TDD Test Driven Development

Unit Testing

Integration Testing

Performance Tests

UI Testing

Разработка через тестирование

Материал из Википедии — свободной энциклопедии

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 31 октября 2018; проверки требуют 7 пр

Разработка через тестирование (англ. *test-driven development, TDD*) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. Кент Бек, считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн и внушает уверенность (англ. *inspires confidence*)[1].

В 1999 году при своём появлении разработка через тестирование была тесно связана с концепцией «сначала тест» (англ. *test-first*), применяемой в экстремальном программировании[2], однако позже выделилась как независимая методология.[3].

Тест — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную.

Модульное тестирование

Материал из Википедии — свободной энциклопедии

[править | править код]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 23 марта 2020; проверки требуют 4 правки.

Модульное тестирование, иногда **блочное тестирование** или **юнит-тестирование** (англ. *unit testing*) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии*, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Например, обновить используемую в проекте библиотеку до актуальной версии можно в любой момент, прогнав тесты и выявив несовместимости.

Интеграционное тестирование

Материал из Википедии — свободной энциклопедии

[править | править код]

Интеграцио́нное тести́рование (англ. *Integration testing*, иногда называется англ. *Integration and Testing*, аббревиатура англ. *I&T*) — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика».

Тестирование производительности

Материал из Википедии — свободной энциклопедии

[править | править код]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 27 апреля 2015; проверки требуют 35 правок.

Тестирование производительности (англ. *Performance Testing*) в инженерии программного обеспечения — тестирование, которое проводится с целью определения, как быстро работает вычислительная система или её часть под определённой нагрузкой. Также может служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов.

Тестирование производительности — это одна из сфер деятельности развивающейся в области информатики инженерии производительности, которая стремится учитывать производительность на стадии моделирования и проектирования системы, перед началом основной стадии кодирования.

2. **UI** — тестирования уже самого интерфейса приложения, всех его свистелок и “не багов, а фич”. Заключается в том, что мы имитируем действия пользователя — клики, переходы по ссылкам, и другие действия подобного плана. Смысл его — в проверке взаимодействия компонентов друг с другом.

Один модуль – одна задача. Вспомним SOLID)))
Мы изучим TDD / Unit Testing / UI Testing

UNIT TEST

```
func add(_ item: ToDoItem) {
    if !toDoItems.contains(item) {
        toDoItems.append(item)
    }
}
```

Проверка на уникальность

```
func testAddWhenItemIsAlreadyAddedDoesNotIncreaseCount() {
    sut.add(ToDoItem(title: "Foo"))
    sut.add(ToDoItem(title: "Foo"))

    XCTAssertEqual(sut.toDoCount, 1)
}
```

Происходит ли добавление в array

```
func testItemAtAddItemToDoCountOne() {
    let item = ToDoItem(title: "")
    sut.add(item)
    XCTAssertEqual(sut.toDoCount, 1, "ToDoCount is not equal 1")
}
```

Что тестировать?

Основные функциональные возможности: классы и методы моделей и их взаимодействие с контроллером.

