одного раза и чтобы каждое решение в программе хотя бы единожды было протестировано при использовании всех возможных исходов.

Покрытие условий. Покрытие условий похоже на покрытие решений. Оно направлено на проверку точности истинных или ложных результатов каждого логического выражения. Этот метод включает в себя тесты, которые проверяют выражения независимо друг от друга. Результаты этих проверок аналогичны тем, что получают при применении метода покрытия решений, за исключением того, что метод покрытия решений более чувствителен к управляющей логике программы.

Обычно тестирование белого ящика основано на анализе управляющей структуры программы. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов графа управления. В этом случае формируются тестовые варианты, в которых:

гарантируется проверка всех независимых маршрутов программы;

проверяются ветви TRUE и FALSE для всех логических решений;

выполняются все циклы в пределах их границ и диапазонов;

анализируется правильность внутренних структур данных.

Недостатки тестирования белого ящика:

количество независимых маршрутов может быть очень велико. Например, если цикл в программе выполняется k раз, а внутри цикла имеется n ветвлений, то количество маршрутов вычисляется по формуле

При n=5 и k=20 . Даже если на разработку выполнения и оценку теста по одному маршруту расходуется 1мс, то на тестирование уйдет свыше 3000 лет.

исчерпывающее тестирование маршрутов не гарантирует соответствие программы исходным требованиям к ней.

в программе могут быть пропущены некоторые маршруты.

нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных.

Достоинства тестирования белого ящика позволяют учесть особенности программных ошибок.

Количество обнаруживаемых ошибок минимально в центре и максимально на периферии программы.

Предварительное предположение о вероятности потока управления или данных в программе часто бывает некорректно. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.

При записи алгоритма программного обеспечения на языке программирования возможно внесение типовых ошибок, как синтаксических, так и логических.

Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из перечисленных причин является аргументом для проведения тестирования по принципу белого ящика, поскольку тесты черного ящика не смогут реагировать на ошибки таких типов.

Способ тестирования базового пути

Тестирование базового пути – это способ, который основан на принципах белого ящика. Автор этого способа – Т. Мак-Кейб.

Способ тестирования базового пути позволяет:

получить оценку комплексной сложности программы;

использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества маршрутов в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Потоковый граф

Для представления программы при тестировании базового пути используется потоковый граф, имеющий следующие особенности:

Граф строится путем отображения управляющей структуры программы. В ходе отображения условные операторы и операторы циклов рассматриваются как отдельные операторы.

Узлы (вершины) потокового графа соответствуют линейным участкам программы и включают один или несколько операторов программы.

Дуги потокового графа отображают поток управления в программе, то есть передачу управления между операторами. Дуга потокового графа представляет собой ориентированное ребро.

Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного – две дуги.

Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в нескольких предикатных узлах.

Составным называется условие, в котором используется одна или несколько булевых операций, то есть OR или AND.

Замкнутые области, образованные дугами и узлами, называются регионами.

Окружающая граф среда рассматривается как дополнительный регион.

Есть три аспекта, которые проверяются в White Box тестировании, а именно:

Было ли программное обеспечение разработано в соответствии с оригинальным дизайном программного обеспечения.

Были ли внедрены меры безопасности в программное обеспечение и надежны ли они.

Выяснить уязвимости указанного программного обеспечения.

Преимущества White Box тестирования

Знание структуры внутреннего кодирования является предпосылкой при которой становится очень легко выяснить, какой тип ввода или какие данные могут помочь в эффективном тестировании приложений.

Еще одно преимущество White Box тестирования заключается в том, что оно помогает в оптимизации кода.

Помогает в удалении дополнительных строк кода, которые могут производить дефекты в коде.

Недостатки White Box тестирования

Так как знание кода и внутренней структуры - это необходимое условие, то для осуществления данного вида тестирования необходим опытный тестер, а это, в свою очередь, увеличивает затраты на программное обеспечение.

Почти невозможно заглянуть в каждый кусок кода, чтобы выяснить скрытые ошибки, которые могут создать проблемы, приводящие к сбою приложения.

Original: http://juice-health.ru/program/software-testing/493-white-box

Gray-box

Проверка «серого ящика» – это метод тестирования программного продукта или приложения с частичным знанием его внутреннего устройства. Для выполнения тестирования «серого ящика» нет необходимости в доступе тестировщика к исходному коду. Тесты пишутся на основе знания алгоритма, архитектуры, внутренних состояний или других высокоуровневых описаний поведения программы.

