**Тестирование программного обеспечения**

Любая программа может содержать в себе ошибки. Компилятор способен выявлять только синтаксические ошибки, но не способен отслеживать семантику. Большинство ошибок проявляется в ходе работы программы, при этом они могут возникать не всегда, а лишь при определенных условиях. Таким образом, успешная компиляция программы и выполнение этой программы в одних и тех же условиях не гарантируют отсутствие ошибок.

Для выявления ошибок в программах ЖЦ разработки ПО предусматривает процесс тестирования, который является достаточно трудоемким и занимает больше времени, чем кодирование. (Г. Майерс дает оценку 1/3 для тестирования, при том, что кодирование занимает примерно 1/6.) Тестируемое ПО обычно называют SUT - Software Under Test. Цель тестирования - не убедиться в безошибочной работоспособности программы, а наоборот - найти ошибки. Поэтому в первую очередь возникает вопрос: а что есть ошибка в программе?

Заметим, что к этому моменту программа уже представляет собой выполнимый процессором набор команд, т.е. с точки зрения процессора она корректна. Даже если при каких-то условиях программа аварийно завершает свое выполнение или "портит" другие процессы, сразу нельзя сказать, что это ошибка в программе - возможно, так было задумано. Таким образом, ошибки необходимо рассматривать с точки зрения пользователя, основываясь на дополнительной информации, т.е. неком описании того, что должна делать программа (это же описание может включать в себя требование о том, чтобы программа никогда не завершалась аварийно и др.).



Рис. 1. V-образная модель жизненного цикла разработки ПО

При рассмотрении вопросов анализа программного кода порой удобнее применять ранее рассмотренную в разделе 1 модель жизненного цикла. Ее часто называют V-образной из-за расположения блоков на рисунке (рис. 1).

Нисходящая левая ветвь модели отражает поэтапную последовательность преобразования одних программных документов в другие: SYS - системных требований в SRD - требования к программному обеспечению, проектированию и формированию DDD - описания архитектуры системы и, наконец, разработке CODE - кода программ. Восходящая правая ветвь отражает процесс верификации разработанного программного обеспечения.

На первом этапе путем тестирования производится модульная верификация (MV), при которой поведение исполняемого программного кода проверяется на соответствие его DDD-описанию. Это наиболее трудоемкая и скрупулезная часть исследования. Она часто требует написания драйверов - моделей модулей, вызывающих процедуры тестируемого модуля, и заглушек - моделей процедур других модулей, вызываемых из тестируемого. Часто в MV отдельно выделяют процесс тестирования межмодульных связей, описанных в DDD.

На втором этапе производится комплексная верификация (CV) реализованного программного обеспечения по отношению к требованиям. Наконец, производится комплексная интеграция (CI) и проверка всей системы: пользователь, аппаратура и программное обеспечение. При грамотном процессе разработки уже на этапах нисходящей ветви для каждого требования определяется, на каком уровне верификации должна будет проводиться проверка его соблюдения.

При этом следует исходить из предположения, что ошибки всегда есть. Тестирование можно считать успешным, если найдены ошибки, а не наоборот. В достаточно сложном ПО все ошибки могут не обнаруживаться даже после длительного тестирования, однако, чем тщательнее ведется тестирование, тем меньше ошибок остается и тем менее вероятно возникновение невыявленных ошибок.

**Тестовый план**

Тестирование обычно проводится снизу вверх, т.е. сначала тестируются отдельные функции, затем целые модули и далее проводится комплексное тестирование всей программы или комплекса программ. Для проведения тестирования разрабатывается тест-план (test-plan) - совокупность тестовых наборов {примеров} (test-case). В каждом тестовом примере производится выполнение тестируемого программного элемента SUT при заданных Input - условиях и входных данных и проверяются все Output - выходные данные на соответствия заданным значениям.

Тестовый пример (набор) должен включать в себя как минимум:

* входы (конкретные значения всех выходных параметров, все необходимые свойства и установки окружения);
* действия (что надо выполнить и в какой последовательности);
* ожидаемый выход (конкретные величины всех возвращаемых значений, все выводы в потоки, сигналы, все изменяемые свойства и установки окружения).

