Автоматическое тестирование

via C# и JS

Декабрь 2017

Флягин Павел
КН-401 ИЕНиМ Урфу

Зачем писать автоматические тесты?

- Удостовериться, что код работает
- А также, что он продолжает работать после очередных изменений
- При ручном тестировании тестировщик может забыть проверить один или несколько тест кейсов
- Тесты - всегда актуальная документация на код для разработчиков
- Удобный подход для знакомства с новой библиотекой

Первый тест

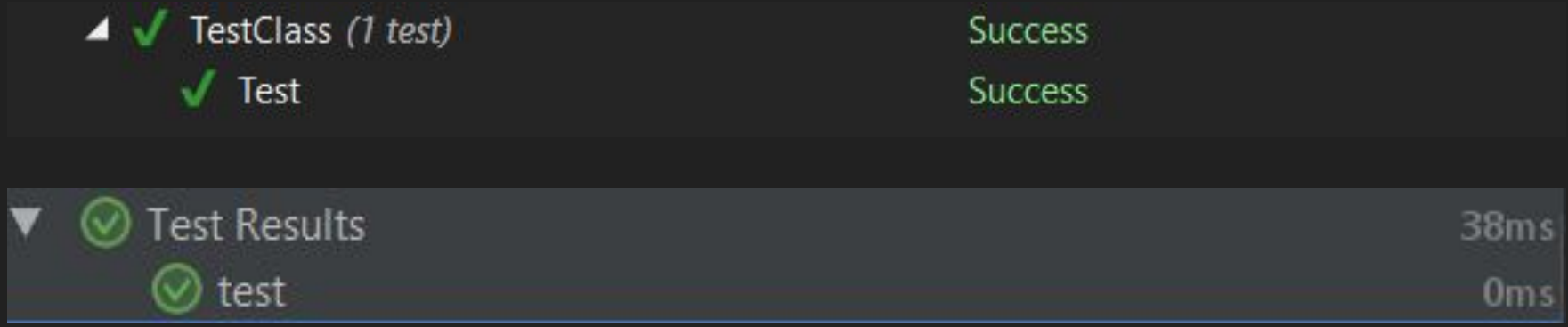
C# NUnit

```
[TestFixture]
0 references
internal class Tests
{
    [Test]
    0 references
    public void Test()
    {
        var calc = new Calculator();
        calc.Add(2);
        calc.Add(2);
        var sum = calc.Result();
        Assert.AreEqual(4, sum);
    }
}
```

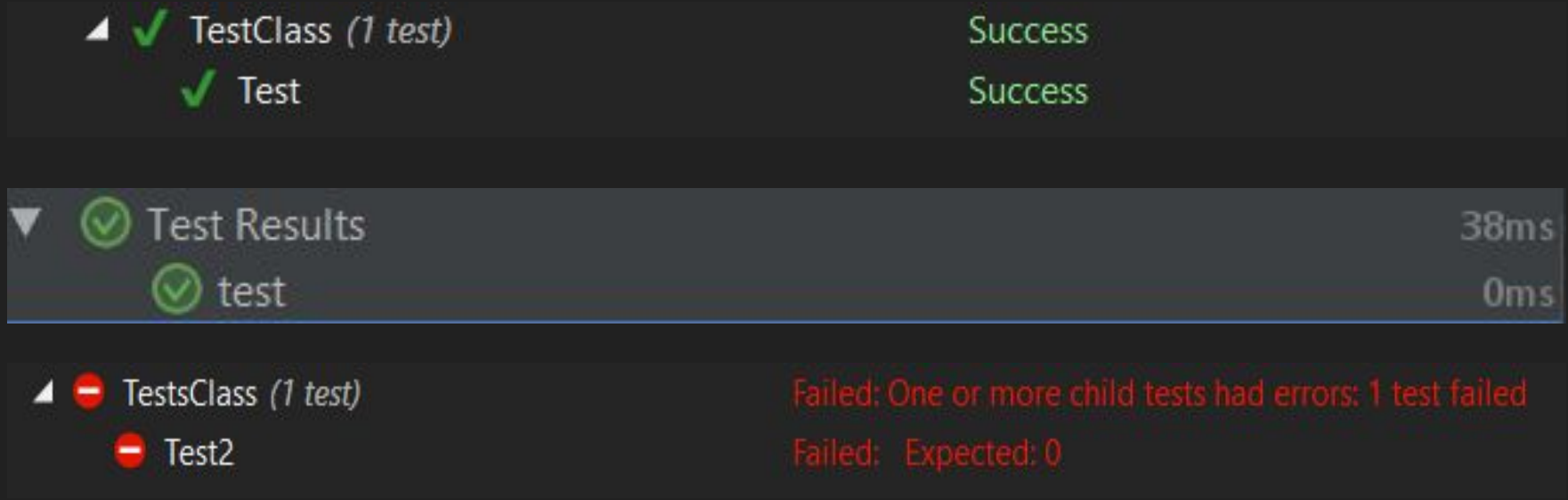
JS Mocha

```
it("test", () => {
    const calc = new Calculator();
    calc.add(2);
    calc.add(2);
    const sum = calc.sum();
    assert.equal(4, sum);
});
```

