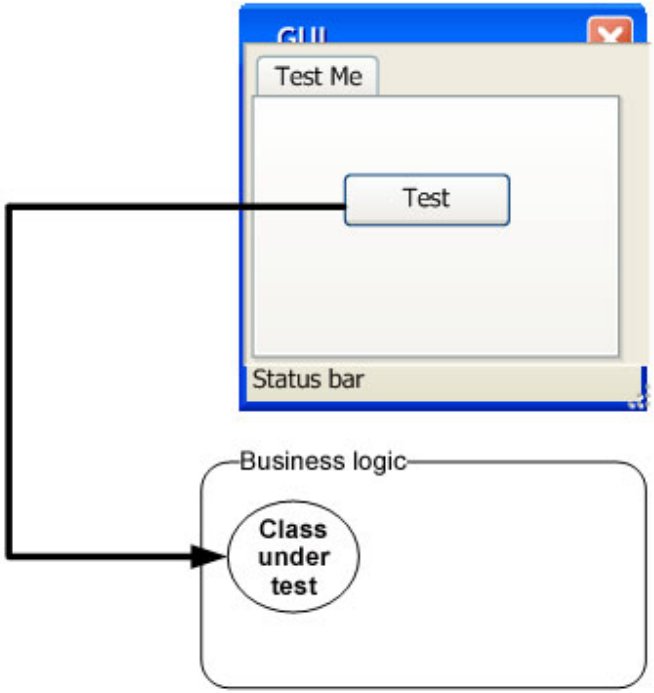
# Что такое unit-тестирование

**Важным этапом создания программного обеспечения является тестирование, в ходе которого выявляются ошибки, проверяется правильность функционирования программы и ее соответствие заявленным требованиям.**



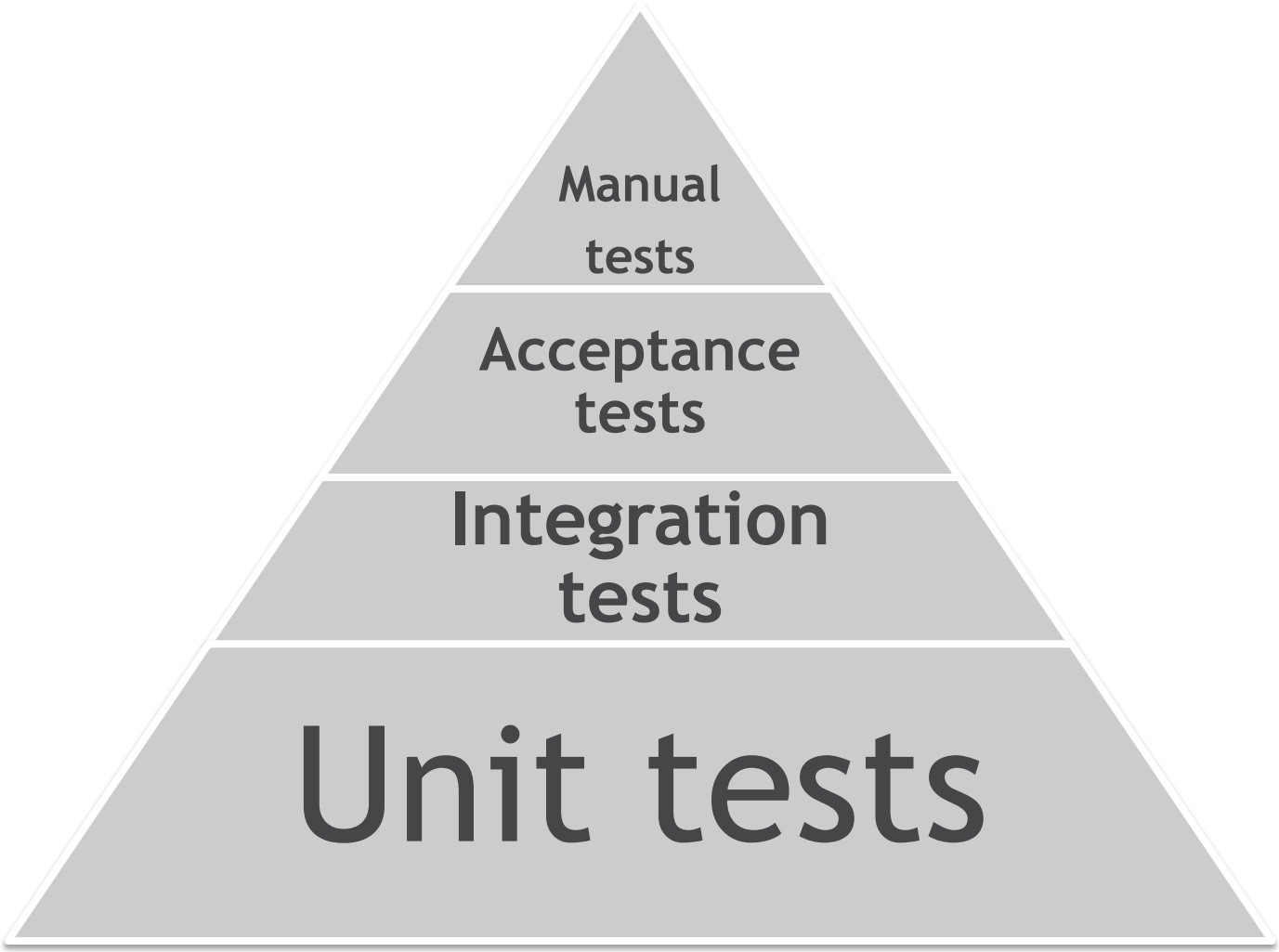
# Что такое unit-тестирование

Модульное тестирование, или юнит-тестирование (англ. unit testing) - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

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.

При модульном тестировании совместно с объектно-ориентированным проектированием, термин «модуль» соответствует понятию «класс», а термин «функция модуля» - понятию «метод класса»

# Что такое unit-тестирование






3

# Что такое unit-тестирование

Ученик спросил великого мастера программирования Летящего Пера:

*"Что превращает тест в юнит-тест?"*

Великий мастер программирования ответил:

*"Если он обращается к базе, значит, он не юнит-тест.*
*Если он обращается к сети, значит, он не юнит-тест.*
*Если он обращается к файловой системе, значит, он не юнит-тест.*
*Если он не может выполняться одновременно с другими тестами, значит, он не юнит тест.*
*Если ты должен делать что-то с окружением, чтобы выполнить тест, значит, он не юнит тест".*

Другой мастер-программист присоединился и начал возражать.
"*Извините, что я спросил*", — сказал ученик.
Позже ночью он получил записку от величайшего мастера-программиста.
Записка гласила:

*"Ответ великого мастера Летящего Пера прекрасный ориентир.*
*Следуй ему, и в большинстве случаев не пожалеешь.*
*Но не стоит застревать на догме.*
*Пиши тест, который должен быть написан".*

Ученик спал хорошо.
Мастера все еще продолжали спорить глубокой ночью.


4

# Когда не нужно писать тесты

Разрабатывается простой сайт-визитка из 5 статических html-страниц и с одной формой отправки письма. На этом заказчик, скорее всего, успокоится, ничего большего ему не нужно. Здесь нет никакой особенной логики, быстрее просто все проверить «руками»

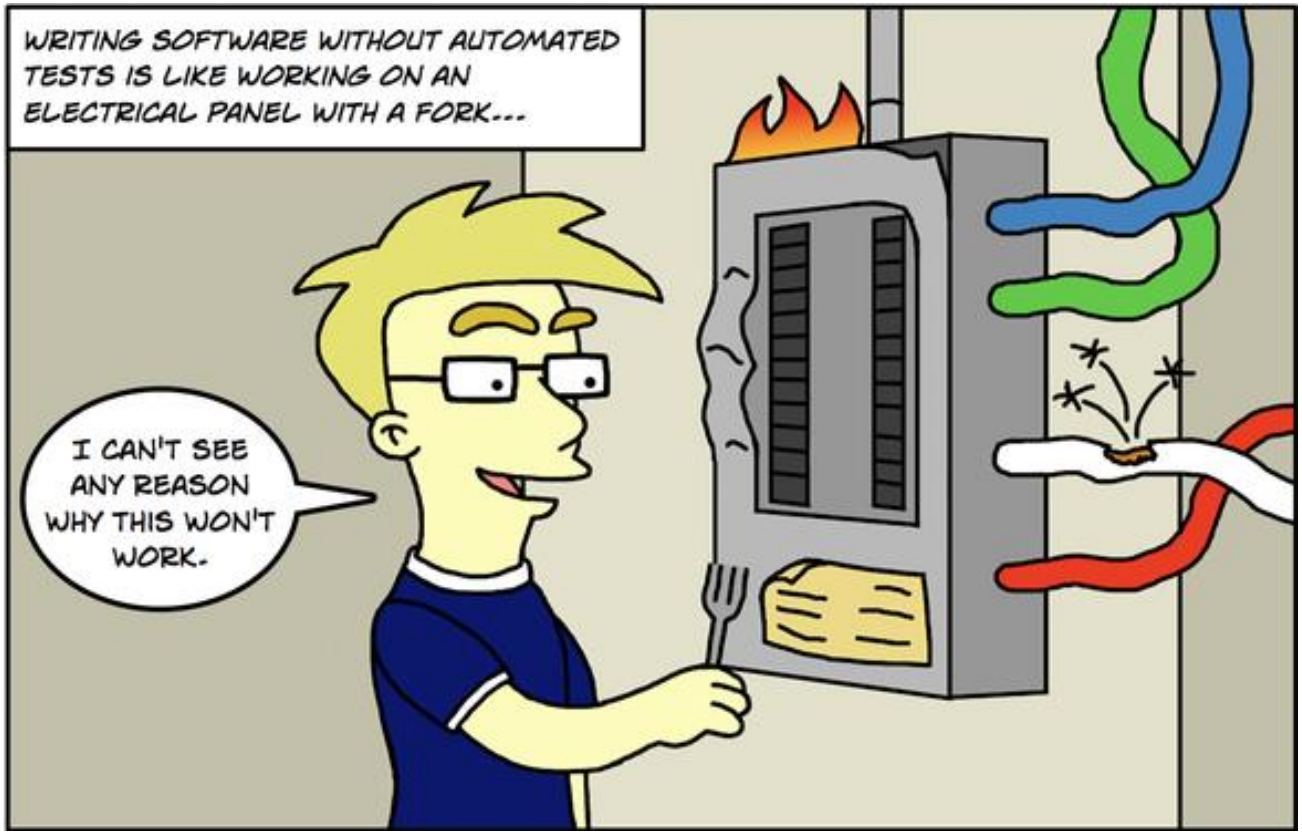
Разрабатывается рекламный сайт/простая флеш-игра или баннер - сложная верстка/анимация или большой объем статики. Никакой логики нет, только представление

Разрабатывается проект для выставки. Срок - от двух недель до месяца, система - комбинация железа и софта, в начале проекта не до конца известно, что именно должно получиться в конце. Софт будет работать 1-2 дня на выставке

Пишется идеальный код без ошибок, наделенный даром предвидения и способный изменить себя сам, вслед за требованиями клиента

# Когда нужно писать тесты

**Любой долгосрочный проект без надлежащего покрытия тестами обречен рано или поздно …**



# Когда нужно писать тесты

...быть переписанным с нуля

# Плюсы и минусы unit-тестирования

Плюсы

- Обеспечивают мгновенную обратную связь
- Помогают документировать код и делать его понимание проще для других разработчиков Позволяют постоянно тестировать код, что сводит к минимуму появление новых ошибок
- Помогают уменьшить количество усилий, необходимых для повторного тестирования
- Поощряют написание слабосвязанного кода

Минусы

- Не проверяют взаимодействие объектов
- Не дают 100%-й гарантии

# Основные правила тестирования

Тесты должны

- быть достоверными;
- не зависеть от окружения, на котором они выполняются;
- легко поддерживаться;
- легко читаться и быть простыми для понимания (новый разработчик должен понять что именно тестируется);
- соблюдать единую конвенцию именования;
- запускаться регулярно в автоматическом режиме.

