Тестирование на основе моделей

В. В. Кулямин

Лекция 2. Основные задачи и виды тестирования

В прошлой лекции тестирование было определено как проверка соответствия поведения программной системы требованиям, выполняемая по результатам реальной работы этой системы в некотором конечном наборе заранее определенных ситуаций [1].

Таким образом, основная задача тестирования — проверка требований. Ее решение чаще всего сводится к последовательному решению ряда промежуточных задач. Кроме того, для решения этих задач в различных обстоятельствах необходимы разные подходы, определяемые технической спецификой и внешними целями — для чего именно тестирование нужно в контексте заданного проекта. Данная лекция посвящена основным целям и задачам тестирования, различным способам его проведения.

Цели тестирования

Тестирование является одним из видов деятельности жизненного цикла программного обеспечения и в рамках всего процесса разработки и сопровождения ПО может использоваться для достижения нескольких целей.

- Наиболее очевидная цель тестирования — ***поиск ошибок***, то есть расхождений между наблюдаемым поведением ПО и требованиями к нему.
  Это наиболее понятная и простая цель, которая первой приходит на ум при разговоре о тестировании.
  Даже весьма опытные разработчики ПО часто считают, что это — его единственная цель. Классическая фраза Дейкстры «тестирование может использоваться для демонстрации наличия ошибок в программе, но не может использоваться для демонстрации их отсутствия» часто интерпретируется именно в том смысле, что единственная польза от тестирования — найденные ошибки.
- Менее очевидная и более сложная для понимания цель тестирования — ***общая оценка качества ПО***. Помимо прямого обнаружения ошибок, оно должно давать информацию о том, где, скорее всего, находятся еще не найденные ошибки, насколько они могут быть серьезны, насколько тестируемая система надежно и корректно работает в целом, насколько корректны и стабильны ее отдельные функции и компоненты.
  Если единственной целью тестирования считать нахождение ошибок, то терпеливое, аккуратное и систематическое тестирование, в ходе которого ошибок не обнаруживается — бесполезно. Это не так, просто потому, что результатом такой работы является ценная информация о свойствах системы, недоступная из других источников, если, конечно, не относиться серьезно к уверенности разработчиков в собственной непогрешимости (обычно считается, что «уж в основном-то все правильно, так, мелочи всякие могут еще не работать»). Такое тестирование, *если оно действительно аккуратное и систематическое*, показывает, что тестируемая система — достаточно хороша. Даже если в ней и есть ошибки (что их нет, доказать при помощи тестирования действительно невозможно), они достаточно редки и случаются в очень специфических ситуациях. Однако проведение подобного тестирования — непростая задача, основная трудность которой — достижение такого уровня аккуратности и систематичности, который позволяет обоснованно делать соответствующие выводы.
  Другой возможной проблемой при понимании тестирования как деятельности, нацеленной только на поиск ошибок, является отсутствие рациональных аргументов для определения момента, когда нужно прекратить тестирование. Если тестировщики занимаются только поиском ошибок, они не могут предоставить руководству проекта обоснованную информацию о том, что «в системе еще очень много недоделок» или,

наоборот, «система уже достаточно надежна». В этом случае решение об остановке тестирования принимается только на основе наличия ресурсов, то есть по остаточному принципу: есть еще есть время и деньги — продолжаем тестировать, нет — прекращаем. Такого рода решения не связаны с реальным качеством системы, и поэтому способны принести большой урон репутации организации-разработчика ПО. Кроме того, они закрепляют плохую практику управления проектами, способствуя перемещению все большей и большей части ресурсов проектов в разработку без тестирования, ведь от тестирования «нет прямой пользы» и его можно проводить на «остатки» ресурсов.

Если руководитель проекта принимает подобные решения, это может означать либо, что он недостаточно компетентен и не воспринимает поступающую от тестировщиков информацию, либо, что тестировщики недостаточно компетентны, чтобы такую информацию предоставить, и тестирование проводится хаотичным, недостаточно эффективным образом, несмотря даже на то, что оно выявляет важные ошибки, — ведь остается непонятным риск наличия еще невыявленных ошибок.

Методом проб и ошибок на практике были выработаны рекомендации по организации тестирования, согласно которым ресурсы для проведения разработки тестов и тестирования необходимо выделять заранее. Их размер варьируется от 20% ресурсов всего проекта при создании достаточно простых систем до 80-90% при разработке критически важных и сложных систем реального времени с жесткими требованиями к их надежности и качеству в целом.

- Еще более редко упоминаемая и долговременная цель тестирования — ***обеспечение стабильного развития системы***, предоставление средств для отслеживания изменений в ней и предсказания возможных проблем системы после внесения в нее тех или иных изменений.

  Хорошие наборы тестов на практике становятся не только средством оценки качества тестируемой системы и сертификации ее пригодности для определенных задач, но и инструментом отслеживания важных изменений при развитии системы и измерения ее общей стабильности и надежности. Для этого, однако, необходимо поддерживать набор тестов в работоспособном состоянии, соответствующем текущим требованиям к системе. При наличии нескольких поддерживаемых версий системы необходимо иметь либо соответствующие версии тестового набора, либо средства его настройки, позволяющие тестировать функциональность, специфичную для каждой из версий. Для сложных систем это возможно только при использовании специфических образцов организации тестов, о которых пойдет речь в следующей лекции.

