Тестирование

Юрий Литвинов
yurii.litvinov@gmail.com

27.04.2018г

Тестирование, зачем

- Любая программа содержит ошибки
- Если программа не содержит ошибок, их содержит алгоритм, который реализует эта программа
- Если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна

Тестирование не позволяет доказать отсутствие ошибок, оно позволяет лишь найти ошибки, которые в программе присутствуют

Пример

Консольный калькулятор, складывающий два двузначных числа

- Называется adder
- Ввод числа заканчивается нажатием на *Enter*
- Программа должна вывести сумму после ввода второго числа

© C. Kaner, Testing Computer Software

Смоук-тест

Что делаем	**Что происходит**
Вводим *adder* и жмём на *Enter*	Экран мигает, внизу появляется знак вопроса
Нажимаем 2	За знаком вопроса появляется цифра 2
Нажимаем *Enter*	В следующей строке появляется знак вопроса
Нажимаем 3	За вторым знаком вопроса появляется цифра 3
Нажимаем *Enter*	В третьей строке появляется 5, несколькими строками ниже — ещё один знак вопроса

Выявленные проблемы

- Нет названия программы на экране, может, мы запустили не то
- Нет никаких инструкций, пользователь без идей, что делать
- Непонятно, как выйти
- Число 5 выведено слева от слагаемых, а не под ними

План дальнейших тестов

Ввод	**Ожидаемый результат**	**Замечания**
99 + 99	198	Пара наибольших допустимых чисел
-99 + -99	-198	Отрицательные числа, почему нет?
99 + -14	85	Большое первое число может влиять на интерпретацию второго
-38 + 99	61	Отрицательное плюс положительное
56 + 99	155	Большое второе число может повлиять на интерпретацию первого
9 + 9	18	Два наибольших числа из одной цифры
0 + 0	0	Программы часто не работают на нулях
0 + 23	23	0 — подозрительная штука, его надо проверить и как первое слагаемое,
-78 + 0	-78	и как второе

План дальнейших тестов (2)

Ввод	**Замечания**
100 + 100	Поведение сразу за диапазоном допустимых значений
Enter + *Enter*	Что будет, если данные не вводить вообще
123456 + 0	Введём побольше цифр
1.2 + 5	Вещественные числа, пользователь может решить, что так можно
A + b	Недопустимые символы, что будет?
Ctrl-A, Ctrl-D, F1, Esc	Управляющие клавиши часто источник проблем в консольных программах

Ещё больше тестов!

- Внутреннее хранение данных — двузначные числа могут хранить в **byte**
  - 99 + 99, этот случай покрыли
- Кодовая страница ввода: символы ’/’, ’0’, ’9’ и ’:’
  - Программист может напутать со строгостью неравенства при проверке
  - Не надо вводить A + b, достаточно граничные символы

Rules of thumb

- Отдельный багрепорт по каждому багу
- Если от двух тестов ожидается один и тот же результат, нужен только один
  - Факторизация пространства состояний необходима
    - Но невозможна — мы не знаем, как оно устроено внутри
  - Из класса тестов выбирается тот, на котором баг вероятнее всего
- Всегда надо записывать, что делали и что происходит
- Проверяйте граничные условия
- Большая часть тестов ошибок не выявит
- Подсистемы обработки ошибок впиливают в последнюю очередь, поэтому они часто полны багов

Жизнь тестировщика — боль

- Самое важное:
  - Любая программа содержит ошибки
  - Если программа ошибок не содержит, их содержит спецификация, которую она реализует
  - Если ни программа, ни спецификация ошибок не содержит, такая программа даром никому не нужна
- Полностью протестировать программу невозможно
  - Комбинаторный взрыв входных данных
  - Экспоненциальное количество путей исполнения
  - Баги, связанные с асинхронностью и многопоточностью
  - Баги, связанные с пользовательским интерфейсом
- Формально доказать корректность программы невозможно

Пример

- Телефон — конечный автомат из 6 состояний
  - 1 — телефон молчит, при входящем вызове (состояние 2) абонент снимает трубку (состояние 3) или звонящий вешает трубку (состояние 5). Ответив на звонок, абонент может нажать *Hold* (состояние 4) или повесить трубку (состояние 5), при этом после нажатия на *Hold* можно ответить на другой звонок (состояние 6)
- По нажатию *Hold* телефон помещает данные в стек
- Если звонящий вешает трубку, пока его линия в *Hold*, данные не снимались со стека
- По возврату в состояние 1 стек очищается
- Глубина стека — 30. Упс.