# Логическое расположение тестов в системе контроля версий

Тесты должны быть частью контроля версий. В зависимости от типа решения, они могут быть организованы по-разному. Общая рекомендация: если приложение монолитное, следует положить все тесты в папку Tests; в случае множества разных компонентов, хранить тесты в папке каждого компонента

# Соответствия между тестируемым и тестирующим кодами

| Объект тестирования | Объект модульного теста |
|---|---|
| **Проект** | Создать проект (Class Library), содержащий тесты, с именем **`ProjectName.Tests`** |
| **Класс** | Для каждого класса, подлежащего тестированию, создать (по крайней мере) один тестирующий класс с именем **`ClassNameTests`**. Такие тестирующие классы называются наборами тестов (test fixtures) |
| **Метод** | Для каждого метода, подлежащего тестированию, создать (по крайней мере) один тестирующий public-метод (тест) с именем **`MethodName_TestConditions_ExpectedBehavior`** |

# Соответствия между тестируемым и тестирующим кодами

| | |
|---|---|
| <PROJECT_NAME>.Core<br><PROJECT_NAME>.Bl<br><PROJECT_NAME>.Web | <PROJECT_NAME>.Core.Tests<br><PROJECT_NAME>.Bl.Tests<br><PROJECT_NAME>.Web.Tests. |
| ProblemResolver | ProblemResolverTests |

```
class Calculator
{
    public void Sum()
    {
        //TODO
    }
}
```

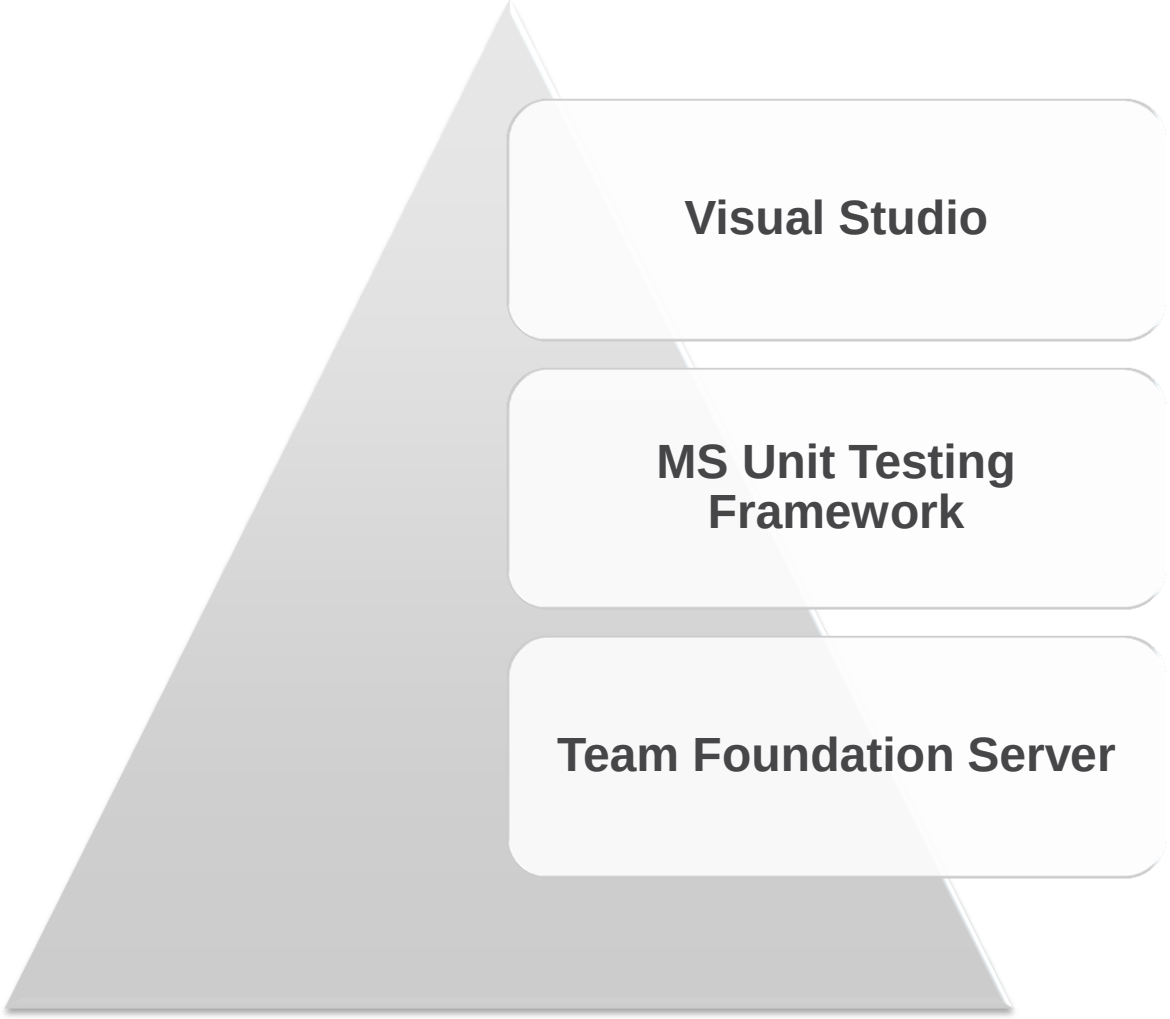
```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        //TODO
    }
}
```

# Инструменты тестирования




13

# Инструменты тестирования. Microsoft



# Сторонние инструменты тестирования

- Среда: Visual Studio, Eclipse
- Фреймворк: NUnit, MbUnit, XUnit.NET, CsUnit (open source)
- Интеграция: TestDriven.Net (free/$), ReSharper (free/$)
- Сервер: CruiseControl (free), TeamCity (free/$)

# Фреймворки для unit-тестирования

Unit-testing Framework – базовый набор средств для написания тестов, предоставляющий следующие возможности

Библиотека

- Разметка тестов
- Проверка различных условий

Test Runner (специальное приложение)

- Выполнение тестов
- Создание отчетов о выполненных тестах

# Примеры тестов с использованием MS Unit Testing Framework

- Библиотека Microsoft.VisualStudio.QualityTools.UnitTestFramework
- Разметка тестов с помощью атрибутов TestClass и TestMethod
- Проверка условий выполняется с помощью методов статического класса Assert


17

# Практика написания модульных тестов. Шаблон Arrange-Act-Assert (AAA)

Шаблон для написания тестов - «Triple A» (Arrange-Act-Assert)

- Arrange (Устанавить) - осуществить настройку входных данных для теста;
- Act (Выполнить) - выполнить действие, результаты которого тестируются;
- Assert (Проверить) - проверить результаты выполнения

# Практика написания модульных тестов. Шаблон Arrange-Act-

```
[TestClass]
public class ProgramTest
{
    [TestMethod]
    public void Sum_2Plus3_3Returned()
    {
   // Arrange
        var target = new ArithmeticUnit();
        target.OperandA = 2;
        target.OperandB = 3;
   //Act
        target.Add();
   //Assert
        Assert.AreEqual(5, target.Result);
     }
}
```

# Тестовое покрытие

| System.Int32 | System.String |
|---|---|
| положительное число | null |
| Отрицательное число | Пустая строка, String.Empty или "" |
| ноль | Один или более пробелов |
| int.MaxValue или 2,147,483,647 | Один или более символов табуляции |
| int.MinValue или -2,147,483,648 | Новая строка или Environment.NewLine |
| | Допустимая строка |
| | неверная строка |
| | символы Unicode, например, китайский язык |

# Row tests или параметризированные тесты

NUnit testing framework

```
[TestCase(12,3,4)]
[TestCase(12,2,6)]
[TestCase(12,4,3)]
public void DivideTest(int n, int d, int q)
{
    Assert.AreEqual( q, n / d );
}

[TestCase(12,3, Result=4)]
[TestCase(12,2, Result=6)]
[TestCase(12,4, Result=3)]
public int DivideTest(int n, int d)
{
  return( n / d );
}
```

# Row tests или параметризированные тесты

NUnit testing framework

```
[Test, TestCaseSource("DivideCases")]
public void DivideTest(int n, int d, int q)
{
    Assert.AreEqual( q, n / d );
}

static object[] DivideCases =
{
    new object[] { 12, 3, 4 },
    new object[] { 12, 2, 6 },
    new object[] { 12, 4, 3 }
};
```

# Практика написания модульных тестов

*Когда пишешь код, думай о тесте.*
*Когда пишешь тест, думай о коде.*
*Когда ты думаешь о коде и тесте как о едином, тестирование просто, а код красив.*

- **Unit-тесты автоматизированы**
- **Unit-тесты пишутся на том же языке, что и тестируемый код**
- **Unit-тесты - простые**
- **Unit-тесты - быстрые**
- **Unit-тесты - независимые**
- **Unit-тесты - надежные**
- **Unit-тесты - точные**

# Практика написания модульных тестов. Борьба с зависимостями

```
public class AccountManagementController :
                                    BaseAdministrationController
{
    private readonly IOrderManager orderManager;
    private readonly IAccountData accountData;
    private readonly IUserManager userManager;
    private readonly FilterParam disabledAccountsFilter;

    public AccountManagementController()
    {
        oms = OrderManagerFactory.GetOrderManager();
        accountData = orderManager.GetComponent<IAccountData>();
        userManager = UserManagerFactory.Get();
        disabledAccountsFilter =
   new FilterParam("Enabled", Expression.Eq, true);
    }
}
```