Задачи тестирования

Чтобы достигать указанных целей управляемым и предсказуемым образом, необходимо уметь решать ряд задач. Эти задачи специфичны для тестирования как особого вида деятельности и всегда должны решаться при его проведении, тем или иным способом. Однако прежде, чем формулировать их необходимо проанализировать точный смысл ряда терминов, повсеместно используемых при разработке тестов и тестировании.

Как уже говорилось, тестирование всегда проводится в некотором конечном наборе заранее определенных ситуаций. Такие ситуации называются *тестовыми ситуациями* и обычно включают несколько элементов.

- Набор внешних воздействий, которые оказываются на систему в такой ситуации. Эти воздействия называют *тестовыми воздействиями*. Примерами тестовых воздействий могут служить вызов интерфейсной функции системы или метода одного из ее объектов, нажатие на кнопку графического интерфейса пользователя, выполнение команды в командной строке, передача сообщения системе по одному из каналов, по которым она такие сообщения ожидает. Обычно воздействия связаны с некоторыми данными, которые они передают в систему — это, например, аргументы вызванной функции или

метода, опции выполненной команды, содержимое полей редактирования формы, где нажимается кнопка, данные пересланного в систему сообщения. Эти данные называют *тестовыми данными*.

- *Внутреннее состояние* системы в этой ситуации. Чаще всего внутреннее состояние сложных программных систем не наблюдаемо полностью извне и не может быть непосредственным образом установлено в определенное значение. Однако для обеспечения полноты проводимых во время тестирования проверок, нужно как-то управлять внутренним состоянием, чтобы иметь возможность проверить поведение системы в разных состояниях.

  Можно заметить, что такая система оказывается в разных состояниях при выполнении разных последовательностей действий. Поэтому для управления внутренним состоянием во время тестирования используют различные последовательности тестовых воздействий, называемые *тестовыми последовательностями*. Таким образом, задача приведения системы в заданное состояние сводится к построению приводящей туда последовательности воздействий.

  Например, при тестировании класса, реализующего очередь каких-либо объектов, необходимо проверить корректность операций добавления и удаления объекта не один раз, а в рамках разных последовательностей, в которых эти операции выполняются при разных заполнениях очереди.

  В еще более сложных случаях, например, при тестировании распределенных систем, состоящих из нескольких работающих параллельно компонентов, состояние, от которого зависят результаы работы в данной ситуации, может включать в себя не только внутренние состояния отдельных компонентов, но и текущее состояние среды передачи сообщений между ними — в каком (необязательно согласованном) состоянии находятся общие данные, какие сообщения уже переданы полностью, какие находятся в стадии приема, какие лежат в буферах для передачи, т.е., уже отправлены, какие еще только передаются в буфер, а какие еще не переданы. Для создания различных внутренних состояний такой системы уже недостаточно последовательностей воздействий — в таких случаях применяются более сложные комбинации воздействий, состоящие, например, из нескольких последовательностей, подаваемых на отдельные параллельно обрабатываемые входы системы (в общем случае получается частично упорядоченное множество воздействий, т.е., между некоторыми воздейсвиями есть порядок, определяющий, какое из них поступило в систему раньше, а между некоторыми парами такого порядка нет).

  Важное отличие состояния системы, работающей детерминировано в рамках одного потока управления, от состояния системы, в которой есть несколько параллельно работающих потоков, в том, что в первом случае, даже если мы не можем наблюдать внутреннее состояние системы, мы можем эффективно его котролировать за счет подачи последовательностей воздействий, т.е., текущее состояние модет быть однозначно полностью вычислено на основе истории. Во втором случае мы не можем ни наблюдать текущее состоянии, ни контролировать его — при подаче одних и и тех же наборов воздействий на внешние входы, даже при соблюдении определенных временных интервалов между ними, в общем случае невозможно предсказать в каком точном состоянии окажется система. Это зависит еще и от внутренних процессов, и от точных моментов времени, в которые она воспринимает те или иные воздействия (а точные такие моменты контролировать извне обычно невозможно, всегда есть некоторая погрешность).

- В тестовую ситуацию входят и *внешние условия*, воспринимаемые системой самостоятельно, без оказания на нее специальных воздействий, и влияющие на ее работу в данной ситуации.

  Например, в автоматическую систему кондиционирования здания могут входить датчики температуры и влажности в помещениях самого здания и на улице. При ее

тестировании не достаточно просто проверить, что команды с пультов управления выполняются ею правильно, но и удостовериться, что это так при различных значениях температуры и влажности, при которых реакции системы на команды должна будет изменяться.

Адекватное моделирование различных внешних условий часто очень сложно. Ведь при этом нужно не только создавать определенные условия (что иногда трудно, например, в требованиях к системе могут быть определены специфические ограничения на ее работу при очень низких температурах или высоком уровне радиации), но иногда и моделировать различные сценарии их изменения, которые, однако, должны быть реалистичны — годятся не произвольные сценарии, а те, что могут случиться на самом деле. Например, температура в комнате обычного здания не может меняться за 10 секунд от -100°C до +100°C, а, скажем, температура элемента внешней облочки искусственного спутника вполне может иногда так себя вести. Или, при тестировании системы управления движением автомобиля можно смоделировать его движение по горячему песку или по обледеневшему асфальту, но вряд ли стоит проверять работоспособность этой системы на дороге, где горячий песок и обледеневший асфальт то и дело сменяют друг друга.