Кроме указанных данных удобно, если каждый тестовый пример имеет дополнительно:

* номер (уникальный номер каждого тестового примера, чтобы на него можно было ссылаться);
* ссылку на требование (если для тестирования используются требования, то указание ссылок на конкретные требования, которые проверяет данный тестовый пример, упростит локализацию ошибок и обеспечит возможность проверки полноты тестирования);
* краткое описание (что проверяет данный тестовый пример).

Для проведения тестирования разрабатывается программа-драйвер (тест), выполняющая все тестовые примеры и сравнивающая выходные значения с ожидаемыми. В результате выполнения теста получается не только общий результат - есть или нет ошибки, но еще и список пройденных и непройденных тестовых примеров, который помогает локализовать ошибки в SUT.

Для упрощения локализации ошибок и последующей модификации тест-плана нужно, чтобы тестовые примеры были независимы друг от друга, т.е. чтобы каждый последующий тестовый пример никак не использовал результаты работы предыдущего. Для этого необходимо провести установки всех начальных условий перед выполнением каждого тестового примера.

**Проблема полноты тестирования**

Основная проблема тестирования ПО заключается в том, что проверить программу при всех возможных условиях функционирования в большинстве случаев невозможно. Это происходит либо в силу ограниченности ресурсов, либо в силу бесконечного количества возможных условий. Например, если рассмотреть функцию умножения двух рациональных чисел, варьируемых от -1000 до +1000, то в интервале от минимального возможного числа до максимального содержится бесконечное количество чисел. Т.е. все возможные значения входов проверить нельзя. Если же учесть, что машина оперирует не всеми этими числами, а различает только 10 знаков после запятой (т.е. множество чисел в интервале дискретно, минимальное отличие двух чисел 0,0000000001), то для проверки всех комбинаций из заданного диапазона понадобится  степени тестовых примера, что является достаточно большим числом для такой простой функции. Если проверяются не все возможные комбинации входных условий, то тестирование является неполным.

В основном для сложных программ тестирование является неполным, но даже неполное тестирование может выявить большинство ошибок, если выработать грамотную стратегию их поиска. Часто используют метод деления входных значений на области эквивалентности, так чтобы внутри каждой области для всех значений программа "вела себя" похоже. Тогда при написании тестовых примеров рассматриваются все значения на границах областей и по одному произвольному значению из каждой области (области определяются для каждого входного параметра).

Этот подход называют методом трех точек. В нашем примере для функции умножения двух чисел можно рассмотреть области [-1000; 0] и [0; +1000]. Деление образовано путем выявления трех особых точек (-1000, 0 и +1000). Такие точки называют критическими точками, в них тестируемая функция может менять свое поведение или потенциально вести себя особо. Т.е. для тестирования функции методом трех точек достаточно проверить  случаев (для каждого входа это точки -1000; 0; 1000 и, например, -500 и 500), что значительно меньше полного перебора. Конечно, при таком подходе возможно, что какие-то ошибки останутся, но вероятность этого будет невелика и зависит от выбора критических точек.

Функции, выполняющие различные сравнения, могут неверно их проводить, поэтому имеет смысл проверять их работу в непосредственной близости к критическим точкам. Для этого берутся значения, отстоящие от критических точек на величину дискретизации значений. Т.е. для примера функции умножения двух чисел, кроме значений метода трех точек, стоит рассмотреть значения -999,9999999999; -0,0000000001; 0,0000000001 и 999,9999999999. Этот подход называют методом пяти точек.

Иная сторона тестирования связана с типизацией переменных, при помощи которых задаются входные данные. Если для входных значений функции используются переменные типа float, а максимальное значение входа ограничено как +1000, то теоретически можно передать на вход и число +1001. Зачастую реакция функции на такое число не будет даже описана. Однако существуют приложения, чье поведение критично даже при передаче им входных значений, выходящих за пределы допустимых (например, авиационные программы, программы управления ядерными реакторами). В этом случае подразумевается, что программа должна вести себя корректно, т.е. не "зависнуть", не "повесить" систему, хотя выходное значение предсказать нельзя. Тестовые примеры, проверяющие поведение программы, в таких случаях, называются тестами на устойчивость (robustness). Если при тестировании методом пяти точек проверять еще и значения, выходящие за пределы допустимых диапазонов, то такой метод будет называться методом семи точек. В примере функции умножения двух чисел кроме значений -1000; -500; -999,9999999999; -0,0000000001; 0,0; 0,0000000001; 500,0; 999,9999999999; 1000 для каждого входа следует взять, например, еще значения -1001 и 100,0000000001.

