Тестирование на основе моделей

В. В. Кулямин

Лекция 1. Качество программного обеспечения и методы контроля качества

Данный курс лекций посвящен тестированию программного обеспечения (ПО).

Многим людям, даже считающим себя хорошими программистами, но не имеющим опыта работы над большими проектами с жесткими сроками готовности и требованиями по качеству и надежности разработанного ПО, часто кажется, что тестирование не представляет особых проблем, по сравнению с созданием программ — сиди себе, пробуй, что работает, а что — нет. К большому сожалению, это не так. Тесты для сложной системы сами по себе сложны. Кроме того, для сложных систем требуется очень много тестов, чтобы проверить корректность работы реализуемых ими функций во многих различных ситуациях, поэтому наборы тестов тоже представляют собой сложные программные комплексы, требующие особых методов для их разработки и сопровождения, и, как это не покажется странным, применения творческих способностей от людей, создающих их.

В этом курс рассказывается о специфических методах построения тестов, которые в последнее время обычно выделяются в особую область — *тестирование на основе моделей*. К значению этого термина мы вернемся несколько позже, а сейчас отметим, что эта область находится на границе между *теоретической информатикой (computer science)* и *инженерией программных систем (software engineering)*. Поэтому она требует как определенной математической подготовки, знакомства с техниками строгого описания свойств программного обеспечения, так и умения применять их на практике и оценивать практическую эффективность используемых подходов на пути к приемлемому компромиссу между полнотой тестирования и затраченными на него ресурсами.

Часто говорят, что тестирование предназначено для поиска ошибок и проверки правильности или надежности создаваемых программных систем, или, более точно, для контроля их *качества*. Поскольку понятие качества программной системы не является вполне очевидным, начнем с его более детального рассмотрения.

Качество программного обеспечения

Можно определить качество ПО, как его пригодность и удобство для решения тех задач, для которых оно создано (так же, как и качество любого инструмента). Однако у программных систем есть две особенности, отличающие их, если не от всех, то от многих других инструментов, используемых человеком в своей деятельности.

- Во-первых, цели создания программной системы чаще всего сложны и включают множество аспектов. Программное обеспечение, за редким исключением, не разрабатывается для решения ровно одной задачи. Гораздо чаще это целый набор связанных задач из некоторой области. Кроме того, в него входят и экономическая эффективность использования данной системы и удобство работы с ней для того персонала, который имеется у организации-заказчика.
- Во-вторых, очень часто программы используются для решения несколько не тех задач, для которых они предназначались при создании. Набор целей, для достижения которых применяется данная система, изменяется со временем, что отражается и в постоянном добавлении новых возможностей и функций в новые версии программ.

Поэтому целостное и четкое понятие качества ПО определить очень нелегко. Вместо этого используют различные *модели качества*, систематизирующие набор аспектов, характеристик и метрик качества, рассмотрение которых необходимо для адекватной оценки качества

разнообразных программ. Такие модели могут изменяться со временем, поскольку изменяются потребности индустрии производства ПО, появляются новые группы возможностей или внимание разработчиков привлекается к новым аспектам качества, ранее считавшимся несущественными.

Наиболее широко на данный момент используется модель качества ПО, зафиксированная в наборе стандартов ISO 9126 [1-4]. В несколько упрощенном виде (при рассмотрении так называемого *внутреннего качества*) эта модель определяет 6 основных характеристик качества программного обеспечения. Каждая характеристика уточняется при помощи некоторого набора более детальных атрибутов.

Рисунок 1. Характеристики и атрибуты качества ПО по ISO 9126.

- *Функциональность*.
  Эта характеристика обозначает способность ПО решать определенный круг задач. Функциональность определяет, что именно делает данная программа. Атрибуты функциональности следующие: функциональная пригодность — способность решать нужный набор задач; точность выдаваемых результатов; защищенность — способность предотвращать доступ к функциям и данным ПО людям или другим системам, у которых нет прав на это; способность к взаимодействию с другими системами; и др.

- *Надежность.*
  Это способность ПО поддерживать определенный уровень работоспособности в заданных условиях.
  Надежность является вероятностной характеристикой работоспособности ПО. Атрибуты ее таковы: зрелость — обратная величина к частоте отказов ПО; устойчивость к отказам, способность выполнять определенные задачи и придерживаться некоторых ограничений даже в случае отказов и сбоев; способность к восстановлению после отказов и среднее время такого восстановления; и др.

- *Удобство использования.*
  Удобство использования показывает, насколько ПО привлекательно, удобно в обучении работе с ним и при выполнении самой работы.
  К атрибутам удобства использования относятся: понятность — показатель, обратный к усилиям, затрачиваемым пользователями на понимание основных понятий и способов работы ПО и их применимости для решения нужных им задач; удобство обучения, обратное к усилиям на обучение работе с системой; удобство работы, обратное к

усилиям на выполнение определенного круга задач; привлекательность, способность привлекать новых пользователей; и др.

- *Производительность.*
  Это способность ПО обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. В соответствии с затратами ресурсов разного вида — времени, памяти, пропускной способности сетевых соединений — выделяются и различные атрибуты производительности.

- *Переносимость.*
  Эта характеристика показывает сохранение работоспособности ПО при изменении его окружения.
  Ее атрибутами являются, например, возможность развертывания или установки ПО в различных окружениях и его адаптируемость — способность приспосабливаться к работе в различных окружениях при помощи действий, зафиксированных в документации.

- *Удобство сопровождения.*
  Удобство сопровождения определяет трудоемкость анализа, исправления ошибок и внесения изменений в ПО.
  Его атрибутами являются, в частности, удобство проведения тестирования, удобство внесения изменений и риск возникновения неожиданных эффектов при изменениях.

В 2011 году принят стандарт ISO 25010 [5], заменяющий ISO 9126-1 и несколько изменяющий набор характеристик и атрибутов внутреннего качества ПО. В его рамках имеются следующие характеристики.

- ***Функциональность*** (теперь называемая functional suitability)
  - Функциональная пригодность (functional appropriateness) — способность ПО решать нужные пользователям задачи;
  - Функциональная полнота (functional completeness) — определяет, насколько полно ПО способно решать нужный набор задач;
  - Точность (functional correctness) — способность выдавать результаты с нужной точностью;

- ***Производительность*** (performance efficiency)
  - Временная эффективность (time behavior) — способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время;
  - Эффективность использования ресурсов (вычислительных, resource utilization) — способность решать нужные задачи с использованием определенных объемов ресурсов (памяти различных видов, устройств ввода-вывода и пр.);
  - Пропускная способность каналов связи (capacity) — способность решать нужные задачи при определенных ограничениях на пропускаемые через каналы связи объемы информации;

- ***Совместимость*** (compatibility)
  - Способность к сосуществованию (co-existence) — из переносимости по ISO 9126, способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы;
  - Способность к взаимодействию (interoperability) — из функциональности по ISO 9126, способность взаимодействовать с нужным набором других систем;

- ***Удобство использования*** (usability)
  - Удобство обучения (learnability) — показатель, обратный усилиям, затрачиваемым пользователями на обучение выполнению определенных задач с помощью ПО;

- Удобство работы (operability) — показатель, обратный усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО;
- Понятность (теперь appropriateness recognizability) — показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание их применимости для решения своих задач;
- Эстетичность (бывшая привлекательность, user interface aesthetics) — способность ПО быть привлекательным для пользователей, не вызывать эстетического отторжения;
- Защищенность от ошибок пользователей (user error protection) — способность игнорировать или исправлять определенные ошибки пользователей;
- Доступность (при различных способностях пользователей, accessibility) — способность поддерживать работу людей с ограниченными возможностями — при нарушении восприятия цветов и сильных дефектах зрения, некоторых нарушениях координации движений, и пр.;

