Лекция «Основы тестирования»

Тестирование и отладка

Тестирование — деятельность на протяжении всего ЖЦ ПО, направленная на:

- анализ и поддержку качества продукта;
- определение соответствия реального ПО бизнес-требованиям.

Исполнители: QA, тестировщики, программисты (в меньшей мере).

Отладка — деятельность, направленная на локализацию и устранение ошибок в ПО.

Исполнители: программисты.

Зачем нужно тестирование?

a. Что бы из-за ошибок в коде программ спутники ценой в миллиарды долларов не улетали в открытый космос вместо выхода на орбиту, истребители не переворачивались при пересечении экватора, не отключались электростанции, не останавливались торги на бирже, медицинские аппараты не облучали пациентов смертельными дозами излучения, чтобы не было синего экрана смерти в Windows. Потому, что людям свойственно ошибаться и с этим надо как-то бороться.

b. Что бы молодые программисты легко находили работу. Опыт — (почти) не нужен, требования — минимальны, возможность вертикального роста, зп более/менее.

Основная задача тестирования

Основная задача тестирования – предоставление информации о соответствии ПО требованиям заказчика. Стоит отметить, что ошибки могут быть допущены и в требованиях.

Входные выходные данные тестирования

Невозможно описать конкретный список документов, который требуется каждый раз, когда вы приступаете к тестированию, но, как правило, на более-менее длительных проектах на входе мы получаем:

- Требования к ПО
- Собственно ПО

на выходе предоставляем:

- Информацию о соответствии ПО требованиям в виде:
  - План тестирования
  - Сценарии тестирования (Test Cases)
  - Описания ошибок (bugs)
  - Протоколы тестирования
  - Отчеты о результатах тестирования
  - Матрица соответствия (Matrix of Traceability)

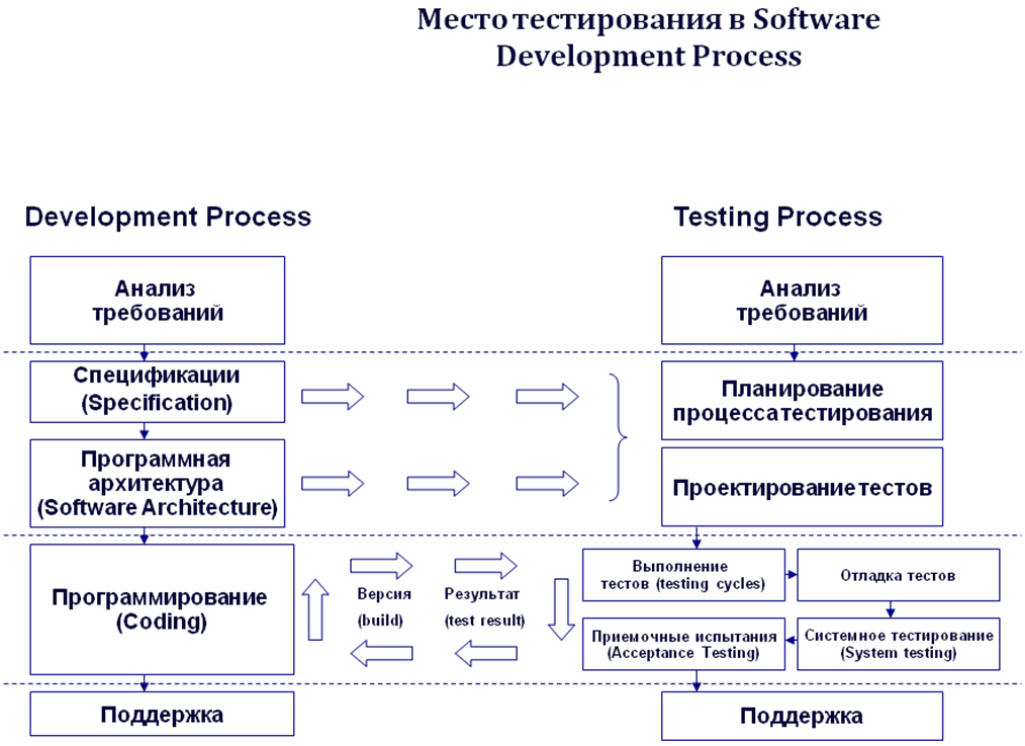
Основной артефакт: программа и методика испытаний (ПМИ).

Место тестирования в процессе разработки программного обеспечения

На схеме ниже отображена деятельность, связанная с тестированием в ходе разработки ПО.

После проведения первого анализа (анализа планируемых требований, степени детальности, оценки временных затрат) по мере появления документированных требований готовятся план тестирования, набор сценариев тестирования и матрица соответствий (если требуются). Далее, по

мере появления готового ПО, выполняются циклы тестирования. Сценарии тестирования прогоняются, корректируются, составляются протоколы тестирования, заводятся описания ошибок (bugs). При необходимости составляются отчеты о результатах тестирования.



План тестирования

План тестирования создается задолго до появления собственно программного обеспечения, которое необходимо будет тестировать и даже может быть вполне до появления конкретных требований к ПО. План тестирования отвечает на вопрос, что мы получаем на входе, что требуется на выходе, и как мы собираемся реализовывать тестирование. Обычно план тестирования выполняют в виде текстового документа rtf/doc/pdf/txt берестяная грамота. Примерный набор пунктов плана тестирования:

1. Уникальное название, версия

2. Введение
3. Документы, на основании которых будет производится тестирование
4. Что мы тестируем (Features to be tested)
5. Что мы не будем тестировать (Features not to be tested)
6. Как мы тестируем (включая виды, этапы и т.п.)
7. Критерии прохождения тестов (Item pass/fail criteria)
8. Артефакты создаваемые в результате тестирования (План тестирования, сценарии, отчеты и т.п….)
9. Порядок задач необходимых для проведения тестирования (связи между задачами, зависимость)
10. Требования к окружению (специфические железки, ОС и т.п.)
11. Кто за что отвечает (менеджмент, подготовка и утверждение сценариев, непосредственное выполнение)
12. Требования к исполнителям , обладание определенными навыками (например надо пройти курсы и т.п.)
13. План. Временной план. Диаграмма Ганта или что-то такое. Основные этапы работы, и их привязка по времени.
14. Риски (описать и оценить риски при проведении тестирования)
15. Список людей / должностей - Кто должен утвердить этот план (менеджер проекта, консультант)

Сценарии тестирования (Test Cases)

Сценарии тестирования – основной документ инженера по тестированию. Сценарии тестирования полностью основаны на требованиях, поскольку главный смысл тестирования — проверить, что продукт соответствует замыслу заказчика (выраженному в требованиях). Можно сказать, что сценарий тестирования — это дочерний документ требований. Начинать создавать сценарии тестирования следует в одно время с началом практических работ над самим продуктом, т.е. сразу после принятия требований, нет необходимости дожидаться появления первой версии тестируемого продукта, даже наоборот, тесты должны опираться на требования, а не на реализованный продукт, поскольку задача тестирования проверить именно выполнение требований. Чаще всего, из-за ограничений по времени инженеры по тестированию начинают писать сценарии тестирования сразу, как только появляются первые версии требований (не дожидаясь их окончательного утверждения) и потом корректируют их по мере изменения требований вплоть до появления финальной версии.