В ряде случаев кроме тестовой системы, осуществляющей тестирование, никакие другие факторы не могут воздействовать на тестируемую систему. Это случай изолированного тестирования, в котором тошько тесты оказывают воздействия на систему и контролируют (на сколько могут) ее поведение, других субъектов контроля нет. Во многих других ситуациях можно смоделировать внешние условия специальными воздействиями, например, отделить от системы датчики температуры и вместо них подавать на соответствующие входы сигналы, соответствующие модельным значениям температуры. Еще в ряде систем все значимые внешние факторы представляются в виде конфигурационных параметров, значения которых определяются в конфигурационных файлах и базах данных при развертывании системы. Во всех этих случаях все значимые внешние условия либо могут быть сымитированы программно, либо могут управляться достаточно простым образом, и тестирование может быть выполнено программными тестами за счет сведения внешних условий к специфическим воздействиям.

Если же часть значимых условий не может управляться через программный интерфейс или информацию, заносимую в файлы и базы данных системы, необходимо специальное оборудование и существенные затраты ресурсов для их моделирования при тестировании. При этом создаются достаточно дорогие имитационные стенды для тестирования поведения системы в различных внешних условиях.

Ситуации, в которых будет проверяться поведение тестируемой системы, и сами выполняемые проверки обычно формализуют и представляют в таком виде, чтобы их мог выполнить любой человек, желательно даже незнакомый с предметной областью и задачами системы, или, еще лучше, компьютер.

Программа или четко описанная процедура, при выполнении которой создается одна или несколько тестовых ситуаций и проверяется правильность поведения системы в этих ситуациях, называется *тестом*. Важно, что тест включает в себя не только инструкцию по достижению определенной ситуации, но и инструкцию по проверке того, что проверяемая система отработала в этой ситуации правильно. Тестирование обычно организуется как выполнение некоторого набора тестов, который так и называется — *тестовый набор*.

Тестовые наборы чаще всего создаются заранее, до проведения тестирования, при разработке тестов. Иногда, однако, используются техники генерации тестов уже в процессе самого тестирования, когда никакого заранее подготовленного набора тестов нет. Такая генерация, тем не менее, всегда основана на каких-то правилах, использующих определенные данные об устройстве тестируемой системы и требованиях к ней. В этих случаях разработкой тестов естественно считать разработку этих правил.

Чтобы уметь целенаправленно и предсказуемым образом создавать полноценные наборы тестов, необходимо уметь решать следующие задачи.

- *Проверка выполнения требований.*
- *Определение критериев полноты тестирования.*
- *Построение полного набора тестовых ситуаций.*
- *Создание отчетов с информацией о результатах тестирования.*
- *Организация тестового набора для обеспечения удобства его модификации, выполнения и анализа получаемых результатов.*

Обсуждению нескольких из этих задач посвящены следующие разделы данной лекции. Критерии полноты тестирования подробно рассматриваются в следующей лекции, а различные техники построения полных наборов тестов являются основным содержанием всех дальнейших лекций.

Проверка выполнения требований

На практике решение этой задачи очень часто затрудняется отсутствием документов с понятным, однозначным, непротиворечивым и полным описанием требований. Чаще всего при разработке тестов приходится заодно уточнять требования к тестируемой системе и дорабатывать представляющие их документы, делая их более ясными и полными, а также устраняя имеющиеся противоречия.

Для построения систематичных и корректных тестов нужно адекватное понимание требований, то есть понимание того, что именно они означают, что из этого может быть проверено и, наконец, что именно должно быть проверено в каждой конкретной тестовой ситуации.

Рассмотрим, например, функцию abs(x), вычисляющую абсолютную величину числа. Вроде бы ясно, что при этом вычисляется — должен возвращаться x, если он неотрицателен, или –x иначе. На языках C, C++, Java или C# это может быть передано так: (x >=0)? x : -x.

Можно, например, проверять, что abs(x) >= 0 для любого x.

Попробуем взять x = –2147483648, например, в Java. Каков будет результат?

Правильный ответ: –2147483648.

Этот неожиданный результат — отрицательное число в качестве абсолютной величины — объясняется тем, что в машинной арифметике 32-битных целых чисел выполнено соотношение –(–2147483648) = –2147483648. Приведенное выше определение abs(x) остается правильным, неверно только заключение о том, что abs(x) >= 0.

Чтобы объяснить полученное «странное» соотношение, надо понимать, чем руководствовались создатели целочисленной машинной арифметики. Им нужно было целые числа, которых бесконечно много, представить в машине как некоторое конечное множество. При этом, однако, надо сохранить основополагающие свойства арифметических действий — сложения, вычитания и умножения. Наиболее похожими на целые числа конечными множествами с таким набором операций являются множества (кольца) классов целых чисел по какому-то модулю, например Z_2 = {[0], [1]}, где [0] — класс четных чисел, а [1] — класс нечетных, или Z_3 = {[0], [1], [2]}, где [n] — класс чисел, имеющих остаток n при делении на 3. Поскольку в машине удобно представлять числа в двоичной записи, для эффективного расхода памяти стоит взять модуль равным степени 2, например 2^{32}. То есть, машинное число n будет обозначать класс чисел, равных n по модулю 2^{32}. Если к тому же хочется, чтобы действия с небольшими числами приводили к привычным результатам: 2+2 = 4, а 3–5 = –2, в качестве представителей классов чисел по модулю 2^{32} должны использоваться небольшие положительные и отрицательные числа. Таким образом, в качестве представителей удобно выбрать числа 0, 1, –1, 2, –2, 3, –3, 4, –4 и т.д. Но, поскольку всего их должно быть 2^{32}, для некоторого числа n, мы в итоге выберем n, но не выберем –n или