Результаты теста



Результаты теста



Результаты теста

TestClass *(1 test)* Success

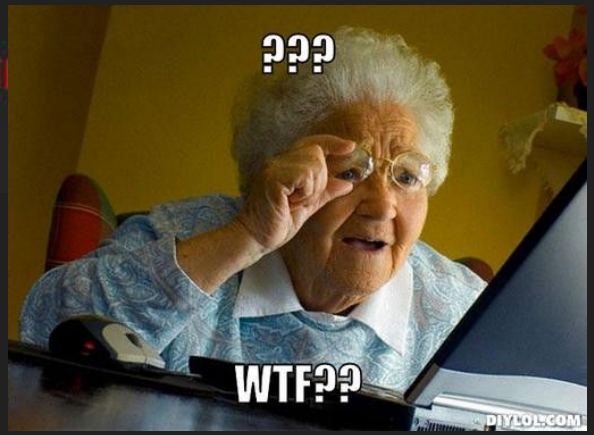
Test Success

Test Results 38ms

test 0ms

TestsClass *(1 test)* Failed: One or more chil

Test2 Failed: Expected: 0



Тесты как спецификация

- Что тестируем (SUT System Under Tests)
- Что ожидаем (expectation)
- (Опционально) При каких условиях (test conditions)

Тесты как спецификация

- Что тестируем (SUT System Under Tests)
- Что ожидаем (expectation)
- (Опционально) При каких условиях (test conditions)

Как достичь?

- Правильное именование тестов
- Группировка тестов

Тесты как спецификация

- Calculator Specification
  - Add
    - Should add given number to accumulated value
    - Should fail if accumulated value overflow
  - Sum
    - Should return 0 by default

- System under test
- Expectation
- Conditions

Тесты как спецификация

- Calculator Specification
  - Add
    - Should add given number to accumulated value
    - Should fail if accumulated value overflow
  - Sum
    - Should return 0 by default