Есть несколько вариантов для оформления сценариев тестирования, многостраничный текст, электронная таблица, база данных. Система хранения сценариев тестирования и их результатов должна обеспечивать возможность:

- в нужный момент внести необходимые изменения, которые сразу же доступны всем заинтересованным участникам процесса;
- быстро и точно составить отчет о состоянии тестирования: получить статистику о покрытии требований тестами, количестве тестов вообще (для планирования сроков), количестве успешно пройденных тестов, найденных ошибках и т.д., причем все это желательно по отдельным компонентам продукта и по различным периодам и фазам тестирования;

Эти пункты будут реализованы, например, если вы используете для сценариев тестирования многостраничный текст, храните файлы в какой-либо системе управления версиями (CVS, SVN и т.п.), храните протоколы выполненных сценариев тестирования и регулярно заполняете Traceability matrix.

Обычно, все требования проекта покрываются несколькими сценариями тестирования, каждый из сценариев состоит из набора test case. Каждый test case из нескольких простых test steps.

Хороший сценарий тестирования помимо test cases должен содержать второстепенную, но при этом полезную информацию: описание документа - кому и зачем он нужен, вкратце об объекте тестирования, связанные с данным сценарием использования документы, глоссарий, общие для всех test cases условия тестирования (например, программно аппаратная конфигурация) и т.д.

Каждый test case в сценарии тестирования должен

- Быть понятен не только автору
  - Иметь единый для всего проекта стиль оформления
  - Иметь осмысленное логичное название и / или комментарий о сути производимого в нем действия
  - Отслеживать изменения (например, changes tracking в MS Word)
- При необходимости иметь отдельно описанные
  - preconditions
  - postconditions
  - тестовых данных (может быть общее описание данных на весь сценарий использования)
- Быть короткими и независимым от предыдущих и последующих test cases (по возможности возвращать систему в исходное положение по завершению)
- Должны содержать ссылки на покрываемые данным test caseом требования

Пример test case

Test №5 Проверка формы A1, A2 A3

Этот тест покрывает требования описанные в 'UC440023 A1 A2 A3 Form.doc' ver. 1.23. NFE 1, AFE 2,3.

В этом тесте мы выставляем допустимые значения для A и проверяем, что B и C отображаются корректно в соответствии с формулой из требований.

tep No	**Action**	**Expected Result**	**Pass / Fail / Blocker**
	Precondition: Зайдите на форму "A1 A2 A3"		
	Установите A1 = X	Убедитесь, что B1 = Y	
	Установите A2 = X	Убедитесь, что B2 = Y	
	Установите A3 = X	Убедитесь, что B3 = Y	
	Postcondition: Установите значения для A1, A2 и A3 назад в 0.		

Колонка Pass / Fail / Blocker заполняется во время выполнения теста. Инженер по тестированию выполняет действие из колнки action. Если результат соответствует записи в Ожидаемый результат - pass. Если нет - fail и ссылка на описание пробелмы (номер bug в bugtracker), об этом ниже. Если шаг по каким-то причинам выполнить невозможно - blocker и причина, по которой шаг невыполним.

Пример Test Summary

Часто в самом же сценарии тестирования хранят форму отчета, которую заполняют после выполнения теста. Вот пример формы отчета:

Project	Аэроэкарус 1
Build Number	255-06
Browser	Type I.E. Version 6.0

Web Server	Tomcat
Operating System	Server Ubuntu Client WinXP
Username	TestUser01
Database	DB504
Run Number	5
Result	Fail
Task Numbers	A1-534, A1 535, A1 536
Tested By	Vasiliy Pupkin
Date	06-May-2010

Матрица Соответствия (Matrix of Traceability)

Еще один важный документ предназначеный для просмотра состояния продукта – это **Матрица Соответствия (Matrix of Traceability)**. Этот документ, как правило, заполняется руководителями команд на стадии планирования, а потом контролируется менеджером по тестированию и менеджером проекта. Документ содержит список требований, для каждого требования указывается дата разработки, набор тестов, которые покрывают реализацию данного требования (только идентификаторы тестов), могут быть результаты тестирования на текущий момент. Этот документ можно расширять очень сильно, но его назначение от этого не изменится. Суть документа – посмотреть, что уже реализовано в продукте (либо на какой стадии находится реализация) и какие проблемы были найдены, убедиться, что мы покрываем 100% требований тестами. Изначально матрица соответствия составляется на этапе планирования и проектирования тестов, корректируется и при необходимости, заполняется результатами в ходе циклов тестирования. Матрицу соответствия удобно вести в электронных таблицах типа excel.

Автоматизация

Существуют специальные системы, которые полностью (или частично) автоматизируют процесс тестирования - хранение/выполнение тестов и ошибок до полного ведения статистики. Такие как HP Quality Center, TestComplete, IBM Rational.

Циклы тестирования

Большие программы пишутся долго. Как правило, требования реализуются поэтапно, постепенно, от версии к версии ПО все больше приближаясь к полному выполнению требований. Тестирование ПО

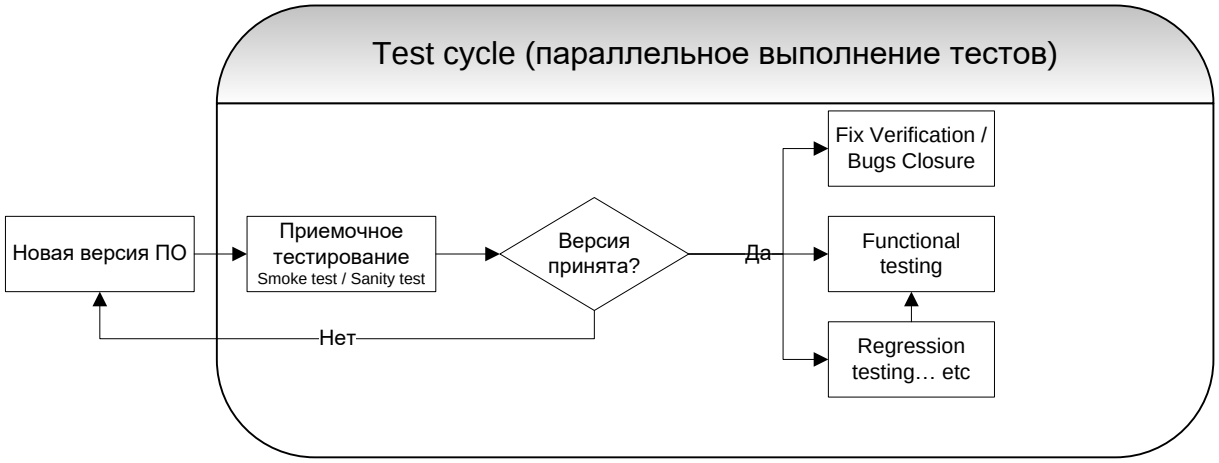
находящегося в разработке часто привязывают к выпуску определенной версии.

Через определенные промежутки времени программисты выпускают одну за другой новые версии программы, соответственно, за время разработки ПО многократно выполняется примерно похожий набор действий связанных с тестированием – цикл тестирования.

Что же такое Test cycle? По сути это процесс выполнения последовательности различных типов тестов для определения работоспособности конкретной версии продукта. По содержанию это набор тестовых сценариеи и их выполнение в процессе тестирования конкретной версии продукта. По назначению это структурированный подход к проверке текущей версии продукта на соответствие реализованным требованиям.

