Тестирование информационных систем

Лекция 5

Проектирование и разработка тестов

Характеристики хорошего теста

Выявление программных ошибок является сложной задачей. Программная ошибка (Software error) может не приводить к наблюдаемому сбою, а, например, порождать другую программную ошибку или переводить процесс работы в некорректное состояние. Сбой (Software failure) порождается наличием одного или нескольких де¬фектов (Software defect) — недостатков в компоненте или системе. Для выявления программных ошибок используются тестовые случаи или тесты.

Тестовым случаем (Test Case) называют документ, который описывает конкретные шаги, условия и параметры, необходимые для анализа реализации тестируемой функции. Каждый тест содержит три базовые части.

- Предусловия (PreConditions) — шаги, которые переводят систему в состояние, пригодном для проведения проверки.
- Описание теста (Description) — шаги, которые переводят систему из состояния в состояние. На основании полученного результата делается вывод о соответствии реализации заявленным требованиям.
- Постусловия (PostConditions) — шаги, которые переводят систему в изначальное положение.

Проверка результата работы объекта тестирования выполняется на основе определения корректного состояния или эталонной модели результата. Эталонная модель определяется используемыми стандартами, спецификациями или ожиданиями пользователя. Эталонная модель может быть представлена множеством различных способов:

- неформальное представление того, «как ПО должно работать»;
- формальная техническая спецификация;
- набор тестовых примеров;
- корректные результаты работы программы;
- другая (априори корректная) реализация той же исходной спецификации.

Для проявления некорректных состояний необходимо создавать тесты, обладающие следующими характеристиками:

- достижение (Reachibility) — тест должен выполнить место в исходном коде, где присутствует программная ошибка;

- повреждение (Corruption) — при выполнении ошибки состояние программы должно испортиться с появлением сбоя;
- распространение (Propagation) — сбой должен распространиться дальше и вызвать неудачу в работе ПО.

Проектирование и создание тестов, которые соответствуют определенным ранее критериям качества и целям тестирования, называется Тест дизайном (Test Design). В ходе тест дизайна необходимо ответить на следующие вопросы:

- “Что тестировать?”
- “Как тестировать?”

V-модель разработки ПО

Разработка тестов неразрывно связана с этапами создания программного продукта. Используемый в классической модели жизненного цикла принцип увеличения детализации проекта находит свое применение и в разработке тестов. На рисунке представлена V-модель разработки ПО.

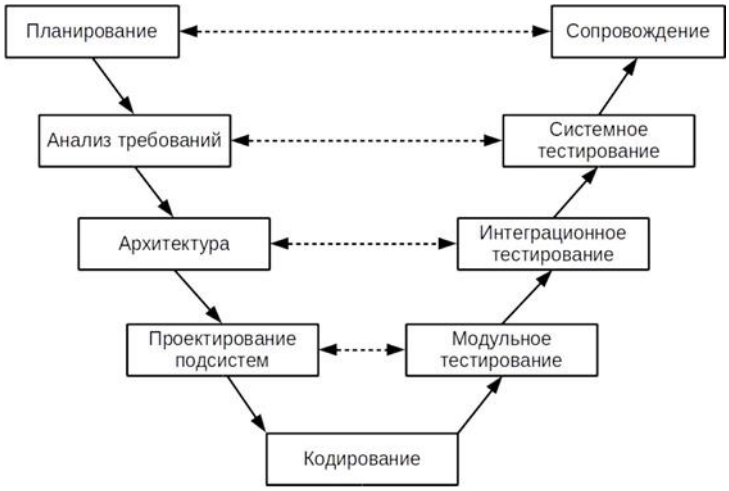

Рис. 1. V-модель разработки ПО

Каждый этап разработки ПО сопровождается соответствующими этапами разработки тестов и выполнением тестирования. В ходе сбора и анализа требований формируются аттестационные и системные тесты, которые используются в ходе приемочного и системного тестирования. Проект архитектуры позволяет определить механизмы взаимодействия между модулями, что приводит нас к созданию интеграционных тестов и затем к интеграционному тестированию. Проектирование модулей сопровождается модульными тестами и модульным тестированием.

Написание кода ПО приводит нас к последовательному усложнению тестируемого объекта: модуль - комбинация модулей - функциональное требование. Таким образом, мы получаем наборы тестов для всестороннего разноуровнего тестирования.

Позитивные и негативные тесты

В ходе разработки тестов формируются наборы, содержащие позитивные и негативные тесты. Позитивные тесты (Positive test) проверяют наличие требуемых действий в тестируемом объекте и их работоспособность. Негативные тесты (Negative test) проверяют действия тестируемого объекта в случае некорректного начала (например, неправильные входные данные или неправильная последовательность вызовов).

Позитивные тесты обладают следующими характеристиками:

- тесты, предназначенные для проверки, что программа выполняет свое основное предназначение;
- тесты на основании «правильных» входных данных;
- тестирование с целью проверки соответствий требованиям.

Для создания позитивных тестов необходимо определить допустимые значения входных параметров, требуемую функциональность и ожидаемый эталонный результат.

Основная задача негативных тестов заключается в определении и минимизации деструктивных воздействий в случае некорректной работы компонент ПО или его окружения. Негативные тесты обладают следующими характеристиками:

- тесты для проверки устойчивости ПО к негативным входным данным;
- тесты на проверку устойчивости ПО к ошибкам пользователя;
- тесты на то, что у программы нет неожиданных побочных эффектов;
- тестирование с целью «сломаем это!».

Методы разработки тестов

Для разработки тестов используются следующие методы:

- на основе внутренней структуры объекта тестирования (белый ящик);
- на основе требуемой функциональности (черный ящик).

