Unit-тестирование

Что такое Unit-тест

Фрагмент кода, написанный разработчиком, для провеки очень маленького, специфичного фрагмента функциональности кода.

Зачем нужно возиться с тестами

- они делают жизнь проще
- они делают дизайн приложения лучше
- они значительно уменьшают время, затрачиваемое на отладку.

Чего мы добиваемся написанием тестов

Ответов на вопросы:

- Делает ли код то, что ожидает от него разработчик?
- Делает ли он это все время?
- Можно ли положиться на код?

Побочные эффекты:

- Документирование кода и способов работы с ним

Как писать юнит-тесты

1. Ответить на вопрос: как мы будем тестировать новый метод.
2. Написать тест и тестируемый класс.
3. Запустить тест.
4. Запустить ВСЕ тесты системы.

Типичные отговорки

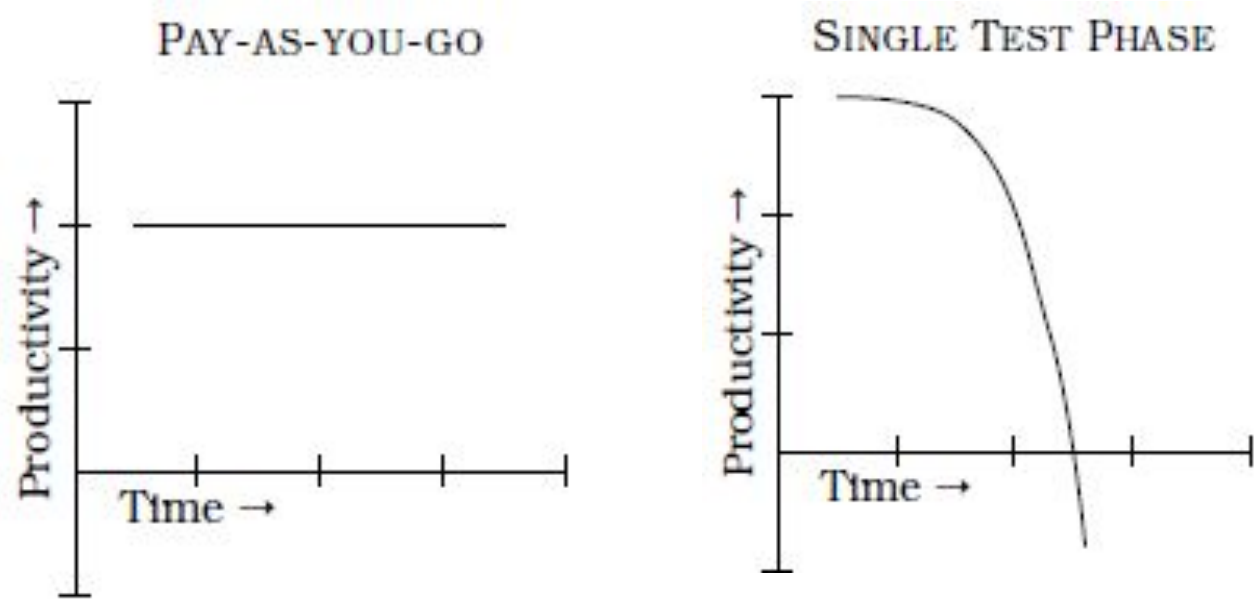
- Написание тестов занимает слишком много времени
- Запуск тестов занимает слишком много времени
- Это не моя работа – тестировать мой код
- Я не знаю точно, как код должен работать, поэтому я не могу написать тест
- Но он же компилируется!
- Мне платят за за написание кода, а не тестов
- Я чувствую вину, оставляя QA без работы
- Моя компания не позволит запустить юнит-тысты на live-системе

Написание тестов занимает слишком много времени

Перед тем, как использовать эту отговорку оцените:

- Сколько времени вы тратите на дебаг кода, написанный вами и вашими коллегами
- Сколько времени тратится на переработку кода после нахождения крупных багов?
- Сколько времени тратится на то, чтобы локализовать обнаруженный баг в коде?

Написание тестов занимает слишком много времени



Что тестировать

Структура теста

- Создать условия, необходимые для тестов (создать объекты, моки, ресурсы и др.)
- Настроить моки (если нужно)
- Вызвать тестируемый метод
- Проверить, что тестируемый метод ведет себя как ожидается
- Прибрать за собой (закрыть ресурсы и т.п.)