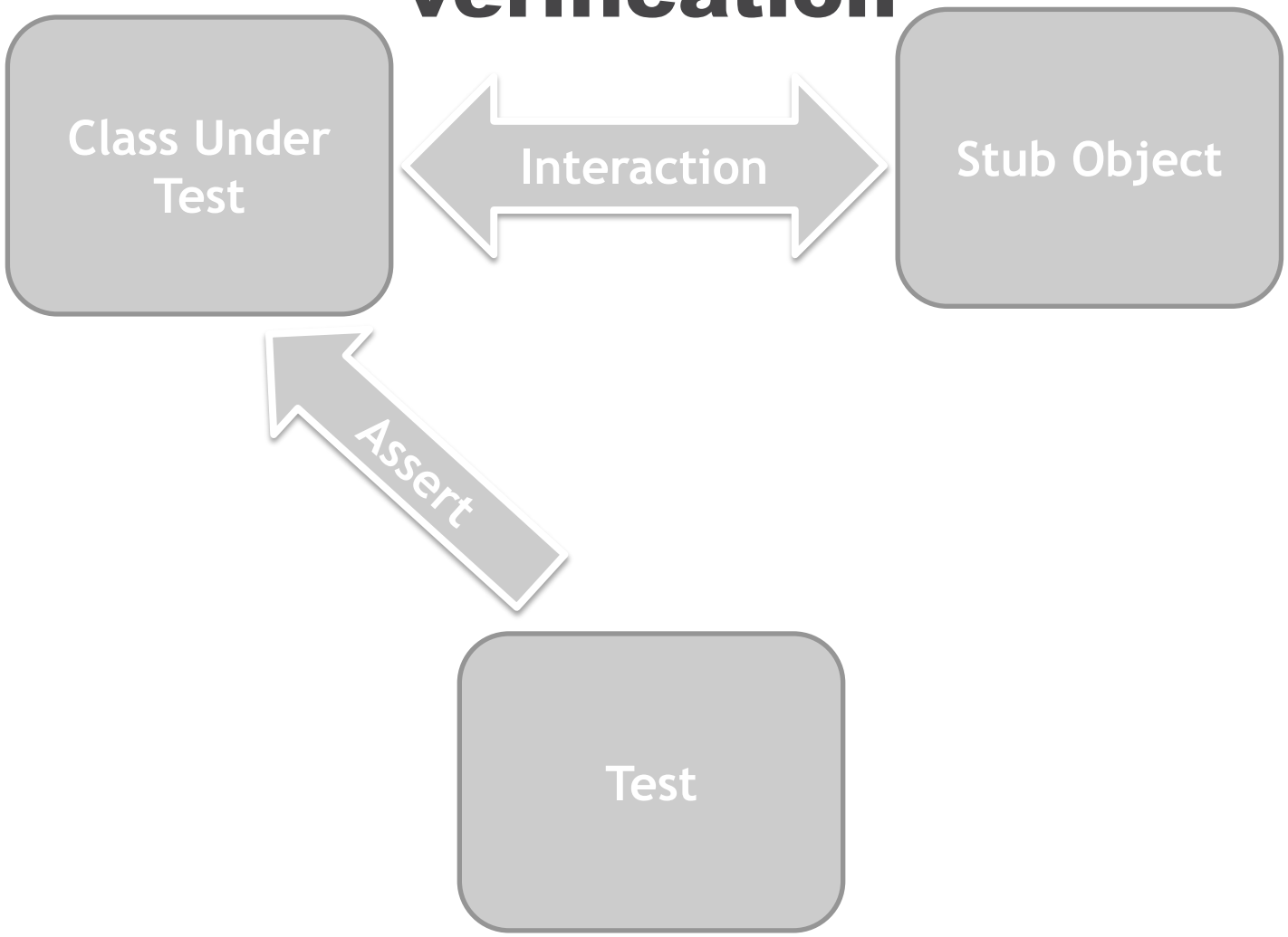
# Понятие о Test Doubles

- **Dummy** – пустые объекты (new object(), null, «Ignored String» и т.д.), которые передаются в вызываемые внутренние методы, но не используются. Предназначены лишь для заполнения параметров методов.
- **Fake** – объекты, имеющие работающие реализации, но в таком виде, который делает их неподходящими для production-кода (например, «In Memory Database»).
- **Stub** – объекты, которые предоставляют заранее заготовленные ответы на вызовы во время выполнения теста и обычно не отвечающие ни на какие другие вызовы, которые не требуются в тесте. Также могут запоминать какую-то дополнительную информацию о количестве вызовов, параметрах и возвращать их потом тесту для проверки.
- **Spy** – используется для тестов взаимодействия, основной функцией является запись данных и вызовов, поступающих из тестируемого объекта для последующей проверки корректности вызова зависимого объекта. Позволяет проверить логику именно нашего тестируемого объекта, без проверок зависимых объектов.
- **Mock** – объекты, которые заменяют реальный объект в условиях теста и позволяют проверять вызовы своих членов как часть системы или unit-теста. Содержат заранее запрограммированные ожидания вызовов, которые они ожидают получить. Применяются в основном для т.н. interaction (behavioral) testing.

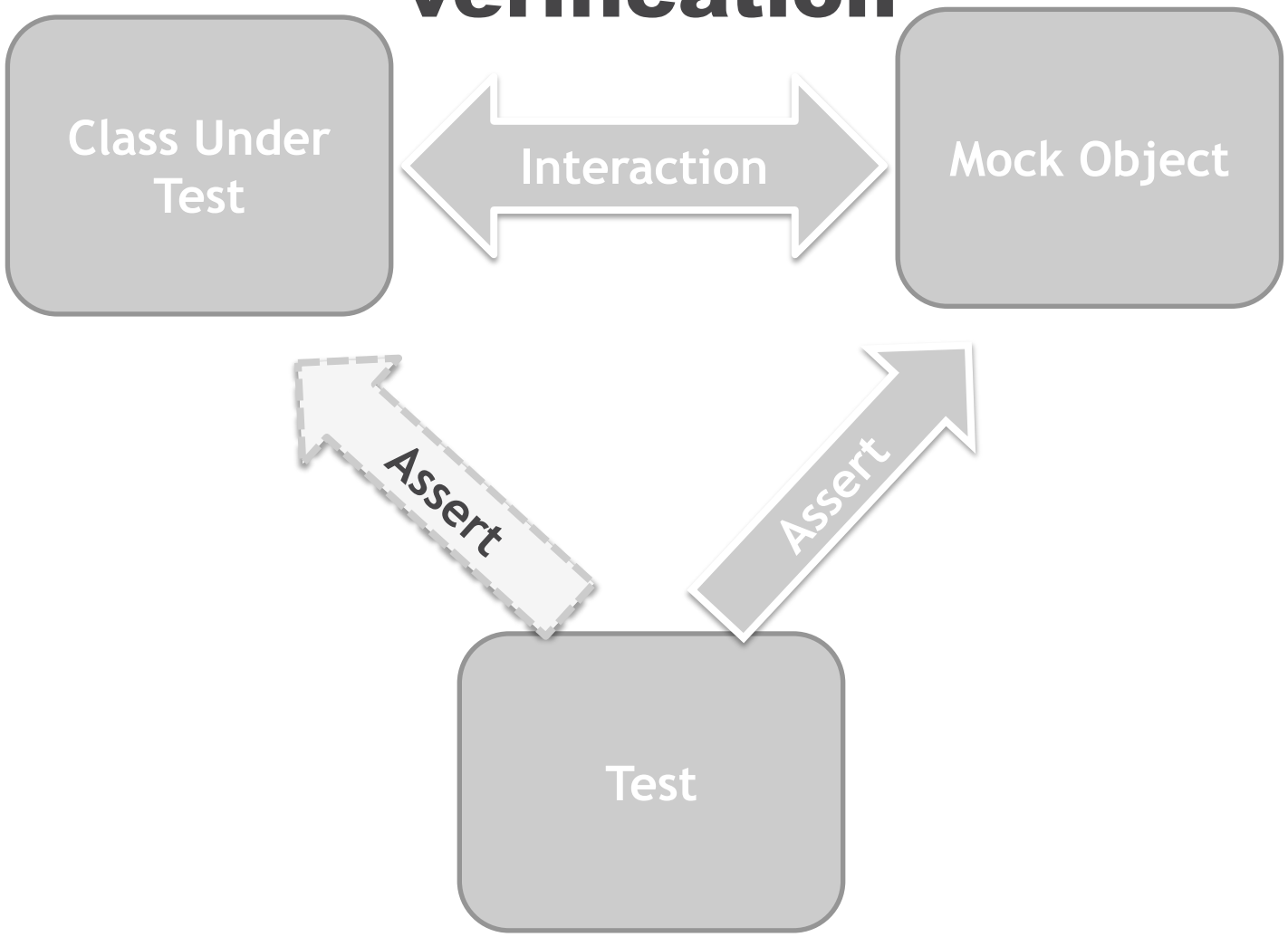
**Gerard Meszaros (Джерард Месарош)** "xUnit test patterns: refactoring test code"

**Roy Osherove (Рой Ошеров)** "The Art of Unit Testing"

# Понятие о Test Doubles. State verification



# Понятие о Test Doubles. State verification



# Понятие о mock-объектах. Dummy-объекты.

Если нужно протестировать метод Foo() класса TestFoo, который делает вызов другого метода Bar() класса TestBar. Предположим, что метод Bar() принимает какой-нибудь объект класса Bla в качестве параметра и потом ничего особого с ним не делает. В таком случае имеет смысл создать пустой объект Bla, передать его в класс TestFoo (например, при помощи широко применяемого паттерна Dependency), а затем уже Foo() при тестировании сам вызовет метод Bar() класса TestBar с переданным пустым объектом.

http://habrahabr.ru/post/150859/
http://sergeyteplyakov.blogspot.com.by/2011/12/blog-post.html
http://sergeyteplyakov.blogspot.com.by/2014/01/microsoft-fakes-state-verification.html

# Понятие о mock-объектах. Dummy-объекты.

```
private class FooDummy : IFoo
{
    public string bar() { return null; }
}
[TestFixture]
public class FooCollectionTest
{
    [Test]
    public void It_Should_Maintain_2_objects()
    {
        FooCollection sut = new FooCollection();
   sut.Add(new FooDummy());
   sut.Add(new FooDummy());
   Assert.AreEqual(2, sut.Count);
    }
}
```

# Понятие о mock-объектах. Stub-объекты.

Stub-объекты (стабы) – это типичные заглушки. Они ничего полезного не делают и умеют лишь возвращать определенные данные в ответ на вызовы своих методов. В нашем примере стаб подменяет класс TestBar и в ответ на вызов Bar() просто бы возвращает какие-то (левые) данные. При этом внутренняя реализация реального метода Bar() просто не вызывается. Реализуется этот подход через интерфейс и создание дополнительного класса StubBar, либо просто через создание StubBar, который является унаследованным от TestBar. В принципе, реализация очень похожа на fake-объект с тем лишь исключением, что стаб ничего полезного, кроме постоянного возвращения каких-то константных данных не требует. Типичная заглушка. Стабам позволяется лишь сохранять у себя внутри какие-нибудь данные, удостоверяющие, что вызовы были произведены или содержащие копии переданных параметров, которые затем может проверить тест.