наоборот — ведь для 0 в этой последовательности уже нет соответствующего отрицательного числа. Соответственно, конец ее выглядит как 2147483647, –2147483647, –2147483648 = -2^{31}. По причинам, связанным с эффективностью и простотой реализации операций, проще выбрать -2^{31}, чем 2^{31} — при этом можно использовать первый бит представления числа в значении его знака, а все вычисления производить побитно, по тем же причинам не стоит вводить специальное число –0, что, например, сделано в арифметике чисел с плавающей точкой. По модулю 2^{32} выполнено $-[-2^{31}] = [2^{31}] = [2^{32}-2^{31}] = [-2^{31}]$, что и соответствует выписанному выше соотношению.

Другой пример. Как ведет себя tg(arctg(x)) при возрастающем x, которое является числом с плавающей точкой двойной точности? При $x <= 10^{16}$ все идет как ожидается: tg(arctg(x)) = x с небольшой погрешностью, а вот дальше, как бы ни было велико x, tg(arctg(x)) дает один и тот же результат $1.633123935319537 \cdot 10^{16}$. Для объяснения этого нужно знать, как устроены числа с плавающей точкой, что служащее для представления π/2 число с плавающей точкой двойной точности X = 884279719003555/562949953421312 меньше π/2 примерно на $6.1232339957367658 \cdot 10^{-17}$, и что получаемый результат является как раз значением $\mathrm{tg}(X) \approx 1/(\pi/2 - X)$, вычисленным с двойной точностью.

Двоичное число с плавающей точкой имеет следующую структуру [2,3].

- Число представлено в виде набора из *n* бит, из которых первый бит является *знаковым битом числа*, следующие *k* бит представляют его *порядок E*, а оставшиеся *(n-k-1)* бит представляют его *мантиссу M*.
- Знаковый бит *S*, порядок *E* и мантисса *M* числа *x* определяют его значение по следующим правилам.
  $x = (-1)^S \cdot 2^e \cdot m$, где
  - *S* — знаковый бит, равный 0 для положительных чисел, и 1 для отрицательных;
  - если $0 < E < 2^k - 1$, то $e = E - 2^{(k-1)} + 1$;
    иначе, если $E = 0$, $e = -2^{(k-1)} + 2$;
    число $b = 2^{(k-1)} - 1$ называется *смещением порядка (exponent bias)*;
  - если $0 < E < 2^k - 1$, то $m = 1 + M/2^{n-k-1}$. Иначе говоря, *m* имеет двоичное представление 1.*M*, т.е. целая часть *m* равна 1, а последовательность цифр дробной части совпадает с последовательностью бит *M*;
    если же $E = 0$, то $m = M/2^{n-k-1}$, или *m* имеет двоичное представление 0.*M*.
    Числа с нулевой экспонентой называются *денормализованными*, а все остальные — *нормализованными*.
- Максимальное возможное значение порядка $E = 2^k - 1$ зарезервировано для представления *исключительных чисел*: положительной и отрицательной бесконечностей, +∞ и -∞, а также специального значения NaN (not-a-number, не число). NaN используется, если результат выполняемых действий нельзя корректно представить ни обычным числом, ни бесконечностью, как, например, результаты 0/0 или (-∞) + (+∞).
  +∞ имеет нулевой знаковый бит, максимальный порядок и нулевую мантиссу; -∞ отличается только единичным знаковым битом.
  Любое число, имеющее максимальный порядок и ненулевую мантиссу, считается представлением NaN.
- Стандарты IEEE 754 и IEEE 854 определяют несколько возможных типов чисел с плавающей точкой, из которых чаще всего используются *числа однократной точности* (single precision), *числа двойной точности* (double precision) и *числа расширенной двойной точности* (double-extended precision).
  Для чисел однократной точности $n = 32$ и $k = 8$. Соответственно, для мантиссы

используется 23 бита и смещение порядка равно 127.

Для чисел двойной точности $n = 64$ и $k = 11$. Для мантиссы используется 52 бита и смещение порядка равно 1023.

Для чисел расширенной двойной точности определенные значения k и n не фиксируются в стандартах, вводятся лишь ограничения $128 \geq n \geq 80$ и $k \geq 15$. В процессорах 32-битной архитектуры Intel $n = 80$ и $k = 15$. При этом для мантиссы используется 64 бита и смещение порядка равно 16383.

Кроме этого, иногда используются числа *четырехкратной точности*, для которых $n = 128$ и $k = 15$. Соответственно, для мантиссы используется 112 бит и смещение порядка равно 16383. Они не специфицированы в стандарте, но имеют аналогичную структуру.

В качестве примера укажем представление числа -17_{10} в виде чисел однократной и двойной точности. Поскольку $-17_{10} = (-1)^1 \cdot 2^{131-127} \cdot 1.0001_2 = (-1)^1 \cdot 2^{1027-1023} \cdot 1.0001_2$, соответственно, оно представляется в виде числа однократной точности как
1 10000011 00010000000000000000000, а в виде числа двойной точности как
1 10000000011 0001000000000000000000000000000000000000000000000000.