В таблице 1 представлены основные особенности методов разработ¬ки. Разработка тестов белым ящиком предполагает непосредствен¬ное участие автора-разработчика, наличие достаточных знаний в по¬нимании логики программной реализации и может быть применена к небольшим объектам. Разработка тестов черным ящиком может быть выполнена без участия автора кода на основе имеющихся специ¬фикаций, а тестированию может быть подвержен большой и сложный объект. Методы не являются взаимозаменяемыми, а дополняют друг друга для проведения качественного тестирования.

Существует возможность комбинации методов разработки тестов (серый ящик, Gray box testing). В этом случае создатель теста знает частично или полностью внутреннее устройство тестируемого объекта, но находится на уровне пользователя.

Отличия черного и белого ящиков

Критерий	Черный Ящик	Белый Ящик
Основной уровень применимости	Приемочное тестирование	Модульное тестирование
Ответственный	Независимый тестировщик	Разработчик
Знание программирования	Не обязательно	Необходимо
Знание реализации	Не обязательно	Необходимо
Знание сценариев использования	Необходимо	Не обязательно
Основа тестовых сценариев	Спецификации	Код

Например, зная особенности реализации модуля, тестировщик создает тестовые сценарии пользовательского уровня, которые покрывают потенциально проблемную область. Серый ящик особенно удобен для проверки интерфейсов взаимодействия объектов (интеграционное тестирование).

Главной проблемой тестирования является определение того, достаточно ли текущего количества тестов для вывода о правильности реализации системы, а также нахождения такого множества тестов, которые обладают таким свойством.

Для решения этой проблемы используют следующие методики:

- эквивалентное разделение (Equivalence Partitioning);
- анализ граничных значений (Boundary Value Analysis);
- причина / следствие (Cause/Effect);
- предугадывание ошибки (Error Guessing);
- исчерпывающее тестирование (Exhaustive Testing);

Число возможных комбинаций входных параметров может быть очень большим. Также число возможных путей следования внутри тестируемого объекта может быть очень большим. В силу ограниченности временных ресурсов, выделенных на проект в целом и этап тестирования в частности, выполнение полного тестирования невозможно. В таких случаях тесты объединяют в группы при выполнении следующих условий:

- тесты, которые предназначены для тестирования одной и той же ошибки;
- при выявлении ошибки в одном из тестов другие тесты, вероятнее всего, тоже это сделают;
- при отсутствии ошибки в одном из тестов в других тестах, вероятнее всего, она также будет отсутствовать;

Тесты, объединенные в группу, называются эквивалентными, а сама группа — классом эквивалентности. Для проведения тестирования достаточно выбрать по одному представителю из классов эквивалентности.

Изменение поведения тестируемого объекта происходит при достижении внешних или внутренних граничных значений. Граничные значения принадлежат одному или нескольким классам эквивалентности или образуют собственный класс эквивалентности. В связи с этим тесты с участием граничных значений являются наилучшими кандидатами.

В ряде случаев необходимо проверить реализованные причинно-следственные связи, например, условия (причин) для получения ответа от ПО (следствие). Использование этой методики построения тестов позволяет проверить работу ПО в динамике (потоки данных, передача управления и т. д.).

Использование накопленного опыта в области разработки и тестирования ПО позволяет проектировщику тестов предугадать места появления ошибок. Например, в поле ввода номера пользователь может попытаться ввести отрицательное значение, ноль или текстовую фразу.

Исчерпывающее тестирование — это крайний случай. В пределах этой методики проверяются все возможные комбинации входных значений. На практике применение исчерпывающего тестирования невозможно в силу того, что количество входных значений бесконечно.

Для примера, рассмотрим функцию двух вещественных переменных, допустимые значения которых лежат в области [0, a] и [0, b]. Полное тестирование функции невозможно в силу бесконечного числа возможных комбинаций значений переменных, однако можно разделить комбинации на следующие группы в соответствии с рисунком 2.

