

Тестирование программистом. Юнит-тесты. Фреймворки для тестирования

Введение

- Программисты
 - не должны надеяться на то, что их код работает правильно
 - должны **доказывать** корректность кода снова и снова
- Лучший способ доказать - автоматизированные тесты
 - Обычно программисты выполняют ручное тестирование
 - Автоматизированный тест
 - Пишется программистом
 - Запускается на компьютере
- Во время тестирования
 - Тестировщик ищет баги
 - Программист убеждается в корректности программы

Unit тесты

- Программисты тестируют сам код, а не результат щелчка по кнопке на сайте
- Unit-тест – блок кода (обычный метод), который вызывает тестируемый блок кода и
 - Тестирует минимально возможный участок кода
 - Класс
 - Метод
 - Проверяет его правильность работы (сравнение ОР и ФР)
- Тестируемый код
 - Тестируемая система (SUT, system under test)
 - Тестируемый класс (CUT, class under test)

Когда пишутся тесты

- Мы создаем тесты по мере написания кода, не ожидая завершения написания всего приложения
 - Также как ручное тестирование
 - У нас может не быть UI или других классов, но мы все равно тестируем наш код

Свойства хорошего unit теста

- Автоматизированный и повторяемый
 - После написания тест должен остаться для последующего использования, чтобы использовать как регрессионное тестирование
- Должен легко запускаться и выполняться быстро
 - Чтобы выполняться как можно чаще и программист не ленился их запускать
- Простым в реализации
 - Чтобы программист не ленился писать юнит-тесты
 - Сложные тесты занимают много времени программиста
 - Написать юнит-тест не сложно, сложнее написать код, который будет поддерживать тестирование

Свойства хорошего unit теста

- Любой участник разработки должен иметь возможность запустить unit тест
 - Поэтому тесты должны сохраняться в dll
- Независимые (могут запускаться независимо)
- Отсутствие побочных эффектов!

Хранение тестов

Тесты можно хранить

- Снаружи проекта как отдельный проект
 - в релиз будет уходить только код
- Внутри рабочего проекта
 - тесты будут поставляться вместе с кодом, что позволит запустить их на пользовательском компьютере

Имя тест-кейса

- Юнит-тесты необходимо сопровождать как и обычный код
 - поэтому важно выбирать правильные имена
- Имя тест-кейса
 - объясняет для чего он нужен
 - другие программисты смогут понять для чего он нужен
 - помогает лучше разобраться нам самим, что мы тестируем
 - не понимая этого, мы не сможем написать тест (также как обычная функция)

Именованние тестов

- Много способов именованния юнит-тестов
 - Бывают соглашения по именованию внутри компании/отдела
- Именованния тестового класса для Foo – FooTest
 - Каждый класс тестирует только одну сущность
- Принцип именованния тестов
[Тестирующийся метод]_[Сценарий]_[Ожидаемое поведение]

Имя теста как спецификация

- `IsAdult_AgeLessThan18_False`
- `ParseInt_Should.Fails_OnNonNumber`
- `Stack_Should.BeEmpty_AfterCreation`
- `When_MandatoryFieldsAreMissing_Expect_StudentAdmissionToFail`

Имя теста как спецификация

ЧТО ДОЛЖНО БЫТЬ В ИМЕНИ ТЕСТА?

- System Under Test

имя класса, имя метода

- Условия

ВХОД, СОСТОЯНИЕ

- Результат

ожидаемое поведение или требования для проверки

<https://dzone.com/articles/7-popular-unit-test-naming>

Фреймворки для тестирования

- Существует большое количество фреймворков для разных ЯП
https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
- Большинство фреймворков очень похожи, т.к. основаны на общей идее и имеет инфраструктуру (иерархию классов)
 - для создания тестов
 - Вспомогательные функции для assert'ов
 - для запуска тестов (test runners)
- Во многих IDE есть поддержка тестовых фреймворков

Фреймворк

- Фреймворк запускает тесты и выдает результат
 - Сколько тестов запустилось
 - Если произошла ошибка
 - Место ошибки
 - Причина ошибки

Дизайн тест-кейсов

AAA - unit тест состоит из 3 частей

- Arrange – создаем все объекты, которые необходимы для выполнения тестирования
- Act – выполняется тестируемый метод
- Assert – сравнение ожидаемого и фактического результата

Правильная структура теста

- Arrange
- Act
- Assert

Arrange – размещение

Act - действия

Assert - утверждение

```
int expected = 9;  
int[] a = {3,9,-  
22,5};
```

```
int actual =  
ArrayManager.Ma  
x(a);
```

```
Assert.AreEqual(e  
xpected, actual)
```

Assert

- `Assert.AreEqual(expected, actual)`
- `VS`
`Assert.AreEqual(actual, expected)`
- `Assert.That(2+2, Is.EqualTo(4))`
- `Assert.That(x, IResolveConstraint)` *// 0_o ?!?*

Список assert'ов

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

Классы для тестирования

Assert

- Сравнение двух значений
- Проверка на истину/ложь

CollectionAssert

- Сравнение двух коллекций
- Проверка элемента в коллекции

StringAssert

- Сравнение строк
- Проверка подстроки в строке

Ожидание исключения

- `Action action = () => { var z = x / y; };`
- `action.ShouldThrow<DivideByZeroException>();`