Некоторые граничные значения таких чисел для разных форматов указаны в Таблице 1. При этом для расширенной двойной точности используются параметры 32-битной архитектуры Intel.

	Общий вид	Однократная точность	Двойная точность	Расширенная двойная точность	Четырехкратная точность
Самое маленькое денормализованное положительное число	$2^{-b-n+k+2}$	2^{-149}	2^{-1074}	2^{-16446}	2^{-16494}
Самое маленькое нормализованное положительное число	2^{-b+1}	2^{-126}	2^{-1022}	2^{-16382}	2^{-16382}
Самое большое положительное число	$2^{b-n+k+1} \cdot (2^{n-k}-1)$	$2^{104} \cdot (2^{24}-1)$	$2^{971} \cdot (2^{53}-1)$	$2^{16319} \cdot (2^{65}-1)$	$2^{16271} \cdot (2^{113}-1)$
Самое большое число, меньшее 1	$1-2^{-n+k}$	$1-2^{-24}$	$1-2^{-53}$	$1-2^{-65}$	$1-2^{-113}$
Самое маленькое число, большее 1	$1+2^{-n+k+1}$	$1+2^{-23}$	$1+2^{-52}$	$1+2^{-64}$	$1+2^{-112}$

Таблица 1. Граничные значения чисел с плавающей точкой.

Заметьте, что половину страницы текста заняло объяснение лишь небольшой «странности» поведения операции взятия противоположного целого числа — одной из простейших и наиболее точно определенных, которые только можно себе представить. Для полного и аккуратного разбора примера с арктангенсом потребовалась бы еще дополнительно пара страниц. При рассмотрении же более сложных функций программных систем, не имеющих таких точных математических аналогов, возникают гораздо более серьезные проблемы.

Именно поэтому здесь в качестве примеров взяты математические функции — любой человек, окончивший два курса ВУЗа по физико-математической или технической специальности, достаточно хорошо представляет себе, что это такое, и при желании может самостоятельно понять, как должна работать такая функция. Если же надо понять, как работают менее однозначно определенные операции, возникает такое количество возможных разнообразных смыслов, что чисто умозрительно решить, какой именно правилен, практически невозможно — необходима документация, зафиксированные в документах

требования, общение с разработчиками системы или с экспертами в данной предметной области.

Например, для адекватного понимания работы операции чтения заданного количества байт из файла нужно уметь четко отвечать на множество вопросов. Что происходит, если файла нет? Что будет, если он есть, но у данного процесса нет прав на работу с ним? Что будет результатом, если размер файла меньше запрашиваемого количества байт? А если другие операции в тоже время записывают данные в этот же файл? И пр., и т.д., и т.п.

В то же время для операций ввода-вывода, как и для большинства других операций, реализованных в рамках широко используемых библиотек, можно, после определенных усилий, достаточно строго определить математическую модель, адекватно описывающую их работу. Для многих же практических примеров — операций биллинговой системы, системы автоматического управления боевым кораблем или гидроэлектростанцией — таких моделей вообще нет, никто никогда не продумывал такие сложные системы во всех их деталях. На формальную проработку их при современных технологиях уйдет времени и усилий гораздо больше, чем это допустимо с точки зрения экономической оправданности таких систем.

Таким образом, одна из наиболее сложных задач — *адекватно понять требования*, понять, что именно обозначает каждое утверждение в документации, которое относится к данной операции. В примере с абсолютной величиной вроде бы удалось сразу написать четкое определение, но *понять* его помогает только приведенный пример «странного» поведения. Только после проведенного дополнительного анализа становится возможным без ошибок выводить следствия из этого определения (первоначальный вывод о том, что абсолютная величина больше 0 был ошибкой). Во многих других случаях даже просто написать четкое определение нелегко. Поэтому всегда необходим вдумчивый *анализ требований*, извлечение всех сведений, которые только можно получить из документации, стандартов, а также из личного общения с экспертами, архитекторами, разработчиками и пользователями тестируемой системы и другими заинтересованными лицами.

Помогают в этом анализе попытки четко определить, как можно проверить требования. Например, попытка проверить правильность вычисление абсолютной величины при помощи тождества abs(x) >= 0 проваливается, и этот факт вынуждает задуматься об основных принципах машинной арифметики. Формулируя требования в проверяемом виде, мы сразу получаем очень хорошее определение правильного поведения, более аккуратное, чем существующие описания для подавляющего большинства программных систем.

Если тесты создаются как обычно, в виде тестовых вариантов, их разработчик сразу пытается определить, как именно проверять требования в той конкретной ситуации, которая возникает в данном тесте. Методы тестирования на основе моделей требуют описывать требования к поведению операции «вообще», в произвольной ситуации, в которой она должна работать. При этом в качестве результата анализа требований получается некоторая ***модель требований*** — целостное, полное, непротиворечивое и точное описание того, что должна делать система. Она указывает, как проверять правильность работы системы и дает возможность разделить формулировку проверяемых требований и придумывание ситуаций для их проверки на два отдельных вида деятельности, повышая качество и того, и другого.

Отчеты о результатах тестирования

Хотя основной целью тестирования является лишь проверка соответствия требованиям, или же, наоборот, проверка наличия ошибок в тестируемой системе, тестирование, после которого остается только информация вида «ошибок нет» или «ошибки есть», практически бесполезно. На практике нужны тесты, после выполнения которых остается достаточно информации о найденных ошибках, чтобы разработчик мог локализовать их в ходе отладки с небольшими усилиями. Кроме того, нужна еще и информация о полноте выполненных тестов. В общем случае отчеты о ходе тестирования должны содержать следующее.