- System under test
- Expectation
- Conditions

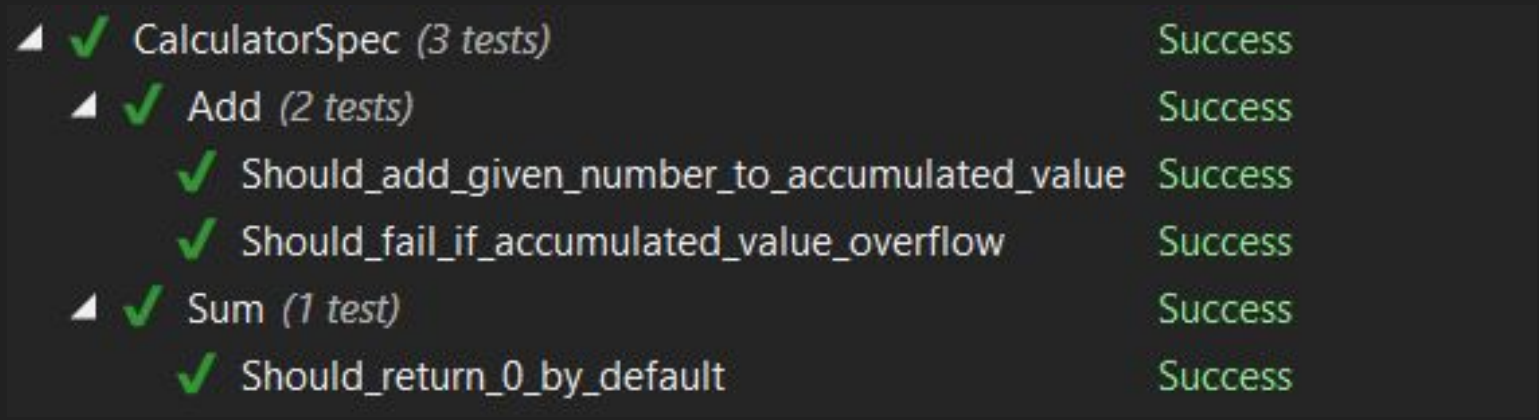
C# реализация

```
[TestFixture]
0 references
internal class CalculatorSpec
{
    [TestFixture]
    0 references
    public class Sum
    {
        0 references
        [Test] public void Should_return_0_by_default() ...
    }
    [TestFixture]
    0 references
    public class Add
    {
        0 references
        [Test] public void Should_add_given_number_to_accumulated_value() ...
        0 references
        [Test] public void Should_fail_if_accumulated_value_overflow() ...
    }
}
```

JS реализация

```
describe("Calculator specification", ()=> {
    describe("add", ()=>{
        it("should add given number to accumulated value",()=>{...});
        it("should fail if NaN parameter passed",()=>{...})
    });
    describe("sum", ()=>{
        it("should return accumulated value",()=>{...});
    })
});
```

Результаты тестов



Результаты тестов

- ✓ CalculatorSpec *(3 tests)* — Success
  - ✓ Add *(2 tests)* — Success
    - ✓ Should_add_given_number_to_accumulated_value — Success
    - ✓ Should_fail_if_accumulated_value_ove
  - ✓ Sum *(1 test)*
    - ✓ Should_return_0_by_default

Структура теста. AAA

- Подготовка (Arrange)
- Действие (Act)
- Проверка (Assert)

```
[Test]
0 references
public void Should_add_given_number_to_accumulated_value()
{
    //arrange
    var calc = new Calculator();

    //act
    calc.Add(2);
    calc.Add(2);

    //assert
    var sum = calc.Result();
    Assert.AreEqual(4, sum);
}
```

Возможные ошибки

Возможные ошибки

```
[Test]
0 references
public void Test()
{
    const int expectedLinesCount = 5;
    var fileLines = File.ReadAllLines("c://my/local/path/file.txt");

    Assert.AreEqual(expectedLinesCount, fileLines.Length);
}
```

Возможные ошибки

```
[Test]
0 references
public void Test()
{
    const int expectedLinesCount = 5;
    var fileLines = File.ReadAllLines("c://my/local/path/file.txt");

    Assert.AreEqual(expectedLinesCount, fileLines.Length);
}
```

Тест работает только на машине разработчика

System.IO.DirectoryNotFoundException : Не удалось найти часть пути "c:\my\local\path\file.txt".

???

Возможные ошибки

```
[Test]
0 references
public void File_should_contain_correct_strins()
{
    var fileLines = File.ReadAllLines("file.txt");

    foreach (var line in fileLines)
        Console.WriteLine(line);
}
```

Возможные ошибки

```
[Test]
0 references
public void File_should_contain_correct_strins()
{
    var fileLines = File.ReadAllLines("file.txt");

    foreach (var line in fileLines)
        Console.WriteLine(line);
}
```

Тест ничего не проверяет. Перманентно зеленый. Даже если изменится значимое содержимое файла, тест будет проходить

???