- ***Надежность*** (reliability)
  - Зрелость (maturity) — показатель, обратный частоте отказов ПО, обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени;
  - Способность к восстановлению (recoverability) — способность восстанавливать определенный уровень работоспособности и целостность данных после отказа, при затрате определенного времени и ресурсов;
  - Устойчивость к ошибкам (fault tolerance) — способность поддерживать заданный уровень работоспособности при отказах и некоторых нарушениях правил взаимодействия с окружением;
  - Работоспособность (availability, иногда также переводится как доступность) — возможность ПО решать задачи и предоставлять пользователям информацию, несмотря на ненадежную работу сетей, отдельных серверов и т.д.;

- ***Защищенность*** (security)
  - Конфиденциальность (confidentiality) — способность ПО защищать свои данные от доступа лиц, которые не имеют к ним допуска;
  - Целостность (integrity) — способность ПО защищать свои данные от изменения теми лицами. которые не имеют на это прав;
  - Строгое выполнение обязательств (неотвергаемость, non-repudiation) — способность дать убедительное подтверждение тому, что заданные операции действительно выполнялись авторизованными пользователями (а их результаты не были внесены каким-либо несанкционированным образом);
  - Авторизуемость (операций, accountability) — возможность проследить, какие пользователи выполняли заданные операции;
  - Аутентичность (authenticity) — способность дать подтверждение собственной идентичности (т.е., отсутствия подмены части кода или модулей по сравнению с проверенными и сертифицированными версиями) и идентичности пользователей (т.е., возможность дать определенные гарантии, что выступающий под некоторым идентификатором пользователя человек, это именно он, или имеющий право действовать от его имени);

- ***Удобство сопровождения*** (maintainability)
  - Удобство проверки (testability) — показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам;
  - Анализируемость (analyzability) — удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий;

    - Модифицируемость (modifiability, удобство внесения изменений + стабильность) — показатель, обратный трудозатратам на выполнение необходимых изменений и риску возникновения неожиданных эффектов после них;
    - Модульность (modularity) — возможность вносить изменения в отдельные модули с минимальным их влиянием на другие;
    - Повторная используемость (reusability) — возможность использовать отдельные модули без модификации в рамках других систем;
- ***Переносимость*** (portability)
    - Адаптируемость (adaptability) — способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных;
    - Удобство замены (replaceability) — возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении;
    - Удобство установки (installability) — способность ПО быть установленным или развернутым в определенном окружении;

Требования к программному обеспечению

Перечисленные характеристики качества ПО представляют собой одну из систематик различных видов *требований* к программным системам. Такая систематика полезна, если необходимо выделить и проанализировать требования для сложной системы. Она позволяет организовать анализ, разбивая возможные требования и ограничения на разные группы и позволяя последовательно рассматривать различные аспекты системы. Сами требования как раз и говорят о том, какие конкретные свойства и характеристики должна иметь система, чтобы с ее помощью можно было успешно решать заданный набор задач.

Требования определяют, какое поведение системы является правильным и желаемым, а какое должно рассматриваться как некорректное. Поэтому они играют первостепенную роль при тестировании — именно они и проверяются с его помощью, а ошибки связаны именно с их нарушениями.

Сами требования определяются при анализе задач, стоящих перед рассматриваемой программной системой и ее окружения. Они могут извлекаться из различных источников.

- При аккуратном следовании стандартам на процессы разработки ПО требования обычно фиксируются в документе, создаваемом при решении о разработке системы и называемом *техническим заданием* (а по-английски — *requirements specification, спецификация требований*). Иногда этот документ создается представителями заказчика, но часто его приходится писать самим разработчикам на основе информации, полученной из других источников.
- Важным источником требований являются *стандарты*, регламентирующие характеристики, функции и состав систем, работающих в определенной предметной области. Такие стандарты создаются на основе опыта, накопленного в большом количестве проектов, в ходе которых такие системы создавались или развивались, и поэтому содержат ценную информацию о желательных характеристиках таких систем. Часто работа системы связана с выполнением каких-то действий, регулируемых существующим законодательством и нормами, действующими в данной области. Нормы и законы тоже являются разновидностью действующих стандартов, хотя обычно они формулируются менее четко и однозначно, чем технические стандарты и правила.
- Разработчики сами могут осознать, что что-то должно быть сделано определенным образом — так обычно возникают *внутренние ограничения задач*, не соблюдая которые,

невозможно решить их правильно. Чаще всего ограничения такого рода можно усмотреть при детальном анализе решаемой задачи.

- Иногда ряд требований к новой системе можно сформулировать на основе анализа *уже существующих систем* для решения схожих задач.
- Б*о*льшая часть требований формулируется на основе явно высказываемых *пожеланий* пользователей системы, их руководителей, заказчиков ее разработки и других заинтересованных лиц.
- Наконец, наиболее нечетким, но, тем не менее, достаточно важным источником требований являются невысказанные явно *потребности и нужды* пользователей создаваемой системы, которые, несмотря на это, все же часто поддаются анализу.

При извлечении требований из перечисленных выше источников их четкость и согласованность возрастают при движении от потребностей и пожеланий к стандартам и техническому заданию.

Тестирование проверяет соответствие требованиям, и поэтому чем более точно и ясно они сформулированы, тем аккуратнее и полнее можно провести тестирование. Если какие-то требования определены нечетко, проверка их тоже может быть выполнена лишь с определенной степенью точностью, и системы, ведущие себя сильно по-разному, могут быть признаны удовлетворяющими ему в одинаковой мере. Получаемые при этом оценки качества таких систем будут во многом субъективными.

В некоторых случаях тестирование кажется возможным и в отсутствии всяких требований, по крайней мере, документально зафиксированных. В этих случаях, однако, используются какие-то свойства, которые считаются само собой разумеющимися и, соответственно, представляющие собой неявные требования. Пример такого свойства — отсутствие сбоев при работе программы. Если такие неявные требования действительно должны быть выполнены, можно проводить тестирование на их основе. Если же это не так, то есть иногда сбой может рассматриваться как корректное поведение системы (если, скажем, вводятся совсем некорректные исходные данные), то подобное тестирование может привести к неправильным выводам о наличии ошибок в системе.

Чтобы требования к программному обеспечению можно было уверенно использовать при его разработке и тестировании, они должны обладать рядом характеристик, которые зафиксированы в стандартах, регламентирующих разработку программного обеспечения.

Два таких стандарта — IEEE 830 [6] и IEEE 1233 [7] — определяют следующие характеристики правильно составленных требований к ПО.

- *Адекватность*, соответствие реальным потребностям пользователей ПО.
- *Однозначность*, отсутствие двусмысленностей и возможностей разного толкования.
- *Полнота* — отражение в требованиях всех существенных потребностей и всех ситуаций, в которых система должна будет функционировать.
- *Непротиворечивость* или согласованность между разными элементами требований.
- *Систематичность* представления — требования должны быть описаны в рамках некоторой системы с четким указанием места каждого требования среди остальных, с определением связей и зависимостей между ними и приоритетности для различных заинтересованных лиц.
- *Прослеживаемость* — требования должны иметь четко определенные связи с модулями разрабатываемой системы, частями проектной документации и тестами, чтобы всегда можно было определить, для выполнения или проверки каких требований создан каждый из этих элементов и насколько он им соответствует.

- *Проверяемость* или возможность для каждого требования однозначно установить при помощи некоторых действий, выполнено это требование или нет.
- *Модифицируемость* или возможность внесения изменений в набор требований с максимально быстрым отслеживанием последствий такой модификации и исправлением всех возникающих при этом дефектов с точки зрения других характеристик.