- Данные обо всех обнаруженных ошибках.

- Тип ошибки по некоторой классификации, который поясняет, что же произошло. Например: затребовано слишком много памяти, функция работает слишком долго, выдается некорректный результат, выполняется запись неверных данных куда-либо, не возвращается управление, разрушение процесса.
- Какая проверка зафиксировала ошибку, что именно было сочтено некорректным, какое требование при этом проверялось.
- Каков наиболее короткий выделяемый сценарий действий, который позволяет повторить эту ошибку. Иногда достаточно указания только одной неправильно сработавшей операции или функции, но в сложных случаях необходимо повторить некоторый набор действий, в совокупности приведший к некорректной работе системы. Операция, при выполнении которой ошибка проявляется, вовсе не обязательно сама содержит ошибку, она может просто использовать некорректные результаты работы других операций.

- Данные о полноте тестирования по нескольким критериям — какие тестовые ситуации возникали в ходе тестирования, а какие нет, каковы значения измерявшихся метрик тестового покрытия.
  Информацию о затронутых элементах тестируемой системы — вызываемых функциях и методах, выполняемых командах, затрагиваемых классах, компонентах, подсистемах — а также о проверяемых в ходе тестирования требованиях всегда полезно иметь, даже когда полнота тестирования определяется другими способами.

Организация тестовых наборов

Поскольку часто набор тестов представляет собой довольно сложную систему, она должна быть хорошо организована. Это особенно важно для тестов, которые планируется поддерживать, сопровождать и пополнять долгое время. При организации тестов нужно также аккуратно выделять отдельные модули, учитывая возможности их повторного использования, как и при проектировании любой другой программной системы. Таким образом, хорошо спроектированный тестовый набор всегда использует те же базовые принципы программной инженерии, что и любая хорошо спроектированная программная система — абстракцию и уточнение, модульность и многократное использование [4].

Помимо этого, хорошая организация тестовых наборов должна подчиняться дополнительным ограничениям, связанным уже с самой природой тестов.

- Тесты должны иметь дополнительную структуру, основанную на их связи с компонентами тестируемой системы. Она может быть представлена, например, в явно указанной в описании каждого теста или в комментариях к нему ссылке на соответствующие компоненты.
  Наличие такой информации облегчает сопровождение тестового набора при небольших изменениях в тестируемой системе, без изменения ее функциональности. В этих случаях становится возможным быстро найти все тесты, подлежащие корректировке в связи с такими изменениями.
  Другая выгода от наличия явных связей с тестируемыми компонентами — возможность минимизации количества повторно выполняемых тестов при небольших изменениях, которые никак не отражаются на интерфейсе системы или ее функциях. При этом можно выполнять только те тесты, которые затрагивают измененные или зависящие от измененных компоненты.
- Тесты должны быть явно связаны и с проверяемыми требованиями.
  Такая привязка позволяет, помимо оценки полноты тестового набора, снижать трудозатраты на его модификацию при изменениях в требованиях, быстро выделяя только те тесты, которые должны измениться.
  Если возникает необходимость проверить только заданное подмножество функций

системы, привязка тестов к требованиям также поможет минимизировать исполняемый тестовый набор.

- Тесты должны быть разбиты на группы по нескольким различным аспектам. Это разбиение может влиять на порядок выполнения тестов.
  - Например, полезно отделять тесты, выполнение которых полностью автоматизировано, от тех, где требуется вмешательство человека. При этом часто автоматические тесты удобно выполнять раньше, поскольку обнаруживаемые ими ошибки могут сделать выполнение тестов под контролем человека более трудным.
  - При тестировании сложных систем часто требуются достаточно сложные сценарии работы теста. Например, тест, проверяющий корректность управления потоками в операционной системе, может запускать много потоков различной конфигурации, в разных потоках использовать различные механизмы синхронизации, проверяя, что они работают корректно. Однако, если ошибка в функции создания потока просто не дает создать новый поток в программе, пытаться выполнить этот тест бессмысленно — большая его часть так и будет выполнена, а та, что будет, может просто попасть в тупик или выдать невнятную информацию о множестве обнаруженных ошибок, анализировать которую будет непросто.

    Избежать такой ситуации поможет простой тест, проверяющий только, что новый поток действительно создается. Прежде, чем выполнять сложные и запутанные тесты, использующие много функций, желательно проверить выполнение некоторых базовых требований каждой из этих функций с помощью простых тестов. При нарушении таких требований, сложный тест не стоит выполнять вообще.

    Такая организация тестов для сложных систем позволяет минимизировать совокупные затраты на выполнение тестов и анализ их результатов.
  - Если тестируемая система может иметь опциональные, необязательные функции, которые, при их наличии, тоже нужно протестировать, появляется необходимость в опциональных проверках, которые при некоторых условиях выполняются, а при других нет.

    Проще всего эта задача решается при помощи введения в тест *конфигурационных параметров*, значения которых могут управлять выполняемыми тестами и проводимыми в них проверками. Конфигурационные параметры могут настраиваться независимо и не изменяться в ходе работы теста, а могут определяться динамически, при помощи специального модуля, проверяющего наличие или отсутствие в системе опциональных функций. Нацеленные на эти функции тесты должны выполняться только после работы такого модуля.