Возможные ошибки

```
[Test]
0 references
public void File_should_contain_five_strings_and_each_line_start_with_dash_and_end_with_dot()
{
    const int expectedLinesCount = 5;

    var fileLines = File.ReadAllLines("c://my/local/path/file.txt");

    Assert.AreEqual(expectedLinesCount, fileLines.Length);
    foreach (var line in fileLines)
    {
        Assert.AreEqual(true, line.StartsWith("-"));
        Assert.AreEqual(true, line.EndsWith("."));
    }
}
```

Тест проверяет слишком много

```
[Test]
0 references
public void File_should_contain_five_strings_and_each_line_start_with_dash_and_end_with_dot()
{
    const int expectedLinesCount = 5;

    var fileLines = File.ReadAllLines("c://my/local/path/file.txt");

    Assert.AreEqual(expectedLinesCount, fileLines.Length);
    foreach (var line in fileLines)
    {
        Assert.AreEqual(true, line.StartsWith("-"));
        Assert.AreEqual(true, line.EndsWith("."));
    }
}
```

Тест проверяет слишком много. Нет единой точки отказа

Expected: True
But was: False

???

Устраняем дублирование

- Параметризованные тесты (Data Driven Tests)
- Выделение общей фазы Arrange, а также фазы сборки ресурсов после теста

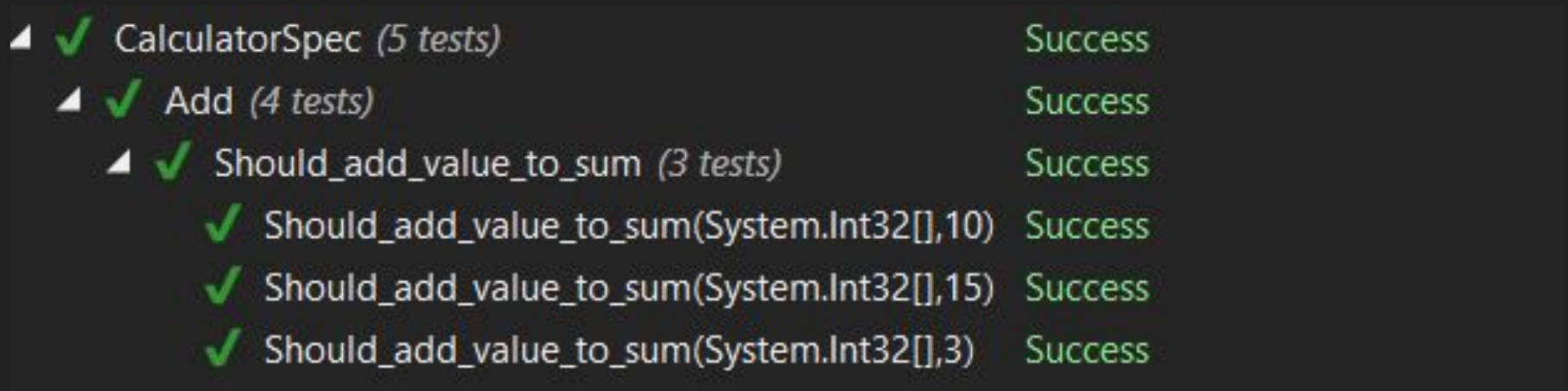
Data Driven Tests C#

```
[TestCase(new[] { 1, 2 }, 3)]
[TestCase(new[] { 5, 10 }, 15)]
[TestCase(new[] { 1, 2, 3, 4 }, 10)]
0 references
public void Should_add_value_to_sum(int[] numbers, int expectedResult)
{
    var calc = new Calculator();

    foreach (var number in numbers)
        calc.Add(number);

    var sum = calc.Result();
    Assert.AreEqual(expectedResult, sum);
}
```