В ходе работы над создаваемой программной системой или переработки уже существующей всегда, в том или ином виде, проводится *анализ требований*, цель которого — подготовить представление требований к ПО, имеющее все указанные характеристики. Анализ требований обычно включает следующие виды деятельности.

- ***Выделение требований***. Его задача — определить полный список требований, уточнить недостаточно четко сформулированные и определить возможные компромиссы в тех случаях, когда различные заинтересованные лица высказывают противоречащие друг другу требования.
  Выделение требований включает определение доступных источников требований, извлечение требований из них, в ходе чего, собственно, и фиксируются отдельные требования, и согласование требований, полученных от разных источников, при возникновении необходимости в этом.
  При извлечении требований может применяться широкий диапазон различных техник — от простого анализа задач, анализа имеющихся документов, до проведения интервью, опросов и семинаров, с использованием специальных методов для фиксации как высказываемых пожеланий и формулировок, так и эмоционального состояния опрашиваемых, чтобы в дальнейшем оценить правдивость и полноту предоставленных сведений.
- ***Систематизация и описание требований***, их сведение в единую систему и составление представляющих ее моделей, отражающих различные аспекты собранных требований. При систематизации особое внимание уделяется полному отражению всех извлеченных сведений в требованиях, определению связей и зависимостей между требованиями, идентификации требований, необходимой, чтобы иметь возможность ссылаться на них из различных проектных документов.
- ***Валидация и верификация требований.***
  Задача этих видов деятельности — проверка необходимых свойств требований к ПО. Валидация представляет собой проверку адекватности и полноты требований, то есть проверяет, что зафиксированные в требованиях ограничения действительно представляют потребности пользователей, заказчиков и других заинтересованных лиц, а также, что все их существенные потребности нашли соответствующее отражение в требованиях.
  Верификация проверяет внутреннюю согласованность, непротиворечивость и однозначность требований, а также их проверяемость и возможность проследить связи требований друг с другом, с кодом, тестами и другими проектными документами.

Ошибки в программном обеспечении

Наиболее наглядными результатами тестирования являются обнаруженные в тестируемой системе ошибки, то есть расхождения между ее реальным поведением и требованиями.

В литературе по программной инженерии слово «ошибка» используется в нескольких различных значениях.

- Иногда, хотя и достаточно редко, так называют произвольный *дефект* программной системы, будь то сбой, полностью разрушающий данные системы, неточно прорисованная буква на кнопке графического интерфейса пользователя или

нестандартное форматирование исходного кода. В англоязычной литературе в этом смысле чаще всего употребляется термин *defect*.

- Часто ошибкой или *сбоем* называют наблюдаемое нарушение требований, проявляющееся при некотором сценарии работы рассматриваемой системы. В английском языке этому значению соответствует термин *failure*.

- Ошибка в коде программы, вызывающая сбой, и состоящая в неправильном использовании какой-то конструкции языка программирования, употреблении лишней конструкции или в пропуске необходимой. В англоязычной литературе ошибка такого рода называется *fault*.
  Ошибка такого рода определена нечетко, в отличие от предыдущего случая, поскольку неправильная работа некоторого кода часто может быть исправлена несколькими разными способами. Если есть несколько конструкций, исправление каждой из которых удаляет эту ошибку, тяжело определить, какая именно из них ошибочна.

- Ошибка аналитика, архитектора или программиста, заключающаяся в неправильном понимании определенного требования или ограничения, в том, что какое-то требование забыто, или, наоборот, используется лишнее требование. По-английски такая ошибка называется обычно *error*.
  Иногда в англоязычной литературе термин *error* употребляется еще и в другом смысле — так называют некорректные, не соответствующие наложенным ограничениям данные, все равно, возвращаемые системой в ответ на какой-либо запрос или возникающие в ходе внутренних вычислений в системе и не видимые извне.

Основной смысл терминов failure, fault и error достаточно тесно связан с основными источниками ошибок, которых тоже три.

- *Неправильное понимание задач.*
  Очень часто люди не понимают или понимают неправильно то, что им пытаются сказать другие. Разработчики ПО тоже не всегда понимают, что именно нужно сделать. Дополнительным источником трудностей может служить отсутствие четкого понимания задач у самих пользователей и заказчиков — чаще всего они лишь приблизительно могут сформулировать проблему или могут попросить сделать несколько не то, что им действительно нужно.
  Ошибки такого рода тяжелее всего обнаруживаются и стоят достаточно дорого.
  Для их предотвращения нужно проводить тщательный анализ предметной области, уточнять каждое сформулированное требование, анализировать его причины и связи с другими требованиями и ограничениями, переформулировать выявленные пожелания и требования, выяснять и уточнять корректность полученных формулировок у пользователей и экспертов в предметной области. Поскольку суть таких ошибок — неадекватное понимание требований, защититься от них помогает постоянный контроль этого понимания на основе формирования различных его следствий и проверки их корректности у экспертов и пользователей.

- *Неправильное решение задач.*
  Даже правильно поняв, что нужно сделать, разработчики часто выбирают неправильный подход к тому, как это делать. Выбираемые решения могут обеспечивать лишь некоторые из требуемых свойств, могут решать поставленную задачу лишь для одного класса ситуаций из нескольких возможных, они могут подходить для данной задачи в теории, но плохо работать на практике, в конкретных обстоятельствах и в том окружении, в которых должна будет работать создаваемая система.
  Помочь в выборе правильного решения может сопоставление альтернативных решений, тщательный их анализ на предмет соответствия всем требованиям, поддержание постоянной связи с пользователями и заказчиками, предоставление им информации о предлагаемых решениях, демонстрация прототипов, анализ пригодности выбираемых

решений для работы именно в том контексте, в котором они будут использоваться, тестирование прототипов и моделей.

- *Неправильный перенос решений в код.*
  Имея правильное решение правильно понятой задачи, люди, тем не менее, способны сделать достаточно много ошибок при его воплощении. Корректному представлению решений в коде могут помешать как обычные опечатки, так и забывчивость программиста или его нежелание отказаться от привычных приемов, которые не дают возможности аккуратно записать именно то, что нужно.
  Хотя такие ошибки обычно более легко устраняются, они способны нанести не меньший ущерб, чем ошибки предыдущих видов.
  С этими ошибками можно справиться при помощи инспектирования кода, взаимного контроля, при котором разработчики внимательно читают код друг друга, опережающей разработки модульных тестов и тестирования.

Среди специалистов в программной инженерии установилось понимание того, что при достижении системой определенного уровня сложности ошибки в ней становятся неустранимыми — сколько бы усилий не было потрачено на их поиск и отладку, гарантировать безошибочность такой системы невозможно. Это связано, в первую очередь, с существованием определенного порога сложности, за пределами которого человек уже не в состоянии представлять себе происходящее в системе во всех деталях, в частности, с невозможностью абсолютно четко сформулировать полный и согласованный набор требований к системе, решающей большой набор сложных задач. С другой стороны, реальные требования к полезному ПО постоянно изменяются в связи с изменениями в потребностях пользователей, бизнеса и общества, для удовлетворения которых создаются программные системы, поэтому разработка точных и согласованных требований может просто не успевать за развитием рынка.

Ошибки в ПО иногда приводят к серьезным инцидентам и значительным убыткам для использующих его организаций. Конечно, чаще возникают мелкие ошибки, не приводящие к серьезным последствиям. Зависимость между количеством ошибок и размером их последствий подчиняется закону Ципфа (Zipf) [9] — количество случающихся сбоев примерно обратно пропорционально величине наносимого ими ущерба.

Далее описывается несколько примеров ошибок в программном обеспечении, имевших серьезные последствия.