Как уже было сказано, для тестирования ПО необходимо обладать информацией о том, что оно должно делать. Это может быть либо подробное описание (требования), либо просто сам код программы (в этом случае подразумевается, что программа должна работать корректно, не "портить память", не завершаться аварийно, не мешать другим процессам). В зависимости от исходной информации о ПО различают два подхода к тестированию - тестирование по требованиям и тестирование по коду.

**Тестирование. Метод "черного ящика"**

Тестирование по требованиям или тестирование "черного ящика" подразумевает неиспользование сведений о структуре исходного кода. Все тестовые примеры составляются только на основе требований, т.е. мы не видим, что находится внутри SUT и как оно работает. Единственное, что доступно - это то, что SUT должно делать, внешние эффекты поведения программного обеспечения.

При тестировании "черного ящика" удобно использовать методы трех, пяти или семи точек. При тестировании по требованиям используется понятие "покрытие" требований тестами. Тест покрывает требования, если он полностью проверяет выполнение каждого отдельного требования. При этом возможны как случаи, когда для проверки одного требования необходимо несколько тестовых примеров, так и случаи, когда достаточно одного тестового примера для проверки нескольких требований.

Рассмотрим, например, следующее требование к функции умножения двух целых чисел.

Если входное значение хотя бы одного множителя выходит за границы диапазона [15 ... 1500], то функция должна вернуть значение 0, в противном случае функция должна вернуть значение произведения двух множителей.

Для тестирования этого требования (если быть точнее, то двух требований) необходимо проверить значения множителей как из диапазона [15 ... 1500], так и вне его.

Тесты для проверки значений из диапазона могут выглядеть, например, следующим образом.

В табл. 1 представлены все комбинации двух входов, каждый из которых принимает по три различных значения (метод трех точек), так как диапазон [15 ... 1500] с точки зрения сложения чисел не имеет критических точек, кроме своих концов.

|  |
| --- |
| Таблица 1. Часть тест-плана |
| Входы | Действия | Ожидаемый выход |  |
| Множитель 1 | Множитель 2 |  |  |
| 15 | 15 | Вызов функции | 225 |
| 15 | 700 | Вызов функции | 10500  |
| 15 | 1500 | Вызов функции | 22500  |
| 700 | 15 | Вызов функции | 10500 |
| 700 | 700 | Вызов функции | 490000 |
| 700 | 1500 | Вызов функции | 1050000  |
| 1500 | 15 | Вызов функции | 22500 |
| 1500 | 700 | Вызов функции | 1050000  |
| 1500 | 1500 | Вызов функции | 2250000  |
| 14 | 14 | Вызов функции | 0 |
| 1501 | 14 | Вызов функции | 0 |
| 14 | 1501 | Вызов функции | 0 |
| 14 | 700 | Вызов функции | 0 |
| 700 | 14 | Вызов функции | 0 |
| 1501 | 700 | Вызов функции | 0 |
| 700 | 1501 | Вызов функции | 0 |

Для проверки поведения функции за границами диапазона стоит проверить, например, значения 14 и 1501 для каждого входа.

Необходимо отметить, что не всегда можно покрыть все требования по тем же причинам, по которым невозможно добиться полноты тестирования, поэтому не всегда возможно доказать правильность работы программы. Кроме того, некоторые требования могут быть так сформулированы, что их нельзя протестировать. Например, требования могут касаться алгоритмов работы (а алгоритмы не видны при тестировании "черного ящика") или каких-нибудь внутренних переменных, не доступных извне. Некоторые требования вообще могут оказаться некорректными при тестировании, хотя могли быть интуитивно понятны при разработке, например требование "программа должна иметь дружественный интерфейс" протестировать невозможно. Для исключения таких ситуаций необходимо уже на этапе составления требований формулировать их как можно более конкретно и однозначно.

Метод тестирования "черного ящика" выявляет все несоответствия между требованиями к ПО и поведением самого ПО.

**Тестирование. Метод "белого ящика"**

Тестирование по коду или тестирование "белого ящика" основывается на проверке кода SUT, когда в ходе выполнения SUT проверяется выполнение каждого блока кода. При тестировании "белого ящика" основная задача - это выполнение всего кода для проверки работоспособности всех его ветвей. При этом ставится задача покрытия кода тестами и рассматриваются разные уровни покрытия. Один из уровней покрытия (покрытие операторов) - это выполнение всех операторов, т.е. при выполнении всех тестовых примеров в итоге должны выполниться все операторы (не в каждом тестовом примере, а по результатам выполнения всех тестовых примеров).