Информация к размышлению

- Программа из сотни строк может иметь 10^{18} путей исполнения
  - Времени жизни вселенной не хватило бы, чтобы их покрыть
- После передачи на тестирование в программах в среднем от 1 до 3 ошибок на 100 строк кода
- В процессе разработки — 1.5 ошибок на 1 строку кода (!)
- Если для исправления ошибки надо изменить не более 10 операторов, с первого раза это делают правильно в 50% случаев
- Если для исправления ошибки надо изменить не более 50 операторов, с первого раза это делают правильно в 20% случаев

Цель тестирования

- Цель тестирования — сделать так, чтобы ошибки исправили
- Основная задача тестировщика — выявить ошибки
  - И “продать” их программистам и руководству
- Тест, который не выявил ошибку — пустая трата времени
- Чем раньше будет выявлена ошибка, тем проще её исправить

Тестирование требований

- Выполняется на этапе планирования
- Цели:
  - Адекватность требований
  - Полнота, непротиворечивость и т.д.
  - Выполнимость, рентабельность
  - Возможность тестирования
- Способы:
  - Обзор аналогов
  - “Дискуссионные группы”
  - Исследование объекта автоматизации

Тестирование архитектуры

- Выполняется на этапе проектирования
- Цели:
  - Проверка соответствия требованиям
  - Проверка ожидаемых качеств
  - Проверка полноты и реалистичности
  - Проверка подсистемы обработки ошибок
- Методы:
  - Ревью
  - Формальный анализ архитектуры

Тестирование “белого ящика”, юнит-тесты

- Coverage
  - Покрытие строк кода — самый слабый критерий
  - Покрытие путей исполнения — тоже не полный
  - Покрытие всех составляющих каждого условия — символьное исполнение?
  - Всё равно особо не поможет (var x = y / z;)
- Стратегии тестирования:
  - “Снизу вверх” — юнит-тесты
  - “Сверху вниз” — интеграционные тесты + моки
- Статическое тестирование
- Мутационное тестирование
- Регрессионное тестирование

Тестирование “чёрного ящика”

- Квалификационное тестирование (смоук-тест)
  - Квалификационные тесты имеет смысл опубликовать
- Оценочное тестирование
- Функциональное тестирование — проверка на соответствие спецификации
- Тестирование целостности — проверка на соответствие пользовательской документации
- Бета-тестирование
- Тестирование инсталлятора
- Приёмка и сертификация
- ...

Примеры видов тестов “чёрного ящика”

- Сверка со спецификацией
- Лабораторные испытания на группе потенциальных пользователей
- Тесты на эргономичность
- Тесты на граничные условия
- Тесты на производительность
- Тесты на переходы между состояниями
- Тесты на асинхронное взаимодействие
- Эксплуатация в реальном режиме
- Нагрузочные тесты
- Тесты на обработку ошибок
- Тесты на защиту от несанкционированного доступа
- Тесты на совместимость импорта/экспорта и программную совместимость
- Тесты на аппаратные конфигурации
- Адаптационное тестирование

Что такое “Ошибка”

- Качество ПО определяется:
  - Возможностями, которыми оно понравится пользователю
  - Недостатками, которые вынудят пользователя купить другое ПО
- **Соответствие спецификации, качество кода и т.д. — не качество!**
- Понятие “ошибка” субъективно

Категории ошибок

- Ошибки пользовательского интерфейса
  - Функциональность — программа не делает того, что от неё ожидает пользователь
    - Пользователей много и все ожидают что-то своё
  - Взаимодействие с пользователем
  - Структура интерфейса
  - Пропущенная функциональность
  - Производительность
  - Выходные данные

Категории ошибок (2)

- Обработка ошибок
- Обработка граничных условий
- Ошибки вычислений
- Ошибки инициализации и первого запуска
- Ошибки управления потоком
- Ошибки хранения, передачи или интерпретации данных
- Гонки
- Ошибки работы под нагрузкой
- Ошибки взаимодействия с аппаратным обеспечением

Категории ошибок (3)