Data Driven Tests C#



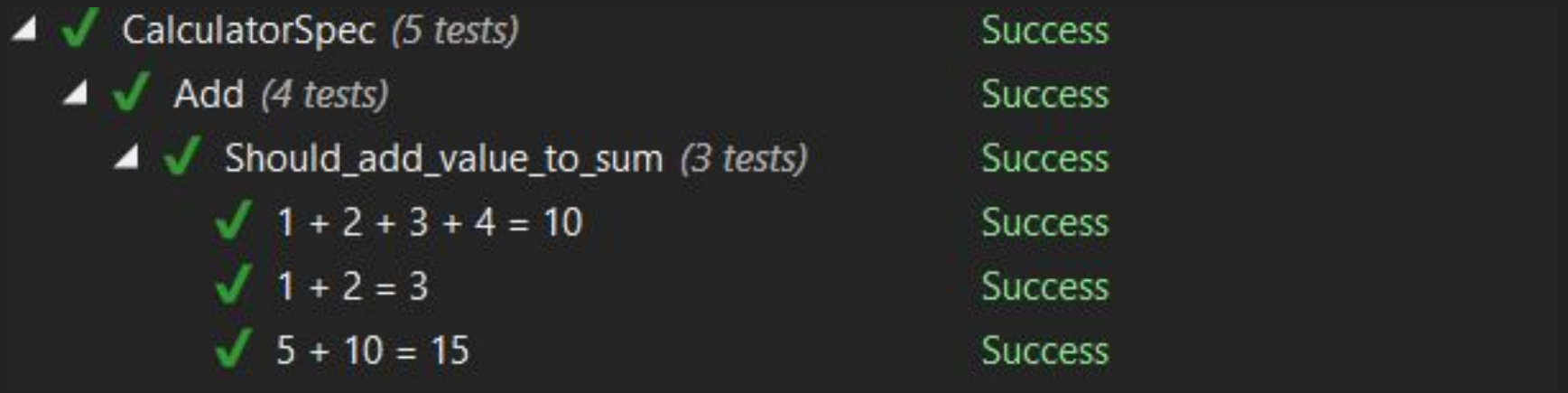
Data Driven Tests C#

```
[TestCase(new[] { 1, 2 }, 3, TestName = "1 + 2 = 3")]
[TestCase(new[] { 5, 10 }, 15, TestName = "5 + 10 = 15")]
[TestCase(new[] { 1, 2, 3, 4 }, 10, TestName = "1 + 2 + 3 + 4 = 10")]
0 references
public void Should_add_value_to_sum(int[] numbers, int expectedResult)
{
    var calc = new Calculator();

    foreach (var number in numbers)
        calc.Add(number);

    var sum = calc.Result();
    Assert.AreEqual(expectedResult, sum);
}
```

Data Driven Tests C#



Data Driven Tests JS

```
[
    {numbers: [1, 2], result: 3},
    {numbers: [10, 5], result: 15},
    {numbers: [1, 2, 3, 4], result: 10},
].forEach((x) => {
    it(`should add given number to accumulated value. ${x.numbers.join('+')}=${x.result}`, () => {
        const calculator = new Calculator();

        for(const number in x.numbers){
            calculator.add(number);
        }

        const sum=calculator.sum();
        assert.equal(x.result, sum);
    });
});
```

Data Driven Tests JS

Настройка окружения

- Выполнить что либо перед запуском всех тестов SUT
- Выполнить что либо перед запуском каждого теста SUT
- Выполнить что либо после запуска каждого теста в SUT
- Выполнить что либо после запуска всех тестов в SUT