Test cycle – явление итерационное и выполняется он для каждой новой версии от начала и по возможности до конца.

В процессе Test cycle выполнение различных типов тестов может быть последовательным, параллельным или комбинированным. Это зависит от множества компонентов: специфики самого проекта, разработки проекта, процесса тестирования, требований заказчика, степени готовности системы и т.д.



Виды тестирования

В рамках цикла тестирования мы описали несколько видов тестирования (sanity test / functional testing / regression testing / bug fix verification). Существует множество видов тестирования, одна из задач при планировании тестирования – определить какие виды тестирования и на каком этапе необходимо провести. Например, если мы тестируем программу, с которой будет работать один человек и скорость работы программы не имеет для него особого значения – нет смысла тратить время на performance testing.

Проверка на пригодность к тестированию (sanity check, smoke testing, приемочное тестирование)

Данный тип тестов является одним из ключевых типов в тестировании и организации Test cycles. Назначение этих тестов заключается в предварительной проверке пригодности новой версии для последующего полного тестирования. Проверяется базовая функциональность и общая стабильность продукта. В случае успешного прохождения данных тестов новая версия принимается для тестирования. В противном случае, отсылается на доработку с указанием того, что не работает.

Приемочное тестирование (ПМИ)

Часто под приемочным тестированием имеется в виду набор сценариев, после успешного выполнения которого заказчик официально принимает ПО от исполнителя. Документ описывающий такое тестирование в России обычно называется ПМИ (Программа и Методика Испытаний) для него есть шаблон ГОСТ.

Проверка исправлений (fixes verification, bug closure)

Эти тесты предназначены для проверки исправлений дефектов, найденных во время предыдущих Test Cycles. На самом деле, данный вид тестов не обязательно выполнять сразу после приема новой версии для тестирования. Однако весьма полезно дать быструю информацию разработчикам о состоянии исправленных дефектов до того как начнется новый цикл.

Функциональное тестирование (functional testing)

Эти тесты могут называться по-разному, однако, их суть проста – проверка соответствия системы предъявляемым к ней требованиям. Пожалуй, основной вид тестирования, без которого никак не обойтись. Если программа не выполняет свои основные функции, то нет смысла говорить о производительности и всем остальном.

Регрессионное тестирование (Regression testing)

Одно из определений успешности регрессионных тестов гласит: «повторное выборочное тестирование системы или компонент для проверки сделанных модификаций не должно приводить к непредусмотренным эффектам». На практике это означает, что если система успешно проходила тесты до внесения модификаций, она должна их проходить и после внесения таковых. Основная проблема регрессионного тестирования заключается в поиске компромисса между имеющимися ресурсами и необходимостью проведения таких тестов по мере внесения каждого изменения. В определенной степени, задача состоит в том, чтобы определить критерии «масштабов» изменений, с достижением которых необходимо проводить регрессионные тесты.

Тестирование производительности (performance testing)

Это тестирование, которое проводится с целью определения, как быстро работает система или её часть под определенной нагрузкой. В тестировании производительности различают следующие направления:

- нагрузочное (load)
- стресс (stress)
- тестирование стабильности (endurance or soak or stability)
- конфигурационное (configuration)

Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих типовые действия с системой.

Стресс-тестирование обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надёжности системы во время экстремальных или диспропорциональных нагрузок и отвечает на вопросы о достаточной производительности системы в случае если текущая нагрузка сильно превысит ожидаемый максимум.

Тестирование стабильности проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти, чтобы выявить потенциальные утечки (memory leak). Также важным, но часто незамеченным, фактором является деградация производительности, смысл которого сводится к тому, чтобы скорость обработки информации и/или время ответа приложения через длительное время работы были такими же или лучше, чем в самом начале теста.

Конфигурационное тестирование - еще один из видов традиционного тестирования производительности. В этом случае вместо того, чтобы тестировать производительность системы с точки зрения подаваемой нагрузки, тестируется эффект влияния на производительность изменений в конфигурации. Хорошим примером такого тестирования могут быть эксперименты с различными методами балансировки нагрузки. Конфигурационное тестирование также может быть совмещено с нагрузочным, стресс- или тестированием стабильности.

Можно придумать еще множество видов тестирования (Installation testing, security testing, usability testing, User Interface testing, localization testing, compatibility testing etc.), однако совершенно не обязательно описывать их все в вашем тестплане. Необходимо выделить то, что нужно в данном, конкретно случае. Стоит обсудить этот вопрос с менеджером проекта.

Проверка реакции на сбои, стрессоустойчивость.

Что произойдет с ПО при внештатной ситуации? Отключение питания, отключение сетевого кабеля, отсутствие необходимых файлов на диске и т.п.

Как тестировать?

Черный ящик

Существует множество подходов к тестированию ПО, один из них – тестирование методом черного ящика. Этот метод предполагает что в процессе тестирования мы имеем дело только со «внешними» аспектами программы, с интерфейсами GUI (Graphic User Interface) или API (Application Programming Interface) и не имеем никаких представлений о внутреннем функционировании ПО. Также существует белый и даже серый ящики, но, в основном, отделы тестирования занимаются тестированием именно черного ящика.

Протестировать все?

Как правило, провести исчерпывающее тестирование разрабатываемого ПО (протестировать полностью) невозможно. Это слишеом трудоемкий процесс. Возьмем поле, в которое можно вводить целые числа от 0 до 100000. Пусть операция ввода занимает около 3 секунд, перебор всех значений займет 300000 секунд - 3.5 дня. Таких полей в системе могут быть тысячи, даже с применением автоматизации на исчерпывающее тестирование могут уйти десятки лет, вряд ли заказчики будут готовы столько ждать и оплачивать такую работу. Чтобы убедиться в работоспособности ПО, можно обойтись перебором значительно меньшего количества вариантов.

Разбиение на классы эквивалентности и граничные значения

Принадлежность двух элементов данных к одному и тому же классу эквивалентности

означает, что с каждым из них функция выполняет одни и те же операции. Т.е. их обрабатывает одна и та же строка кода. Представим себе, что в программе есть вот такой код обрабатывающий входные данные (текстовое поле на форме), которые присваиваются переменной А:

```
If  A > 100 then
….do something 1….
Else
….do something 2….
```

Можно определить два класса эквивалентности для А: - А меньше ста и А больше ста.

Проверив значения из обоих групп мы можем убедиться, что ПО корректно обрабатывает весь диапазон допустимых значений (мы проверили

обе строчки и do smth 1 и do smth 2). Входные данные часто делят на два класса эквивалентности - *правильные и неправильные данные.* Обычно, в проектной документации для большинства функций определены *правильные* и *неправильные* входные данные.

Весьма перспективной областью для поиска ошибок являются границы классов эквивалентности

функции. Как правило, анализ, который ведет к определению классов эквивалентности, определяет и эти границы. Включение нескольких пограничных значений в тестовые случаи для данной функции поможет проверить удовлетворение ожиданий (требований) пользователя.

Например, если мы знаем из требований, что на форме есть текстовое поле, значение которого передается в переменную типа byte (0..255), то правильными данными для этого поля будут целые числа от 0 до 255, а неправильными все остальное: числа меньше нуля, больше 255ти, буквы, спецсимволы, пустые значения (если система позволяет их ввести). Граничными значениями в данном случае можно назвать -1, 0, 255, 256. Кроме того в текстовое поле можно ввести множество букв a-z, A-Z, букв локального алфавита а-я,А-Я, спецсимволы ~`!@#$%^&*()_+=-“ ’;:/|\.,><, или даже не ввести ничего. Полезно проверить, что произойдет с системой в случае ввода недопустимых данных. Как правило, это поведение должно быть описано в требованиях к системе.