Right-BICEP

- **Right** – верны ли результаты?
- **B** – **Boundary** верны ли граничные условия?
- **I - Inverse** можно ли проверить обратные связи?
- **C - Cross-check** можно ли проверить результат другими методами?
- **E - Error conditions** можно ли вызвать ошибочные состояния искусственно?
- **P - Performance** удовлетворительна ли производительность?

Right
верны ли результаты

- Если результаты выполнения кода верны, то как я об этом узнаю?

B – Boundary
Проверка граничных условий

CORRECT

- **C**onformance – верен ли формат значения?
- **O**rdering – является ли правильный набор данных упорядоченным?
- **R**ange – лежит ли значение в пределах конкретных минимального и максимального значения?
- **R**eference – ссылается ли код на что-либо внешнее, что не находится под контролем данного кода? Зависит ли от состояния зависимостей, состояние объекта? Зависит ли код от каких-либо условий.
- **E**xistence -
- **C**ardinality – количественная проврка результата (нужное ли количество возвращается/используется методами)?
- **T**ime – все ли происходит в нужном порядке? В нужное время? В пределах допустимого времени? Правильно ли обрабатываются конкурентные условия?

I - Inverse
проверка обратных связей

```
public void testSquareRootUsingInverse() {
  double x = mySquareRoot(4.0);
  assertEquals(4.0, x * x, 0.0001);
}
```

C - Cross-check
проверка резултат другими методами

```
public void testSquareRootUsingStd() {
   double number = 3880900.0;
   double root1 = mySquareRoot(number);
   double root2 = Math.sqrt(number);
   assertEquals(root2, root1, 0.0001);
}
```

E – Force Error conditions
вызов ошибочных состояний

Типичные ошибочные состояния

- Кончилась память
- Кончилось место на диске
- Проблемы с синхронизацией времени
- Доступность и ошибки сетей
- Система под нагрузкой
- Ограниченная цветовая палитра
- Высокое/низкое разрешение экрана

P – Performance
параметры производительности

- Производительность алгоритма
- Скорость выделения ресурсов
- Скорость доступа к ресурсам
- Время обработки запроса
- Потребляемая память

Свойства хорошего теста

A-TRIP:

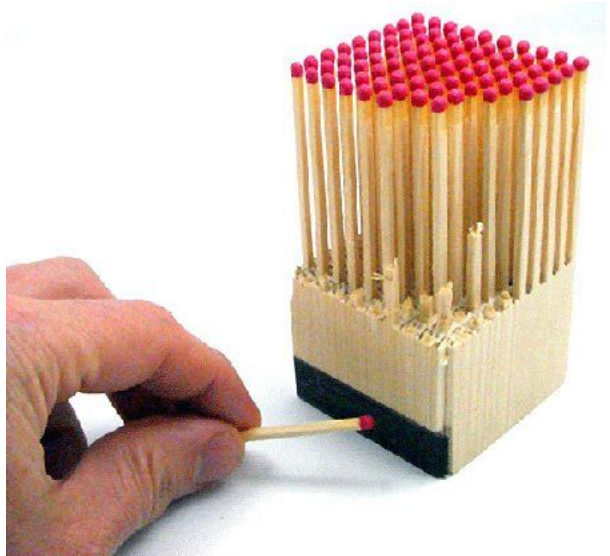
- **A**utomatic – тесты должны запускаться автоматически и проверять результаты автоматически.
- **T**horough – тестируйте все, что может сломаться.
- **R**epeatable –тесты должны запускаться снова и снова, в любом порядке и всегда возвращать одинаковые результаты.
- **I**ndependent – тесты должны быть независимы (от окружения и друг от друга).
- **P**rofessional – тест, это код и он должен быть таким же качественным, как и обычный код.