# Понятие о mock-объектах. Dummy-объекты.

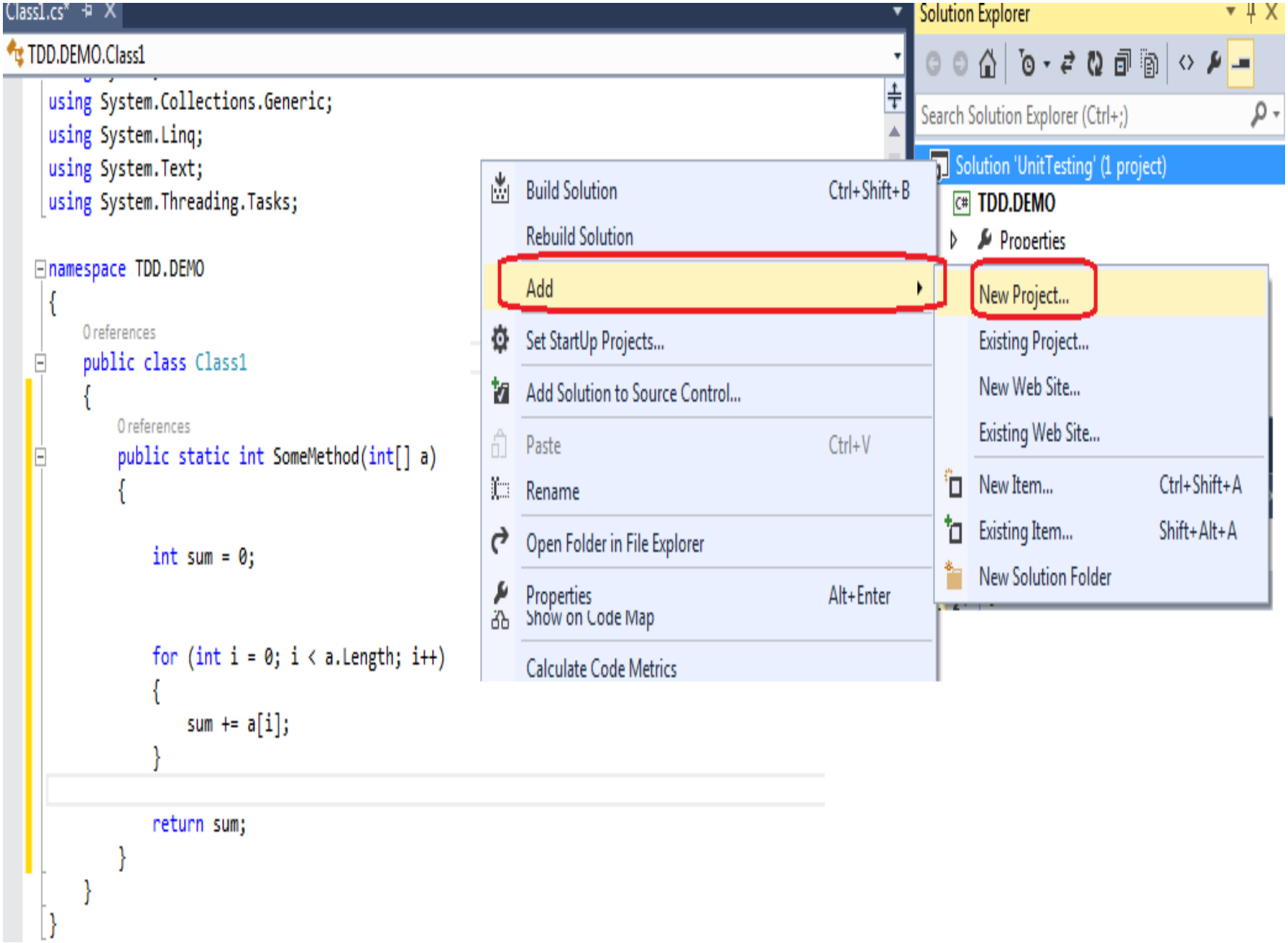
```
private class FooStub : IFoo
{
    public string Bar() {return "test"; }
}
[TestFixture]
public class FooCollectionTest
{
    [Test]
    public void It_Should_Maintain_2_objects()
    {
        FooCollection sut = new FooCollection();
   sut.add(new FooStub());
   sut.add(new FooStub());
   Assert.Equals("testtest", sut.Joined());
    }
}
```

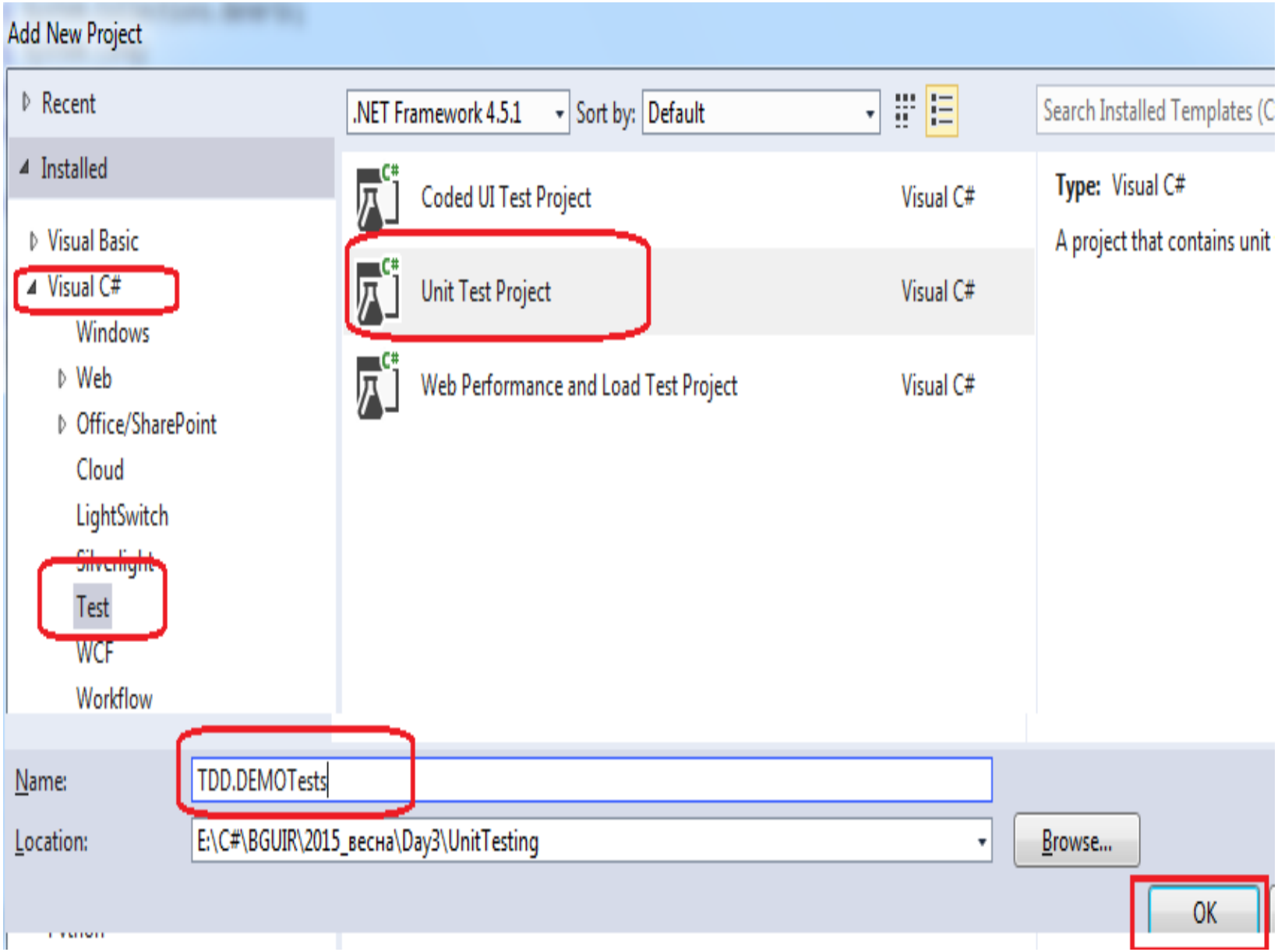
# Понятие о mock-объектах. Fake-объекты.

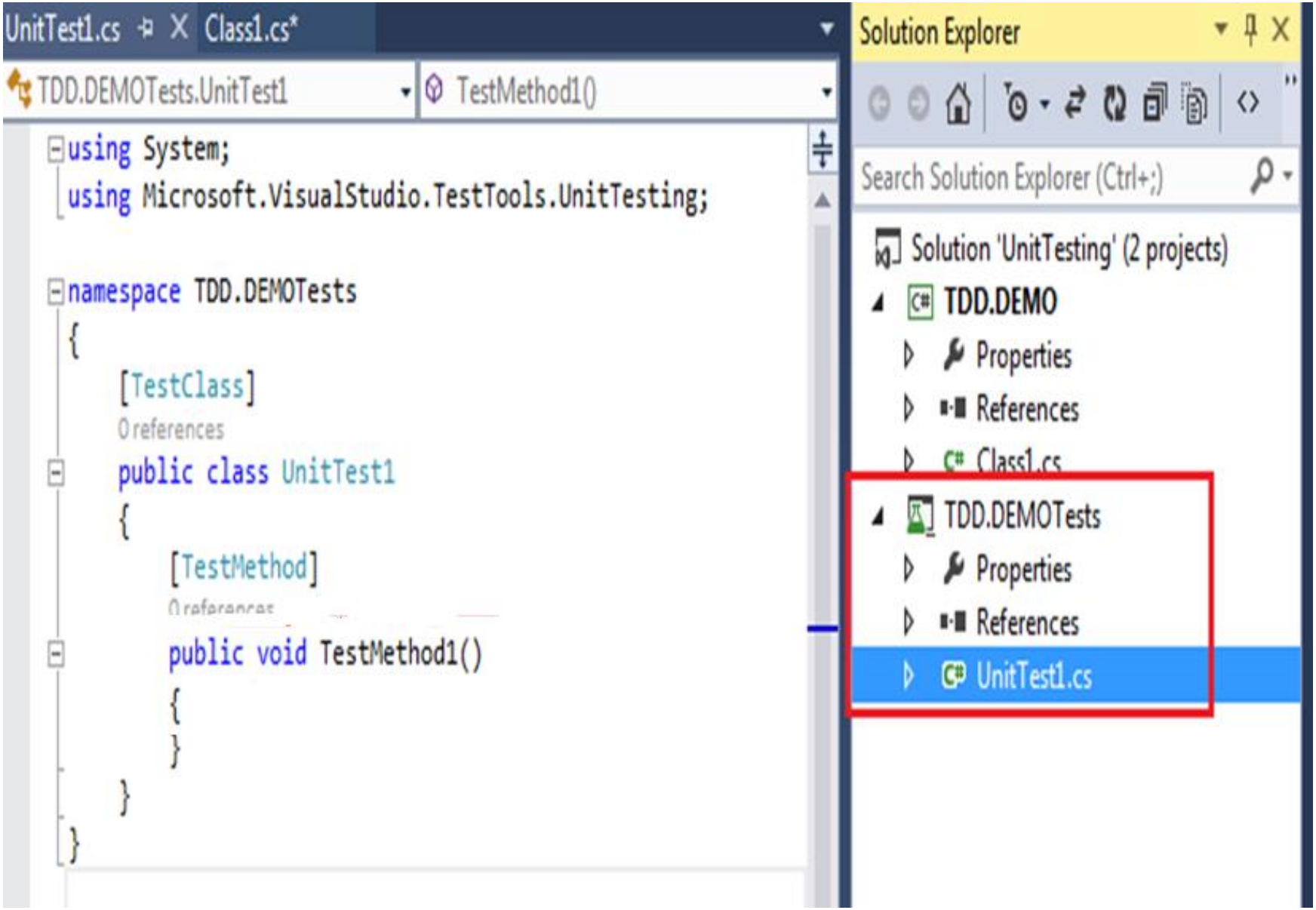
Иногда метод Bar() выполняет какие-то действия с объектом Bla ( например, сохраняет данные в базу или вызывает веб-сервис). В таких случаях объект класса TestBar должен быть dummy-объектом. Нужно научить его в ответ на запрос сохранения данных просто выполнить какой-то простой код (допустим, сохранение во внутреннюю коллекцию). В таких случаях можно выделить интерфейс ITestBar, который будет реализовывать класс TestBar и дополнительный класс FakeBar. При unit-тестировании мы просто будем создавать объект класса FakeBar и передавать его в класс с методом Foo() через интерфейс. Естественно, при этом класс Bar будет по-прежнему создаваться в реальном приложении, а FakeBar будет использован лишь в тестировании. Это иллюстрация fake-объекта.