Набор проверок для поля типа byte

Предположим, что в требованиях описано поле на форме ввода (или на входе функции API) типа byte. Разрешается вводить в поле целые числа от 0 до 255 не больше 3х символов. В случае ввода неправильных данных система должна выдать сообщение об ошибке. Ниже набор вариантов данных, которые я бы проверил для этого поля.

Значение	Тип проверки
Позитивное тестирование	
0, 255	Граничные значения
22,124, 201	Одно или несколько случайных значений из середины промежутка *правильных данных*
Негативное тестирование	
-1, 256	Граничные значения
-55, 2434*, -23424*	Одно или несколько значений из промежутка *неправильных данных*

4444	Недопустимые данные. Условие на длину поля – не больше 3х символов. *Вообще говоря, мы уже проверили условие на длину поля в предыдущей проверке. Здесь этот пункт что бы не забыть, что длину полей тоже стоит проверять.*
~`!@#$%^&*()_+=-“ ’;:/|\.,><.	Недопустимые данные. Спецсимволы, буквы
A-Z,a-z,A-Я,а-я	Недопустимые данные. Буквы.
NULL,	Недопустимые данные. Пустое значение, пробел
12 3; 12,2; 12.2; 23/3; 23 3; а34, 34а	Недопустимые данные. Смешанные данные, дробные числа, числа и буквы, пробел перед числом или в середине.

Как писать сценарии тестирования?

Прежде всего, стоит начать с базового теста. Простейший путь в приложении. Вводим минимально необходимый набор данных, совершаем наиболее вероятный набор операции в наиболее вероятном порядке. Проверяем, что для основных действий система обрабатывает данные в соответствие с требованиями. Набор таких тестов часто используют в качестве Smoke test. Затем, одно за одним, покрывает тестами все остальные требования к системе.

Например, при работе с базами данных, как правило, встречаются три действия Add New, Update, Delete. Покопавшись в требованиях или посовещавшись с аналитиками / менеджером / заказчиком и т.п. можно выяснить, что 90% деятельности пользователя заключается в операции Update и 10% Add New и Delete. Соответственно, Update попадает в базовый тест, Add New, Delete в последующие тесты. Убедившись, что основные операции работают, следует более детально разобраться с обработкой входных параметров.

Поочередно проверяем весь набор входных данных для всех предполагаемых данной функциональностью операций. Выбираем одно поле (параметр функции) и, оставляя остальные поля неизменными, вводим для него различные значения - пограничные значения, позитивное негативное

тестирование. Проверяем, что система обрабатывает данные в соответствие с требованиями.

Проверяем различные комбинации входных данных. Часто бывает, что входных комбинаций слишком много, и нет возможности проверить их все. Тогда необходимо проанализировать связи между входными данными, какие правила и ограничения существуют в системе явно и неявно, и поверить их. Например, если мы вводим в систему длины сторон треугольника – стоит проверить, что система корректно обработает ситуацию, когда сумма двух сторон окажется меньше третьей, даже если этот случай не описан в требованиях явно.

При написании тестов стоит попробовать представить себя на месте пользователя системы. Как бы поступил человек, который увидел эту программу в первый раз? Какие операции и в каком порядке будут производить пользователи в системе? Иногда требования могут быть сформулированы прямо в виде таких User Stories.

При проверке программного обеспечения, связанного с математическими вычислениями стоит отдельно обратить внимание на округление в расчетах.

Checklist

Иногда, вместо подробных сценариев тестирования пишутся краткие чеклисты, с перечислением функциональности, которую надо не забыть проверить при очередном цикле тестирования.

Например:

- Логин/логаут Админом, пользователем
- Добавить запись
- Удалить запись
-
-

Bugs - Дефекты

Как описать несоответствие между ПО и требованиями, передать эту информацию аналитикам или программистам? Как убедиться, в дальнейшем, что ошибки не забыты – исправлены, отложены? Ответ – bugs, дефекты.

Почему bug? 9 сентября 1945 года ученые Гарвардского университета, тестировавшие вычислительную машину Mark II Aiken Relay Calculator, нашли мотылька, застрявшего между контактами электромеханического реле. Проделанная работа требовала описания, и слово было найдено — debugging (дебаггинг, дословно: избавление от насекомого).

Описание дефекта (ошибки, bug)

В современном мире ошибки обычно хранятся в специальных программах - bug-trackerax типа jira, bugzilla и т.п. потому, что это удобно. Можно вести описание ошибок и в excel и на бумажке,

но зачем? Это не очень удобно.

Обычно регистрационная информация содержит следующие данные:

- **Идентификационный номер (ID)**, уникальный для каждого бага.
- **Заголовок (Headline** или **Title)** – краткое описание бага, касающееся его сути. Описание должно быть коротким, уникальным, отражающим суть.
- **Серьезность (Severity)** – позволяет менеджеру проекта оценить, насколько серьезно данная ошибка влияет на проект. (Critical / Major / Minor) Выставляется тестером, но может быть скорректирована менеджером/ аналитиком и т.п.
- **Приоритет (Priority)**, позволяющий разработчику оценить очередность исправления данной ошибки). Как правило выставляется менеджером.
- **Статус ошибки (Status)** – отражает на какой стадии находится работа над данной ошибкой.

(Ready for test / In Development / Analysis)

- **Имя тестера сообщившего об ошибке и дату**, как правило, заполняется автоматически.
- **Дополнительная информация** о дате исправления и имени разработчика, исправившего баг; информация о проверке, ее дате и имени тестера, проверившего факт исправления, информация о человеке ответственном за работу с дефектом в данный момент и т.д.
- **Описание (Description)** – подразумевает более подробное сообщение о том, в чем именно заключается ошибка. Это основная часть описания проблемы – она должна состоять из трех частей:
  - Пошаговое описание как привести **систему и окружение** в состояние необходимое для проявления ошибки
  - Пошаговое описание как привести **тестируемый продукт** в состояние необходимое для проявления ошибки
  - Непосредственно описание ошибки
- **Вложения (Attachments)** – прикрепление к описанию ошибки скриншотов (снимков экрана) и логов (автоматических отчетов и журналов работы программы), если их присутствие целесообразно и полезно разработчику для понимания и принятия факта ошибки.