- Ошибка в системе управления космическим аппаратом Mariner 1 [10].
  Эта ошибка привела к уничтожению одного из первых кораблей, направлявшегося к Венере, через несколько минут после запуска 22 июля 1962 года.
  В ходе полета антенна связи вышла из строя, связь со службой управления была потеряна, и управление полетом взял на себя бортовой компьютер. Однако в одной из формул для расчета положения было забыто усреднение скорости по нескольким последовательным измеренным значениям — в результате небольшие колебания скорости, связанные с неточностью измерительной аппаратуры, стали рассматриваться системой как серьезные, она стала предпринимать «корректирующие» действия, в результате чего корабль сошел с курса и был уничтожен.

- Ошибка в программном обеспечении, управляющем аппаратом радиационной терапии Therac-25 [11].
  За 1985-1987 годы зафиксировано 6 инцидентов со смертельным исходом, связанных с его работой. В трех из них непосредственной причиной смерти пациентов было признано именно их повышенное облучение из-за ошибки в программной системе управления аппаратом.
  Аппарат имел два режима облучения — мягкое облучение электронами и рентгеновское облучение. Во втором случае с источника электронных лучей снимался фильтр, который

ослаблял их интенсивность, но между пациентом и источником излучения устанавливался специальный экран, падая на который мощные электронные лучи вызывали рентгеновское излучение.
Ошибка проявлялась, когда оператор сначала включал первый режим, а потом слишком быстро переключал аппарат на второй. При этом ослабляющий фильтр снимался, а экран не устанавливался, и пациент подвергался очень интенсивному облучению электронными лучами. Кроме того, оператору при этом сообщалось, что пациент не получил никакой дозы, что не позволяло адекватно среагировать на происходящее.
Ошибка возникала лишь иногда и была связана с несинхронизованным выполнением модулей, управлявших различными элементами аппарата. При эксплуатационном тестировании она не была обнаружена, поскольку операторы тогда еще не научились переключать режимы достаточно быстро.

- Ошибка в системе управления космическим аппаратом Фобос 1 [12].
Привела к потере связи с кораблем, уже находившимся на пути к Марсу, 2 сентября 1988 года. Корабль перестал ориентироваться в пространстве, не смог сориентировать солнечные батареи и израсходовал аккумуляторы, поскольку были отключены навигационные приборы для определения положения относительно Солнца.
Команда отключения приборов была в тестовой подпрограмме, использовавшейся на Земле для проверки работоспособности отдельных систем, удалить этот код не успели перед вылетом, поскольку он был записан в памяти, предназначенной только для чтения, а для ее замены требовалось существенное время. Казалось, однако, что в ходе полета эта программа никогда не будет вызвана. Но при передаче команд по корректировке курса 29 августа 1988 года была допущена ошибка — пропущен один символ, что привело к выполнению этой тестовой программы.
Другой корабль этой серии, Фобос 2, был также потерян из-за какой-то ошибки в системе управления 27 марта 1989 года, уже на орбите вокруг Марса.

- 25 февраля 1991 года во время Первой войны в Персидском заливе американская система ПВО Patriot не смогла сбить иракскую ракету Скад, которая в результате попала в барак американской армии, убив 28 человек и ранив около ста [13].
Причиной промаха Patriot, как выяснилось, было накопление ошибок округления за время работы системы. Время в ней измерялось в десятых долях секунды, а числа были представлены в 24-битном двоичном формате с плавающей точкой. При представлении 1/10 как двоичной дроби с 24-мя цифрами возникает небольшая ошибка.
В рассматриваемом случае система Patriot работала без перезагрузки около 100 часов. За это время накопление погрешности определения времени дало ошибку около 1/3 секунды. Поскольку ракета Скад летит со скоростью 1700 м/с, ошибка в 1/10 секунды при расчете ее траектории уже не дает возможности ее сбить.

- Многочисленные ошибки в системе управления двигателями и навигационной системе считаются наиболее вероятной причиной катастрофы вертолета Chinook ZD 576 [14], произошедшей 2 июня 1994 года на мысе Кинтайр. В этой катастрофе погибли 25 экспертов и высокопоставленных сотрудников отдела разведки Великобритании в Северной Ирландии, что на значительное время парализовало работу этого отдела.

- Ошибка в системе управления ракеты Ариан-5 привела к ее уничтожению при первом запуске этой ракеты 4 июня 1996 года [15].
Долгое время эта ошибка, приведшая к убыткам в размере 500 миллионов долларов США, считалась самой дорогостоящей ошибкой в программной системе.
Ошибка состояла в том, что без изменений использовался модуль расчета траектории из системы управления ракетой Ариан-4. В нем горизонтальная составляющая скорости ракеты представлялась 16-битным числом. Ариан-5 могла выдерживать более значительные ускорения и большую кривизну траектории, из-за чего в ходе полета значение горизонтальной скорости вышло за пределы представимых 16-ю битами чисел.

Специальной процедуры обработки такой ситуации не было, поэтому возникшее исключение обрабатывалось модулем обработки общих сбоев, который остановил данный процесс и запустил новый с теми же исходными данными, что вновь привело к той же ошибке. В результате система не смогла вычислить правильное текущее положение ракеты и стала использовать ранее полученные данные. Это привело к попытке «скорректировать» курс и «болтанию» ракеты, после чего она была уничтожена.

- Ошибка в системе управления космическим аппаратом Mars Climate Orbiter [16].
  Привела к его выходу на слишком низкую орбиту вокруг Марса 23 сентября 1999 года и к последовавшему за этим разрушению.
  Необходимые корректировки к движению корабля рассчитывались специальной программой на Земле и после передавались в виде команд двигателям аппарата. Ошибка состояла в том, что управляющая программа на Земле использовала значения импульсов в фунтах силы на секунду, а бортовая система передавала ей значения, измеренные в Ньютонах на секунду. В результате были использованы неправильные команды корректировки.
- Одной из причин сбоя в электроснабжении северо-востока Северной Америки 14 августа 2003 года, на несколько часов оставившего без электричества 50 миллионов человек и приведшего к потерям на сумму около 6 миллиардов долларов США, была ошибка в программной системе оповещения о сбоях на электростанции, связанная с неаккуратной синхронизацией параллельно работающих процессов [17].

Можно отметить, что в большинстве примеров ошибок, имевших тяжелые последствия, нельзя однозначно приписать всю вину за случившееся ровно одному недочету или дефекту, одному месту в коде. К таким последствиям чаще всего приводят ошибки системного характера, затрагивающие многие элементы системы и различные аспекты ее устройства. Это значит, что при анализе такого происшествия обычно выявляется множество частных ошибок, нарушений действующих правил, недочетов в инструкциях и требованиях, которые совместно привели к создавшейся ситуации.

Даже если ограничиться рассмотрением только ПО, часто одно проявление ошибки (failure) может выявить несколько дефектов, находящихся в разных местах. Такие ошибки возникают обычно в тех ситуациях, в которых поведение системы недостаточно четко определяется требованиями (а иногда и вообще никак не определяется, вследствие неполного понимания задачи). Поэтому разработчики различных модулей ПО имеют возможность по-разному интерпретировать те части требований, которые относятся непосредственно к их модулям, а также иметь разные мнения по поводу области ответственности каждого из взаимодействующих модулей в данной ситуации. Если различия в их понимании не выявляются достаточно рано, при разработке системы, то становятся «минами замедленного действия» в ее коде.

При подготовке и проведении тестирования эти закономерности стоит учитывать, это помогает как находить ошибки быстрее и с меньшими трудозатратами, так и более аккуратно анализировать их последствия и устранять все возможные связанные с ними эффекты.