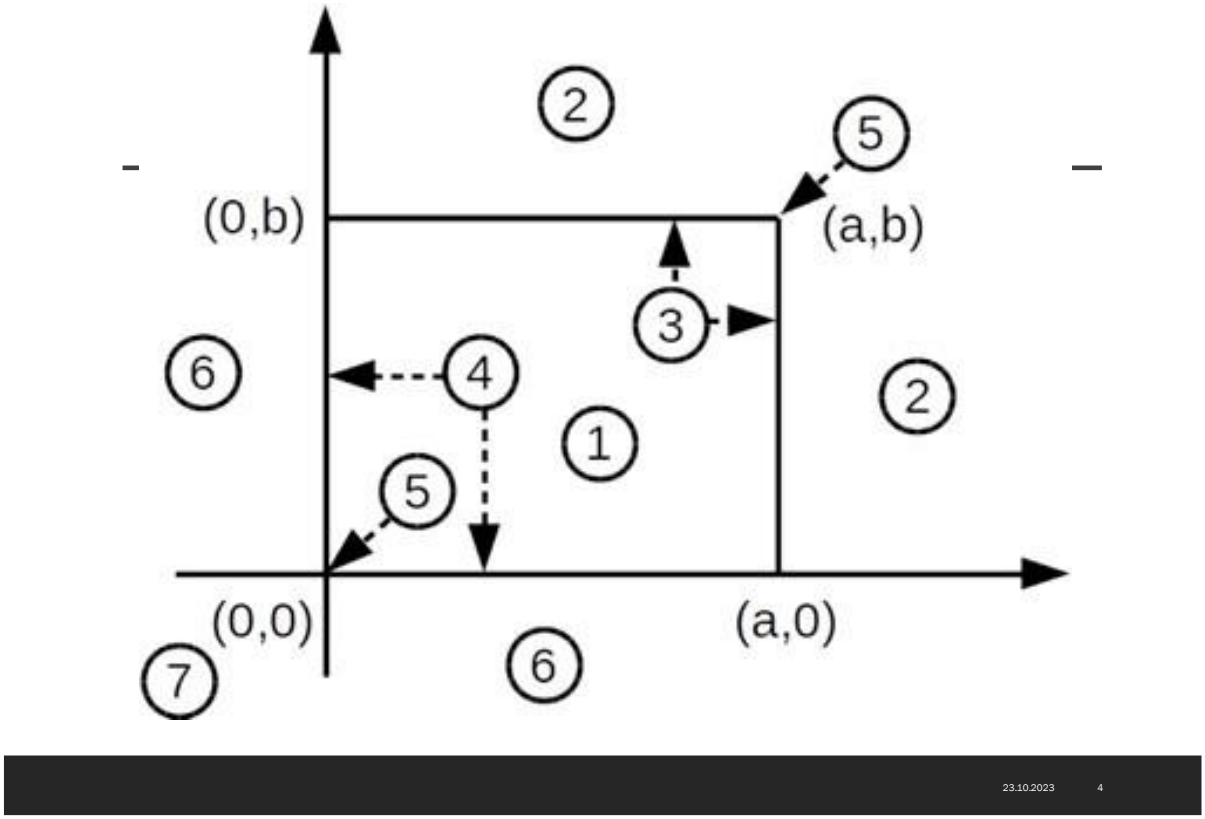

Рис. 2. Пример разбиения на классы эквивалентности

1) Значения переменных внутри области допустимых значений. Такая комбинация подойдет для проверки работоспособности исследуемой функции. Такие тесты называются общими.

2) Одно значение или оба превышают верхнюю границу области допустимых значений. Эта группа тестов подходит для проверки реакции функции на некорректные входные данные. Такие тесты называются негативными.

3) Одно значение находится на верхней границе, а другое — внутри области допустимых значений. Такие тесты называются краевыми.

4) Одно значение находится на нижней границе, а другое — внутри области допустимых значений. Группы 3 и 4 можно объединить, если поведение функции одинаково.

5) Оба значения находятся на верхней или нижней границе одновременно. В первом случае мы проверяем работу функции на максимальных значениях входных параметров, а во втором случае — на минимальных. Такие тесты называются специальными.

6) Одно значение меньше нижней границы, а второе внутри области допустимых значений. Эта группа выделена в связи с

изменением знака входного параметра, что может повлиять на работу функции.

7) Оба значения меньше нижней границы области допустимых значений. Группы 6 и 7 можно объединить, если поведение функции одинаково.

Таким образом, для исследуемой функции достаточно 7 тестов, что позволит выполнить тестирование с минимальными затратами и охватить все потенциально возможные места нахождения ошибок. Более детальный анализ содержания функции позволит улучшить тестирование за счет проверки внутренних граничных точек.

Для поиска классов эквивалентности можно воспользоваться следующими критериями:

- Одни и те же значения входных данных.
- Выполняются одинаковые функции программы.
- Одни и те же значения выходных данных.
- Блок обработки ошибок программы вызывается всеми тестами.
- Блок обработки ошибок программы не вызывается ни одним тестом.

2.5. Модульное тестирование

Модульное (блочное) тестирование нацелено на независимую проверку работы компонент (модулей, блоков, объектов, классов, функций и т. д.) ПО. Тестирование модулей выполняется изолированно, без интеграции с другими модулями ПО. В связи с этим для выполнения тестирования требуется реализовывать и/или подключать заглушки, эмуляторы и другие вспомогательные инструменты, заменяющие полностью или частично реальные компоненты ПО.

Разработка тестов основывается на внутренней структуре модуля (белый ящик). Задачами модульного тестирования являются поиск

дефектов, связанных с алгоритмическими ошибками, ошибками кодирования алгоритмов, выполнением условных и циклических операторов, использованием переменных и ресурсов. Проверка правильности трактовки данных, реализации интерфейса взаимодействия, совместимости, производительности и других аспектов выполняется на других этапах тестирования.

Модульные тесты запускаются с использованием специальной программы-драйвера, выполняющего следующие функции в соответствии с рисунком 3:

- чтение входных параметров теста;
- подготовка окружения модуля: заглушек и других вспомогательных инструментов;
- запуск тестируемого модуля;

- чтение результатов работы модуля.

