

# **ЛЕКЦИЯ**

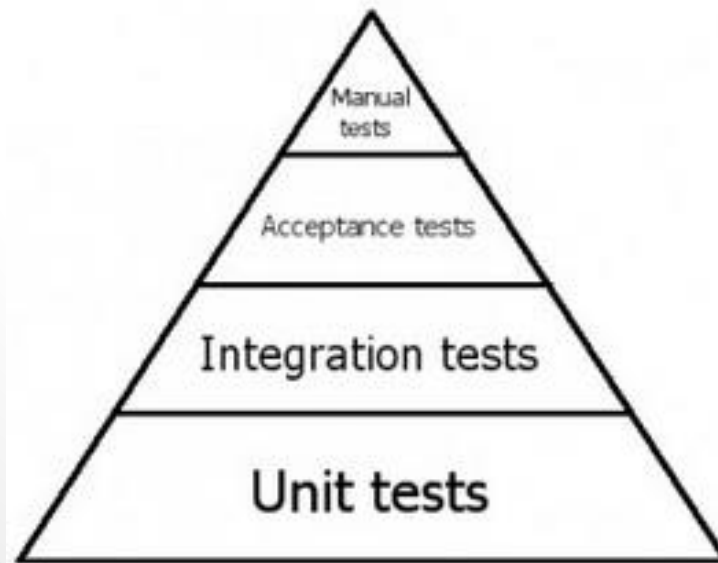
## **Модульное тестирование**

Лектор Крамар Ю.М.

# Модульное тестирование

Модульное тестирование или юнит-тестирование (unit testing) – процесс, 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.



# Не нужно писать тесты, если

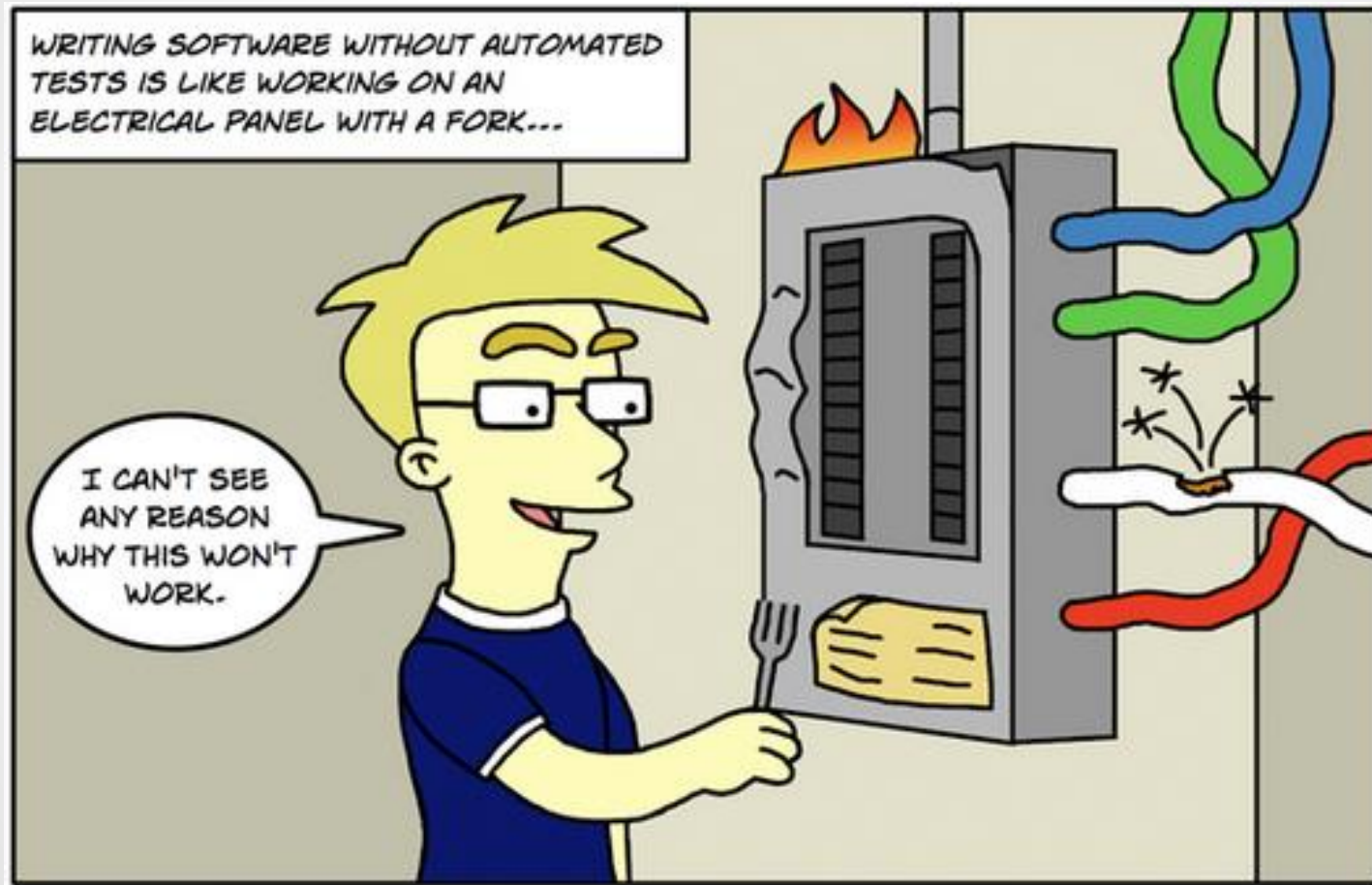
1. Вы делаете простой сайт-визитку из 5 статических html-страниц и с одной формой отправки письма. На этом заказчик, скорее всего, успокоится, ничего большего ему не нужно. Здесь нет никакой особенной логики, быстрее просто все проверить «руками»
2. Вы занимаетесь рекламным сайтом/простыми флеш-играми или баннерами – сложная верстка/анимация или большой объем статики. Никакой логики нет, только представление
3. Вы делаете проект для выставки. Срок – от двух недель до месяца, ваша 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
4. Вы всегда пишете код без ошибок, обладаете идеальной памятью и даром предвидения. Ваш код настолько крут, что изменяет себя сам, вслед за требованиями клиента. Иногда код объясняет клиенту, что его требования не нужно реализовывать

# Не нужно писать тесты, если

Если на фото не Вы, то...(((



# Модульное тестирование





# Модульное тестирование



# Проекты старше 1 года

- Без покрытия тестами.
- С тестами, которые никто не запускает и не поддерживает
- С серьезным покрытием. Все тесты проходят.