Automatic

- Не пишите тесты, которые требуют ввода пользователя
- Если метод требует ресурс (базу данных, соединение по сети), сделайте заклушку или мок для тестирования такого метода
- Если запуск всех тестов требует длительного времени, то при разработке целесообразно использовать отдельный сервер, который будет запускать такие тесты и рассылать отчеты разработчикам (Continuous Integration)

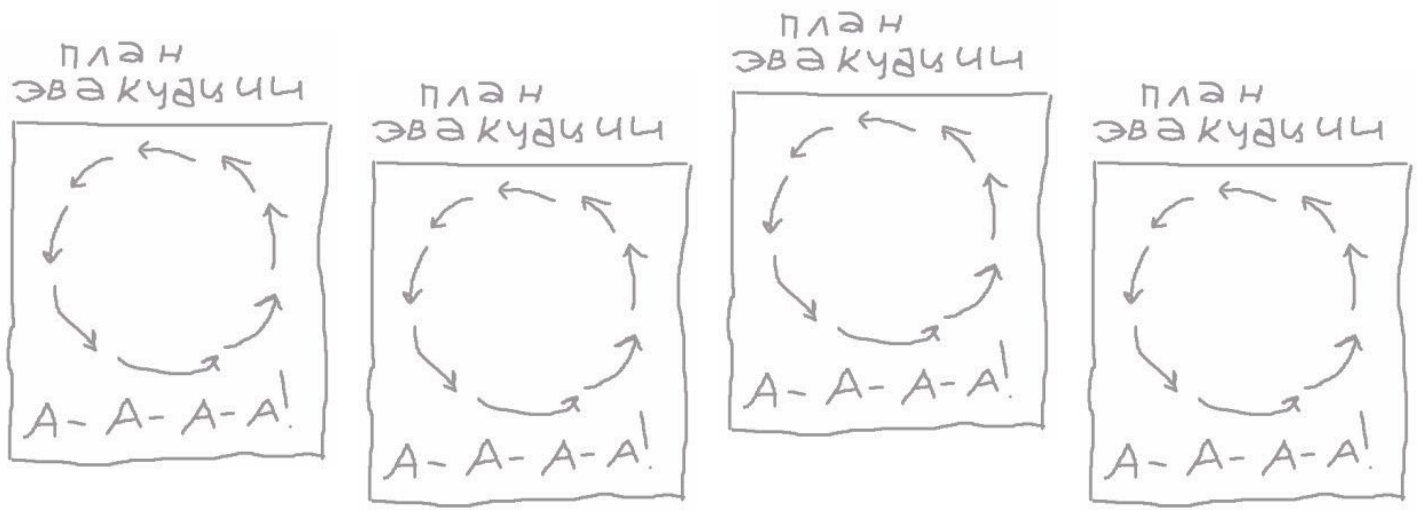
Thorough (целостность)

- Чем больше процент покрытия тестами кода, тем меньше проблем.
- Баги имеют свойство группироваться в проблемных местах ▫ иногда проще и дешевле переписать такие проблемные места с нуля.

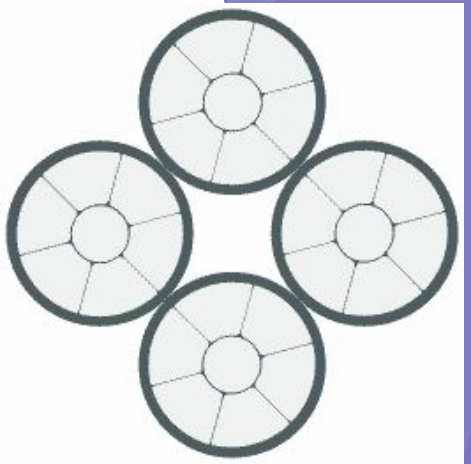


Repeatable

- Результаты тестов не должны зависеть от чего-то, находящегося под вашим непосредственным контролем (ресурсы, данные из баз данных, внешних систем и т.п.)
- Используйте моки и заглушки для изоляции теста от таких систем



Independent



- Убедитесь, что вы тестируете только одну вещь одновременно.
- Разбивайте сложные операции на несколько более мелких и тестируйте их независимо.
- Некоторая функциональность может иметь несколько тестов, проверяющих различные ее аспекты (например, тест с применением некорректных значений, тест метода обычном режиме для обработки единичных значений, тест для метода в пакетном режие для массовых операций).
- Вы не должны зависеть от порядка запуска других тестов.

Professional

- Используйте все принципы хорошего дизайна: DRY (Don’t Repeat Yourself), сохраняйте инкапсуляцию, уменьшайте связность компонентов.
- Не пишите тесты в линейном процедурном стиле, если можно использовать преимущества ООП.
- Некоторые связанные методы тестирования можно инкапсулировать в отдельные классы.
- Если необходимо, создавайте фреймворки для тестирования.
- Не тратьте время на тестирование банальностей (не нужно писать юнит-тесты для get/set-методов).
- Хорошее покрытие кода ~1:1 (1 строчка кода к одной строчке теста).