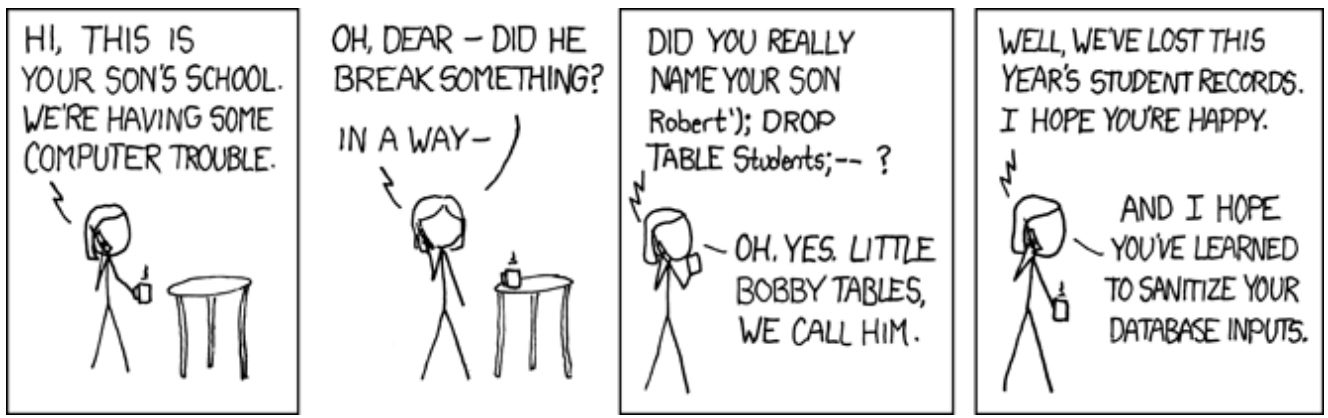
- Ошибки версионного контроля
- Ошибки документации
  - Да, её тоже тестируют
    - Более того, надо внимательно проверять вообще всё, что “кладётся в коробку”
- Ошибки тестирования
  - Тесты тоже содержат код, следовательно, ошибки

Test plan

- План тестирования — формальный документ, описывающий, что, как и когда надо проверить
- Нужен, чтобы:
  - Ничего не забыть и не пропустить
  - Проанализировать программу и выбрать лучшие тесты
  - Обеспечить взаимодействие между членами команды
  - Выполнить оценку трудозатрат
  - Скорректировать спецификацию и архитектуру системы
- Стандарты IEEE-829 и IEEE-29119
  - Как правило, не нужны
  - План полезен настолько, насколько он помогает организации тестирования и поиску ошибок