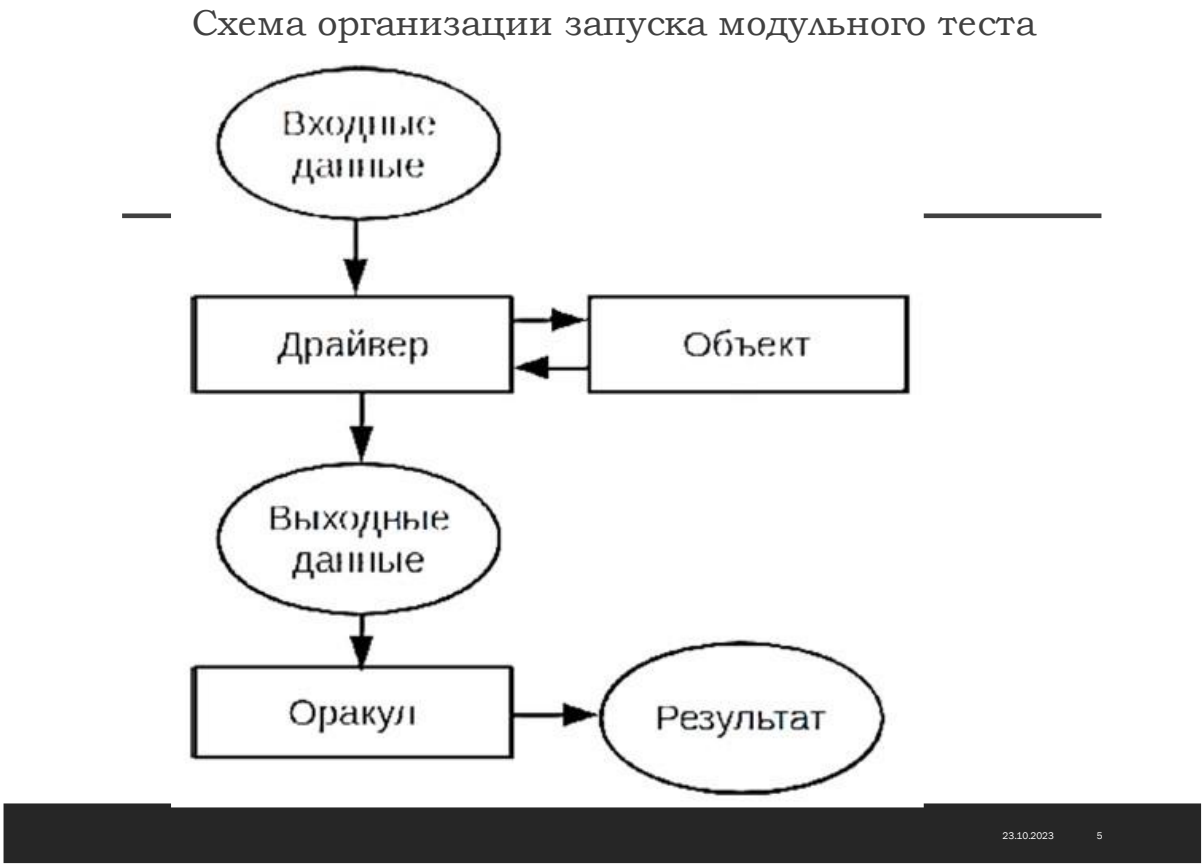

Рис. 3. Схема организации запуска модульного теста

Анализ результата работы модуля и сравнение с эталоном выполняется с помощью специального модуля (оракула). Оракул позволяет

выявить требуемые элементы из множества результирующей информации, выполнить сравнение с эталонным результатом и сделать вывод о получении или отсутствии дефекта.

Интеграционное тестирование

Интеграционное тестирование (Integration testing) направлено на проверку взаимодействия между частями (модулями) приложения. На этапе интеграционного тестирования выполняется поиск ошибок, связанных с трактовкой данных, реализацией интерфейса взаимодействия и совместимостью компонент приложения. Как правило, для интеграционного тестирования применяется метод серого ящика: известны все характеристики взаимосвязей между модулями, но модули закрыты для анализа.

В ходе интеграционного тестирования выполняется объединение модулей в блоки. Существуют два основных подхода к проведению интеграции:

- восходящая интеграция (Bottom Up Integration);
- нисходящая интеграция (Top Down Integration).

Восходящая интеграция предполагает объединение модулей низкого уровня в группы, группы с группами или модулями и с получением в итоге целого приложения в соответствии с рисунком 4. Нисходящая интеграция предполагает последовательное присоединение модулей к группе, содержащей управляющий модуль в соответствии с рисунком 5. Преимущества и недостатки представленных подходов отражены в таблице 2.

В ряде случаев предпочтительным является использование смешанной интеграции (Mixed Integration), когда модули собираются в блоки (восходящая интеграция), а блоки присоединяются к управ¬ляющему модулю (нисходящая интеграция). Смешанная интеграция позволяет минимизировать недостатки традиционных подходов.

Также существует подход «Большого взрыва» («Big Bang» Integration), который подразумевает сбор всех модулей в одну систему и проведение интеграционного тестирования. Данный подход может сохранить время, однако может привести к одномоментному появлению большого количества ошибок.