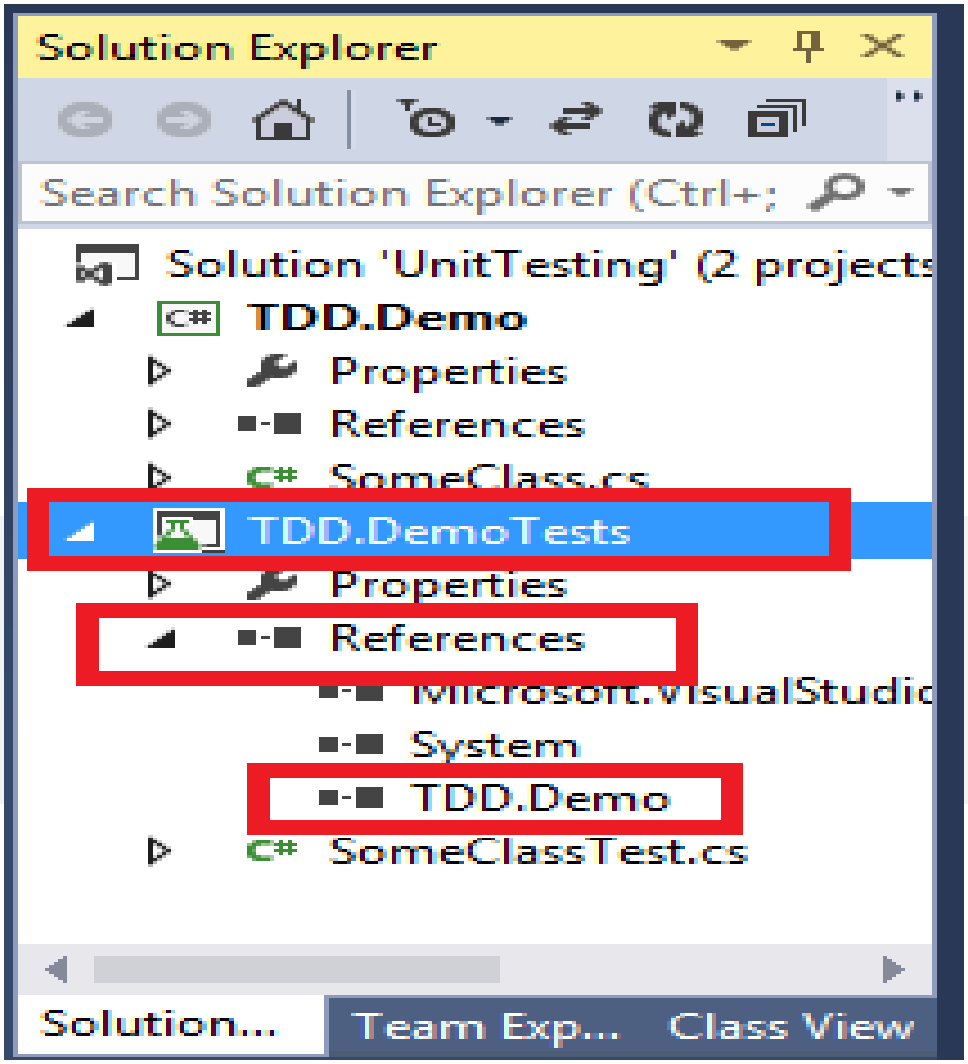
# Понятие о mock-объектах. Moke-объекты.

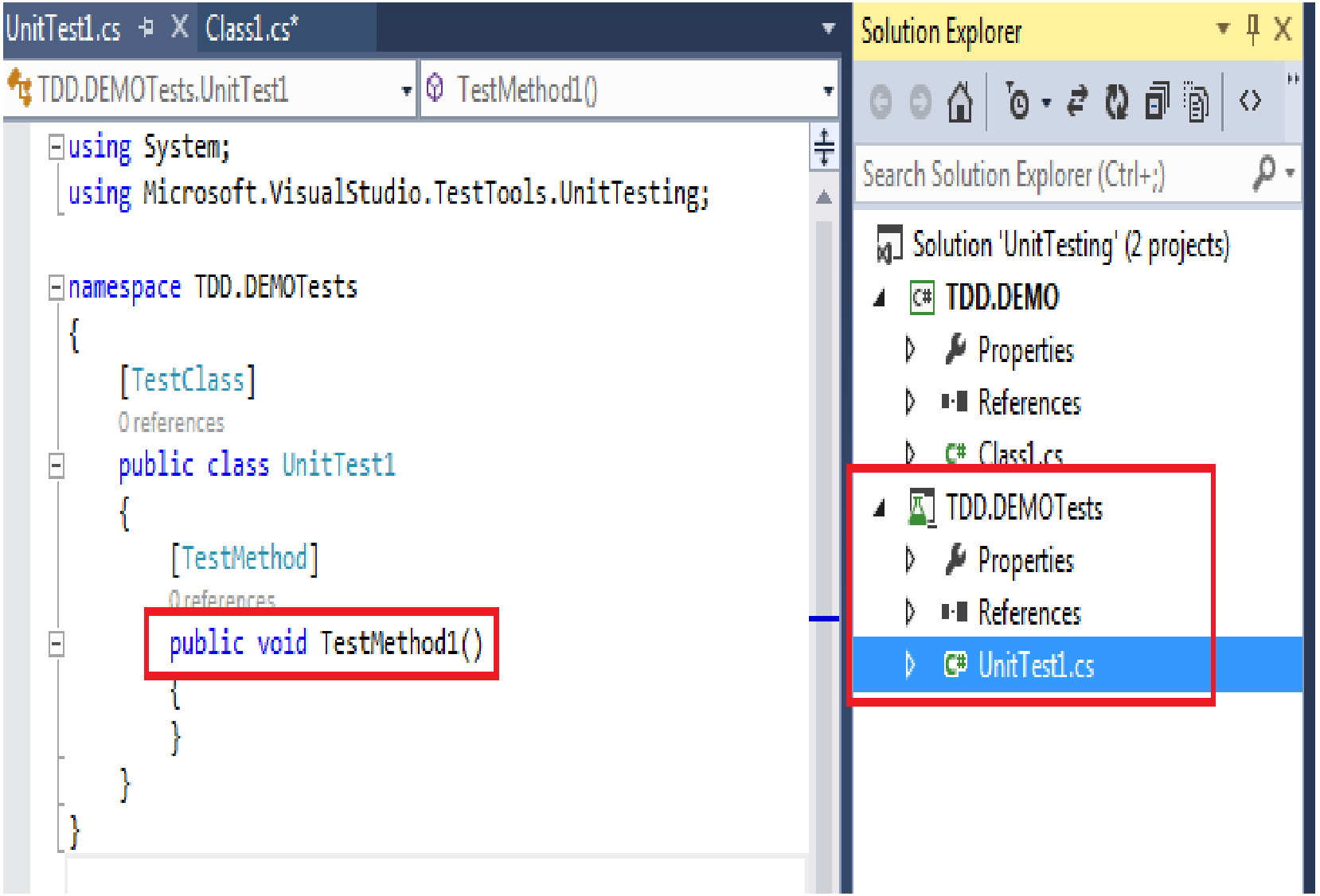
Иногда метод Bar() выполняет какие-то действия с объектом Bla ( например, сохраняет данные в базу или вызывает веб-сервис). В таких случаях объект класса TestBar должен быть dummy-объектом. Нужно научить его в ответ на запрос сохранения данных просто выполнить какой-то простой код (допустим, сохранение во внутреннюю коллекцию). В таких случаях можно выделить интерфейс ITestBar, который будет реализовывать класс TestBar и дополнительный класс FakeBar. При unit-тестировании мы просто будем создавать объект класса FakeBar и передавать его в класс с методом Foo() через интерфейс. Естественно, при этом класс Bar будет по-прежнему создаваться в реальном приложении, а FakeBar будет использован лишь в тестировании. Это иллюстрация fake-объекта.