Литература

[1] ISO/IEC 9126-1:2001. Software engineering — Software product quality — Part 1: Quality model, 2001.

[2] ISO/IEC TR 9126-2:2003 Software engineering — Product quality — Part 2: External metrics, 2003.

[3] ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics, 2003.

[4] ISO/IEC TR 9126-4:2004 Software engineering — Product quality — Part 4: Quality in use metrics, 2003.
[5] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.
[6] IEEE 830-1998. Recommended Practice for Software Requirements Specifications. New York: IEEE, 1998.
[7] IEEE 1233-1998. Guide for Developing System Requirements Specifications. New York: IEEE, 1998.
[8] В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт 13, ИСП РАН, Москва, 2006. http://panda.ispras.ru/~kuliamin/docs/Req-2006-ru.pdf.
[9] Статья Wikipedia о законе Ципфа http://en.wikipedia.org/wiki/Zipf%27s_law.
[10] http://nssdc.gsfc.nasa.gov/nmc/tmp/MARIN1.html.
[11] N. Levenson, C. S. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, 26(7):18-41, July 1993.
[12] R. Z. Sagdeev, A. V. Zakharov. Brief history of the Phobos mission. Nature 341:581-585, 1989.
[13] G. N. Lewis, S. Fetter, L. Gronlund. Casualties and Damage from Scud Attacks in the 1991 Gulf War, 1993. http://web.mit.edu/ssp/Publications/working_papers/wp93-2.pdf.
[14] http://www.publications.parliament.uk/pa/ld200102/ldselect/ldchin/25/2501.htm.
[15] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.
[16] Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.
[17] http://www.nyiso.com/public/webdocs/newsroom/press_releases/2005/blackout_rpt_final.pdf.

Как защититься от ошибок

В прошлой лекции шла речь о качестве программного обеспечения (ПО) и его возможных дефектах — ошибках. Каким образом можно обеспечить качество и, тем самым, защититься от ошибок в ПО? Эту задачу решают виды деятельности в рамках разработки и сопровождения ПО, связанные с *обеспечением качества*. Любой систематический подход к обеспечению качества должен включать следующие три пункта.

- Методы предотвращения ошибок
- Методы обнаружения ошибок
- Методы исправления ошибок

Если ошибки не обнаружить или не исправить, то они не исчезнут, поэтому последние два пункта необходимы. Методы предотвращения ошибок нужны для снижения нагрузки на работу по оставшимся пунктам — иначе в сложных случаях можно оказаться в ситуации, в которой количество обнаруживаемых ошибок значительно превышает возможности их исправления (на практике так тоже бывает, но величину этого превышения стараются держать ограниченной). Также, стоит помнить, что самые совершенные методы предотвращения ошибок не способны справиться со всеми их видами — все равно ошибки возникают и их надо уметь находить, т.е., наличие высокоэффективных технологий предотвращения ошибок не делает работу по их обнаружению и исправлению ненужной.

Примеры методов предотвращения ошибок.

- Стандартизация интерфейсов широко используемых библиотек, протоколов взаимодействия и создание хорошей документации по их интерфейсам. Примерами таких стандартов являются POSIX, стандартная библиотека Java JDK. Хорошая, соответствующая реальной работе описываемыхого ПО и одновременно

лишенная неполноты, двусмысленностей и рассогласованности документация по стандартам позволяет разработчикам, участвующим в реализации билиотек точнее и полнее понимать, какую функциональность нужно обеспечить и избегать ошибок в ее реализации и несогласованностей при реализации одного стандарта разными разработчиками. Она же дает возможность тем программистам, которые пользуются этими библиотеками, с меньшими усилиями, точнее и полнее понимать их функции, не тратить время на эксперименты и отладку с целью выяснения точных правил их работы, избегать затрат на создание кода, переносимого между несколькими несогласованными реализациями одного стандарта.

- Разработка новых конструкций языков программирования, позволяющих эффективно проверять больше свойств корректности программ, и устранение конструкций, вызывающих многочисленные ошибки.
  Например, частое использование инструкции goto без ограничений на место перехода приводит к запутанному коду, с большим количеством ошибок. Поэтому в большинстве современных языков эта инструкция отсутсвует или употрбеляется крайне ограниченно.
  В Eiffel предложено использование конструкций, которые позволяют избавиться от одной из наиболее часто встречающихся ошибок — попытке обращения к методу или атрибуту по пустой ссылке (NullPointerException). Решение состоит в использовании необнуляемых типов (тип ссыок на объекты, которые не могут быть пустыми) и конструкции завершения цикла при обнаружении пустой ссылки в обрабатываемой им коллекции. После введения этих конструкций компилятор объявляет ошибкой любой проход по ссылке обнуляемого типа, сделанный вне блока, в начале которого стоит проверка этой ссылки на равенство null.
- Внедерние стандартов кодирования, делающих код программ более понятным и позволяющих тратить меньше времени и ресурсов на понимание и внесение изменений в программы.
- Регулярное предварительное обсуждение реализуемых проектных и программистких решений в группе, позволяющее избегать ошибок, связанных с пропуском в коде обработки специфических ситуаций и игнорированием важных элементов требований.

Исправление ошибок по существу мало отличается от собственно разработки кода, используя дополнительно специфические техники отладки — локализации ошибок при помощи постепенного сужения области анализа.

Методы контроля качества ПО

Данный курс посвящен тестированию — одному из наиболее широко используемых методов контроля качества ПО, или, методов обнаружения ошибок, однако оно — не единственный такой метод.

Методы *контроля качества* предназначены для проверки различных характеристик качества ПО и поиска различных дефектов, связанных с недостаточным качеством. Они обычно разделяются на верификацию и валидацию.

В рамках *верификации* качество некоторого артефакта (документа, модели, элемента кода и пр.) проверяется за счет сопоставления его с другим артефактом, на основе которого первый должен был быть разработан или которому он должен соответствовать (а также за счет проверки его соответсвия принятым нормам и стандартам разработки). Так, верифицировать код можно, имея на руках описание требований, проектные решения или стандарты кодирования, верифицировать проектный документ можно с помощью требований или стандартов на оформление подобных документов. Верифицировать можно и реальное

поведение системы — сопоставляя его с требованиями, проектными решениями, принятыми стандартами функционирования систем такого рода.

Валидация обозначает проверку некоторого артефакта разработки на соответствие конечным целям, для достижения которых это ПО предназначено, т. е., нуждам и потребностям его пользователей и заказчиков. При валидации ПО проверяется, что оно действительно решает нужные пользователям задачи и удовлетворяет их потребности (даже если эти задачи и потребности описаны плохо и неполно). Валидация обычно проводится представителями заказчика, пользователями, экспертами в предметной области, т.е. людьми, обладающими достаточной компетентностью, чтобы судить о достижении поставленных целей. Если же эти цели формализовать, описать точно и полно, то проверка на соответствие полученному документу будет верификацией. Валидация необходима, потому что обычно согласованное, полное и точное описание задач сложной системы практически невозможно, разрабатываемые документы со спецификациями требований и пр. являются только приближениями к такому описанию. При валидации, могут использоваться те же техники, что и при выявления требований, поскольку цели этих видов деятельности похожи — преобразовать неясные и неформальные пожелания и представления о работе ПО в более точную форму (при валидации — в оценку проверяемых характеристик качества).

Различие между верификацией и валидацией проиллюстрировано на Рис. 1.