Обычно так и происходит: спагетти-кодом и

У нас тесты в системе есть, но что они проверяют, и какой от них ожидается результат, неизвестно. Система уже лучше. Присутствует какая-никакая атомарность, есть понимание,

Тестов много, каждый из них – атомарный: один тест проверяет только одну вещь. Тест является спецификацией метода класса, контрактом: какие входные параметры ожидает этот метод, и что остальные компоненты системы ждут от него на выходе. Проект не зависит от людей. Разработчики могут приходить и уходить. Система надежно протестирована и сама рассказывает о себе путем тестов.

# Модульное тестирование

Для того, чтобы проект оставался в рамках проектов 3го типа, его юнит-тесты должны:

1. Быть достоверными
2. Не зависеть от окружения, на котором они выполняются
3. Легко поддерживаться
4. Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется)
5. Соблюдать единую конвенцию именования
6. Запускаться регулярно в автоматическом режиме



# Модульное тестирование

- логическое расположение тестов в VCS

Тесты должны быть частью контроля версий. В зависимости от типа решения, они могут быть организованы по-разному. Например, если приложение монолитное, разместить все тесты в папку Tests; если у много разных компонентов, хранить тесты в папке каждого компонента.

# Модульное тестирование

- **именование проектов с тестами, классов тестов, методов тестов**
  - Одна из лучших практик: добавить к каждому проекту его собственный тестовый проект.  
Например, есть части системы `<PROJECT_NAME>.Core`, `<PROJECT_NAME>.BI` и `<PROJECT_NAME>.Web`? Добавьте еще `<PROJECT_NAME>.Core.Tests`, `<PROJECT_NAME>.BI.Tests` и `<PROJECT_NAME>.Web.Tests`.
  - Есть класс `ProblemResolver`? Добавьте в тестовый проект `ProblemResolverTests`. Каждый тестирующий класс должен тестировать только одну сущность.
  - Названия методов отражают Что именно тестируется? Каковы входные параметры? Могут ли возникать ошибки и исключительные ситуации?  
Например, способ именования методов: `[Тестируемый метод]_[Сценарий]_[Ожидаемое поведение]`.  
Предположим, что у нас есть класс `Calculator`, а у него есть метод `Sum`, который должен складывать два числа.  
В этом случае наш тестирующий класс будет выглядеть так:

# Модульное тестирование

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

# Модульное тестирование

- Использование фреймворков

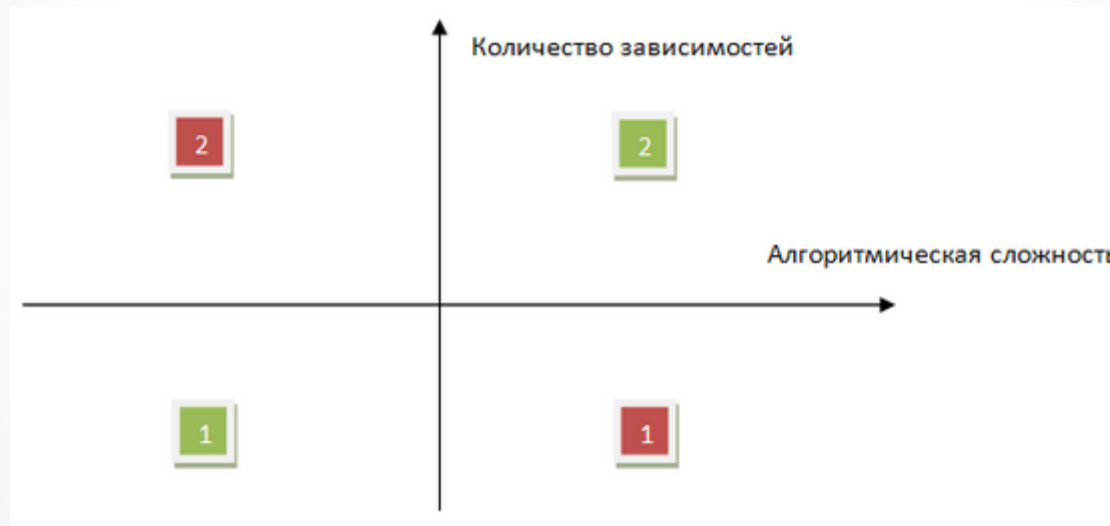
MsTest (интегрирован в Visual Studio, но не лучшее решение)

Nunit

XUnit

# Модульное тестирование

- Что подлежит тестированию?
  - Простой код без зависимостей.
  - Сложный код с большим количеством зависимостей.



- + Сложный код без зависимостей.
- + Не очень сложный код с зависимостями.



# Модульное тестирование

- единый стиль написания тела теста

Отлично зарекомендовал себя подход AAA (*arrange, act, assert*)

```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        // arrange
        var calc = new Calculator();

        // act
        var res = calc.Sum(2,5);

        // assert
        Assert.AreEqual(7, res);
    }
}
```

VS

```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        Assert.AreEqual(7, new Calculator().sum(2,5));
    }
}
```

# Модульное тестирование

- каждый тест должен проверять только одну вещь

Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.

Если не придерживаться этого правила, тесты станут нечитаемыми, и вскоре окажется очень сложно их поддерживать.

# Модульное тестирование

- причины, почему тест перестал проходить
  1. Ошибка в продакшн-коде: это баг, его нужно завести в баг-трекере и починить.
  2. Баг в тесте: видимо, продакшн-код изменился, а тест написан с ошибкой (например, тестирует слишком много или не то, что было нужно). Возможно, что раньше он проходил ошибочно. Разберитесь и почините тест.
  3. Смена требований. Если требования изменились слишком сильно – тест должен упасть. Это правильно и нормально. Вам нужно разобраться с новыми требованиями и исправить тест. Или удалить, если он больше не актуален.

# Модульное тестирование

## Модульные тесты

- 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

# Модульное тестирование

## + юнит-тестов:

1. Юнит-тесты можно запускать каждый раз сразу после внесения изменений в код, это позволит выявить дефект фактически при написании кода
2. Грамотно созданные юнит-тесты могут служить документацией к коду
3. Юнит-тесты упрощают процесс рефакторинга

## – юнит-тестов:

1. Надо выделять время на их создание и поддержку (изменение тестов)
2. Тесты должны быть написаны грамотно
3. Недостаточно получить корректные результаты выполнения юнит-тестов по отдельности. Требуется еще интегрированное тестирование для выявления ошибок работы модулей программного кода при их взаимодействии друг с другом.