Class1.cs*
TDD.DEMO.Class1
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace TDD.DEMO
{
0 references
public class Class1
{
0 references
public static int SomeMethod(int[] a)
{
int sum = 0;
for (int i = 0; i < a.Length; i++)
{
sum += a[i];
}
return sum;
}
}
}
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UnitTesting' (1 project)
TDD.DEMO
Properties
Build Solution Ctrl+Shift+B
Rebuild Solution
Add
Set StartUp Projects...
Add Solution to Source Control...
Paste Ctrl+V
Rename
Open Folder in File Explorer
Properties Alt+Enter
Show on Code Map
Calculate Code Metrics
New Project...
Existing Project...
New Web Site...
Existing Web Site...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Solution Folder


Add New Project
Recent
Installed
Visual Basic
Visual C#
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Test
WCF
Workflow
.NET Framework 4.5.1
Sort by:
Default
Search Installed Templates (C
Coded UI Test Project
Visual C#
Unit Test Project
Visual C#
Web Performance and Load Test Project
Visual C#
Type: Visual C#
A project that contains unit
Name:
TDD.DEMOTests
Location:
E:\C#\BGUIR\2015_весна\Day3\UnitTesting
Browse...
OK


UnitTest1.cs
Class1.cs*
TDD.DEMOTests.UnitTest1
TestMethod1()
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace TDD.DEMOTests
{
[TestClass]
0 references
public class UnitTest1
{
[TestMethod]
0 references
public void TestMethod1()
{
}
}
}
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UnitTesting' (2 projects)
TDD.DEMO
Properties
References
Class1.cs
TDD.DEMOTests
Properties
References
UnitTest1.cs


Solution Explorer
Search Solution Explorer (Ctrl+;
Solution 'UnitTesting' (2 projects
TDD.Demo
Properties
References
SomeClass.cs
TDD.DemoTests
Properties
References
Microsoft.VisualStudio
System
TDD.Demo
SomeClassTest.cs
Solution...
Team Exp...
Class View


UnitTest1.cs
Class1.cs*
TDD.DEMOTests.UnitTest1
TestMethod1()
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace TDD.DEMOTests
{
[TestClass]
0 references
public class UnitTest1
{
[TestMethod]
0 references
public void TestMethod1()
{
}
}
}
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UnitTesting' (2 projects)
TDD.DEMO
Properties
References
Class1.cs
TDD.DEMOTests
Properties
References
UnitTest1.cs

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDD.Demo;

namespace TDD.DemoTests
{
    [TestClass]
    ссылок 0
    public class SomeClassTest
    {
        [TestMethod]
        | ссылок 0
        public void SameMethod_All_Positive_Numbers()
        {


        }

    }
}
```

```
namespace TDD.DemoTests
{
    [TestClass]
    ссылок 0
    public class SomeClassTest
    {
        [TestMethod]
        | ссылок 0
        public void SameMethod_All_Positive_Numbers()
        {
            var a = new int[] { 1, 2, 3, 4, 5 };

            int expected = 15;

            int actual = SomeClass.SomeMethod(a);

            Assert.AreEqual(expected, actual);
```

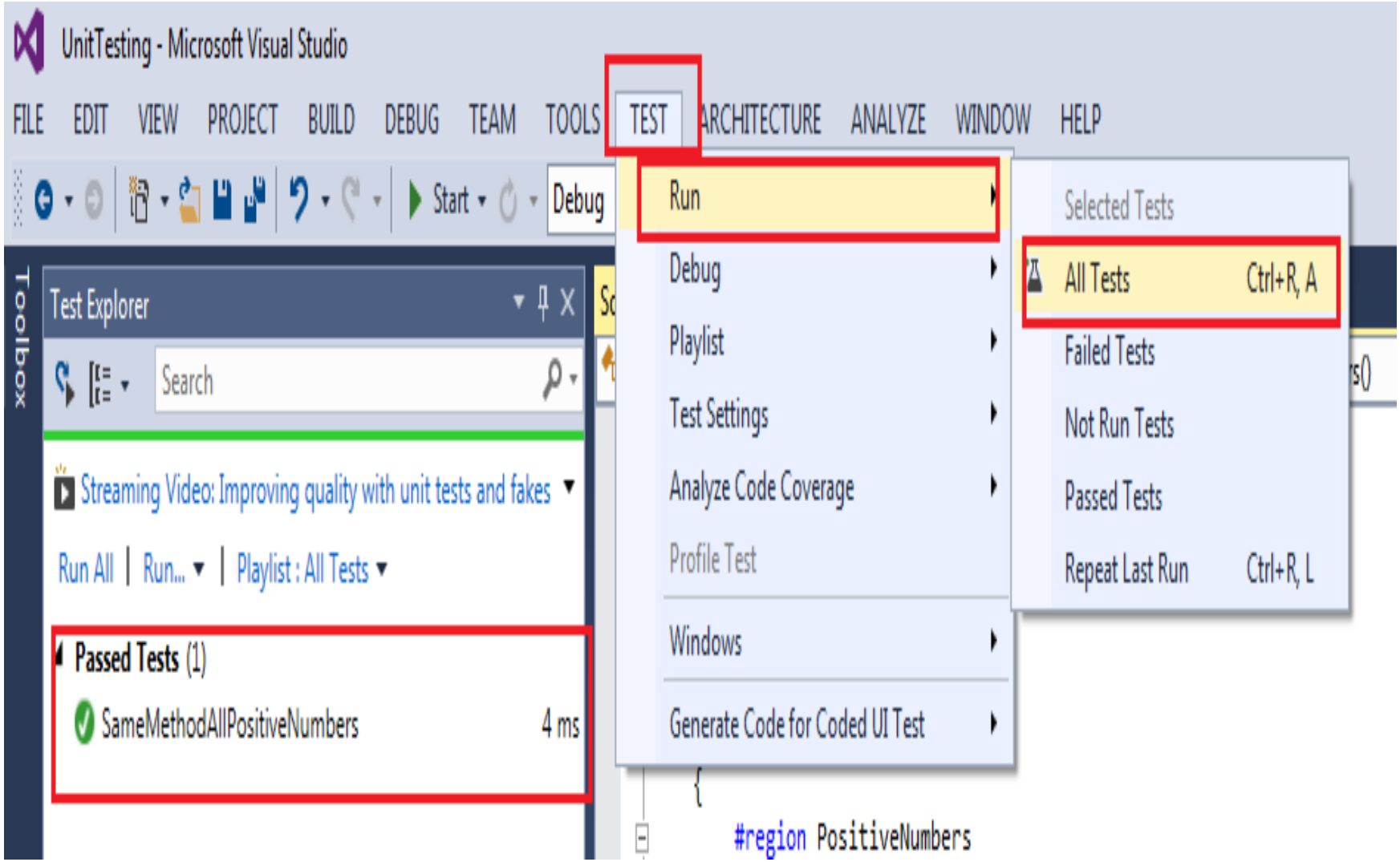
```
Assert.AreEqual(expected, actual);
```

void Assert.AreEqual<int>(int expected, int actual) (+ 17 overload(s))

Verifies that two specified generic type data are equal by using the equality operator. The assertion fails if they are not equal.

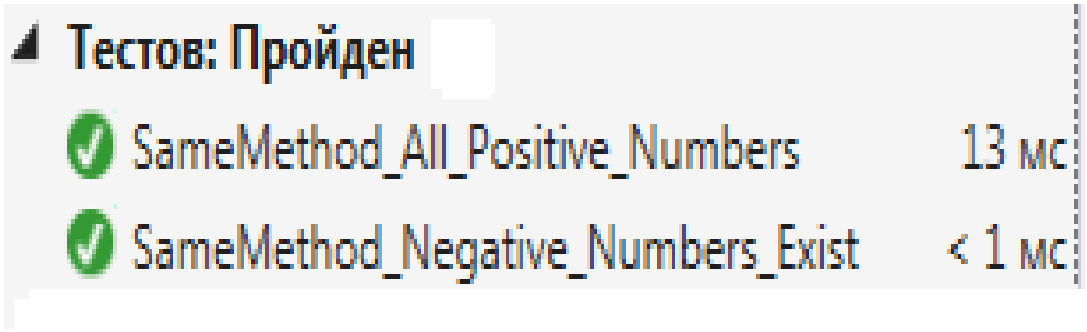
Exceptions:

Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException


UnitTesting - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start
Debug
Toolbox
Test Explorer
Search
Streaming Video: Improving quality with unit tests and fakes
Run All | Run... | Playlist : All Tests
Passed Tests (1)
SameMethodAllPositiveNumbers 4 ms
Run
Debug
Playlist
Test Settings
Analyze Code Coverage
Profile Test
Windows
Generate Code for Coded UI Test
Selected Tests
All Tests Ctrl+R, A
Failed Tests
Not Run Tests
Passed Tests
Repeat Last Run Ctrl+R, L
{
#region PositiveNumbers

```
[TestMethod]
ссылок 0
public void SameMethod_Negative_Numbers_Exist()
{
    var a = new int[] { 1, 2, 3, 4, 5, -15 };

    int expected = 0;

    int actual = SomeClass.SomeMethod(a);

    Assert.AreEqual(expected, actual);
}
```

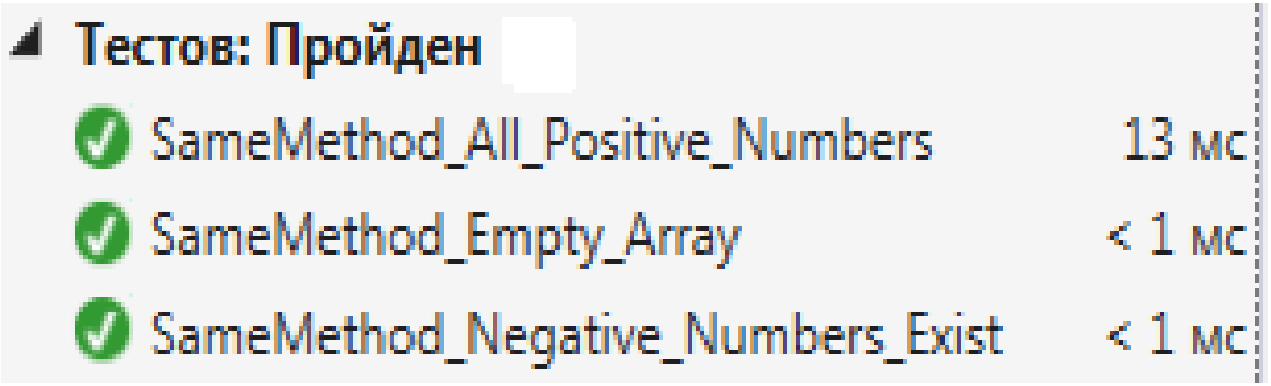
```
[TestMethod]
ссылок 0
public void SameMethod_Negative_Numbers_Exist()
{
    var a = new int[] { 1, 2, 3, 4, 5, -15 };

    int expected = 0;

    int actual = SomeClass.SomeMethod(a);

    Assert.AreEqual(expected, actual);
}
```



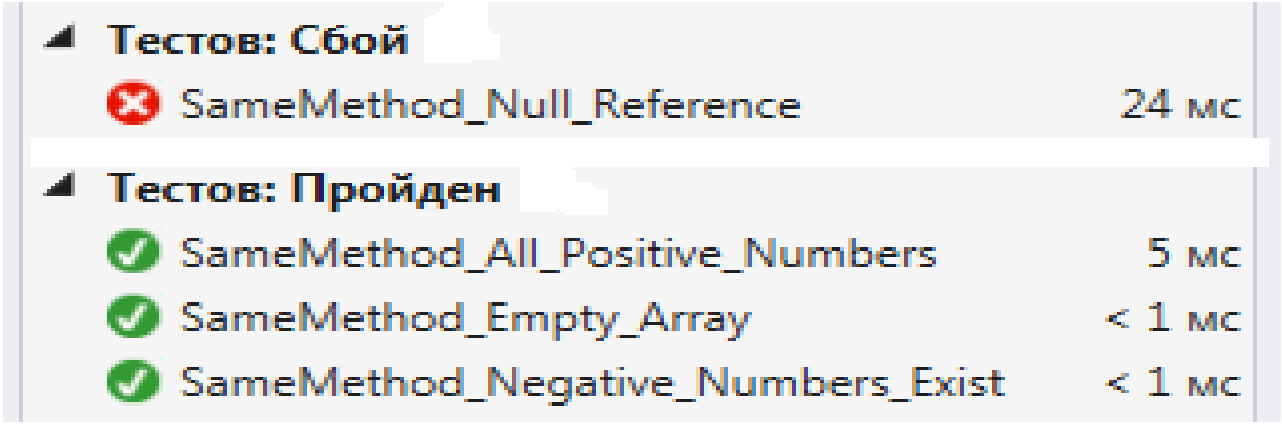


45