Несмотря на все сопутствующие активности – bug является основным продуктом деятельности тестера, bug должен быть хорошо оформлен:

1. ***Bug должен четко и не двусмысленно описывать проблему***: у разработчика не должно возникать вопросов по поводу ошибки после прочтения описания. Для этого перед отправкой тестер должен:

a. прочесть описание сам (а лучше два раза)

b. приложить всю сопутствующую информацию, которая может облегчить понимание проблемы

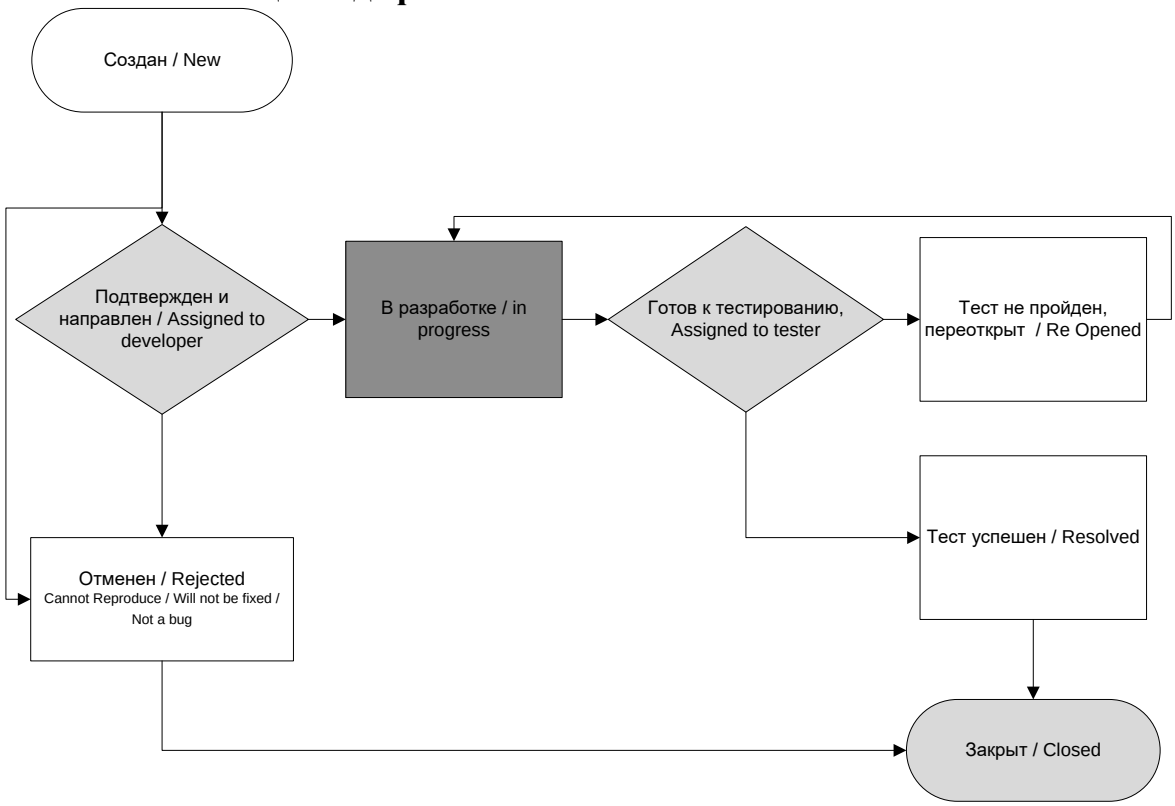
i. точные данные, используемые для ввода
ii. вырезки из журнала событий программы (logs)
iii. скриншоты

Все это должно быть прикреплено к описанию проблемы (bug).

2. ***Описывать нужно именно проблему, а не то, что за проблему принято*** – первая инстанция для проверки корректности работы приложения это проектная документация – надо внимательно сравнить описываемое состояние с реальным состоянием программы. Если документация не содержит достаточной информации для того, что бы убедится в том, что проблема действительно присутствует – надо обратиться к руководителю команды тестеров, к аналитикам, к руководителю проекта. Сепаратные переговоры с разработчиком, который ответственен за тестируемую функциональность, не приветствуются и должны носить только консультативный характер, но ни в коем случае не руководство к действию.

3. ***Проблема должна быть четко локализована*** – после того как проблема описана, надо попробовать её повторить в соответствии с описанием (а лучше два раза). Если проблема не повторяется постоянно, то скорее всего это означает, что вы пропустили какую-то важную информацию в описании (состояние системы на машине тестера, либо состояние сервера…, либо шаг описания который был пропущен) либо плохо локализовал проблему.

Жизненный цикл дефекта



Кто:

Инженер по тестированию Tester	Менеджер по тестированию / ведущий Аналитик / ПМ Test Team Lead / Senior Analyst / PM	Разработчик Developer

А можно просто так потетсировать, без всего вот этого?

Существуют десятки и сотни успешных проектов, в которых не пишутся сценарии тестирования, и даже проекты в которых нет требований зафиксированных в виде конкретных документов. Можно сэкономить время и силы, разработать простое приложение и убедиться, что оно работает не создавая кучу лишней документации. Однако следует понимать, что даже если у вас нет четких требований, оформленных в виде документа, чтобы что-то проверить, нужно иметь эти требования хотя бы у себя в голове. Так или иначе, вам придется сформулировать, хотя бы только для себя, что конкретно вы проверяете. Аналогично, даже если вы не описывали сценарии тестирования в виде отдельного документа, каждый раз работая с ПО вы выполняете те или иные шаги в определенном порядке и ждете соответствующего отклика от программного обеспечения. Т.е. у вас есть сценарий тестирования, вы просто храните его у себя в голове. Если зафиксировать его в виде документа, скрипта или хотя бы чеклиста, то гораздо проще будет не забыть какую-то из проверок в следующий раз, передать задание на тестирование другому исполнителю. К тому же, в процессе фиксации своих идей в документе, вполне вероятно, вам придут в голову еще более гениальные идеи проверок. Иногда записать что-то имеет смысл только ради самого процесса записи. Чем больше и сложнее становится ваш проект, тем сложнее удержать в голове всю информацию о требованиях и тестировании, а значит будут ошибки. При разработке больших сложных приложений почти всегда готовят планы и используют сценарии тестирования в том или ином виде. По этой же причине международными и локальными организациями разработаны специальные стандарты, требующие в том числе подробного оформления тестовой документации для разработки критического ПО (от которого зависит жизнь или безопасность человека, экология, сохранность секретной информации и т.д.).

Итог: Вполне вероятно, вы можете обойтись без всей описанной выше документации при разработке небольшого проекта. Вам совершенно точно не обойтись без всего этого, если вы собрались разрабатывать медицинское ПО, бортовое ПО для авиации, автомобилей, или полетов на Марс, для атомных электростанций или нефтяных платформ и т.д. Если не здравый смысл, то требования заказчиков и государственных регуляторов заставят вас написать план и тест кейсы.

Основы тестирования

Определение

Тестирование — деятельность на протяжении всего ЖЦ ПО, направленная на:

- анализ и поддержку качества продукта;
- определение соответствия реального ПО бизнес-требованиям.

Исполнители: QA, тестировщики, программисты (в меньшей мере).

Отладка — деятельность, направленная на локализацию и устранение ошибок в ПО.

Исполнители: программисты.