Схема восходящей итерации

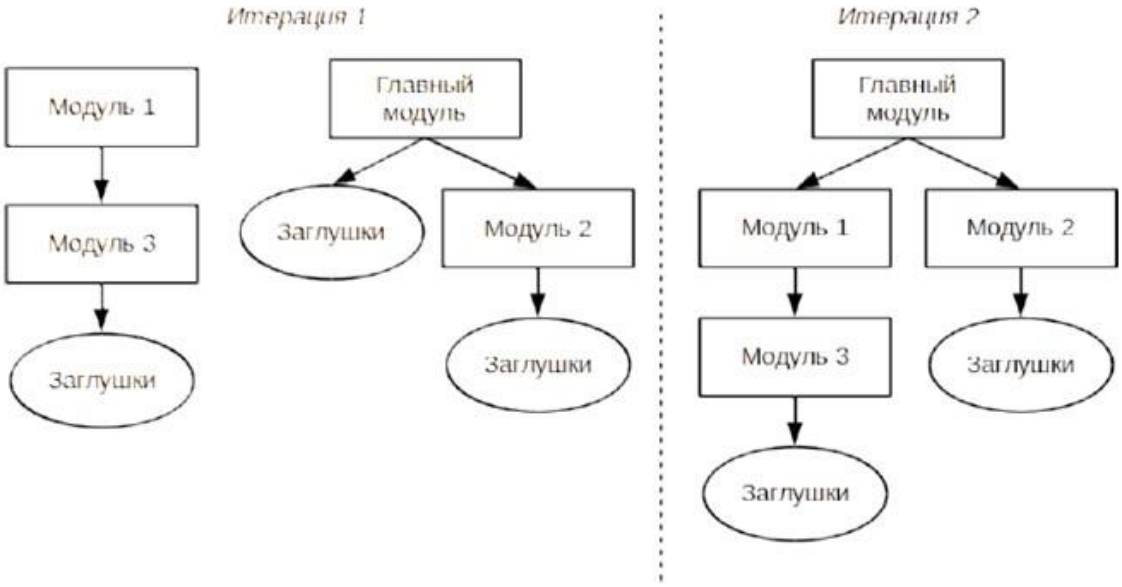

Схема нисходящей итерации

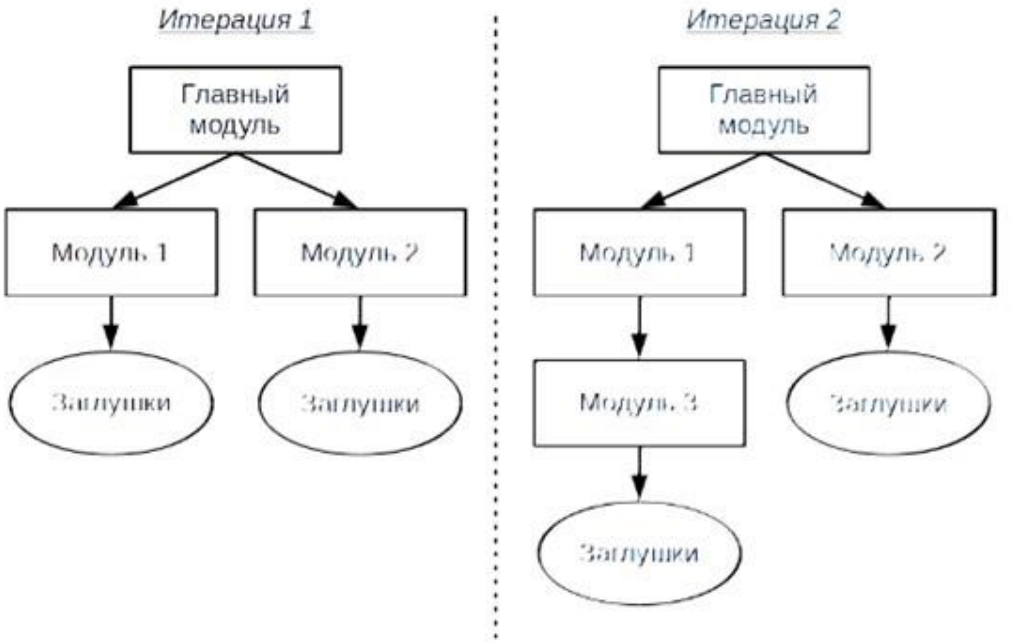

Таблица 2. Сравнение способов интеграции

Сравнение способов интеграции

	Восходящее	Нисходящее
Преимущества	Возможность ранней проверки корректности низкоуровневого поведения Не требуется написание заглушек Просто определить требования ко входам/выходам модулей	Возможность ранней проверки корректности высокоуровневого поведения Модули могут добавляться по одному, независимо друг от друга Не требуется разработка множества драйверов Можно разрабатывать систему как в глубину, так и в ширину
Недостатки	Отложенная проверка высокоуровневого поведения Требуется разработка драйверов При замене драйвера на модуль высокого уровня может произойти «мини-Большой Взрыв» числа найденных ошибок	Отложенная проверка низкоуровневого поведения Требуется разработка «заглушек» Крайне сложно корректно сформулировать требования ко входам/выходам частичной системы

Системное тестирование

Системное тестирование завершает проверку реализации приложения. В ходе системного тестирования проводится функциональное тестирование, а также характеристики разработанного ПО, в том числе устойчивость, производительность, надежность и безопасность. Для системного тестирования применяется подход черного ящика: приложение рассматривается как единое целое, на вход подаются реальные данные, работа приложения анализируется по полученным результатам.

На этапе системного тестирования выявляются ошибки, связанные с неправильной реализацией функций ПО, неправильным взаимодействием с другими системами, аппаратным обеспечением, неправильным распределением памяти, отсутствием корректного освобождения ресурсов и т. п. Источником данных выступают техническое задание на разработку приложения, спецификации на компоненты приложения и его окружения и используемые стандарты.

Пользовательское тестирование

На данном этапе к тестированию приложения подключаются сторонние участники, включая будущих пользователей и экспертов. По результатам тестирования принимается решение о внедрении.

Существуют различные классификации пользовательского тестирования: по организации процесса (модерируемое или немодерируемое), по местоположению (в окружении разработчика или заказчика), по используемому методу (сценарии работы, навигация, интерфейс пользователя).

В ходе модерируемого тестирования работа пользователя над ПО контролируется специалистом-модератором. Модератор может помочь пользователю с выполнением требуемых задач, оценить как работу ПО, так и реакцию пользователя. К недостаткам модерируемого тестирования относятся высокая стоимость, возможность влияния модератора на результаты теста и «неестественность» поведения пользователя под наблюдением. Немодерируемое тестирование позволяет пользователю выполнять задачи без привлечения сторонних специалистов, что обеспечивает возможность проведения массового тестирования. Для проведения немодерируемого тестирования требуется аудио-видео фиксация действий и комментариев пользователя с последующим анализом.

Тестирование в окружении разработчика позволяет привлечь большое количество разработчиков, но ограниченное количество пользователей. С другой стороны, тестирование в окружении заказчика максимально приближено к реальным условиям работы ПО и позволяет пользователям не отвлекаться на новую обстановку.

В ходе тестирования используются следующие методы сбора информации от пользователей: анкетирование, наблюдение, интервью,

видеозапись. Выбор метода осуществляется в зависимости от требуемой информации и степени вовлечения пользователя.

Пользовательское тестирование строится по следующей схеме:

- определение цели тестирования;
- формулировка задания и инструкции для пользователей;
- определение перечня респондентов;
- проведение тестирования;
- оценка результатов.

Принципы тестирования

Разработка правильных и эффективных тестов - достаточно непростое занятие. Принципы тестирования, представленные ниже, были разработаны в последние 40 лет и являются общим руководством для тестирования в целом [7].

1. Тестирование может найти ошибки (Testing shows presence of defects).

Тестирование может найти ошибки в ПО, но не доказать их отсутствие. Однако важно находить варианты тестов, которые будут выявлять как можно больше ошибок. Это позволит снизить вероят¬ность появления ошибок в ПО. Ни в коем случае нельзя утвержать, что ПО не содержит ошибок даже если тестирование не выявило их.