# Модульное тестирование

**Для измерения успешности внедрения юнит-тестов в проекте следует использовать две метрики:**

1. Количество багов в новых релизах (в т.ч. и регрессии)
2. Покрытие кода

Первая показывает, есть ли у наших действий результат, или мы впустую расходует время, которое могли бы потратить на фичи. Вторая – как много еще предстоит сделать.

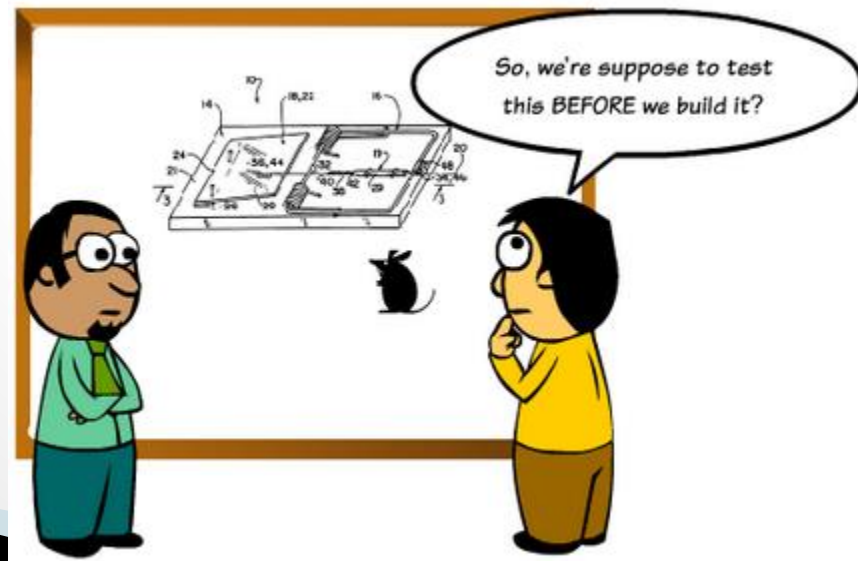
Наиболее популярные инструменты для измерения покрытия кода на .NET платформе:

1. dotTrace
2. NCover
3. встроенный в студию Test Coverage

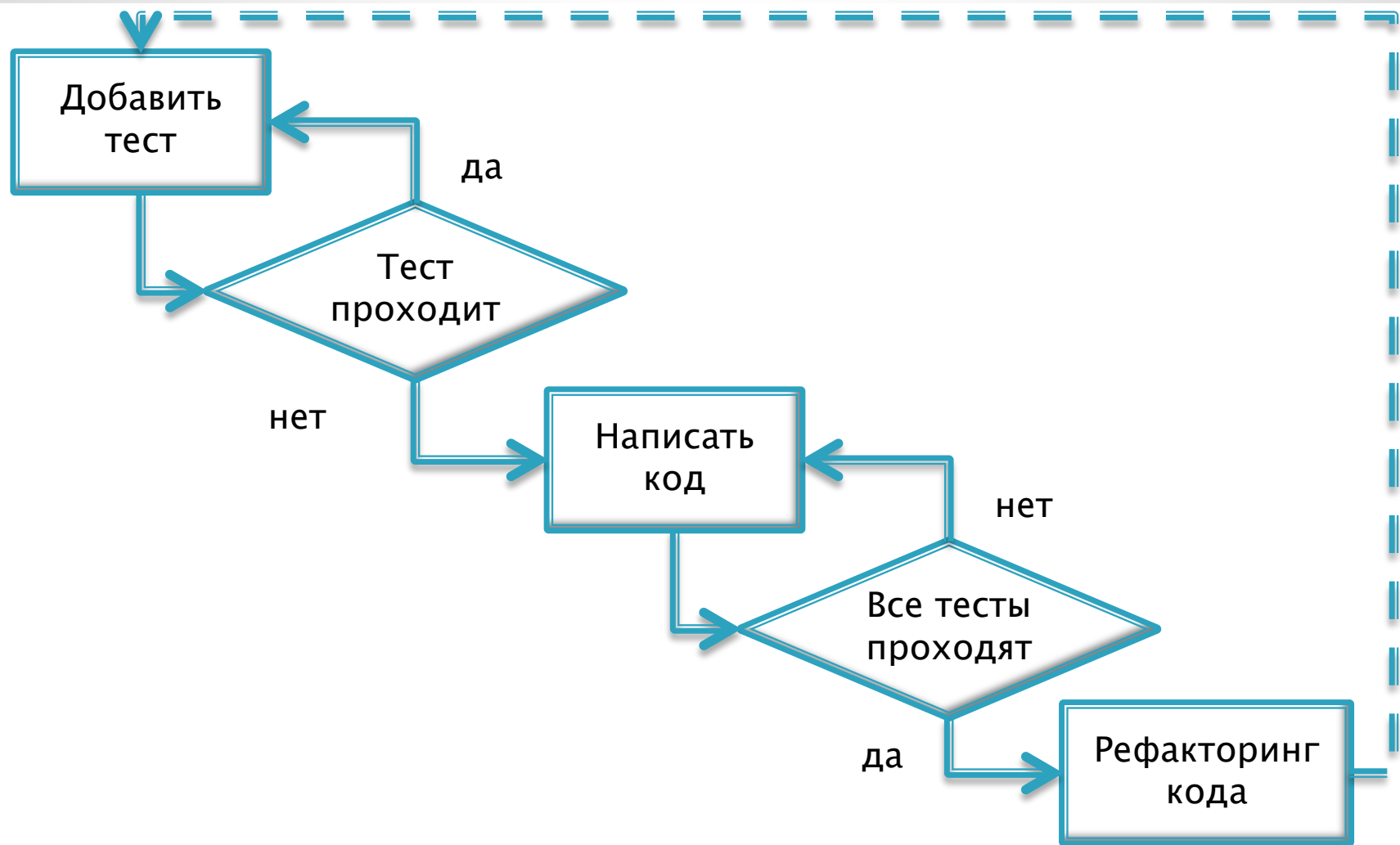
# Test-driven development

Разработка через тестирование (test-driven development, TDD) происходит в несколько этапов:

- Добавление теста
- Запуск всех тестов: убедиться, что новые тесты не проходят
- Написание кода
- Запуск всех тестов: убедиться, что все тесты проходят
- Рефакторинг
- Повторение цикла



# Test-driven development



# Test-driven development

Шаблон для написания тестов – «Triple A» (Arrange-Act-Assert), согласно шаблону тест состоит из трех частей

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

# Тестирование метода

```
// method under test
public void Debit(double amount)
{
    if(amount > m_balance)
    {
        throw new ArgumentOutOfRangeException("amount");
    }
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }
    m_balance += amount;
}
```



Create Unit Test



# Тестирование метода

Create Unit Test



```
// unit test code
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests
{
    [TestClass]
    public class BankAccountTests
    {
        [TestMethod]
        public void DebitTest()
        {
        }
    }
}
```

# Test-driven development

Требования к тестовому классу и методу:

1. Атрибут [TestClass] является обязательным для платформы модульных тестов Microsoft для управляемого кода в любом классе, содержащем методы модульных тестов, которые необходимо выполнить в обозревателе тестов.
2. Каждый метод теста, который требуется выполнять с помощью обозревателя тестов, должен иметь атрибут [TestMethod].
3. Метод должен вернуть void.
4. Метод не должен иметь параметров

# Test-driven development

Create Unit Test



```
// unit test code
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    account.Debit(debitAmount);

    // assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

# Test-driven development

Create Unit Test



```
//unit test method
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = -100.00;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    account.Debit(debitAmount);

    // assert is handled by ExpectedException
}
```

# Test-driven development

Create Unit Test



```
[TestMethod]
public void Debit_WhenAmountIsGreaterThanBalance_ShouldThrowArgumentOutOfRangeException()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 20.0;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    try
    {
        account.Debit(debitAmount);
    }
    catch (ArgumentOutOfRangeException e)
    {
        // assert
        StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
        return;
    }
    Assert.Fail("No exception was thrown.")
}
```



# Типы классов Assert

Пространство имен [Microsoft.VisualStudio.TestTools.UnitTesting](#) предоставляет несколько типов классов Assert.

## [Assert](#)

В методе теста можно вызывать любое число методов класса Assert, таких как `Assert.AreEqual()`. Класс Assert содержит много методов для выбора, и многие из этих методов имеют несколько перегрузок.

## [CollectionAssert](#)

Класс `CollectionAssert` служит для сравнения коллекций объектов и проверки состояния одной или нескольких коллекций.

## [StringAssert](#)

Класс `StringAssert` служит для сравнения строк. Этот класс содержит различные полезные методы, такие как `StringAssert.Contains`, `StringAssert.Matches` и `StringAssert.StartsWith`.

## [AssertFailedException](#)

Исключение `AssertFailedException` возникает в случае невыполнения теста. Причиной невозможности выполнения теста может быть истечение времени ожидания, непредвиденное исключение или оператор Assert, создающий результат "Ошибка".

## [AssertInconclusiveException](#)

Исключение `AssertInconclusiveException` возникает при каждом результате теста с неопределенным результатом. Как правило, оператор `Assert.Inconclusive` добавляется к тесту, над которым еще ведется работа, для обозначения его неготовности к выполнению.

## [ExpectedExceptionAttribute](#)

Если необходимо, чтобы метод теста проверял, что исключение, возникающее в этом методе, на самом деле является требуемым исключением, включите в метод тест атрибут `ExpectedExceptionAttribute`.

# Demo

Calculator.cs

C# Calculator Calculator.Calculator Div(int a, int b)

```
using System;