```
[TestMethod]
0 ссылок
public void SameMethod_Empty_Array()
{
    var a = new int[] { };

    int expected = 0;

    int actual = SomeClass.SomeMethod(a);

    Assert.AreEqual(expected, actual);
}
```

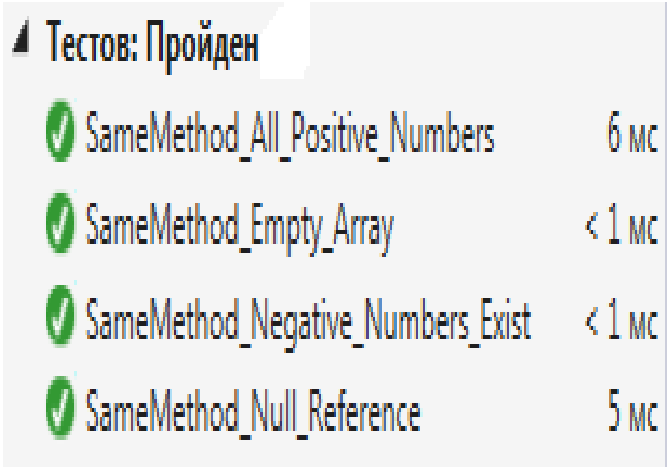


```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
ссылок 0
public void SameMethod_Null_Reference()
{
    int[] a = null;

    int expected = 0;

    int actual = SomeClass.SomeMethod(a);

    Assert.AreEqual(expected, actual);
}
```



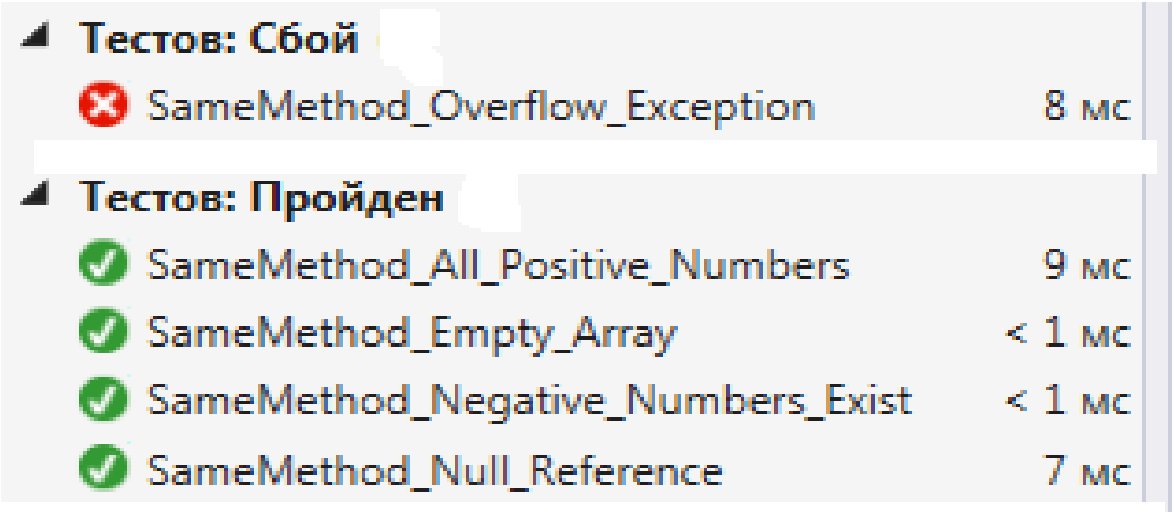


47




```
namespace TDD.Demo
{
    ссылок 7
    public class SomeClass
    {
        ссылок 7 | 5/5 пройдены
        public static int SomeMethod(int[] a)
        {
            if (a == null) throw new ArgumentNullException("a");

            int sum = 0;

            for (int i = 0; i < a.Length; i++)
            {
                sum += a[i];
            }

            return sum;
        }
    }
}
```

```
[TestMethod]
[ExpectedException(typeof(OverflowException))]
ссылок 0
public void SameMethod_Overflow_Exception()
{
    int[] a = { 1, int.MaxValue };
    int actual = SomeClass.SomeMethod(a);
}
```






49


```
namespace TDD.Demo
{
    ссылок 7
    public class SomeClass
    {
        ссылок 7 | 5/5 пройдены
        public static int SomeMethod(int[] a)
        {
            if (a == null) throw new ArgumentNullException("a");

            int sum = 0;

            checked
            {
                for (int i = 0; i < a.Length; i++)
                {
                    sum += a[i];
                }