2. Полное тестирование невозможно (Exhaustive testing is impossible).

Нет возможности провести полное тестирование, включающее весь возможный ввод пользователя и состояния системы. Но необходимо правильно расставлять приоритеты и анализировать риски. Это может позволить более эффективно обеспечить качество ПО.

3. Раннее тестирование (Early testing).

Тестирование необходимо начинать как можно раньше. На разных этапах жизненого цикла разработки ПО оно должно преследовать определенные цели.

4. Скопление дефектов (Early testing).

Разные модули системы могут содержать разное количество дефектов, то есть плотность скопления дефектов в разных элемен¬тах программы может отличаться. Усилия по тестированию должны распределяться пропорционально фактической плотности дефектов. В основном, большую часть критических дефектов находят в ограни¬ченном количестве модулей. Это проявление принципа Парето: 80% дефектов содержатся в 20% модулей.

5. Парадокс пестицида (Pesticide paradox).

Прогоняя одни и те же тесты вновь и вновь, Вы столкнетесь с тем, что они находят все меньше новых ошибок. Поскольку ПО эволюционирует, многие из ранее найденных дефектов исправляют и старые тест-кейсы больше не срабатывают.

Чтобы преодолеть этот парадокс, необходимо периодически вно¬сить изменения в используемые наборы тестов, рецензировать и кор-ректировать их с тем, чтобы они отвечали новому состоянию ПО и позволяли находить как можно большее количество дефектов.

6. Тестирование зависит от контекста (Testing is context dependent).

Выбор методологии, техники и типа тестирования будет напрямую зависеть от природы самого ПО. Например, ПО для медицинских нужд требует гораздо более строгой и тщательной проверки, чем, например, сайт магазина. Из тех же соображений, сайт с большой посещаемостью должен пройти через серьезное тестирование производительности, чтобы показать возможность работы в условиях высокой нагрузки.

7. Заблуждение об отсутствии ошибок (Absence—of—errors fallacy).

Тот факт, что тестирование не обнаружило дефектов, еще не значит, что ПО готово к публикации. Нахождение и исправление дефектов будут не важны, если ПО окажется неудобным в использовании, и не будет удовлетворять ожиданиям и потребностям пользователя.

Структура документации тестирования

На рисунке 6 представлены основные документы тестирования. Разберем более подробно каждый из них.

Рис. 6. Структура документации тестирования

План тестирования

План тестирования (Test plan) — это основной документ этапа тестирования, который описывает работы по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Основные вопросы, которые план тестирования должен раскрыть:

- Что необходимо тестировать: включает в себя абсолютно все аспекты ПО, которые могут быть протестированы.
- Что будет тестироваться: подмножество пунктов из первого вопроса, которые будут протестированы.

Из первого вопроса исключаются аспекты, исходя из анализа сроков, бюджета, приоритетов и пр. В идеальном случае множество первого и второго вопросов совпадают.

- Как будет проходить тестирование: выбор стратегии тестирования (ручное тестирование, написание автоматических тестов, подбор групп пользователей для тестирования и др.).
- Когда будет проходить тестирование: сроки тестирования для каждого компонента ПО.

• Критерии окончания тестирования: результат, который должен быть получен в результате тестирования (отчет, список ошибок, каким способом представлены эти документы и др.).

План тестирования может создаваться либо как полноценный продукт, либо как инструмент. В первом случае требуются значительные ресурсы для его создания, но затем он может использоваться вне команды разработчиков (например, на стороне заказчика). Обычно выполняется по какому-либо стандарту. Во втором случае в тестовый план включается только то, что помогает в организации процесса тестирования и выявлении ошибок. Все, что не отвечает этим задачам, избыточно.

Но в любом случае необходимость создания плана тестирования заключается в повышение качества продукта. Для этого ставятся следующие цели:

• облегчение тестирования (контроль полноты тестирования и его эффективности, отсутствие повторяющихся тестов, поиск наилучших методов для тестирования);

• организация взаимодействия между участниками команды, отвечающих за проведение тестирования;

• удобная структура для организации, планирования и управления.

План тестирования включает в себя:

• Тестовые ресурсы (список совместимого оборудования, программ и ОС).

• Перечень функций и подсистем, подлежащих тестированию:

— списки отчетов и экранных форм;

— списки входных и выходных переменных;

— списки возможностей и функций;

— списки сообщений об ошибках;

— список файлов программы.

• Тестовую стратегию, включающую:

— Анализ функций и подсистем с целью определения наиболее слабых мест, то есть областей функциональности тестируемой системы, где появление дефектов наиболее вероятно.

— Определение стратегии выбора входных данных для тестирования. Так как множество возможных входных донных программного продукта, как правило, практически бесконечно, выбор конечного подмножества, достаточного для проведения исчерпывающего тестирования, является сложной задачей. Для ее решения могут быть применены такие методы, как покрытие классов входных и выходных данных, анализ крайних значений, покрытие модели использования, анализ временной линии и тому подобные. Выбранную стратегию необходимо обосновать и задокументировть.

— Определение потребности в автоматизированной системе тестирования и дизайн такой системы.

- Расписание тестовых циклов.
- Фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения.
- Определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать. Например, метрик, оценивающих степень покрытия тестами набора требований, степень покрытия кода тестируемой системы, количество и уро-вень серьезности дефектов, объем тестового кода и другие характеристики.

В тестовом плане определяются и документируются различные типы тестов. Типы тестов могут быть классифицированы по двум категориям: по тому, что подвергается тестированию (по виду подси¬стемы), и по способу выбора входных данных.