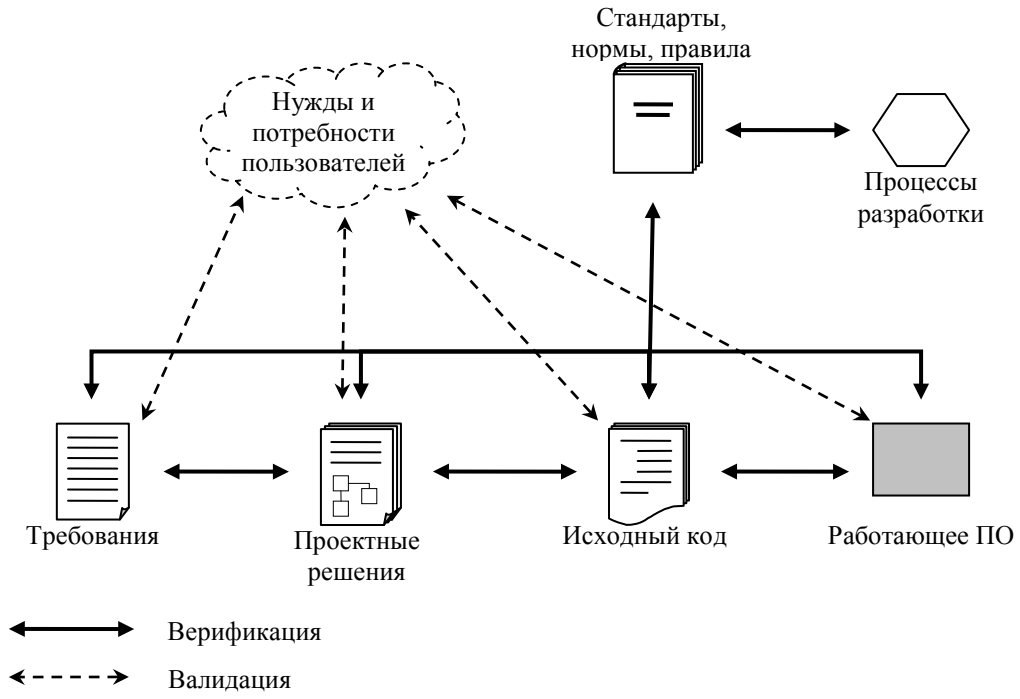

Рисунок 2. Соотношение верификации и валидации.

Методы верификации существенно строже, они чаще могут быть формализованы и автоматизированы. Скорее по историческим, чем по содержательным причинам, методы верификации делятся на следующие группы [1].

- *Методы экспертизы* (*review, inspection*). При экспертизе верификацию проводит человек, обладающий значительным опытом проведения такого рода проверок (часто также в экспертизу включаются неопытные сотрудники с целью их обучения). Экспертиза бывает общей, нацеленной на выявление любых дефектов и ошибок, или специализированной, направленной на оценку отдельных характеристик качества (например, гибкости архитектуры, удобства использования или защищенности ПО). Обычно в качестве видов экспертиз выделяют организационные экспертизы (management review), технические экспертизы (technical review), сквозной контроль

(walkthrough), инспекции (inspection) и аудиты (audit).

Экспертиза применима к любым свойствам ПО и любым артефактам жизненного цикла и на любом этапе проекта, хотя для разных целей могут использоваться разные ее виды. Она позволяет выявлять практически любые виды ошибок, причем делать это на этапе подготовки соответствующего артефакта, тем самым минимизируя время существования дефекта и его последствия для качества итогового продукта. В то же время экспертиза весьма трудоемка, требует активного участия опытных людей и не может быть автоматизирована. Эффективность экспертизы существенно зависит от опыта и мотивации ее участников, организации процесса, а также от обеспечения корректного взаимодействия между различными участниками. Это накладывает дополнительные ограничения на распределение ресурсов в проекте и может приводить к конфликтам между разработчиками, если руководство проекта обращает мало внимания на коммуникативные аспекты проведения экспертиз

- *Методы статического анализа* (*static analysis*). Такие методы используют автоматический анализ кода, моделей или других документов разработки с целью проверки выполнения формализованных правил их оформления (синтаксической корректности) и поиска часто встречающихся ошибок по определенным шаблонам (разыменования нулевых указателей, обращения к неинициализированным данным, деления на 0, взаимной блокировки параллельных процессов и пр.).

  Статический анализ выполняется с помощью специализированных инструментов, техники статического анализа кода, которые достигли достаточной зрелости и поддерживаются эффективными алгоритмами, чаще всего включаются в состав компиляторов. Однако, статический анализ обычно способен выявлять лишь ограниченный набор видов ошибок.

  Основной проблемой многих техник статического анализа является следующая дилемма: либо используются строгие методы анализа, не допускающие пропуска ошибок (тех типов, что ищутся), но приводящие к большому количеству сообщений о возможных ошибках, которые таковыми не являются (ложные сообщения об ошибках), либо набор сообщений об ошибках является точным, но возникает возможность пропустить ошибку. Обычно используются компромиссные решения, позволяющие обнаруживатькак можно больше ошибок, но допускающие не слишком высокий процент ложных сообщений. Для выявления ложных сообщений об ошибках привлекаются достаточно опытные разработчики, поэтому высокий процет таких сообщений означает большие трудозатраты на анализ результатов работы инструментов. В тех случаях, когда определенная техника статического анализа включается в компилятор, часто прибегают к объявлению всех обнаруженных с ее помощью проблем ошибками (даже если при их наличии программу можно скомпилировать в работоспособный код).

- *Методы динамического анализа* (*dynamic analysis*). Эти методы выполняют верификацию реальной работы ПО или работы его кода или исполнимой модели (из которой код получают автоматизированной трансляцией) в специализированном окружении (например, при отсутствии нужного оборудования или внешних систем производится их эмуляция). При этом собирается информация о результатах работы в различных ситуациях, и эта информация далее подвергается анализу на предмет соответсвия требованиям или проектным решениям. Обычно к таким методам относят *тестирование*, при котором работа ПО проверяется на заранее выбранном наборе ситуаций; *мониторинг*, в рамках которого работа ПО протоколируется и оценивается при его обычной или пробной эксплуатации; а также *профилирование*, специфический вид мониторинга временных характеристики работы ПО и использования им отдельных ресурсов. К динамическим методам анализа относят также и имитационное тестирование и имитационный мониторинг, при которых ПО

выполняется в рамках какого-то модельного окружения, построенного с использованием симуляторов и эмуляторов.
Для применения динамических методов необходимо иметь работающую систему или хотя бы некоторые ее компоненты, или же их прототипы, поэтому нельзя использовать их на первых стадиях разработки. Зато с их помощью можно контролировать характеристики работы системы в ее реальном окружении, которые часто невозможно аккуратно проанализировать с помощью других подходов. Динамически методы позволяют обнаруживать в ПО только ошибки, проявляющиеся при его работе, а, скажем, дефекты удобства сопровождения найти не помогут, однако, обнаруживаемые ими ошибки обычно считаются более серьезными. Еще одна проблема динамических методов — получаемые с их помощью результаты сильно засисят от полноты и разнообразия возникающих в ходе выполнения проверяемой системы ситуаций. Если набор исследованных ситуаций недостаточно широк, выводы о качестве системы, сделанные только на основе их выполнения, будут необоснованы. Методы статического анализа этим недостатком не обладают — они позволяют исследовать даже самые запутанные ситуации, которые крайне нелегко воспроизвести при работе проверяемой системы.