Наиболее мощной техникой структуризации тестовых наборов является выделение модулей или компонентов, отвечающих каждый за решение своей специфической задачи.

Виды компонентов тестового набора

- Оракулы.

  Тестовым оракулом называется компонент теста, принимающий решение о том, правильно или неправильно сработала тестируемая система в данном тесте.

  Способы построения оракулов.
  - Чаще всего роль оракула выполняет человек. Он просто анализирует тестовые данные и результаты теста и сопоставляет их со своим пониманием требований.
  - Достаточно часто используются оракулы на основе таблиц тестовых данных и правильных результатов. Такой оракул находит в таблице строку, в соответствующих столбцах которой записаны использованные тестовые данные, и

проверяет, что полученный результат равен тому, который находится в столбце результатов в этой же строке.

- Если есть другая реализация той же функции, которую мы собираемся тестировать, можно построить оракул, использующий сравнение результатов, полученных от тестируемой системы, с выдаваемыми этой реализацией. Другая реализация может быть прототипной, не такой эффективной, как тестируемая, но зато правильной. В качестве другой реализации может использоваться предыдущая версия тестируемой системы, если к ней нет нареканий по поводу корректности работы.
- При наличии нескольких версий тестируемой системы или нескольких других реализаций проверяемой функции можно использовать оракул, построенный на базе голосования. Правильным результатом при этом считается тот, что вернуло большинство из имеющихся реализаций, и результат тестируемой системы сравнивается с ним.
- Если тестируемая функция имеет обратную, ее можно использовать в оракуле — вычислять по результатам входные данные и проверять их совпадение с использованными реально входными данными.
- Самый общий случай — проверка каких-то свойств результата. На основе требований формулируются ожидаемые свойства результата, и оракул проверяет именно их.

  Например, если от операции требуется возвращать список клиентов, имеющих задолженность, превышающую заданное значение, такой оракул для каждого из клиентов в полученном списке проверяет, действительно ли его задолженность превышает это значение.

- Генераторы тестовых данных.

  Генератор тестовых данных отвечает за построение входных данных, используемых тестами.

  Способы организации генераторов.

  - Пул данных. Заранее приготовленные данные сохраняются в виде коллекции или таблицы и используются по мере необходимости.
  - Другой генератор с фильтрацией. Генератор данных, удовлетворяющих некоторому условию можно организовать, используя простой генератор данных этого типа и фильтрацию по данному условию — если поставляемые простым генератором данные удовлетворяют условию, они передаются вовне, если нет — вычисляется следующая порция данных.
  - Составной генератор. Генератор данных сложной структуры строится из генераторов элементов этой структуры. Например, если нужно построить объекты, у которых есть три поля, можно по отдельности строить значения полей, а потом объединять их в объект. При объединении можно использовать декартово произведение множеств значений, то есть строить по объекту на каждую создаваемую тройку значений полей, или использовать «диагональ» — объединять только одновременно сгенерированные тройки значений полей.

    В составных генераторах данных, удовлетворяющих некоторым ограничениям целостности (например, значение первого поля всегда должно быть больше значения второго), используются фильтры.

- Тестовые адаптеры.

  Тестовый адаптер предназначен для соединения теста с тестируемой системой в тех случаях, когда интерфейс тестируемой системы не соответствует тому, на который рассчитан тест.

  Такая ситуация может сложиться, если тесты для старой версии нужно перенести на

новую версию системы, причем изменений в проверяемой функциональности мало, но есть изменения в интерфейсах — функции или команды названы по-другому, в них изменен порядок параметров или типы параметров получили другие имена.
Другая возможность — тесты создавались заранее, до разработки системы или параллельно ей, на основе требований и проектной документации, а когда была разработана сама тестируемая система, часть ее интерфейсов изменилась по сравнению с проектом.
Третий случай использования адаптеров — тестирование на соответствие некоторому общему стандарту, для которого может быть много реализаций от разных поставщиков. Такая ситуация, например, в телекоммуникационном ПО — стандарты на протоколы взаимодействия фиксированы, но есть много разных разработчиков этого ПО и их системы должны успешно взаимодействовать друг с другом. При этом тестовый набор для стандарта делается при самых общих предположениях об интерфейсе (просто наличие определенных функций и структура данных их параметров) — такой тестовый набор называется *абстрактным*. Чтобы использовать абстрактный тестовый набор для тестирования некоторой реализации, он дополняется набором адаптеров, привязывающих использованный в нем абстрактный интерфейс к конкретному интерфейсу данной реализации.

- Заглушки.
  Тестовой заглушкой называется компонент, играющий роль необходимого тестируемой системе компонента, который еще не разработан.
  При интеграционном и модульном тестировании иногда приходится тестировать отдельно компоненты, которым для работоспособности необходимы другие. Эти другие компоненты могут быть еще не готовы или же они не включаются в тест, поскольку сильно усложнили бы его. Чтобы тестируемый компонент мог работать во время тестирования, вместо отсутствующих компонентов подставляются заглушки, которые имеют такой же интерфейс, что и отсутствующие компоненты, но устроены как можно более просто, например, возвращают один и тот же результат или генерируют его случайно.

Виды тестирования

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества — тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование отдельных атрибутов — защищенности, функциональной пригодности и пр. Кроме того, особо выделяют ***нагрузочное*** или ***стрессовое тестирование***, проверяющее работоспособность, надежность ПО и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных, и пр.