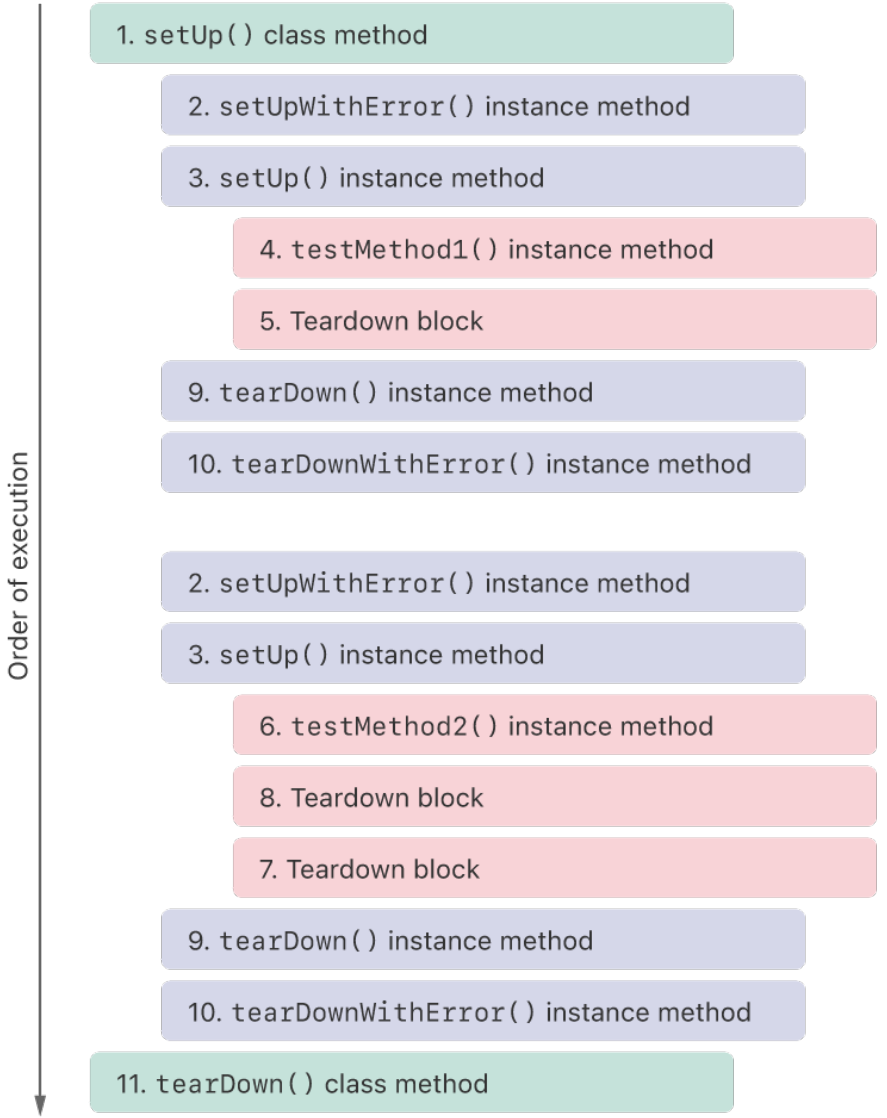
Наиболее распространенные рабочие процессы пользовательского интерфейса.

Bug Fixes

Best Practices for Testing

FIRST принципы

- **Fast**: тесты должны выполняться быстро.
- **Independent/Isolated**: тесты должны быть независимы и изолированы друг от друга.
- **Repeatable** (воспроизводимость / повторяемость): тесты должны выдавать одни и те же результаты при каждом запуске.
- **Self-validating**: тесты должны быть полностью автоматизированы. Результат тестирования должен быть либо успешным либо провальным.
- **Timely**: в идеале тесты должны быть написаны до написания тестируемого продакшн кода (разработка на основе тестов: TDD)