- *Формальные методы верификации* (*formal verification*) используют формальные математические модели проверяемого артефакта и того, на соответствие которому проводится проверка. Чаще всего это модели, соответственно, кода или проектных решений и требований. К формальным методам относятся *дедуктивный анализ* и *проверка моделей*. В рамках дедуктивного анализа свойства кода или проектных решений, а также требования формализуются в виде наборов логических утверждений (могущих включать элементы логики высших порядков и различных алгебраических структур), и из первого набора I (решения) пытаются формально вывести второй S (требования), т.е., пытаются строго доказать выводимость $I \vdash S$. Проверка моделей основана на представлении проверяемых решений в виде автоматных моделей M, а требований к ним — в виде наборов логических утверждений S (обычно с элементами временной или модальной логики). Далее выполняется автоматическая проверка выполнения $M \models S$ этих утверждений на полученных моделях с помощью специализированных инструментов. При этом обнаруживаемые ошибки могут быть сразу же оформлены как контрпримеры, конкретные сценарии работы проверяемой модели, при которых сформулированные требования нарушаются.
Формальные методы применимы только к тем свойствам, которые выражены в рамках некоторой математической модели, а также к тем артефактам, для которых можно построить адекватную формальную модель. Для использования таких методов необходимо затратить значительные усилия на построение формальных моделей. К тому же, построить такие модели и провести их анализ могут только специалисты по формальным методам, которых не так много, и чьи услуги стоят достаточно дорого. Построение формальных моделей нельзя автоматизировать, для этого всегда необходим человек. Анализ их свойств в значительной мере может быть автоматизирован, и сейчас уже есть инструменты, способные анализировать формальные модели промышленного уровня сложности, однако чтобы эффективно пользоваться ими часто тоже требуется очень специфический набор навыков и знаний (в специфических разделах математической логики и алгебры).
Тем не менее, в ряде областей, где последствия ошибки в системе могут оказаться чрезвычайно дорогими, формальные методы верификации активно используются. Они способны обнаруживать сложные ошибки, практически не выявляемые с помощью экспертиз или тестирования. Кроме того, формализация требований и проектных решений возможна только при их глубоком понимании, и поэтому вынуждает провести тщательнейший их анализ, для чего часто необходима

совместная работа специалистов по формальным методам и экспертов в предметной области.

В последние 15-20 лет появились основанные на формальных методах инструменты, решающие ряд задач верификации функциональных требований, но зато способные эффективно работать в промышленных проектах.

Гораздо чаще, чем к программам, формальные методы верификации на практике применяются к аппаратному обеспечению. Их использование в этой области имеет более долгую историю, что привело к созданию более зрелых методик и инструментов. Это обусловлено более высокой стоимостью ошибок для аппаратного обеспечения, более однородной его структурой и более простыми базовыми элементами моделей аппаратного обеспечения, более широким многократным использованием проектной информации.

Приведенная классификация методов верификации не является полностью строгой. За последнее десятилетие активно развиваются инструменты верификации ПО, использующие техники, относящиеся одновременно к нескольким из перечисленных пунктов. В качестве примеров таких методов можно привести следующие: расширенный статический анализ, построение полных тестов на основе формальных моделей, синтетическое структурное тестирование.

Тестирование на основе формальных моделей [2,3] использует для построения тестов формальные модели требований к ПО и принятых проектных решений. В рамках тестов проверяются ограничения, описанные в моделях, а критерий полноты, на базе которого строится набор тестов, обычно использует структуру модели и имеет формальное обоснование, позволяющее утверждать, что при соблюдении определенных гипотез (обычно касающихся представимости реального поведения в выбранном формализме, ограниченности реализации, т.е., отсутсвия в ней неограниченно сильно отличающихся от модели элементов поведения, и ее однородности, т.е., отсутсвия в ней большого количества незатрагиваемых моделью деталей) такой набор тестов действительно способен выявить все ошибки (все отклонения в поведении тестируемой реализации от модели).

Расширенный статический анализ [4] (extended static checking) использует формальные спецификации поведения отдельных функций и операций и дедуктивный анализ для проверки выполнения этих спецификаций кодом.

Синтетическое структурное тестирование [5-8] использует элементы формальных спецификаций и статического анализа для генерации тестов, на основе результатов работы которых, опять же, может быть проведен более глубокий статический анализ. Наиболее широко известным представителем этой группы техник является метод DART (Directed Automated Random Testing) [7,8].

Определение тестирования

Для тестирования в литературе можно найти много определений, выделяющих несколько различающиеся понятия, например, следующие.

- Процесс выполнения программы с намерением найти ошибки (Glenford J. Myers [9]).
- Любая деятельность, направленная на оценку некоторых характеристик программного обеспечения и проверку того, что они соответствуют требуемым результатам (William C. Hetzel [10]).
- Процесс анализа программного обеспечения с целью обнаружения расхождений между его реальными и требуемыми свойствами и оценки его характеристик [11].
- Это не деятельность, а дисциплина ума, позволяющая получить программное обеспечение с низкими рисками небольшими усилиями (Boris Beizer [12]).

- Процесс выполнения программного обеспечения в определенных условиях, наблюдения или протоколирования результатов его работы и оценки некоторых его аспектов [13].
- Процесс выполнения программного обеспечения для проверки того, что оно удовлетворяет специфицированным требованиям, и для нахождения ошибок [14]
- Технический анализ программного обеспечения, проводимый эмпирически, для получения информации о его качестве с точки зрения определенного круга заинтересованных лиц (Cem Kaner [15]).
- Проверка соответствия между реальным и ожидаемым поведением программного обеспечения в конечном наборе ситуаций, выбранных определенным образом [16].
- Формальный процесс, выполняемый специализированной командой, при котором поведение программного обеспечения проверяется при его выполнении на компьютере в рамках утвержденных тестовых процедур и вариантов [17].
- Процесс, состоящий из всех деятельностей жизненного цикла, статических и динамических, связанных с оценкой программного обеспечения и относящихся к нему рабочих материалов (а также планированием и подготовкой этой оценки) для определения их соответствия заданным требованиям, демонстрации их пригодности для поставленных целей и обнаружения дефектов [18].

Большинство этих определений пытаются описать понятие тестирования вне контекста других возможных методов верификации, и часто в результате получается само понятие верификации.

Для целей этого курса наиболее подходящим является определение из SWEBOK [16], указанное в восьмом пункте списка. Мы будем использовать его далее в немного уточненном виде.

Тестированием называется проверка соответствия поведения проверяемой системы требованиям, выполняемая по результатам реальной работы этой системы в некотором конечном наборе специально созданных ситуаций [16]. При этом проверяемая система обычно называется *тестируемой системой* (system under test или SUT по-английски).

В этом определении можно отметить следующие аспекты.

- ***Проверка соответствия требованиям.***
  Тестирование в этом смысле возможно лишь при наличии требований к программе. Если от системы ничего не требуется, она может делать все, что угодно, и это нельзя считать неправильным.
  Тестирование позволяет проконтролировать качество программной системы ровно настолько, насколько полно, четко и недвусмысленно определены требований к ней. В тех случаях, когда явные требования не сформулированы, тестирование можно использовать, только основываясь на некоторых неявно подразумеваемых, но желательных свойствах тестируемой системы, например, отсутствии сбоев в ее работе. Однако бывают ситуации, в которых системы определенного типа наоборот, должны демонстрировать сбои (встроенные системы или базовое ПО в определенных случаях должны просто падать, поскольку попытка обработки в значительной мере некорректных данных может существенно снизить общую эффективность таких систем). Поэтому даже в тех случаях, когда пытаются тестировать что-то, не описывая проверяемые требования явно, их выявление и четкая формулировка позволяют полнее понять, что именно и в каких случаях нужно проверить и сэкономить массу усилий при дальнейшем сопровождении и развитии проверяемой системы.