Рассматриваемые в данном курсе методы построения тестов ориентированы в большей мере на тестирование функциональности. Их можно использовать и для тестирования переносимости, производительности или надежности. Однако при их использовании нужно иметь в виду следующее.

- Для тестирования производительности необходим дополнительный анализ факторов возможного снижения производительности — объема входных данных, объема базы данных системы, количества пользователей, количества процессов и потоков в системе, объема данных, передаваемых между компонентами системы и пр. Полнота такого

тестирования сильно зависит от используемого в тестах набора факторов. Выделение адекватного набора аспектов, влияющих на производительность, не рассматривается в данном курсе.

- При тестировании надежности должны измеряться статистические показатели работы системы и должны использоваться близкие к реальным по своим статическим свойствам входные данные. Вопросы статистического моделирования различных аспектов входных данных, поведения пользователей и других систем, которые могут оказывать влияние на работоспособность тестируемой системы, не рассматриваются в данном курсе. Также не обсуждаются методы определения статистических показателей ее работы.

Тестирование удобства использования сильно отличается от других видов тестирования, поскольку всегда связано с оценкой удобства выполняемых действий и представления их результатов в системе. По этой причине при таком тестировании всегда используются экспертные оценки, требующие вовлечения квалифицированных специалистов по удобству использования. Автоматизировать этот вид тестирования в той же мере, как другие, не удается.

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды.

- ***Тестирование черного ящика***, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется ***тестированием на соответствие*** (conformance testing). Частным случаем его является ***функциональное тестирование*** — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности.
  Еще одним примером тестирования на соответствие является ***аттестационное*** или ***сертификационное тестирование***, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.
- ***Тестирование белого ящика***, оно же ***структурное тестирование*** — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).
- Тестирование, при котором используются как требования, так и знания о внутреннем устройстве системы, используется на практике чаще, чем указанные выше крайние разновидности. Оно иногда называется ***тестированием серого ящика***, но термин этот упоминается реже, чем тестирование черного или белого ящика.
- Тестирование, нацеленное на определенные ошибки. Тесты для такого тестирования строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота тестирования определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, ***тестирование на отказ*** (smoke testing), в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками.

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено. Эти же разновидности тестирования можно связать с фазой жизненного цикла тестируемой системы, на которой они выполняются.

- ***Модульное тестирование*** (unit testing) предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют ***программные контракты*** — ***предусловия***, описывающие для каждой операции, на каких входных данных она предназначена работать, ***постусловия***, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и ***инварианты,*** определяющие критерии целостности внутренних данных модуля.
  Модульное тестирование является важной составной частью ***отладочного тестирования***, выполняемого разработчиками для отладки написанного ими кода.
- ***Интеграционное тестирование*** (integration testing) предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций. Интеграционное тестирование также используется при отладке, но на более позднем этапе разработки.
  При интеграционном тестировании могут использоваться различные стратегии присоединения новых компонентов к тестируемой системе.
  - При стратегии «сверху вниз» сначала тестируют модули, находящиеся на самом верхнем уровне и непосредственно взаимодействующие с пользователями или внешними системами. Затем к ним постепенно добавляют модули, вызываемые ими, выполняя тестирование после добавления каждого модуля, затем — модули следующих уровней. На каждом шаге, кроме последнего, в котором участвуют все модули системы, вместо отсутствующих модулей используются заглушки.
  - При стратегии «снизу вверх» сначала тестируются модули нижнего уровня, не зависящие от других модулей системы, затем добавляются модули, зависящие от них, и т.д., вплоть до модулей самого верхнего уровня. При этом заглушки используются редко, только в тех случаях, когда только что добавленный в тестируемую систему модуль зависит от других модулей того же уровня.

  Часто применяются смешанные стратегии — часть этапов интеграции выполняется «сверху вниз», часть — «снизу вверх».
  Метод, которым настоятельно не рекомендуется пользоваться, — проведение интеграционного тестирования сразу для всех модулей большой системы, без предварительной отладки взаимодействия внутри отдельных групп модулей.
- ***Системное тестирование*** (system testing) предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях.
  Системное тестирование выполняется через внешние интерфейсы ПО и тесно связано с ***тестированием пользовательского интерфейса*** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида тестирования являются ***тестирование графического пользовательского интерфейса*** (Graphical User Interface, GUI) и ***пользовательского интерфейса Web-приложений*** (WebUI).
  Если интеграционное и модульное тестирование чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя тестирование через API в этом случае также вполне возможно.

Особняком стоит ***регрессионное тестирование***, используемое для проверки того, что вносимые небольшие изменения и исправления ошибок не нарушают стабильность и не

снижают работоспособность системы. Регрессионное тестирование используется на этапе сопровождения, после внесения изменений и исправлений в систему, при выпуске ее очередной версии.

Литература

[1] IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004.

[2] IEEE 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. NY: IEEE, 1985.

[3] D. Goldberg. *What Every Computer Scientist Should Know about Floating-Point Arithmetic*. ACM Computing Surveys, 23(1):5-48, 1991.

[4] В. В. Кулямин. Технологии программирования. Компонентный подход. М: Интернет-университет информационных технологий — БИНОМ. Лаборатория знаний, 2007. http://www.ispras.ru/~kuliamin/lectures-sdt/Lecture01.pdf.