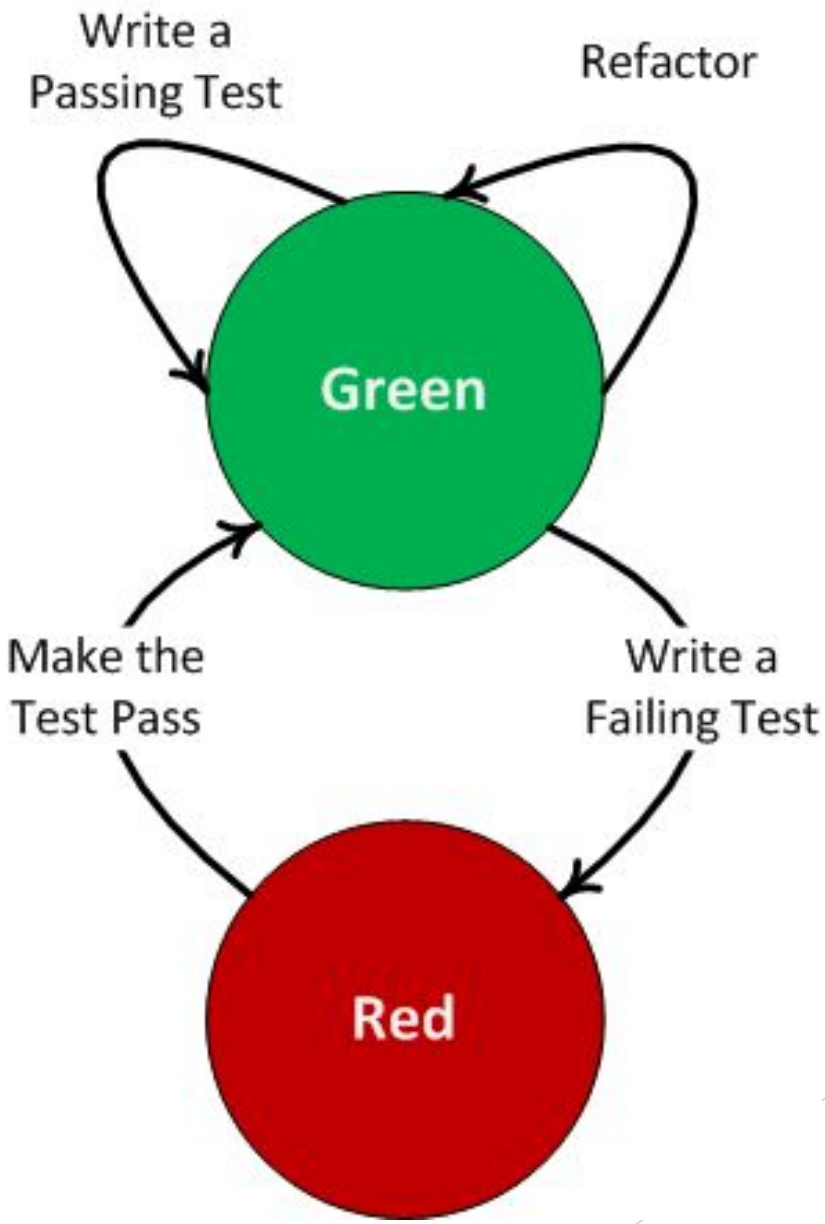
Тестирование тестов

- Не тестируйте тесты – чаще всего в этом нет необходимости.
- Улучшайте тесты по мере исправления багов.

Как исправлять баги

- Идентифицировать баг
- Написать тест, который «поломается» от этого бага, чтобы подтвердить наличие бага.
- Исправить код так, чтобы тест выполнился.
- Запустить **ВСЕ** остальные тесты.

Red-Green-Refactor



Общие принципы

- Тестируйте все, что может сломаться
- Тестируйте все, что уже сломалось
- Новый код признается виновным, пока не доказана его невиновность.
- Тестового кода должно быть как минимум сколько же, сколько и production-кода.
- Запускайте юнит-тесты локально при каждой компиляции.
- Запускайте все юнит-тесты перед check-in-ом в репозиторий.

Вопросы, которые стоит задавать при написании кода и тестов

- Если код выполнился правильно, как я об этом узнаю?
- Как я собираюсь тестировать это?
- Что еще может пойти не так?
- Может ли такая проблема «всплыть» где-нибудь еще?