Типы тестирования по виду подсистемы или продукта:

- Тестирование основной функциональности, когда тестированию подвергается собственно система, являющаяся основным выпускаемым продуктом.
- Тестирование инсталляции включает тестирование сценариев первичной инсталляции системы, сценариев повторной инсталляции (поверх уже существующей копии), тестирование деинсталляции, тестирование инсталляции в условиях наличия ошибок в инсталлируемом пакете, в окружении или в сценарии и т. п.
- Тестирование пользовательской документации включает проверку полноты и понятности описания правил и особенностей использования продукта, наличие описания всех сценариев и функциональности, синтаксис и грамматику языка, работоспособность примеров и т. п.

Типы тестирования по способу выбора входных значений:

- Функциональное тестирование, при котором проверяется:

— Покрытие функциональных требований.

— Покрытие сценариев использования.

- Стрессовое тестирование, при котором проверяются экстремальные режимы использования продукта.
- Тестирование граничных значений.
- Тестирование производительности.
- Тестирование на соответствие стандартам.
- Тестирование совместимости с другими программно-аппаратными комплексами.

- Тестирование работы с окружением.
- Тестирование работы на конкретной платформе.

В реальных разработках используются и комбинируются различные типы тестов для обеспечения спланированного качества продукта.

Тестовый отчет

В результате тестирования (после каждого этапа тестирования) формируется документ, который называется тестовым отчетом. Он должен содержать:

- Что было запланировано для тестирования и что удалось протестировать.
- Время тестирования.
- Выполненные тесты и результат их выполнения.
- Найденные ошибки и повторно найденные ошибки.
- Найденные отклонения от разработки программного обеспечения.
- Заключение о результате проведенного этапа тестирования.

Матрица соответствия требований

Матрица соответствия требований (traceability matrix, трассиру- емость требования в тестах) — это двумерная таблица, содержащая соответствие функциональных требований ПО и подготовленных тестовых сценариев. В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки. Матрица соответствия требований используется руководителями тестирования ПО для валидации покрытия продукта тестами и является неотъемлемой частью тест-плана.

Лист проверки

Лист проверки (чек-лист, check list) — это документ, описывающий что должно быть протестировано. Лист проверки может быть абсолютно разного уровня детализации. На сколько детальным будет чек-лист, зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности продукта. Как правило, лист проверки содержит только действия (шаги), без ожидаемого результата. Лист проверки менее формализован, чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Лист проверки обычно используется в гибких подходах в тестировании

Отчет об ошибке

Ошибкой ПО считают либо расхождение между программой и ее спецификацией в том случае, когда спецификация существует и она правильная, либо когда программа не делает того, чего пользователь от нее вполне обоснованно ожидает.

Все ошибки можно разделить по следующим категориям:

- Ошибки пользовательского интерфейса:

— Отсутствие или неправильная работа ожидаемой функ¬ции.

— Взаимодействие программы с пользователем. Например, возможность ввести неправильный тип данных там, где есть такие ограничения.

— Организация программы (легко ли найти нужную функ¬цию).

— Низкая производительность ПО.

— Выходные данные (правильно ли формируются отчеты).

- Недостаточно качественная обработка ошибок.
- Ошибки, связанные с обработкой граничных условий.
- Ошибки вычислений и алгоритмов.
- Ошибки управления потоком (последовательность действий).
- Ошибки передачи или интерпретации данных (взаимодействие с другим ПО).
- Ошибки, связанные с пиковыми нагрузками на ПО.
- Ошибки, возникающие на специфичном аппаратном обеспечении.
- Ошибки в описании ПО (его документации), обычно возникают при переработки функционала ПО.
- Ошибки тестирования.

Цель создания отчета об ошибке заключается в том, чтобы ее исправить. Создание отчета необходимо для всех участвующих лиц в разработке ПО — от заказчика до разработчика. Кроме того, это помогает осуществлять анализ разработки ПО во времени.

Структура отчета об ошибке

Отчет об ошибке может включать в себя следующие разделы:

- программа или модуль (указывается при наличии нескольких компонент ПО (например, серверная или клиентская части) или реализаций ПО под несколько платформ);
- версия программы или модуля;

- тип отчета (может включать в себя ошибку или запрос на добавление новой функциональности);
- важность отчета (варианты значения: критический, высокий, средний, низкий);
- описание;
- повторяемость;
- последовательность шагов для воспроизведения ошибки;
- предлагаемое исправление;
- автор отчета;
- ответственный за исправление;
- комментарии пользователей ПО;
- текущее состояние отчета (варианты значения: открытый, закрытый);
- приоритет отчета (отмечается руководителем разработки ПО в отличие от важности, которую указывает автор отчета);
- срок выполнения работ (указывается руководителем разработки ПО).

Составление подробных отчетов и сбор полной информации об ошибках является очень важным элементом тестирования.

Анализ воспроизводимости

Прежде чем отправить отчет об ошибке, необходимо произвести анализ воспроизводимости, который включает в себя несколько пунктов:

- максимально подробно записать последовательность действий, приводящих к ошибке;
- выявить наиболее серьезные проблемы (повысить важность);
- найти кратчайший путь воспроизведения (облегчить отладку);
- найти альтернативные действия, приводящие к этому результату;
- выявить связанные проблемы;
- проверить более ранние версии (поиск модификаций в коде);
- проверить на нескольких конфигурациях.