Настройка окружения C#

```
[TestFixture]
0 references
internal class NUnitLifeCycleTests
{
    0 references
    [OneTimeSetUp] public void BeforeAllTests() { Console.WriteLine("OneTimeSetUp"); }
    0 references
    [SetUp] public void BeforeEachTest() { Console.WriteLine("SetUp"); }
    0 references
    [TearDown] public void AfterEachTest() { Console.WriteLine("TearDown"); }
    0 references
    [OneTimeTearDown] public void AfterAllTests() { Console.WriteLine("OneTimeTearDown"); }
    0 references
    [Test] public void Test1() {Console.WriteLine("test1"); }
    0 references
    [Test] public void Test2() { Console.WriteLine("test2"); }
}
```

Настройка окружения C#

- OneTimeSetUp
- SetUp
- test1
- TearDown
- SetUp
- test2
- TearDown
- OneTimeTearDown

Настройка окружения JS

```
describe("mocha life cycle tests", () => {
    before(() => { console.log("before all"); });
    beforeEach(() => { console.log("before each"); });
    afterEach(() => { console.log("after each"); });
    after(() => { console.log("after all") });
    it("test1", () => { console.log("test1")});
    it("test2", () => { console.log("test2"); });
});
```

Настройка окружения JS

- before
- beforeEach
- test1
- afterEach
- beforeEach
- test2
- afterEach
- after

Делаем тесты читабельнее

C# FluentAssertions http://fluentassertions.com

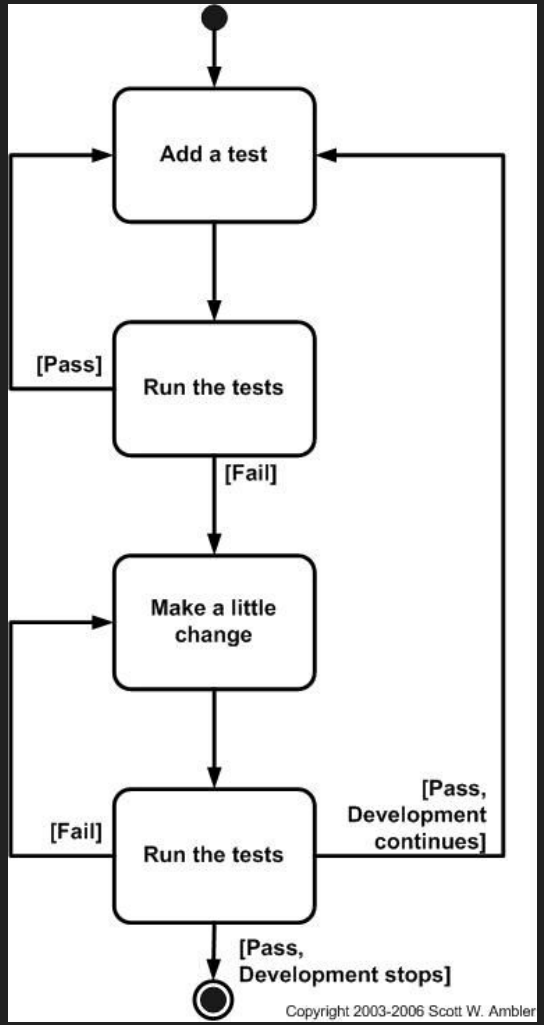
- (2 + 2).Should().Be(4);
- array.Should().HaveCount(3);
- complexObject.ShouldBeEquivalentTo(anotherObject);

JS Chai http://chaijs.com

- (2+2).should.be.equal(2);
- [1,2,3].should.to.have.lengthOf(3)
- complexObject.should.be.to.deep.equal(anotherObject);

Test Driven Development

- Не всегда после написания кода его легко протестировать.
- Не всегда после написания кода, при тестировании, учитывают все юзкейсы
- Не всегда после написания кода вспоминают написать тесты.