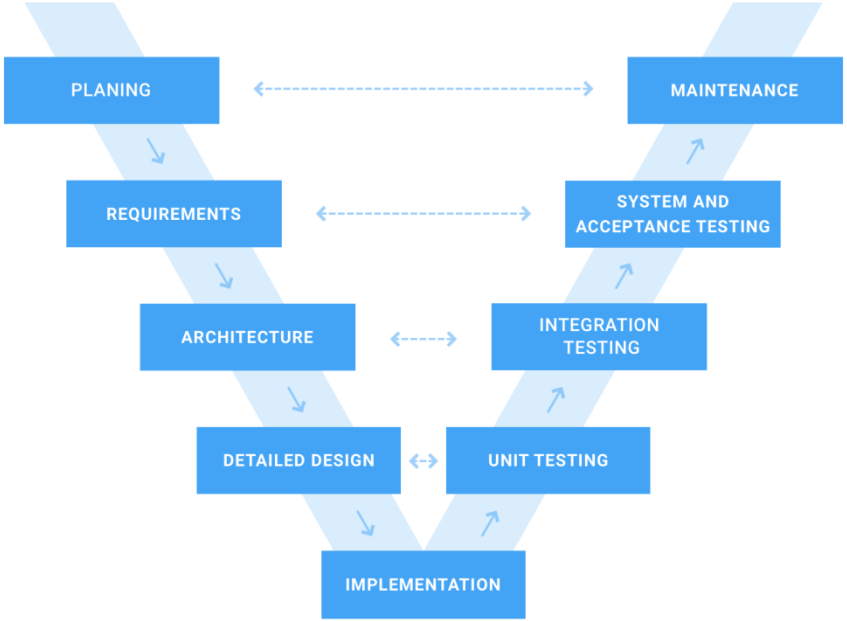
Процесс тестирования начинается также в начале ЖЦ продукта: составляются планы тестирования, определяются методики тестирования, после получения требований создаются первые версии сценариев тестирования и т.д..

Уровни тестирования

Вспоминаем V-Model!

- Разбили на модули — проверяем каждый модуль
  Часто выполняется программистом
- Проверяем связи между компонентами, а также взаимодействие с некоторыми частями системы
- Проверяем, что вся система в целом работает правильно
- Проверяем, что все требования заказчика удовлетворены

Подробнее:
http://www.protesting.ru/testing/testlevels.html



Функциональное тестирование

Функциональные требования — ожидаемое поведение ПО; те функции, которое ПО должно выполнять.

Функциональное тестирование — тестирование в целях проверки реализации функциональных требований.

Проводится на всех уровнях тестирования.

Тестами должны быть покрыты все функциональные требования!

Проверяются:

- корректные наборы данных — система должна правильно выполнять свои функции;
- некорректные наборы данных — система должна быть устойчива к ошибкам.

Test-case

Test-case — сценарий теста для проверки определенного функционала.

Могут объединяться в тестовый комплект (набор), один комплект закрывает одно требование к продукту. По итогу все требования заказчика должны быть покрыты тест-кейсами. Нежелательна зависимость одного тест-кейса от другого.

Атрибуты тест-кейса (и, по сути, структура):

- id
- названия (раскрывающее суть)
- предусловия — условия, которые должны быть выполнены, чтобы начать прохождение теста
- шаги — четко сформулированная последовательность действий для получения результата
- ожидаемый результат — кейс проверяет вполне конкретную ситуацию, потому результат должен быть четко определен в единственном экземпляре

Результат теста может быть положительным (если фактический результат совпал с ожидаемым), отрицательным (не совпал, ошибка), неопределенным (тест блокирован, либо из-за того, что нельзя продолжить тестирование после некоторых шагов, либо потому что не выполнены предусловия).

Test-case

Test #5 “Загрузка текста из файла на форму”

Предусловия: нет

Шаги:

1) Открыть сайт TextEditor: в адресной строке браузера ввести адрес 127.0.0.1:55500, нажать Enter;

2) На форме нажать кнопку «открыть файл»;

3) В появившемся диалоге выбора файла выбрать файл «TestFile4a.txt», расположение файла — C:/Tests/TestFile4a.txt;

4) Нажать ок.

Ожидаемый результат: в виджете просмотра содержимого файла отображается текст «аааа».

(либо прям скрин того, как это должно выглядеть)

Smoke-тесты

Минимальный набор простых тестов на явные ошибки. Если завалилось на них — нет смысла тестить дальше. Выполняется и программистом (перед отправлением на тестирование, чтобы проверить свои изменения), и тестировщиком, принявшим билд.

Вполне очевидный смок-тест — сборка программы.

Пытаемся покрыть простыми тестами как можно больше функционала.

Легко автоматизируется.

Sanity-тесты

Чуть более глубокие тесты, чем дымовые. Более подробно проверяются основные функции программы.

Покрывают меньше кода, чем дымовые, но проверяют более серьезно.

Автоматизируются сложнее дымовых.

Stub и Mock

Stub — объект-пустышка, частично реализующий логику реального объекта на уровне «валидные данные на входе — валидные на выходе». Как правило, код содержит тривиальную логику, имитирующую работу нескольких методов реального объекта. Имитирует интерфейс реального модуля, но внутри объекта — хардкод.

Mock — Stub + логирование состояний для анализа после тестов.

Граница между ними размыта и в разных источниках определяется по-разному, поэтому часто все объекты-заглушки (dummy, fake, stub, mock) называют mock.

Примеры моков:

- запрос к серверу в клиент-серверной системе: данные могут выдаваться моком из файла с тестовыми данными;
- запись в БД: мок проверяет валидность переданных данных и пишет данные в файл (Fake) или ничего не делает, но пишет сообщение об успешной запись в лог (Mock).

Автоматизация тестирования

Для более-менее простых тестов (смок-тесты, некоторые санитарные тесты, часть модульных тестов) необязательно выполнять человеку, затрачивая человеко-часы на рутинную работу, можно написать программу-тестер, которая будет выполнять тесты автоматизированно.

Автоматизированное тестирование — шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи специализированных библиотек, фреймворков и инструментов (хотя для простых случаев можно обойтись и возможностями языка).

Хорошо сочетается с системами CI/CD.

Примеры библиотек и фреймворков:

- Gtest, Catch, Catch2 (рекомендуем) — для C++;
- JUnit — для Java, возможно и для Kotlin;
- Unittest — для Python;
- Selenium — для поведенческого тестирования веб-приложений (реализован на основе разбора DOM).

Автоматизация тестирования

На примере тестирования в Python 3+. (пример использования Catch2 для C++ представлен в файле практического занятия с прошлой недели).

Пусть нужно протестировать реализацию операции сложения в классе рациональных чисел (Rational).

```python
import unittest

from rational_numbers import Rational


class RationalNumbersTest(unittest.TestCase):

    # Test addition
    def test_add_two_positive(self):
        self.assertEqual(Rational(1, 2) + Rational(2, 3), Rational(7, 6))

    def test_add_positive_and_negative(self):
        self.assertEqual(Rational(1, 2) + Rational(-2, 3), Rational(-1, 6))

    def test_add_two_negative(self):
        self.assertEqual(Rational(-1, 2) + Rational(-2, 3), Rational(-7, 6))

    def test_add_opposite(self):
        self.assertEqual(Rational(1, 2) + Rational(-1, 2), Rational(0, 1))


if __name__ == '__main__':
    unittest.main()
```

Тест сделаны на основе проверки утверждений, сравнивается фактический результат и ожидаемый. Если результаты не будут совпадать, то тест упадет с ошибкой.

Функционал unittest велик — можно проверять пробросы исключений, пропускать тесты, запускать серию одного и того же теста и т.д..

Если мы будем дрегать эту программу в CI на стейдже тестирования, мы сможем проверить базовый функционал разрабатываемого класса.

Выводы

- Тестированию посвящена отдельная дисциплина в магистратуре, которую теперь ведет тот самый Власов Е.Е.!
- Про-тестинг — неплохой сайт для начального ознакомления с теорией мира тестирования и поддержки качества
- Ваша задача — разработать тест-кейсы, покрывающие требования к вашему курсовому проекту, определить способы тестирования (что сможете автоматизировать, что потребует участия человека)

Методологии разработки с использованием тестирования

Методология разработки

Методология разработки — набор правил и приемов, использующихся при разработке, которые позволяют достичь определенных качеств процесса реализации или разрабатываемого продукта (скорость разработки, непрерывная поставка версий, качество продукта и т.д.).