namespace Calculator
{
    0 references
    public class Calculator
    {
        0 references
        public int Sum(int a, int b)
        {
            return a + b;
        }
        0 references
        public int Mult(int a, int b)
        {
            return a * b;
        }
        0 references
        public double Div(int a, int b)
        {
            return a / b;
        }
    }
}
```

Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'Calculator' (1 project)

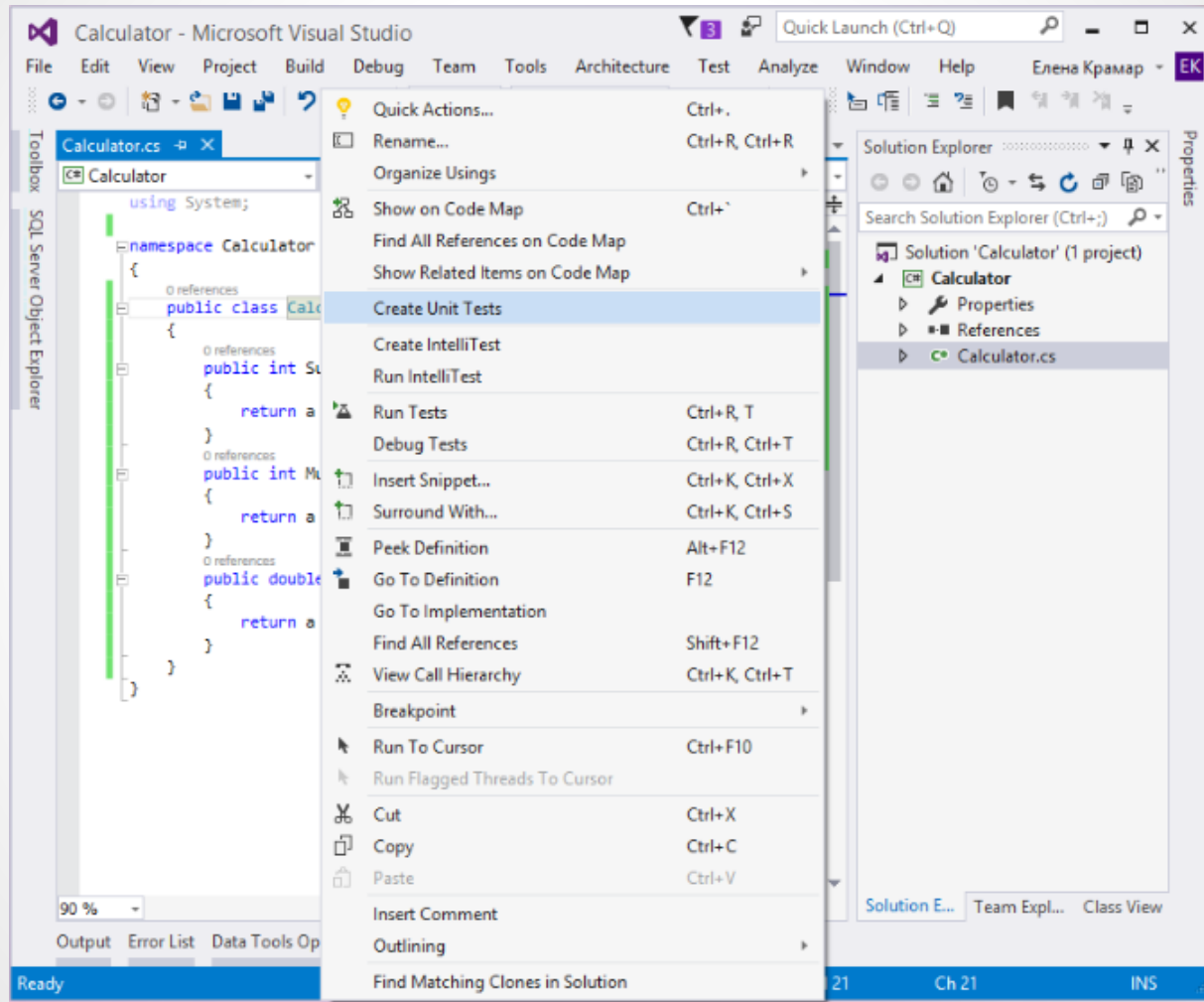
- C# Calculator
  - Properties
  - References
  - C# Calculator.cs

Output

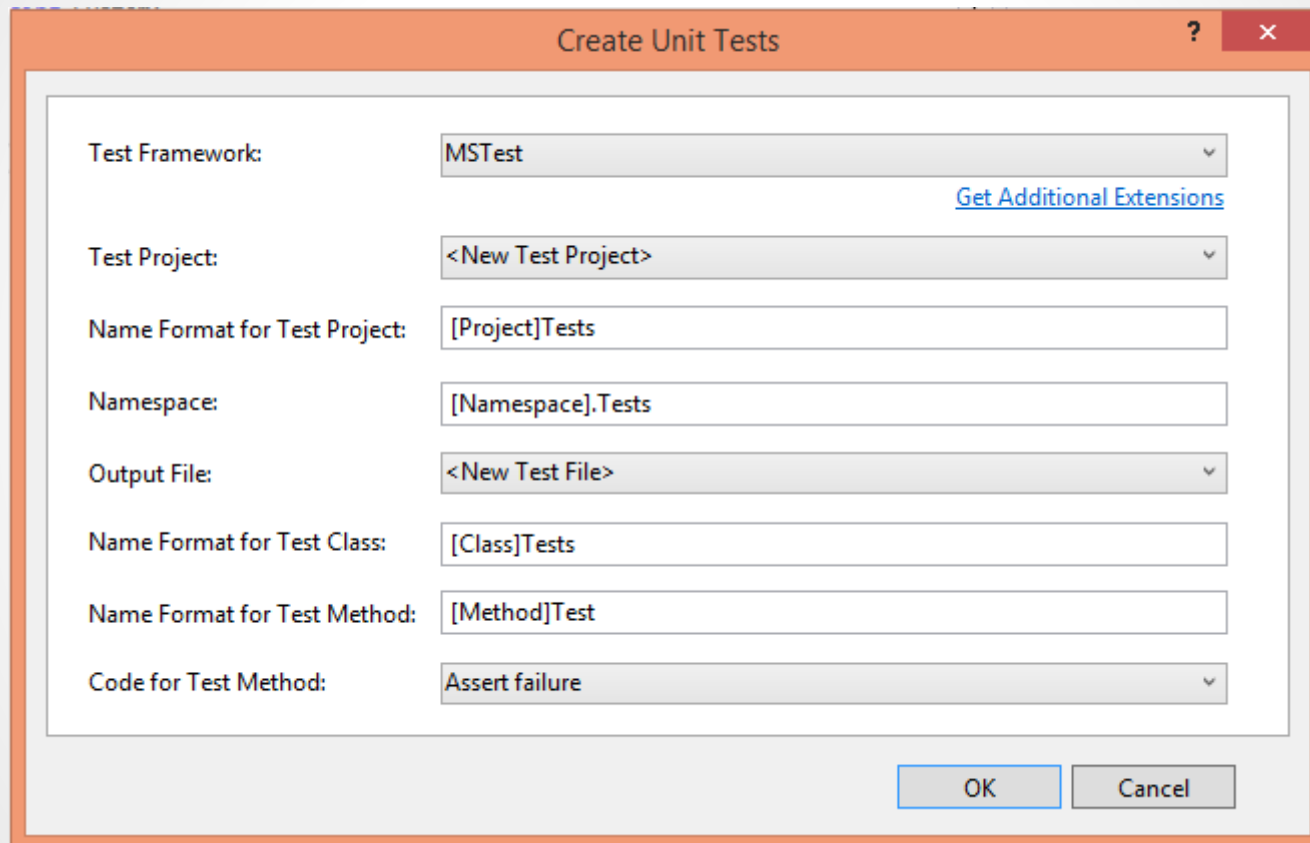
Show output from: Build

```
1>----- Build started: Project: Calculator, Configuration: Debug Any CPU -----
1> Calculator -> D:\Dropbox\KPIcity\Semester_6\ProgS_6\Technologies Of Computer Design\Calculator\Calcula
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