Например, для написанного ниже фрагмента программы, где A, B и C рассматриваются как входные значения:

X = 0;

if ((A<=B) || (A>C)) X = 5;

достаточно одного тестового примера (ТП1: A=1, B=2, C=3). В этом случае выполнятся все операторы. Но если программист допустил ошибку и неверно написал условие, например так:

if ((A<=B) || (A>B)) X = 5;

то тогда код будет работать неверно (переменной X всегда будет присваиваться значение 5), хотя показанный выше тестовый пример приведет к выполнению всех операторов и не выявит ошибки.

Для выявления таких ошибок требуется выполнить другой уровень покрытия - по условиям.

Покрытие по условиям требует проверок всех условий на TRUE/FALSE, т.е. каждое условие в ходе тестирования должно проверяться на оба возможных значения. Для покрытия по условиям приведенного примера кода необходимо уже два тестовых примера:

ТП1: A=1, B=2, C=3;

ТП2: A=3, B=2, C=3.

Эти тестовые примеры позволяют найти ошибку.

Не все блоки кода всегда удается покрыть тестами. Это может быть связано с защитным программированием (когда входные значения функции проверяются на корректность, но передаются ей только корректные данные, так как передающая функция тоже проверяет их корректность); операторами выхода (закрывающая скобка "}" после оператора exit); мертвым кодом (код, который заведомо никогда не выполняется).

Проверка таких блоков кода и анализ их "безопасности" может происходить без выполнения самого кода, в этом случае группа экспертов читает и анализирует программный код, делая выводы о существовании или несуществовании ошибок. Результаты подобного анализа могут установить причины появления непокрытого кода.

Одной из причин может быть несовершенство тест-плана. Другой - пробелы в требованиях. И в том и в другом случае сам тестируемый код может не требовать изменений, но должны быть дописаны новые требования или изменены уже существующие и/или расширен тест-план. В обоих случаях процесс верификации повторяется.

**Заглушки**

Часто возникает необходимость тестировать модули, использующие процедуры других модулей, которые могут влиять на результат тестирования или значительно усложнять его получение. Допустим, имеется функция climatControl, обрабатывающая информацию с датчика температуры и управляющая нагревателем. Значение температуры она получает при помощи функции getTemperature, которая в свою очередь опрашивает температурный датчик. Пусть имеется следующее требование: "Если температура становится ниже 23 °C, то функция climatControl должна включить нагреватель". Для тестирования этого требования необходимо передать функции некое значение температуры, меньшее 23, допустим 20. Но эта величина не является входным значением функции climatControl, она задается возвращаемым значением функции getTemperature. Один из путей - "заставить" функцию getTemperature возвратить нужное значение, например подключив реальный датчик температуры и охладив его до 20°С. Но при этом нет гарантии, что датчик и сама функция getTemperature работает правильно. А что если она возвращает значение температуры в фаренгейтах (20°С равно 68°F, т.е. функция вернет 68 вместо ожидаемых 20)? Это приведет к тому, что функция climatControl не включит нагреватель, даже если не содержит в себе ошибок.

В таких ситуациях реальная функция getTemperature заменяется новой функцией, которая разрабатывается специально для конкретного (конкретных) тестового примера, и выполняет только то, что необходимо для теста, возвращая нужное значение. Т.е. в данном примере функция getTemperature заменяется на функцию, которая не будет опрашивать датчик, а сразу вернет значение 20. Функции или программы, которыми заменяются используемые при тестировании SUT функции и программы, называют заглушками.

Стоит отметить, что если SUT использует какие-либо функции, которые не заменяются заглушками в ходе тестирования, то такие функции также должны быть предварительно протестированы. Это приводит к так называемому "восходящему" тестированию. Что, в свою очередь, предполагает такую же последовательность реализации модулей системы. Чаще как раз подобные getTemperature процедуры реализуются в последний момент, так как зависят от периферийного оборудования, которое не изготавливается, а закупается или характеристики которого специально подгоняются под условия эксплуатации системы. В этих случаях написание моделей (заглушек) программ неизбежно.