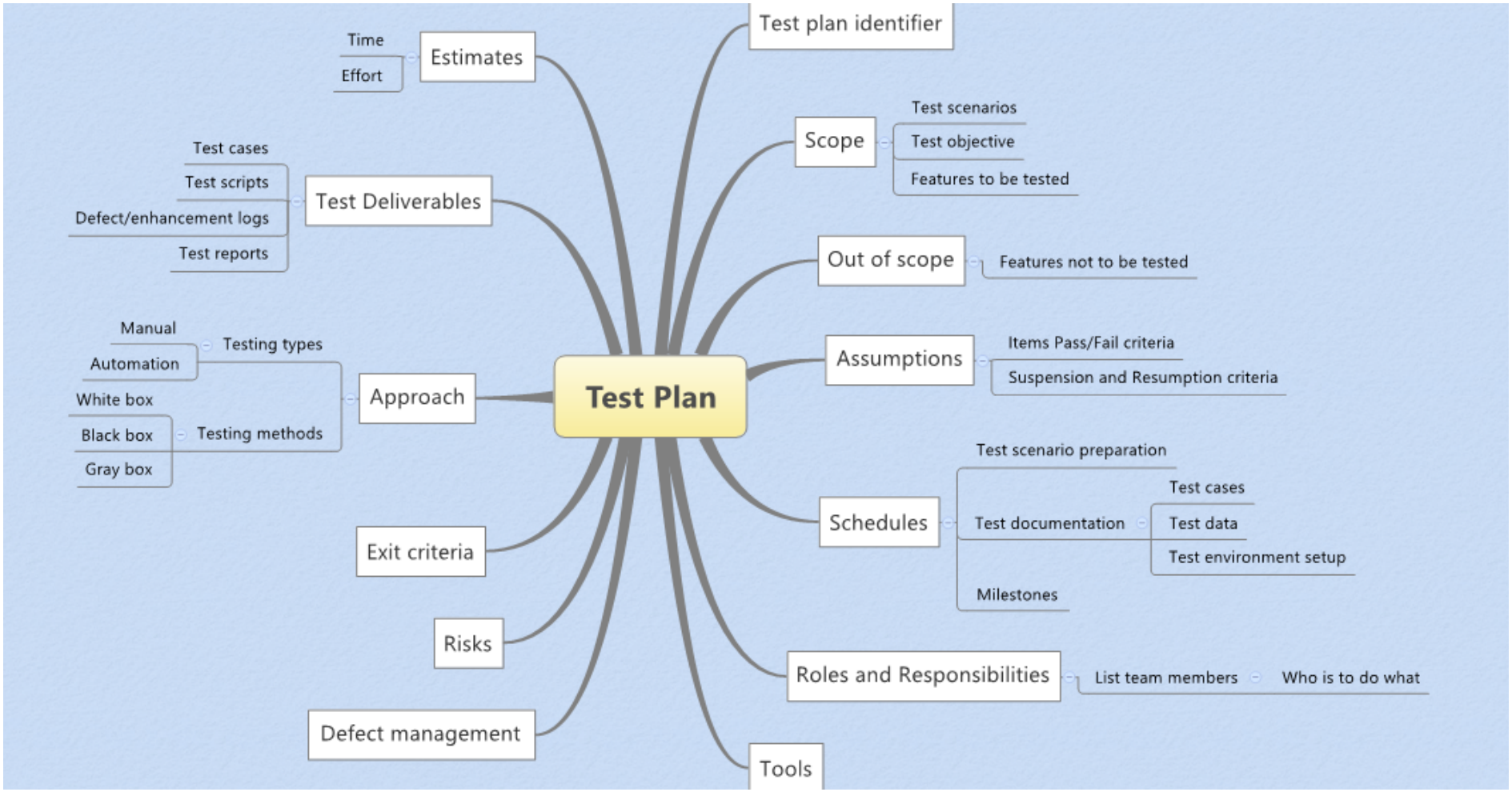
Какие тесты фиксировать в плане

- Тестирование “белого ящика” часто оставляют программистам
- Что оно упускает:
  - Ошибки, связанные со временем — гонки, асинхронность
  - Особые стечения данных



  - Всё, что относится к UI
  - Ошибки, связанные с конфигурацией и совместимостью
  - Аппаратные ошибки

Составляющие тест-плана



Как писать тест-план

1. Проанализировать продукт
   - Список фич
   - Список экранов
   - Список файлов
2. Определиться со стратегией тестирования
3. Выписать типы ошибок, на которые хотим вообще проверять
4. Определиться с тем, что мы считаем ок, а что не ок
5. Прикинуть способы проверки штук из п. 1 на ошибки из п. 3
6. Оценить трудоёмкость
7. Посмотреть, нельзя ли сэкономить (урезанием функциональности, например)
8. Назначить ресурсы, составить календарный план

См. https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html

Модульные тесты

- Тест на каждый отдельный метод, функцию, иногда класс
- Пишутся программистами
- Запускаются часто (как минимум, после каждого коммита)
- Должны работать быстро
- Должны всегда проходить
- Принято не продолжать разработку, если юнит-тест не проходит
- Помогают быстро искать ошибки (вы ещё помните, что исправляли), рефакторить код (“ремни безопасности”), продумывать архитектуру (мешанину невозможно оттестировать), документировать код (каждый тест — это рабочий пример вызова)

Почему модульные тесты полезны

- Помогают искать ошибки
  - Особо эффективны, если налажен процесс Continuous Integration
- Облегчают изменение программы
  - Помогают при рефакторинге
- Тесты — документация к коду
- Помогают улучшить архитектуру
- НЕ доказывают отсутствие ошибок в программе

Best practices

- Независимость тестов
  - Желательно, чтобы поломка одного куска функциональности ломала один тест
- Тесты должны работать быстро
  - И запускаться после каждой сборки
    - Continuous Integration!
- Тестов должно быть много
  - Следить за Code coverage
- Каждый тест должен проверять конкретный тестовый сценарий
  - Никаких try-catch внутри теста
    - @Test(expected = NullPointerException.class)
    - Любая нормальная библиотека юнит-тестирования умеет ожидать исключения
- Test-driven development

Hamcrest

```
assertThat(someString, is(not(equalTo(someOtherString))));
assertThat(list, everyItem(greaterThan(1)));
assertThat(cat.getKittens(), hasItem(someKitten));
assertThat("test",
    anyOf(is("testing"), containsString("est")));
assertThat(x,
    allOf(greaterThan(0), lessThanOrEqualTo(10)));
```

Mock-объекты

- Объекты-заглушки, симулирующие поведение реальных объектов и контролирующие обращения к своим методам
  - Как правило, такие объекты создаются с помощью библиотек
- Используются, когда реальные объекты использовать
  - Слишком долго
  - Слишком опасно
  - Слишком трудно
  - Для добавления детерминизма в тестовый сценарий
  - Пока реального объекта ещё нет
  - Для изоляции тестируемого объекта
- Для mock-объекта требуется, чтобы был интерфейс, который он мог бы реализовать, и какой-то механизм внедрения объекта

Пример: Mockito

```
@Test
public void test() throws Exception {
    // Arrange, prepare behaviour
    Helper aMock = mock(Helper.class);
    when(aMock.isCalled()).thenReturn(true);
    // Act
    testee.doSomething(aMock);
    // Assert - verify interactions (optional)
    verify(aMock).isCalled();
}
```

Соотношение тестов