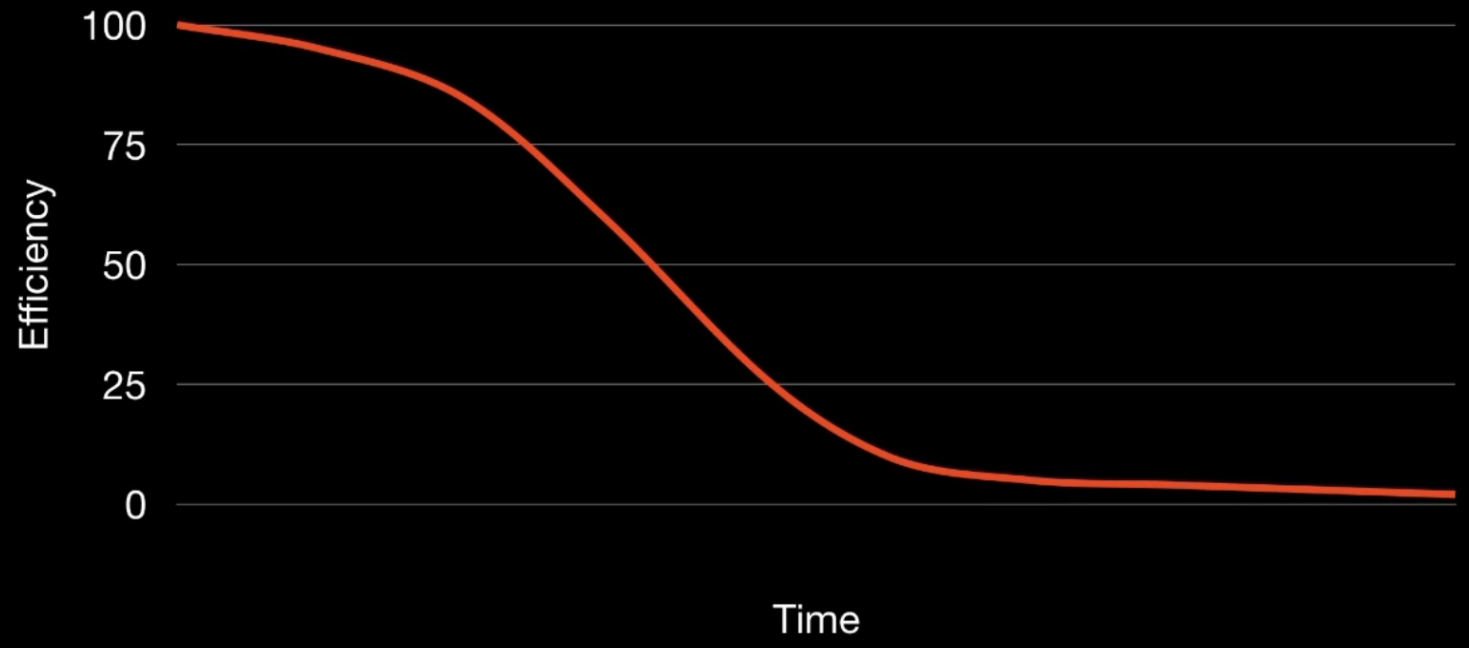

Order of execution
1. setUp() class method
2. setUpWithError() instance method
3. setUp() instance method
4. testMethod1() instance method
5. Teardown block
9. tearDown() instance method
10. tearDownWithError() instance method
2. setUpWithError() instance method
3. setUp() instance method
6. testMethod2() instance method
8. Teardown block
7. Teardown block
9. tearDown() instance method
10. tearDownWithError() instance method
11. tearDown() class method

Unit Testing и TDD одно и то же?

Unit Testing - процесс в программировании, позволяющий проверить на корректность единицы исходного кода.

TDD (Test Driven Development) - методология разработки ПО, основанная на повторении очень коротких циклов разработки. И первое, что пишется - тест.

TDD ot not TDD?




TDD
Time
Написание кода
Дебаггинг
не TDD
Time
Написание код
Дебаггинг

Плюсы TDD:

- Вы пишите модульный код, потому что ваш код будет основан на тестах.
- Ваши тесты являются документацией к вашему коду.
- Используя тесты, вы вкладываете свое время в разработку, а не в дебаггинг.
- Вы не будете боятся менять или рефакторить ваш код, так как он будет покрыт тестами.
- Вы контролируете каждую строчку кода, которую пишите.

Минусы TDD:

- Вы пишите значительно больше кода, чем при разработке без использования тестов.
- Больше поддержки.
- Нет гарантии, что код без ошибок.
- Время разработки.

Три правила Дядюшки Боба

Роберт Сесил Мартин

Нельзя писать продакшн код, пока не написан для него тест.
Нельзя продолжать писать тест, если он уже не проходит или вызывает ошибку компиляции.
Нельзя писать больше кода, чем достаточно для прохождения теста.

TDD Workflow

Red. Сначала вы пишите тест.

Green. Пишите достаточный код для того, чтобы тест был пройден.

Refactor. Проверьте, есть ли код, который нужно рефакторить.