            }
            return sum;
        }
    }
}
```

## Практика unit-тестирования

- Unit-тесты автоматизированы
- Unit-тесты пишутся на том же языке, что и тестируемый код
- Unit-тесты – простые
- Unit-тесты – быстрые
- Unit-тесты – независимые
- Unit-тесты – надежные
- Unit-тесты – точные

## Простота

- ✓ Говорящее название
- ✓ 1 Assert
- ✓ Макс 10 строк
- ✓ Никакой логики
  - Циклы
  - Многопоточность
  - Условные операторы

  ...

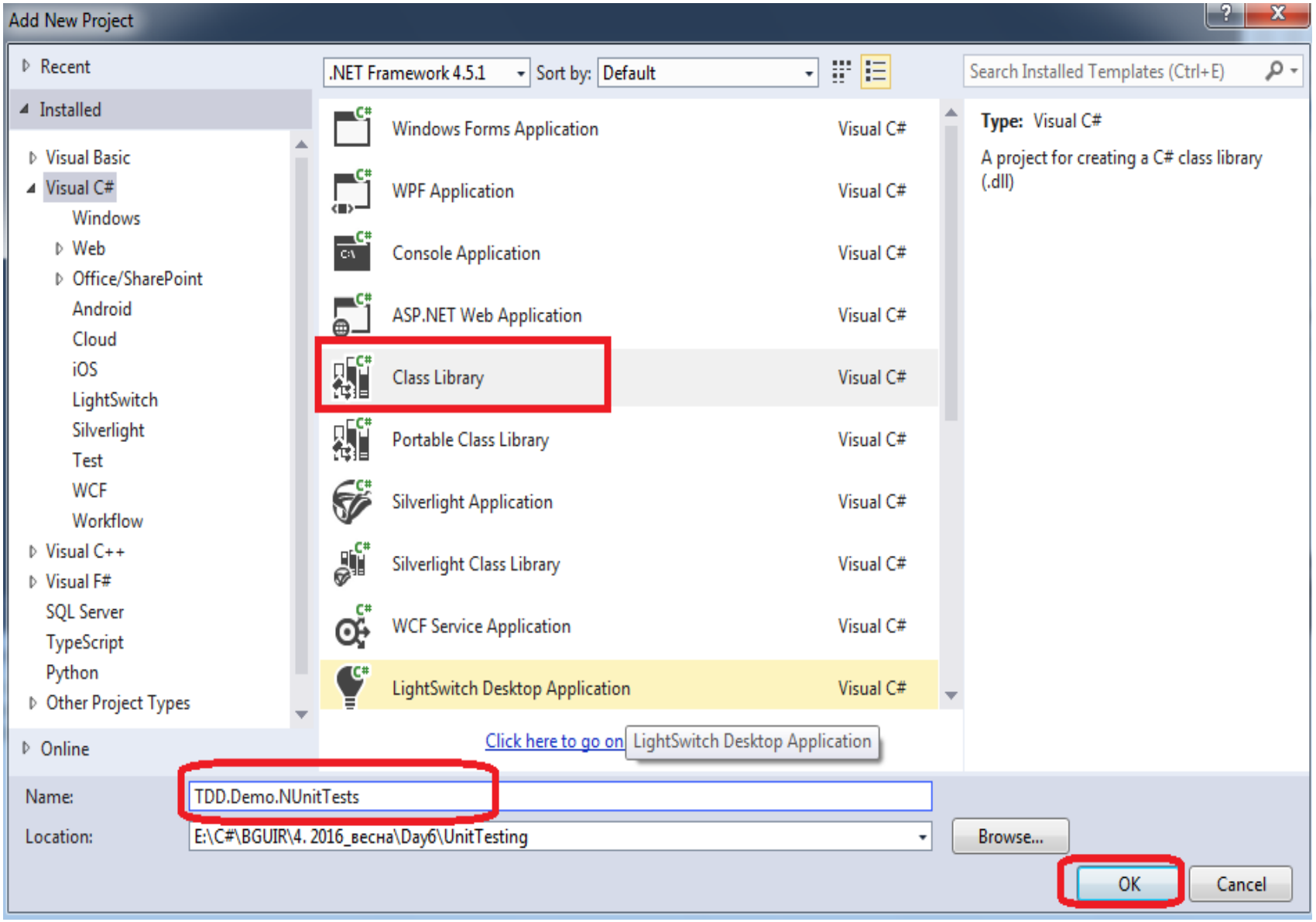
## Простота

- ✓ 10 секунд на весь набор тестов

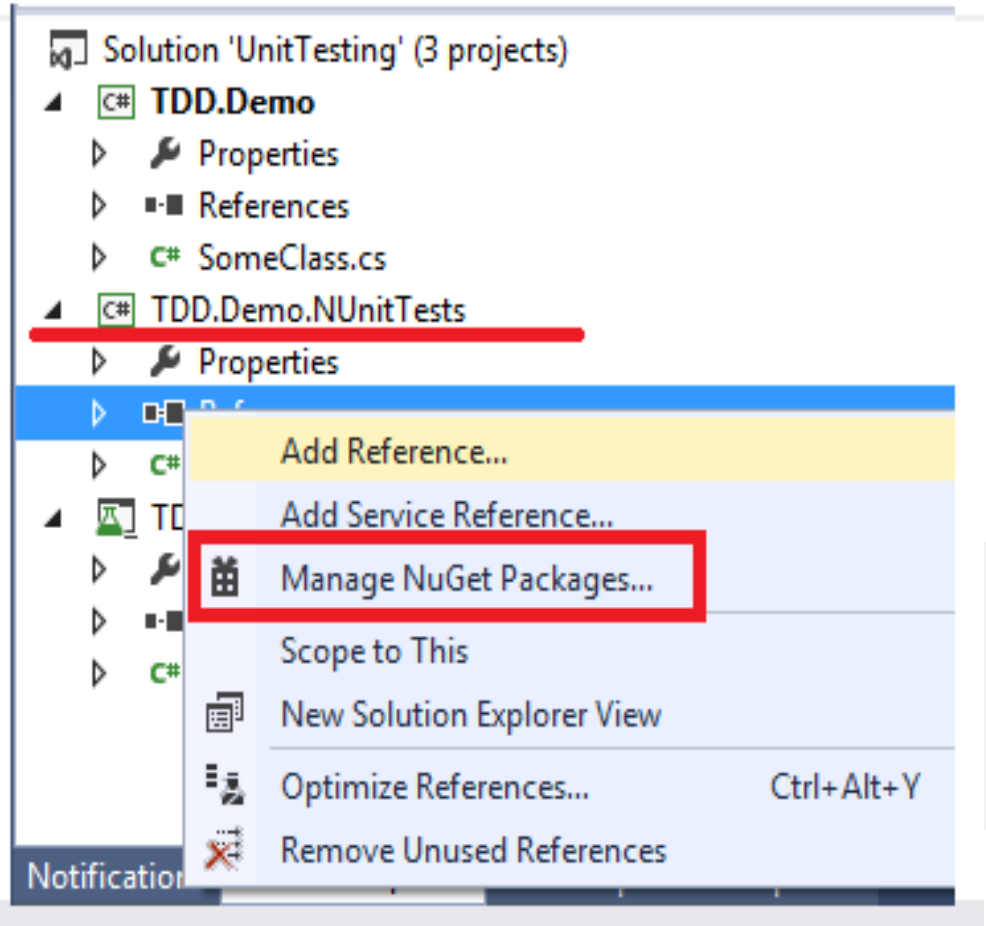
## Надежность

- ✓ Тестируется только код
- ✓ Падают только при ошибках в коде
- ✓ Выполняются в любом окружении
- ✓ Все тесты изолированы

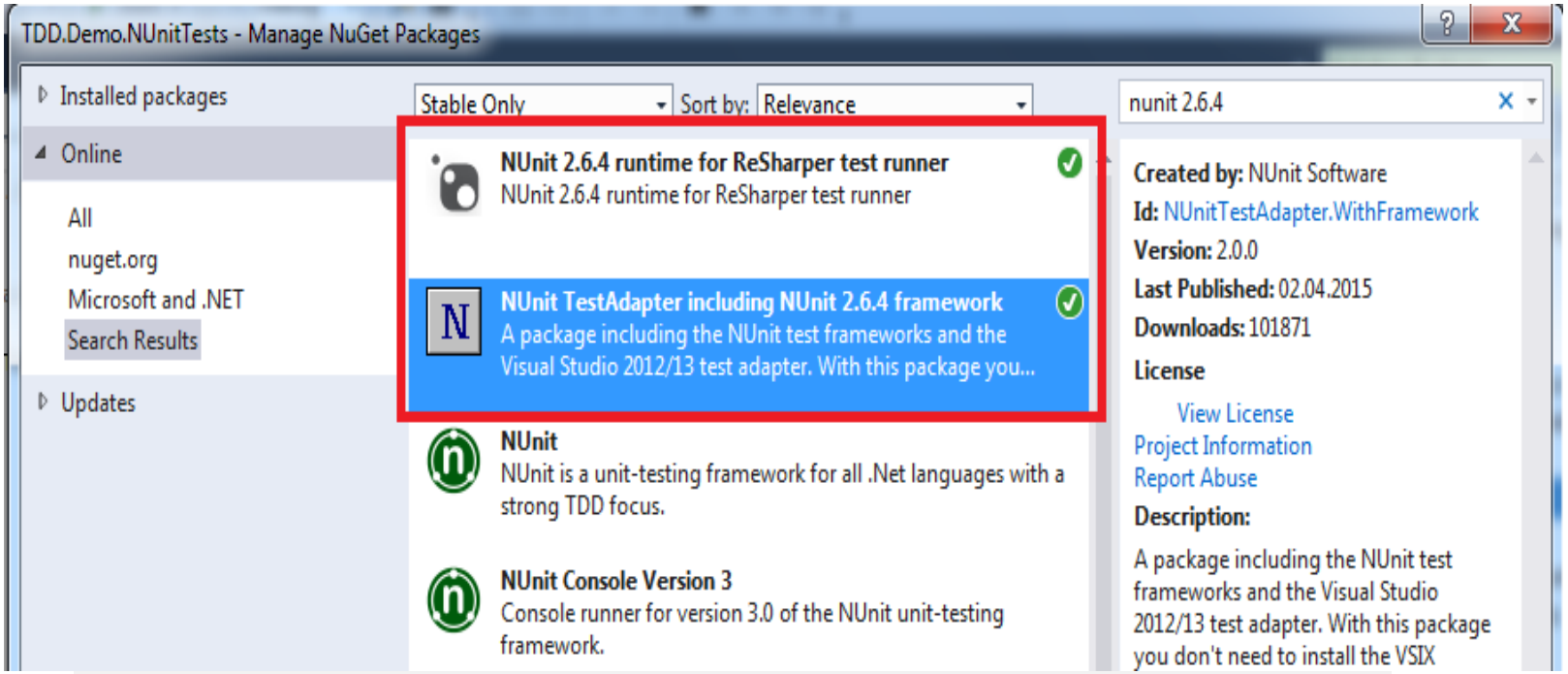
# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ



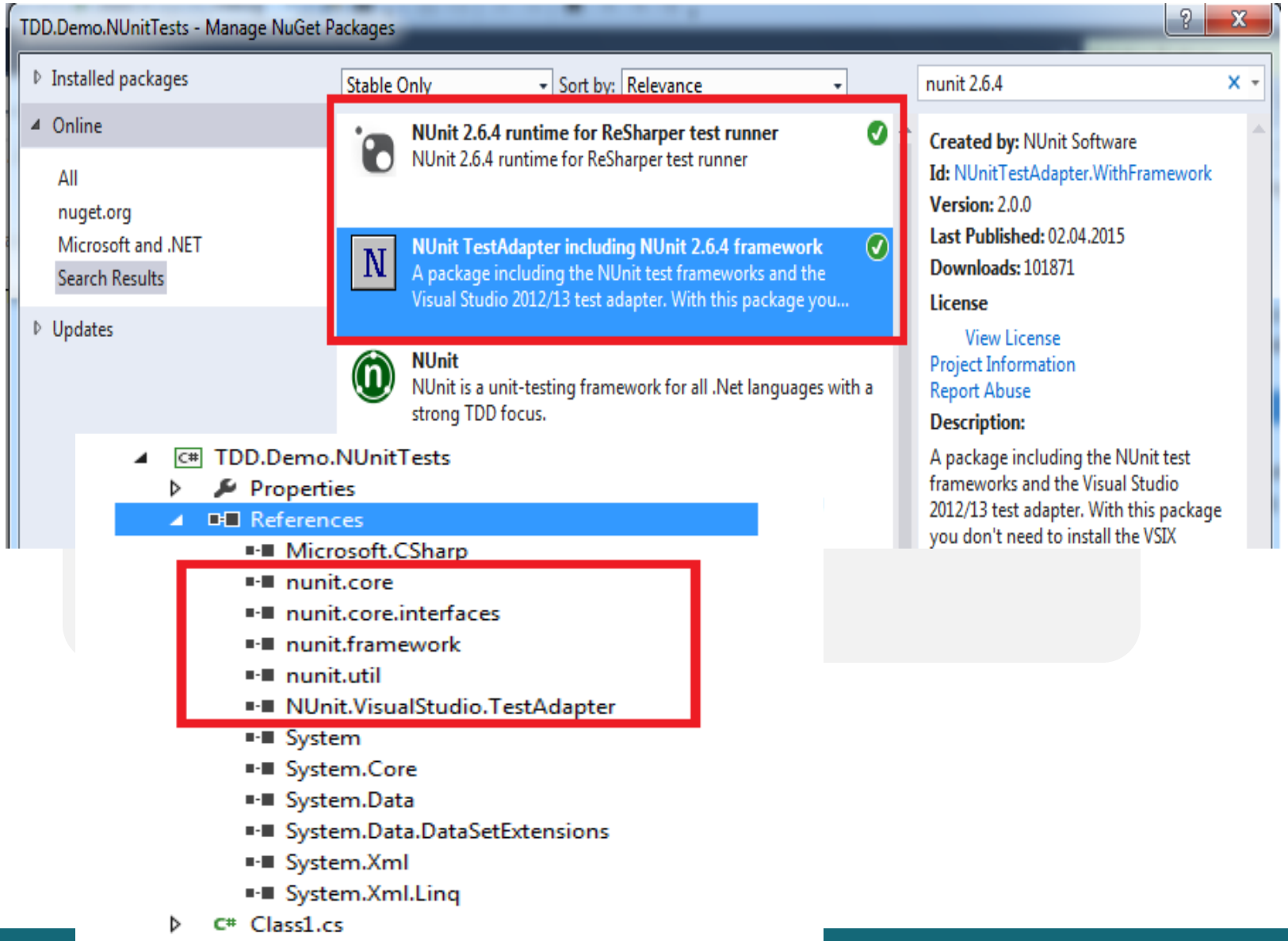
# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ



# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ



# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ



# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

```
using System.Text;
using System.Threading.Tasks;
using NUnit.Framework;
using TDD.Demo;
namespace TDD.Demo.NUnitTests
{
    0 references
    public class SomeClassTest
    {
        [TestCase(new int[] { 1, 2, 3, 4, 5 }, Result = 15)]
        [TestCase(new int[] { 1, 2, 3, 4, 5, -15 }, Result = 0)]
        [TestCase(null, ExpectedException = typeof(ArgumentNullException))]
        [TestCase(new int[] {1, int.MaxValue}, ExpectedException = typeof(OverflowException))]
        0 references
        public static int SomeMethod_Test( int[] a)
        {
            return TDD.Demo.SomeClass.SomeMethod(a);
        }
```


59

# ПАРАМЕТРИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

```
public IEnumerable<TestCaseData> TestData
{
    get
    {
        yield return new TestCaseData(new int[] { 1, 2, 3, 4, 5 }).Returns(15);
        yield return new TestCaseData(new int[] { 1, 2, 3, 4, 5, -15} ).Returns(0);
        yield return new TestCaseData(null).Throws(typeof(ArgumentNullException));
        yield return new TestCaseData(new int[] { 1, int.MaxValue }).Throws(typeof(OverflowException));
    }
}
[Test, TestCaseSource("TestData")]
0 references
public static int SomeMethod_Test_Yeild(int[] a)
{
    return TDD.Demo.SomeClass.SomeMethod(a);
}
```

# СПАСИБО ЗА ВНИМАНИЕ!

# ВОПРОСЫ?

Объявление и вызов методов в C#

**Author:** **Саркисян Гаяне Феликсовна**
**gayane.f.sarkisyan@gmail.com**