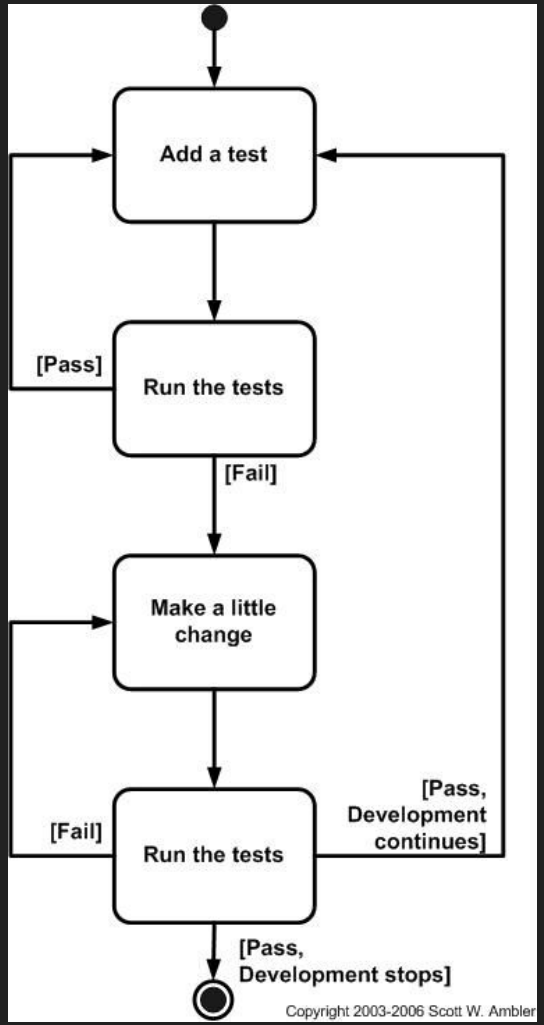


Test Driven Development

- Не всегда после написания кода его легко протестировать.
- Не всегда после написания кода, при тестировании, учитывают все юзкейсы
- Не всегда после написания кода вспоминают написать тесты.

Решение: **Писать тесты параллельно с кодом**

1. Тест (кода нет, тест красный)
2. Минимальный код, чтобы тест стал зеленым
3. Рефакторинг
4. Goto 1



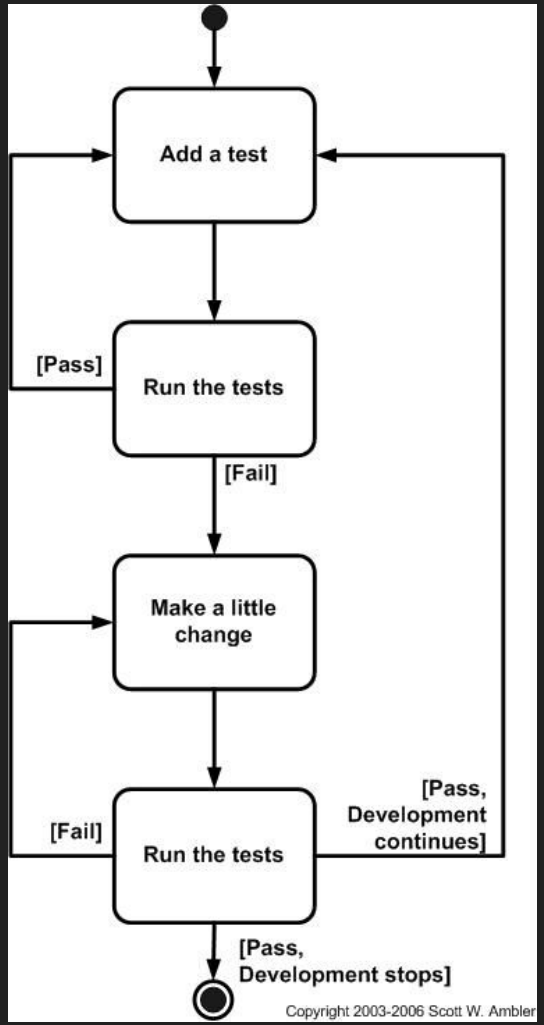
Test Driven Development

- Не всегда после написания кода его легко протестировать.
- Не всегда после написания кода, при тестировании, учитывают все юзкейсы
- Не всегда после написания кода вспоминают написать тесты.