- выбирается в зависимости от специфики продукта и компетенций команды;
- допускаются мотивированные отклонения или дополнения к принципам методологии в начале работы над продуктом.

Разработка через тестирование (TDD)

TDD — методология разработки, основанная на повторении коротких итераций:

- Пишется тест, закрывающий желаемое изменение

Тест пишется на основе требований к системе. Также могут изменяться уже существующие тесты. Измененные тесты не должны проходить по условию.

- Пишется такой код, который пройдет все тесты

Подгоняем поведение под тест, возможно неидеальным способом.

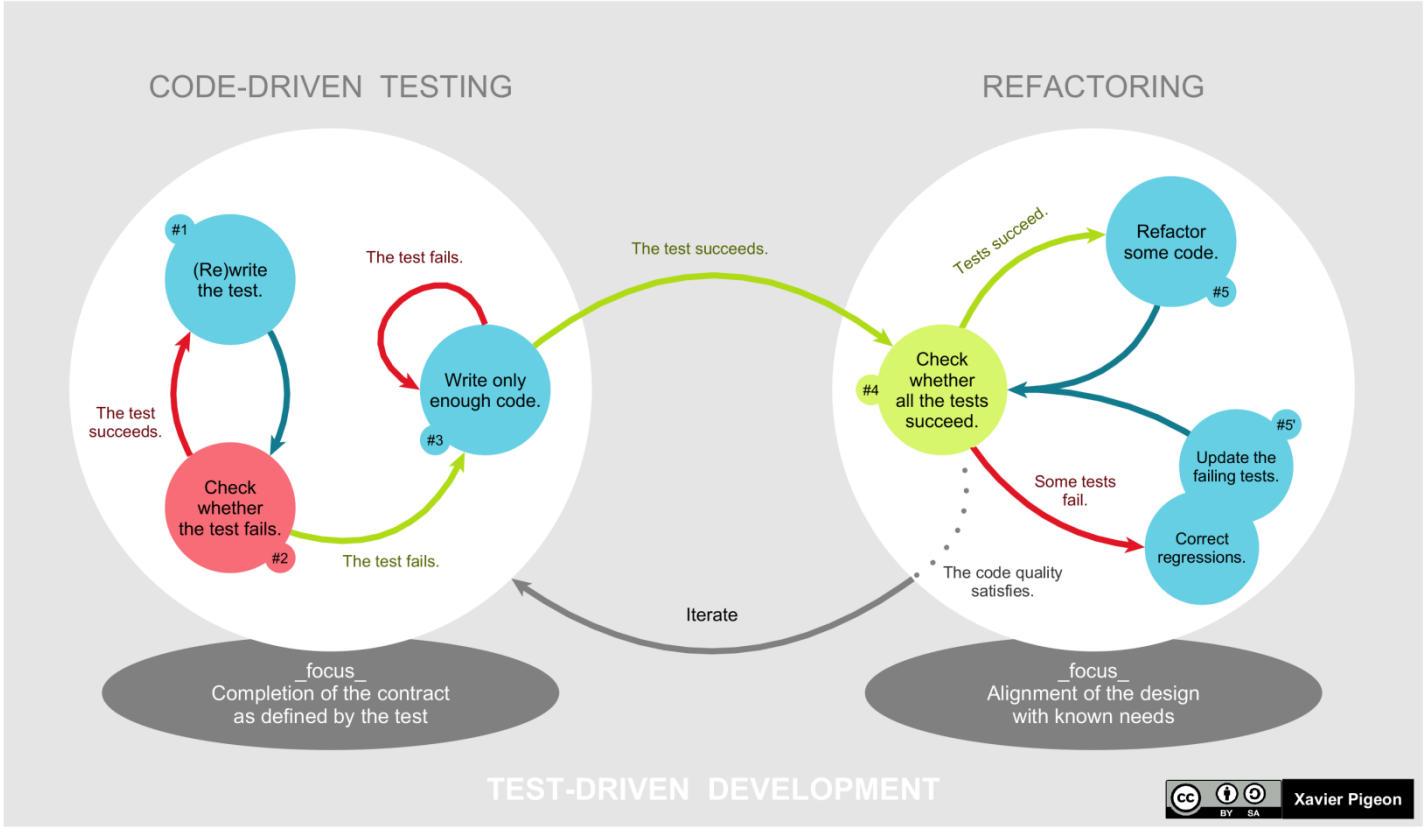
- Рефакторинг получившегося решения

Если все тесты проходят, можно изменить неидеальное решение на более чистое и идиоматическое. Рефакторинг может производиться раз в несколько итераций.

С каждой итерацией тесты все больше и больше уточняется, приводя к полной реализации функций системы.

В основном тесты для разработки являются модульными тестами.

TDD


CODE-DRIVEN TESTING
REFACTORING
#1
(Re)write the test.
The test fails.
The test succeeds.
Write only enough code.
#3
The test succeeds.
Check whether the test fails.
#2
The test fails.
Tests succeed.
Refactor some code.
#5
#4
Check whether all the tests succeed.
#5'
Update the failing tests.
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus
Completion of the contract as defined by the test
focus
Alignment of the design with known needs
TEST-DRIVEN DEVELOPMENT
Xavier Pigeon

TDD

Достоинства

- фокусировка на требованиях;
- тесты являются документацией;
- продукт обладает только необходимой пользователю функциональностью;
- интерфейс к модулю проектируется при создании теста;
- модули обладают меньшей связностью (так как сильно-связанные модули сложнее тестировать, а программисты ленивые);
- увеличение временных затрат на создание тестов компенсируется уменьшением временных затрат на отладку;
- уменьшается риск пропуска ошибок.

Недостатки

- тесты пишет программист: в случае недостаточно формализованных требований, неправильная трактовка требования приведет к неправильной реализации при полной уверенности, что все идет хорошо;
- ложное ощущение безопасности приводит к снижению интенсивности на высоких уровнях тестирования;
- обнаружение поздних ошибок приводит к необходимости изменения уже существующих тестов;
- тесты могут содержать ошибки;
- сложно полноценно использовать для разработки GUI-приложений.

TDD

Приемы TDD:

- выполнение тестов не должно занимать много времени, чтобы у программиста не пропадала мотивация писать и запускать тесты;
- количество тяжеловесных интеграционных тестов меньше, чем количество модульных; частота запуска интеграционных тестов меньше, чем модульных;
- при зависимости тестируемого модуля от других следует использовать mock-объекты, следовательно каждый модуль должен иметь две реализации: тестовую и реальную;
- при ошибке теста код не отлаживается, а происходит откат к прошлому коммиту: так как тесты уточняются постепенно, нет риска потерять большой объем кода;
- для кода, закрывающего тест, активно используются принципы KISS (сделай это проще), YAGNI (отказ от избыточной функциональности), а также «подделай, пока не сделал нормально»: если решение будет плохим, оно будет исправлено на этапе рефакторинга.

Разработка через поведение (BDD)

BDD — расширение TDD, «мостик» между техническими специалистами (программисты, тестировщики, системные администраторы…) и управляющим персоналом (аналитики, менеджеры).