# Demo



# Demo



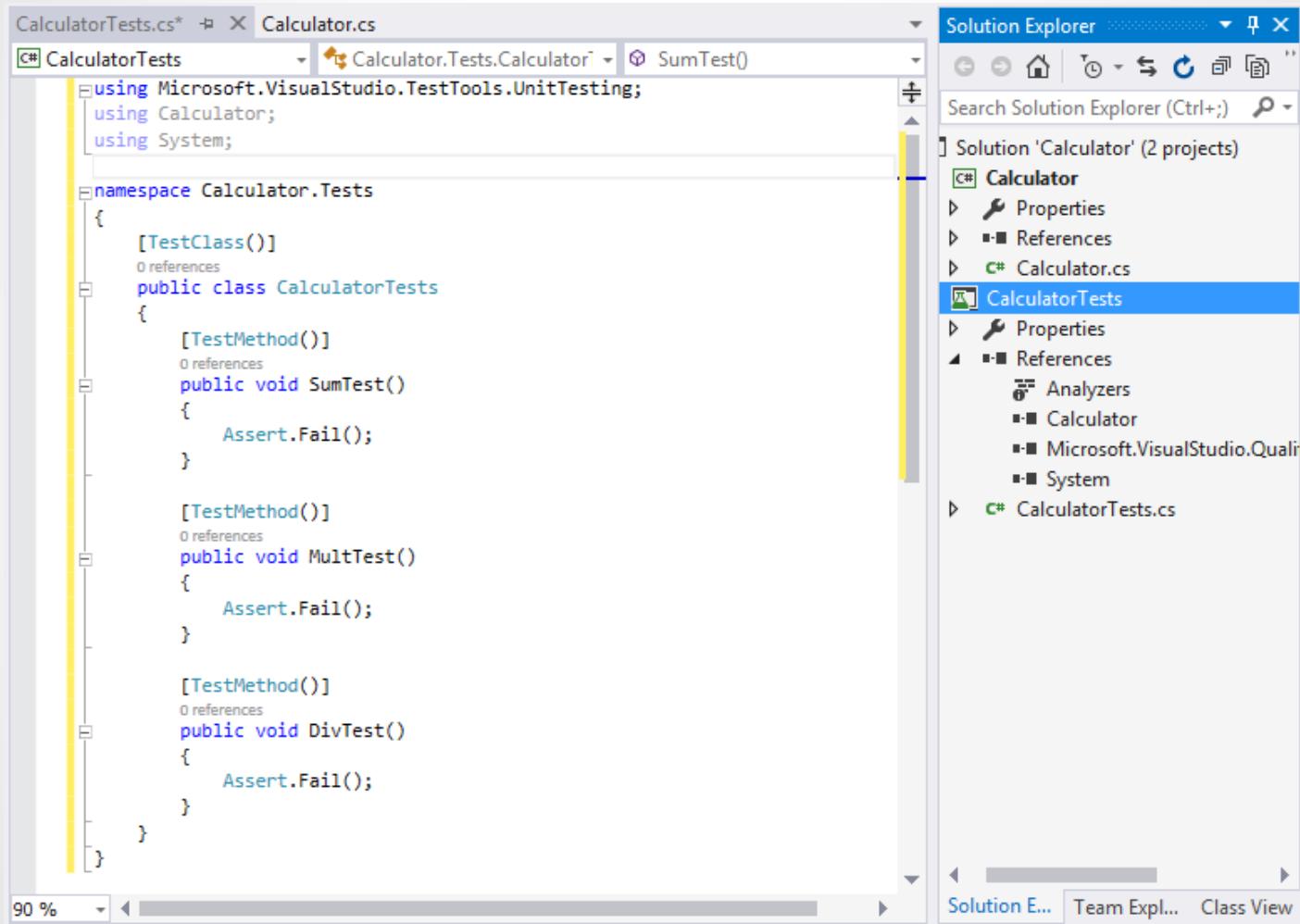
The image shows a 'Create Unit Tests' dialog box with an orange title bar. It contains several configuration options for creating unit tests, each with a label and a corresponding input field (either a dropdown menu or a text box). At the bottom right, there are 'OK' and 'Cancel' buttons.

Label	Value
Test Framework:	MSTest
Test Project:	<New Test Project>
Name Format for Test Project:	[Project]Tests
Namespace:	[Namespace].Tests
Output File:	<New Test File>
Name Format for Test Class:	[Class]Tests
Name Format for Test Method:	[Method]Test
Code for Test Method:	Assert failure

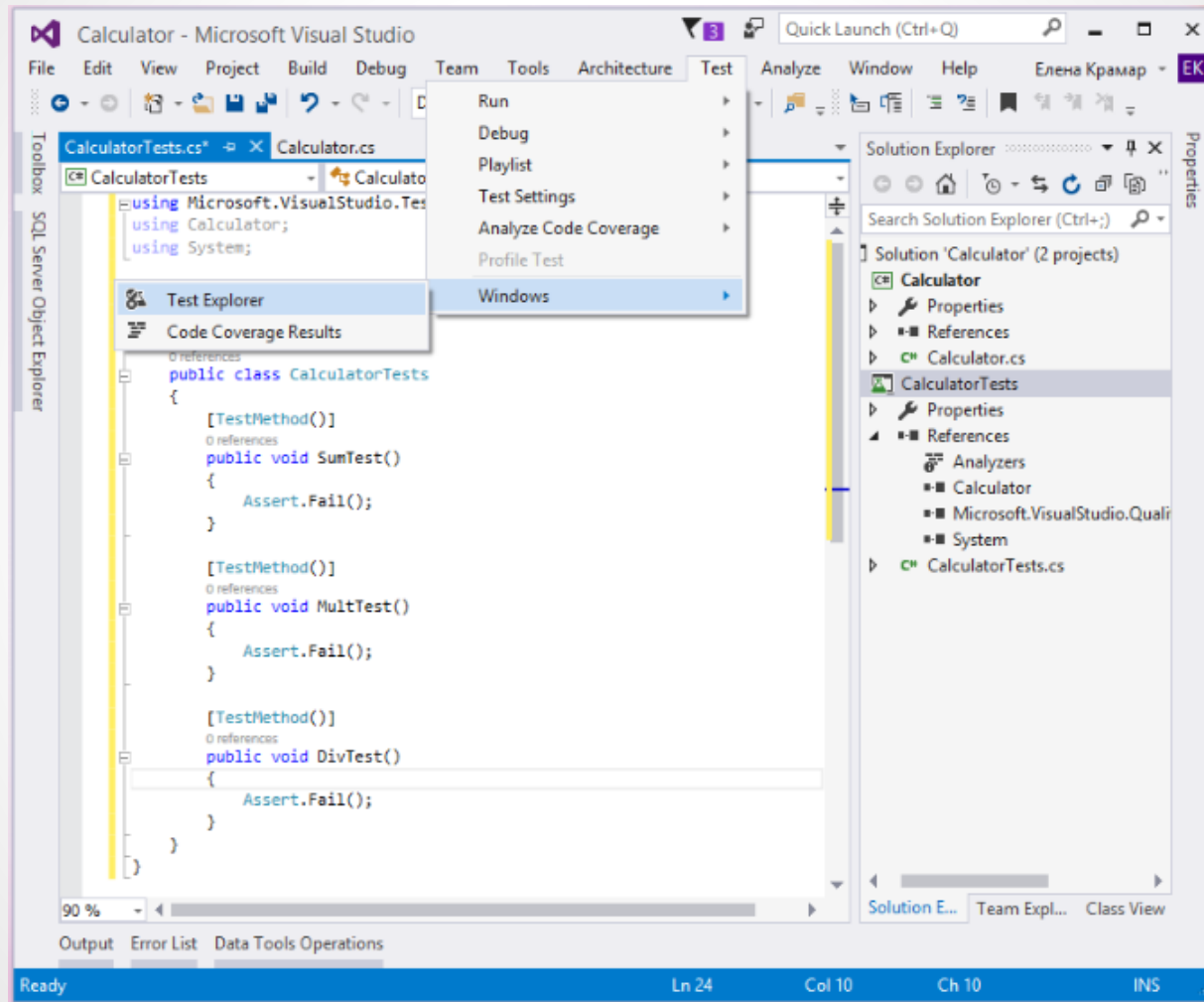
[Get Additional Extensions](#)