Решение: **Писать тесты параллельно с кодом**

1. Тест (кода нет, тест красный)
2. Минимальный код, чтобы тест стал зеленым
3. Рефакторинг
4. Goto 1

Опасность: Написать тест, который реализовать нетривиально (больше 2-5 минут). Реализация должна быть очевидной. Начинать нужно с простейших тестов и двигаться от простого к сложному



Test Driven Development

Плюсы:

- Код делает только то, что нужно и делает это правильно
- ~100% покрытие тестами
- Упрощает решение сложных задач

Минусы:

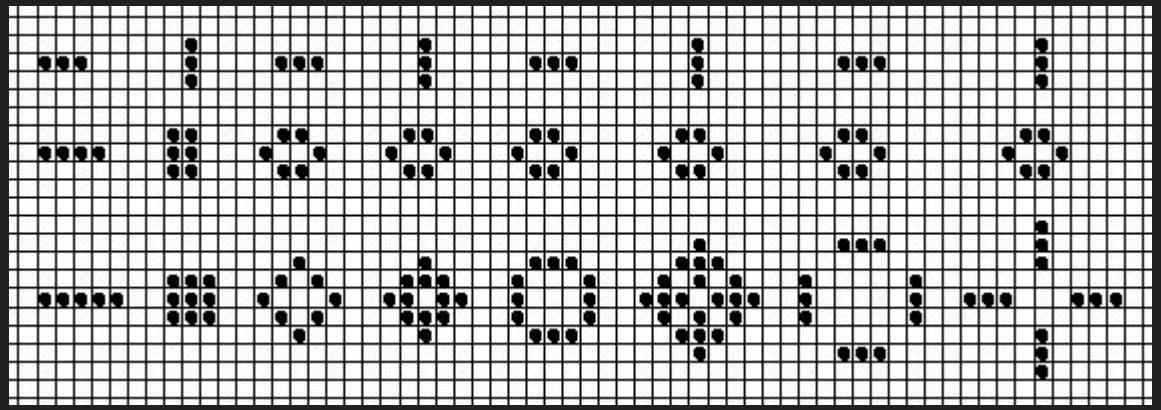
- Увеличивает время разработки
- Далеко не всегда удается соблюдать все формальности
- Мешает полету мысли

Test Driven Development

Наилучшие Use Cases:

- Сложная задача
- Исправление бага (сначала нужно показать, что баг был(тест красный), а потом, что он исправлен (тест зеленый))
- Парная разработка

Практика. Игра жизнь



https://github.com/Panya911/testing-via-cs-and-js

Обязательные требования:

- На каждое правило игры должен быть тест
- Тесты должны быть читабельные и правильно именованные

Необязательные требования:

- Попрактиковаться в TDD. Сначала тест, потом реализация
- PingPong. Один человек пишет тест, второй заставляет его проходить и пишет следующий тест. И так до конца
- Постараться избавиться от дублирования в тестах

Правила:
Дано клеточное поле размера N на M с заданной конфигурацией клеток. Каждая клетка может быть живой или мертвой. Поле замкнуто (зациклено).
Производится серия ходов.
На каждом ходе клетка меняет свое состояние в соответствии со следующими правилами:

- Клетка живая:
  - если клетка имеет 2 или 3 живых соседа, то клетка остается жить,
  - иначе умирает
- Клетка мертвая:
  - если клетка имеет ровно три живых соседа, она оживает
  - иначе остается мертвой

Итоги

- Тесты - хорошо
  - Доверие к работоспособности
  - Легкость изменения (быстрая обратная связь о том, что что-то сломалось)
  - Сокращает время разработки в перспективе (смотри пункт выше)
- Читаемые тесты - еще лучше
  - Доверие к тестам
  - Тесты как спецификация
- TDD - прекрасно
  - Упрощает разработку сложных задач
  - Система делает то, что заявлено, и только это
  - ~ 100% покрытие тестами

Вопросы?