Поведение и результаты работы системы описываются на языке предметной области, близком к естественному, понятном специалистам разных лагерей.

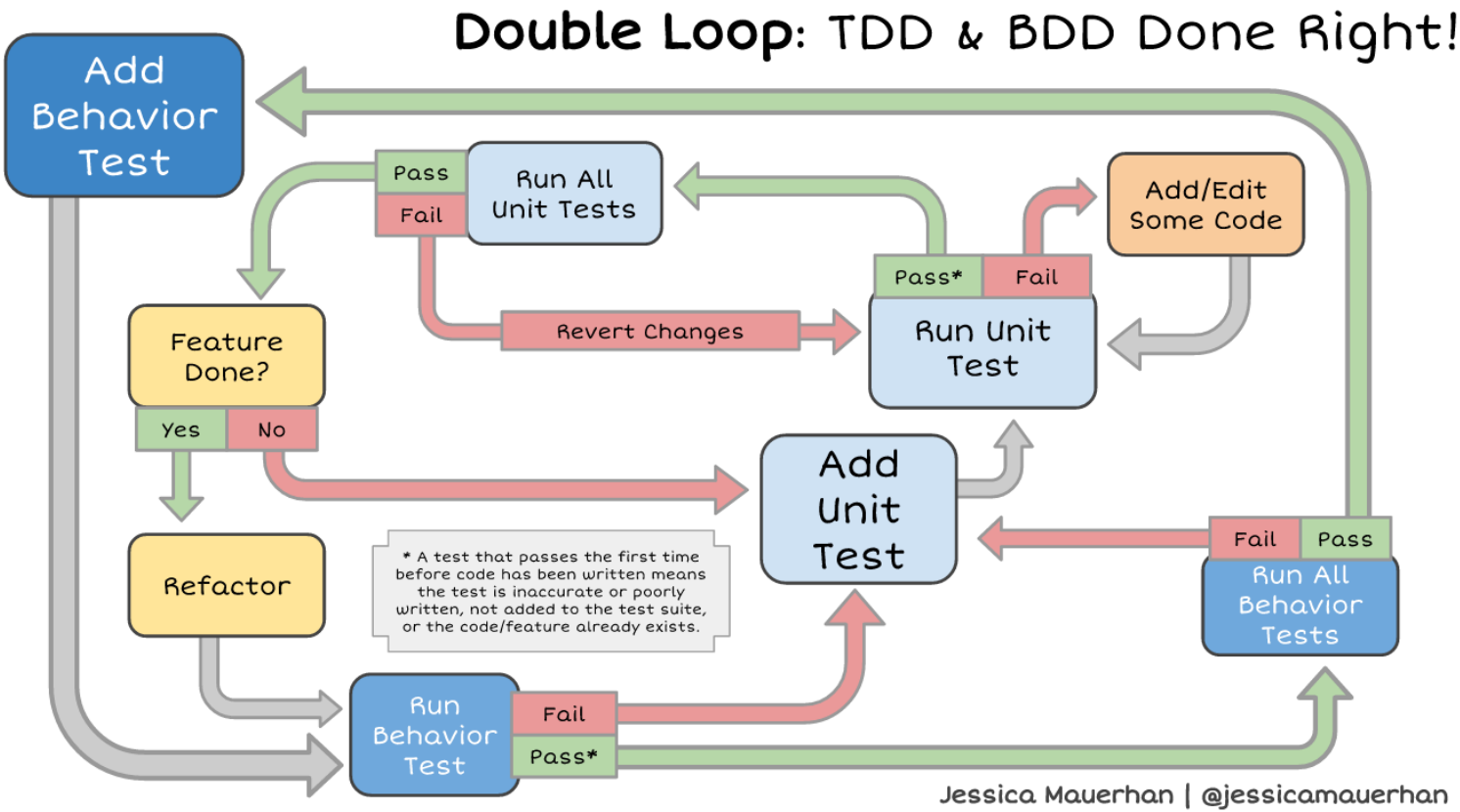
Используется для сложных предметных областей.

Акценты BDD:

- бизнес-цели;
- заинтересованные лица;
- сценарии использования системы.

На основе сценариев создаются тесты (автоматически или вручную).

BDD


Double Loop: TDD & BDD Done Right!
Add Behavior Test
Pass
Fail
Run All Unit Tests
Add/Edit Some Code
Pass*
Fail
Revert Changes
Run Unit Test
Feature Done?
Yes
No
Add Unit Test
Refactor
* A test that passes the first time before code has been written means the test is inaccurate or poorly written, not added to the test suite, or the code/feature already exists.
Fail
Pass
Run All Behavior Tests
Run Behavior Test
Fail
Pass*
Jessica Mauerhan | @jessicamauerhan

BDD

Достоинства

- упрощение коммуникации между всеми участниками проекта, т.к. поведение описано понятным языком;
- описания бизнес-целей и их сценарии являются документацией;
- упрощение интерпретации результатов тестов;
- сценарии легко портируются между языками и платформами;
- при наличии фреймворков для языка, из «человеческого» описания можно получить почти готовые тесты.

Недостатки

- долго, дорого;
- этим никто не будет заниматься, кроме программистов (потому что фу, сложно);
- требует предварительного глубокого изучения предметной области всеми участниками проекта (а не только аналитиками);
- даже при наличии фреймворка с парсером описаний поведения, адаптация этого описания к тесту может занимать больше времени, чем написание теста с нуля;
- усложнение изменения тестов (т.к. добавился новый уровень абстракции).

BDD: спецификация Gherkin

Ключевое слово	**Русскоязычная адаптация**	**Описание**
Story/Feature	История	Каждая новая спецификация начинается с этого ключевого слова, после которого через двоеточие в сослагательной форме пишется имя истории.
As a	Как (в роли)	Роль того лица в бизнес-модели, которому данная функциональность интересна.
In order to	Чтобы достичь	В краткой форме какие цели преследует лицо.
I want to	Я хочу, чтобы	В краткой форме описывается конечный результат.
Scenario	Сценарий	Каждый сценарий одной истории начинается с этого слова и порядковый номера, после которого через двоеточие в сослагательной форме пишется цель сценария.
Given	Дано	Начальные условия.
When	Когда (*прим.*: что-то происходит)	Событие, которое инициирует данный сценарий.
Then	Тогда	Результат, который пользователь должен наблюдать в конечном итоге.
And, But	И, Но	Для связки предложений в начальных условиях, триггерах и результатах

BDD: пример описания бизнес-цели

```
Feature: доступ к воронке конверсии

As a администратор
In order to анализ вовлеченности AND выявление слабых этапов
I хочу видеть графическую интерпретацию воронки конверсии AND
   детализацию переходов пользователя по сайту

Scenario 1: смена состояния пользователя должна сохраняться
Given пользователь зашел на сайт
When пользователь меняет состояние # н-р переходит в каталог
Then в отчете должна появиться запись вида “<user-id> <time>
     <state>”
Scenario 2: …
```

Список источников

1. https://javarush.ru/groups/posts/6-chto-takoe-tdd-i-moduljhnoe-testirovanie-
2. https://en.wikipedia.org/wiki/Test-driven_development
3. https://ekaterinagoltsova.github.io/posts/tdd-and-bdd/
4. https://en.wikipedia.org/wiki/Behavior-driven_development
5. https://habr.com/ru/post/459620/
6. https://cucumber.io/docs/gherkin/reference/#keywords