OK Cancel

# Demo



# Demo



# Demo

**Test Explorer**

Streaming Video: Configure contir

Run All | Run... | Playlist: All Tests

**Not Run Tests (3)**

- DivTest
- MultTest
- SumTest

**CalculatorTests.cs**

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Calculator;
using System;

namespace Calculator.Tests
{
    [TestClass()]
    0 references
    public class CalculatorTests
    {
        [TestMethod()]
        0 references
        public void SumTest()
        {
            Assert.Fail();
        }

        [TestMethod()]
        0 references
        public void MultTest()
        {
            Assert.Fail();
        }

        [TestMethod()]
    }
```

**Solution Explorer**

Solution 'Calculator' (2 projects)

- Calculator
  - Properties
  - References
  - Calculator.cs
- CalculatorTests
  - Properties
  - References
    - Analyzers
    - Calculator
    - Microsoft.VisualStudio.QualityTools.TestTools.UnitTesting
    - System
  - CalculatorTests.cs

**Output**

Show output from: Build

```
1>----- Build started: Project: CalculatorTests, Configuration: Debug Any CPU -----
1> CalculatorTests -> D:\Dropbox\KPIcity\Semester_6\ProgS_6\Technologies Of Computer Design\Calculator\Ca
===== Build: 1 succeeded, 0 failed, 1 up-to-date, 0 skipped =====
1
```



# Demo

The screenshot displays the Visual Studio IDE with three main panels:

- Test Explorer (Left):** Shows a list of failed tests under the heading "Failed Tests (3)". The tests are DivTest (< 1 ms), MultTest (1 ms), and SumTest (79 ms). A "Summary" section at the bottom indicates "Last Test Run Failed (Total Run Time 0:)" and "3 Tests Failed".
- Code Editor (Center):** Displays the source code for `CalculatorTests.cs`. The code is as follows:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Calculator;
using System;

namespace Calculator.Tests
{
    [TestClass]
    0 references
    public class CalculatorTests
    {
        [TestMethod]
        0 references
        public void SumTest()
        {
            Assert.Fail();
        }

        [TestMethod]
        0 references
        public void MultTest()
        {
            Assert.Fail();
        }

        [TestMethod]
        0 references
        public void DivTest()
        {
            Assert.Fail();
        }
    }
}
```
- Solution Explorer (Right):** Shows the project structure for the "Calculator" solution, which contains two projects: "Calculator" and "CalculatorTests". The "CalculatorTests" project is currently selected.

# Demo

The screenshot displays the Visual Studio interface during a test run. The **Test Explorer** on the left shows a summary of test results: 1 failed test (MultTest) and 2 passed tests (DivTest, SumTest). The **Code** window in the center shows the **CalculatorTests.cs** file with three test methods: **SumTest** (passed), **MultTest** (failed), and **DivTest** (passed). The **Solution Explorer** on the right shows the project structure, including the **Calculator** project and the **CalculatorTests** project.

**Test Explorer**

Search

Streaming Video: Configure contir

Run All | Run... | Playlist: All Tests

**Failed Tests (1)**

- MultTest 82 ms

**Passed Tests (2)**

- DivTest < 1 ms
- SumTest 11 ms

**Summary**

Last Test Run Failed (Total Run Time 0:00:00)

- 1 Test Failed
- 2 Tests Passed

**CalculatorTests.cs**

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Calculator;
using System;

namespace Calculator.Tests
{
    [TestClass]
    0 references
    public class CalculatorTests
    {
        [TestMethod]
        0 references
        public void SumTest()
        {
            Assert.IsTrue(true);
        }

        [TestMethod]
        0 references
        public void MultTest()
        {
            Assert.Fail();
        }

        [TestMethod]
        0 references
        public void DivTest()
        {
            Assert.IsTrue(true);
        }
    }
}
```

**Solution Explorer**

Search Solution Explorer (Ctrl+;)

Solution 'Calculator' (2 projects)

- Calculator
  - Properties
  - References
  - Calculator.cs
- CalculatorTests
  - Properties
  - References
    - Analyzers
    - Calculator
    - Microsoft.VisualStudio.QualityTools.TestTools.UnitTesting
    - System
  - CalculatorTests.cs

Solution Explorer | Team Explorer | Class View

# Demo

**Test Explorer**

Search

Streaming Video: Configure contir

Run All | Run... | Playlist: All Tests

**Failed Tests (1)**

- MultTest 69 ms

**Passed Tests (2)**

- DivTest < 1 ms
- Sum\_2Plus5\_7Returned 8 ms

**Summary**

Last Test Run Failed (Total Run Time 0:00:00.000)

- 1 Test Failed
- 2 Tests Passed

**CalculatorTests.cs**

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Calculator;
using System;

namespace Calculator.Tests
{
    [TestClass()]
    public class CalculatorTests
    {
        [TestMethod()]
        public void Sum_2Plus5_7Returned()
        {
            // arrange
            var calc = new Calculator();

            // act
            var res = calc.Sum(2, 5);

            // assert
            Assert.AreEqual(7, res);
        }

        [TestMethod()]
        public void MultTest()
        {
            Assert.Fail();
        }

        [TestMethod()]
        public void DivTest()
        {
            // ...
        }
    }
}
```

**Solution Explorer**

Search Solution Explorer (Ctrl+;)

Solution 'Calculator' (2 projects)

- Calculator
  - Properties
  - References
  - Calculator.cs
- CalculatorTests
  - Properties
  - References
    - Analyzers
    - Calculator
    - Microsoft.VisualStudio.QualityTools.TestTools.UnitTesting
    - System
  - CalculatorTests.cs

# Demo

The screenshot displays the Visual Studio IDE with the Test Explorer on the left, the code editor in the center, and the Solution Explorer on the right.