Виды тестирования «серого ящика»:

- Матричное тестирование.
- Регрессионное тестирование.
- Шаблонное тестирование (pattern).
- Тестирование с помощью ортогонального массива.

Достоинства метода:

- Тестирование серого ящика включает в себя плюсы тестирования «черного» и «белого». Другими словами, тестировщик смотрит на объект тестирования с позиции «черного» ящика, но при этом проводит анализ на основе тех данных, что он знает о системе.
- Тестировщик может проектировать и использовать более сложные сценарии тестирования.
- Тестировщик работает совместно с разработчиком, что позволяет на начальном этапе убрать избыточные тест-кейсы. Это сокращает время функционального и нефункционального тестирования и положительно влияет на общее качество продукта.
- Предоставляет разработчику достаточно времени для исправления дефектов.

Недостатки метода:

- Возможность анализа кода и тестового покрытия ограничена, так как доступ к исходному коду отсутствует.
- Тесты могут быть избыточными в том случае, когда разработчик также проверяет свой код Unit-тестами.
- Нельзя протестировать все возможные потоки ввода и вывода, поскольку на это требуется слишком много времени

В нашем примере у каждого клиента мог быть набор дополнительных функций (capabilities):

- «can_vpn» – клиент мог подключиться к VPN;
- «can_double_vpn» – клиент получал возможность подключиться к VPN, используя функцию DoubleVPN;
- «can_port_forward» – клиент имел дополнительный порт для входящих подключений на стороне сервера;
- «can_promo1» – клиент имел доступ к дружественному сервису.

Для удобства проверки разработчики предусмотрели возможность тестировщикам читать набор разрешенных функций из таблицы capabilities для каждого клиента. Тестировщики ставили тарифный план (подписку) и проверяли правильность изменения флагов в этой таблице. Без использования методики «серого ящика» проверка возможности для клиента совершить VPN-соединение в сочетании с дополнительными функциями потребовала бы гораздо больших затрат времени и труда.

Подведем итоги

Из представленной информации можно понять, что метод «серого ящика» помогает в следующих случаях:

- когда нет возможности использовать «белый ящик»;
- когда необходимо более полное покрытие по сравнению с «черным ящиком».

Используя этот метод, тестировщики получают доступ к проектной документации и могут подготовить и создать более точные и полные тест-кейсы и сценарии тестирования. Наибольшая эффективность применения «серого ящика» достигается при тестировании web-приложений, web-сервисов, безопасности, GUI, а также для функционального тестирования.

Смысл тестирования белого ящика

При определённом усердии можно добиться того, что тесты, написанные вручную или сгенерированные автоматически, будут покрывать все ветви тестируемого кода, то есть обеспечат 100% покрытие. Тем самым мы сможем с уверенностью сказать, что белый ящик делает то, что он делает. Хм. Секундочку. А в чём, собственно, смысл такого тестирования, спросит внимательный читатель? Ведь для любого содержимого белого

ящика будут построены тесты, которые только лишь подтверждают, что белый ящик работает каким-то определённым образом.

В некоторых случаях такой набор тестов всё же может иметь смысл:

1. Если белый ящик является прототипом и существует ненулевая вероятность ошибки для действительной реализации в другой среде.
2. Если существует несколько реализаций одинаковой логики (на разных языках или в разных окружениях).
3. Код белого ящика эволюционирует или подвергается рефакторингу, при этом мы хотели бы обнаруживать отличия от эталонного поведения.
4. Мы хотим обнаружить подмножества входных данных, обеспечивающие требуемые нам результаты.

Следует иметь в виду некоторые особенности тестирования, основанного на реализации, в отличие от тестирования на основе спецификации. Во-первых, если изначальная реализация не поддерживала некоторую функциональность, которую можно было бы ожидать, основываясь на спецификации, то наши тесты не заметят её отсутствия. Во-вторых, если такая функциональность присутствовала, но работала иначе, чем указано в спецификации (то есть, с ошибками), то наши тесты не просто этих ошибок не обнаружат, а напротив, ошибки будут "кодифицированы" в тестах. И если последующие/альтернативные реализации попробуют исправить ошибки, то такие тесты не позволят этого просто так сделать.