**Процесс тестирования**

Как уже говорилось, тест состоит из набора тестовых примеров. Перед написанием теста необходимо подготовить сценарий каждого тестового примера. Такие сценарии описываются в тест-плане. Тест-план представляет собой документ, последовательно определяющий всю совокупность тест-примеров с указанием конкретных значений всех входных данных, действий, а также ожидаемых значений выходных данных (см. выше про то, что включает в себя тестовый пример).

В тест-плане отражается основная логика тестирования, а сам тест (набор тестовых примеров) является реализацией тест-плана. Вполне возможно построить универсальный тест-драйвер, который будет использовать в качестве входных данных тест-план, создавать на его основе все необходимые условия для каждого тестового примера и выполнять заданные сценарии, сравнивая реальные выходные значения с ожидаемыми. Этот подход часто применяется для автоматизации процесса тестирования.

Если в тест-плане для каждого тестового примера указываются ссылки на тестируемые им требования, то можно говорить о трассировке или о соответствии тестового примера тест-плана требованию. Аналогично при кодировании теста для каждого тест-примера удобно указывать ссылку на номер его описания в тест-плане.

Такая трассировка позволит при выявлении ошибки каким-либо тестовым примером определить, перейдя к тест-плану, при каких условиях проявилась ошибка, и далее, перейдя к требованиям, выяснить, какие требования нарушены. Так как в ходе разработки документация может меняться, то при трассировке необходимо сохранять соответствие версий трассируемых документов. Это, в свою очередь, позволяет определять область влияния внесенных изменений, например определить, какие тесты должны быть модифицированы или перепроверены при изменении части требований к системе.

Результатом выполнения теста является отчет о прогоне теста, в котором отражается, сколько всего было выполнено тестовых примеров, сколько ошибок найдено, какие тестовые примеры выявили ошибки. Удобно помещать в отчет о прогоне теста ожидаемые и реальные значения всех выходных данных после выполнения каждого тестового примера, чтобы в случае выявления ошибки можно было их сравнить.

**Пример верификации**

**Третий этап работы**

Тестирование программной реализации

**Соображения**

Общие требования к тест-плану состоят в определении входных воздействий на программу достаточных для контроля всех требований по проверке функций программной реализации как "черного ящика" (полнота покрытия требований). Дополнительно желательно определить входные воздействия, обеспечивающие выполнение всех ветвей программной реализации (программа проверяется как "белый ящик"). Часто удобно представлять тест-план в виде таблицы.

**Выводы**

Далее приведен неполный пример тест-плана (табл. 2).

|  |
| --- |
| Таблица 2. Пример тест-плана |
| ВХОД | ВЫХОД | Проверяемое ТРЕБОВАНИЕ | Комментарии |
| "ABCD EFG,HIJ." | "BCAD FGE,IJH." | 4.1.2 | Длина первого слова равна 4 |
| " ABCD EFG,HIJ." | " BCAD FGE,IJH." | 4.1.1; 4.1.2 | Ведущие пробелы; длина первого слова равна 4 |
| " ..,,." | " ..,,." | 4.1.3 | Строка состоит из одних разделителей |
| "ABCDE FG,HIJ." | "BCAED GF,IJH." | 4.1.2 | Длина первого слова равна 5; длина второго слова равна 2 |
| "1234567890.........2,,,,,,,,,8" | "2315648970.........2,,,,,,,,,8" | 4.1; 4.1.2; 4.4 | Длина первого слова равна 10; длина остальных слов равна 1; общая длина строки 80 символов |
| "1234567890.........2,,,,,,,,,8,," | Сообщение "Ошибка во входной строке" | 4.3.3 | Длина строки больше 80 символов |
| "1\*2" | Сообщение "Ошибка во входной строке" | 4.3.2 | Строка содержит недопустимый символ |
| "...123" | Сообщение "Ошибка во входной строке" | 4.3.1 | Ведущие разделители не пробелы |
| "" | Сообщение "Работа закончена" | 4.2 | Введена пустая строка |

**Замечание**

Приведенный выше тест-план не проверяет (явно) вывод предупреждающего сообщения о вводе новой строки. Такое требование не было явно определено в разделе 4. Это недоработка требований по проверке функций программной реализации. Хороший тестировщик должен сам включить подобную проверку в свой тест-план. Дополнительные пункты плана явно зависят от конкретного программного кода и поэтому не приведены в данном примере.