Жизненный цикл отчета

Этапы жизненного цикла отчета об ошибке:

- неподтвержденная (unconfirmed). После публикации отчет об ошибке попадает на первичное рассмотрение, в результате которого ответственные лица решают допустить ли этот отчет к анализу или отправить на доработку, например из-за отсутствия некоторых данных;

• новая (new). Руководители проекта или ответственные лица анализируют ошибку и либо назначают ответственного за ее исправление, либо переводят ее в один из статусов: повторная (duplicate), отклонена (rejected), не ошибка (not a bug), после чего закрывают ее;

• назначен ответственный (assigned). Ошибка ждет начала работы над ней;

• открытая (open). Этот статус ошибки означает, что ответственный за исправление начал работу над ней;

• исправленная (fixed). Ошибка исправлена и требует повторного тестирования;

• проверена (verified). Исправления протестированы и могу быть опубликованы. Если в результате тестирования были выявлены другие ошибки, связанные с исправлениями, или ошибка была воспроизведена, то ошибка возвращается в статус «открытая»;

• опубликована (published). Версия ПО с исправленной ошибкой опубликована;

• закрыта (closed). Ошибка закрыта;

На рисунке 7 представлен жизненный цикл отчета об ошибке.

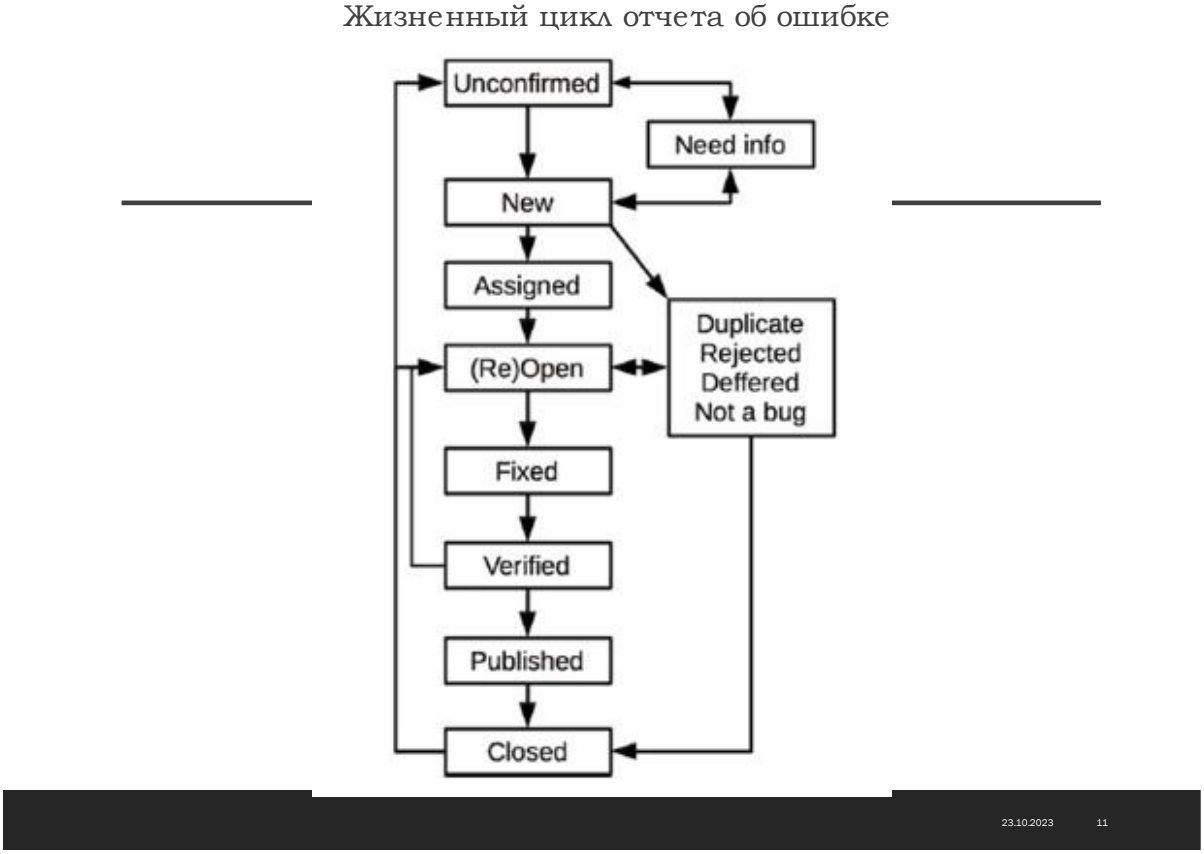

Рис. 7. Жизненный цикл отчета об ошибке

Системы отслеживания ошибок

Как видно из вышеприведенного списка разделов отчета об ошибке, его составление является достаточно трудоемкой задачей. Еще более трудоемкой задачей является ведение таких отчетов и поддержка их актуального состояния. Упростить эти задачи для разработчиков ПО призваны системы отслеживания ошибок. Они позволяют:

- автоматизировать часть работ (например, заполнение некоторых полей отчета, возможность создания списков программ, версий ПО и др., возможность создания специализированных наборов важности, текущего состояния, приоритетов и др.);
- организовать взаимодействие между пользователями, заказчиком и командой разработчиков;
- оценивать производительность работ и выполнения сроков по устранению ошибок;
- оценивать состояние проекта;
- интегрироваться с системами контроля версий кода для указа¬ний мест кода, проводящих к ошибке или решающих ее;
- составлять различного рода статистические отчеты;
- использовать расширенный набор возможностей (оповещения по электронной почте или СМС, возможность управления проектами, настройка прав доступа, поддержка добавления файлов, поддержка комментариев, распределение работ, ведение подзадач и др.).

Системы отслеживания ошибок являются полезными не только для программистов. Отчеты о «работе над ошибками» могут использовать менеджеры проекта. Для управляющих процессом разработки программного обеспечения такие отчеты позволяют судить о производительности программистов, при работе по улучшению работы ПО.

Приведем несколько примеров систем отслеживания ошибок:

- Bugzilla (https://www.bugzilla.org/),
- Redmine (https://www.redmine.org/),
- Trac (https://trac.edgewall.org/),
- Atlassian JIRA (https://ru.atlassian.com/software/jira).

При выборе системы отслеживания ошибок также учитывают следующие факторы: лицензия, стоимости, язык интерфейса, наличие базы знаний ошибок, набор функциональности и др.