- ***Результаты реальной работы системы.***
  Тестирование основывается на результатах реальной работы тестируемой системы. Проверяемая программа должна выполняться, чтобы проводимую при этом проверку можно было считать тестированием.
  По сути, это означает, что тестирование является разновидностью динамического анализа.
  Использование реальной работы тестируемой системы позволяет применять тестирование для проверки корректности поведения системы в ее рабочем окружении, на месте ее эксплуатации, что невозможно сделать при помощи других методов контроля качества ПО.
  Бывает, однако, имитационное тестирование, при проведении которого основные действия (определение проверяемых совйств, выбор критериев полноты, разработка тестов, выполнение тестов, анализ результатов) примерно такие же, как и при обычном тестировании, но проверяется работа не самой системы, а какой-то ее исполнимой модели или прототипа. Важно, что и в этой ситуации должна использоваться модель, способная исполняться (на симуляторе или виртуальной машине).

- ***Специально созданные ситуации.***
  Тестирование всегда выполняется в специально созданных ситуациях.
  Такие ситуации называются *тестовыми ситуациями*, а процедура или программа, при выполнении которой создается одна или несколько тестовых ситуаций и проверяется правильность поведения системы в них, называется *тестом*. Подготовка к проведению тестирования всегда включает подготовку или разработку тестов.
  Составляющие тестовых ситуаций будут рассматриваться в следующих лекциях, а различным методам разработки тестов посвящено основное содержание этого курса.
  Эта характеристика отличает тестирование от пассивного наблюдения за поведением системы или *мониторинга* (passive testing, runtime verification), при котором собираются и проверяются результаты реальной работы программы, но эта работа не управляется, не направляется на возникновение определенных ситуаций.

- ***Конечный набор ситуаций.***
  Тестирование всегда выполняется в конечном наборе ситуаций. Более того, возможное количество ситуаций, возникающих во время тестирования, ограничивается практическими соображениями достижения приемлемого компромисса между затратами времени и ресурсов проекта на разработку тестов и тестирование и пользой от него — количеством обнаруживаемых ошибок, полнотой и адекватностью получаемой оценки качества тестируемой системы.
  Практически значимые системы сейчас настолько сложны, что количество ситуаций, которые необходимо испытать для полной проверки одной такой системы, превосходит возможности сколь угодно щедро финансируемого проекта и потребует не одной человеческой жизни для выполнения. Поэтому полное тестирование, хотя и возможно теоретически, в силу конечности любой вычислительной системы, практически совершенно невыполнимо.
  С одной стороны, это означает, что проведение тестирования никогда не дает полной гарантии корректности тестируемой системы, полного отсутствия в ней ошибок. С другой стороны, это обстоятельство делает чрезвычайно важным выбор тестов для выполнения. За счет выбора тестов можно получить как большой набор, выполняющийся долго и не дающий существенной информации о качестве тестируемой системы, так и компактный, но проверяющий большое количество разнообразных аспектов поведения системы и позволяющий оценить ее качество достаточно адекватно (хотя и без абсолютных гарантий).
  Критически важно для правильного выбора тестов использовать адекватный, отражающий реальную ситуацию *критерий полноты тестирования*. Различные

способы определения таких критериев и методы выбора тестов в соответствии с ними также рассматриваются в следующих лекциях данного курса.

- Тестирование — это разновидность именно верификации, а не просто анализа. Отличие в том, что анализ дает какие-то результаты в виде чисел или качественных характеристик, а верификация (в данном случае) должна ответить на вопрос о соответствии или несоответствии требованиям. В результате проведения тестирования важно получить ответ в виде «поведение системы в таких-то и таких-то ситуациях неправильно, не соответствует требованиям», а не просто набор числовых характеристик, которые дальше нужно интерпретировать отдельно.
- Ряд специфических видов тестирования, прежде всего, тестирование удобства использования, не укладываются в данное определение. Это связано с тем фактом, что для большинства систем получить аккуратное и полное описание требований к удобству использования невозможно — большинство таких требований остаются неявными, их извлечение и формализация требуют слишком больших усилий. Поэтому для контроля удобства использования используют специфическое тестирование, в виде выполнения ряда сценариев определенным образом отобранными пользователями проверяемой системы, в ходе которого протоколируются их действия и проблемы, связанные с восприятием информации от системы и поиском нужных для выполнения очередной задачи элементов интерфейса. Источником информации для выявления проблем здесь являются не отдельно сформулированные требования, а само поведение пользователей — т.е., это разновидность валидации, а не верификации. Организация такого тестирования сильно отличается от обычной, поэтому в данном курсе эта его разновидность не рассматривается.

Перечисленные аспекты определяют как достоинства, так и недостатки тестирования по сравнению с другими методами контроля качества программного обеспечения. В отличие от аналитической верификации, тестирование не может гарантировать полного отсутствия ошибок в коде системы, но зато может проверить корректность ее работы на месте эксплуатации, при взаимодействии с другими системами, что сделать при помощи аналитической верификации крайне тяжело. Тестирование всегда ограничено по ресурсам, но и предоставляет гибкие возможности управления полнотой и объемом проводимых проверок за счет разнообразных техник разработки и выбора тестов.

Для успешного проведения тестирования огромное значение имеют требования к тестируемой системе, использовавшиеся в ходе тестирования тестовые ситуации и критерии полноты тестирования. В общем случае и требования, и критерии полноты представляются в виде некоторых *моделей* (не обязательно полностью формальных), на основе которых выбираются тесты и выполняются проверки. Даже при отсутствии явных таких моделей, они присутствуют неявно, в сознании проектировщиков тестов и тестировщиков всегда есть какие-то критерии проверки правильности поведения тестируемой программы, представляющие требования, и критерии продолжения или прекращения тестирования после получения ряда результатов, основанные на понимании его неполноты или достаточной полноты. Поэтому можно сказать, что *тестирование всегда проводится на основе некоторых моделей*, быть может, не представленных явно.

Тестирование на основе моделей, которому посвящен этот курс, отличается только тем, что *все используемые при разработке, выборе и выполнении тестов модели формулируются явно*. Наиболее важную роль среди моделей, используемых при тестировании, играют модели требований, описывающие желательное поведение системы, и модели ситуаций или критерии полноты тестирования, определяющие набор разрабатываемых или выбираемых для выполнения тестов.

Литература

[18] В. В. Кулямин. Методы верификации программного обеспечения. 2008.

[19] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, eds. *Model Based Testing of Reactive Systems*. LNCS 3472, Springer, 2005.

[20] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[21] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. *Extended static checking.* Technical Report SRC-RR-159, Digital Equipment Corporation, Systems Research Center, 1998.

[22] C. Csallner, Y. Smaragdakis. *DSD-Crasher: A hybrid analysis tool for bug finding*. Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 245-254. ACM, July 2006.

[23] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. *Feedback-Directed Random Test Generation*. Proc. of International Conference on Software Engineering, pp. 75-84, 2007.

[24] P. Godefroid, N. Klarlund, K. Sen. DART: Directed Automated Random Testing. ACM SIGPLAN Notices - Proceedings of PLDI 2005, 40(6):213-223, 2005.

[25] K. Sen, G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. LNCS 4144:419-423, Spriger, 2006.

[26] Glenford J. Myers. Software Reliability. Principles and Practices. Wiley, 1976.

[27] William C. Hetzel. The Complete Guide to Software Testing. Wiley, 1984.

[28] IEEE Standard 829: Standard for Software Test Documentation. IEEE, 1983.

[29] B. Beizer. Software Testing Techniques. 2nd edition. Int. Thomson Publishing, 1990.

[30] ANSI/IEEE 610.12-1990. Glossary of Software Engineering Terminology. NY:IEEE, 1987.

[31] BCS 7925-1. Glossary of terms used in Software Testing. 1995.

[32] Cem Kaner. Black box testing course. 1999.

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

[34] Daniel Galin. Software Quality Assurance. From theory to implementation. 2004.

[35] ISTQB Glossary of Terms used in Software Testing. 2006.