**Test Explorer (Left):**

- Search bar
- Streaming Video: Configure contri...
- Run All | Run... | Playlist: All Tests
- Failed Tests (1)**
  - ✖ MultTest 70 ms
- Passed Tests (2)**
  - ✔ Div\_DivisionByZero\_Should... 1 ms
  - ✔ Sum\_2Plus5\_7Returned 8 ms
- Summary**
  - Last Test Run Failed (Total Run Time 0:00:00.000)
  - ✖ 1 Test Failed
  - ✔ 2 Tests Passed

**Code Editor (Center):**

Files: CalculatorTests.cs | Calculator.cs

Selected file: CalculatorTests.cs

```
// arrange
var calc = new Calculator();

// act
var res = calc.Sum(2, 5);

// assert
Assert.AreEqual(7, res);
}

[TestMethod()]
0 references
public void MultTest()
{
    Assert.Fail();
}

[TestMethod()]
[ExpectedException(typeof(System.DivideByZeroException))]
0 references
public void Div_DivisionByZero_ShouldThrowDivideByZero()
{
    // arrange
    var calc = new Calculator();

    // act
    var res = calc.Div(10, 0);

    // assert is handled by ExpectedException
}
```

**Solution Explorer (Right):**

- Solution 'Calc...
- Calculator
- Properties
- References
- Calculator
- Properties
- References
- Analysis
- Calculator
- Micro
- System
- Calculator

# Demo

The screenshot displays the Visual Studio IDE with the Test Explorer on the left, the CalculatorTests.cs file in the center, and the Solution Explorer on the right.

**Test Explorer (Left):**

- Search bar
- Buttons: Run All, Run..., Playlist: All Tests
- Failed Tests (2):**
  - Div\_DivisionByZero\_Should... 3 ms
  - MultTest 75 ms
- Passed Tests (2):**
  - Div\_DivisionByZero\_Should... 1 ms
  - Sum\_2Plus5\_7Returned 10 ms
- Div\_DivisionByZero\_ShouldThrow**
  - Source: CalculatorTests.cs line 45
  - Test Failed - Div\_DivisionByZero\_S
  - Message: Ошибка в StringAssert.Contains. Строка "Попытка деления на ноль." не содержит строку "zero".
  - Elapsed time: 3 ms
  - StackTrace: CalculatorTests.Div\_DivisionBy

**CalculatorTests.cs (Center):**

```
var calc = new Calculator();

// act
var res = calc.Div(10, 0);

// assert is handled by ExpectedException
}

[TestMethod()]
public void Div_DivisionByZero_ShouldThrowDivideByZero2()
{
    // arrange
    var calc = new Calculator();

    // act
    try
    {
        var res = calc.Div(10, 0);
    }
    catch (DivideByZeroException e)
    {
        StringAssert.Contains(e.Message, "zero");
        return;
    }

    Assert.Fail("No exception was thrown");
}
```

**Solution Explorer (Right):**

- Solution 'Calc'
- Calculator
- Properties
- References
- Calculator
- Properties
- References
- Analysis
- Calculator
- Micro
- System
- Calculator

# Demo

The screenshot displays the Visual Studio IDE with the Test Explorer on the left, the main code editor in the center, and the Solution Explorer on the right.

**Test Explorer (Left):**

- Search bar at the top.
- Buttons: Run All, Run..., Playlist: All Tests.
- Failed Tests (1):** MultTest (71 ms).
- Passed Tests (3):**
  - Div\_DivisionByZero\_Should... (1 ms)
  - Div\_DivisionByZero\_Sho... (< 1 ms)** (Selected)
  - Sum\_2Plus5\_7Returned (8 ms)

**Test Details (Bottom Left):**

**Div\_DivisionByZero\_ShouldThrow**

Source: CalculatorTests.cs line 45

✓ Test Passed - Div\_DivisionByZero\_

Elapsed time: < 1 ms

**Code Editor (Center):**

Files: CalculatorTests.cs, Calculator.cs

Selected Item: Div\_DivisionByZero\_S

```
var calc = new Calculator();

// act
var res = calc.Div(10, 0);

// assert is handled by ExpectedException
}

[TestMethod()]
0 references
public void Div_DivisionByZero_ShouldThrowDivideByZero2()
{
    // arrange
    var calc = new Calculator();

    // act
    try
    {
        var res = calc.Div(10, 0);
    }
    catch (DivideByZeroException e)
    {
        StringAssert.Contains(e.Message, "нуль");
        return;
    }

    Assert.Fail("No exception was thrown");
}
```

**Solution Explorer (Right):**

Solution 'Calc'

- Calculator
- Properties
- References
- Calculator
- Properties
- References
- Analysis
- Calculator
- Micro
- System
- Calculator

# Demo

The screenshot displays the Visual Studio IDE with three main panels:

- Test Explorer:** Shows test results for 'CalculatorTests'.
  - Failed Tests (1):** MultTest (81 ms).
  - Passed Tests (3):** Div\_10Div2\_5Returned (1 ms), Div\_DivisionByZero\_Should... (1 ms), Sum\_2Plus5\_7Returned (14 ms).
  - Summary:** Last Test Run Failed (Total Run Time 0:00:01.14). 1 Test Failed, 4 Tests Passed.
- CalculatorTests.cs:** Shows the test code for 'Div\_DivisionByZero\_ShouldThrowDivideByZero()'. The test is marked as failed with a red 'X' icon.

```
[TestMethod()]
[ExpectedException(typeof(System.DivideByZeroException))]
public void Div_DivisionByZero_ShouldThrowDivideByZero()
{
    // arrange
    var calc = new Calculator();

    // act
    double res = calc.Div(100, 20);

    // assert
    Assert.AreEqual(5, res, 0.001, "Wrong division result")
}
```
- Code Coverage Results:** A table showing coverage for the 'Calculator' class and its methods.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Calculator	2	33,33%	4	66,67%
Calculator	2	33,33%	4	66,67%
Sum(int, int)	0	0,00%	2	100,00%
Mult(int, int)	2	100,00%	0	0,00%
Div(int, int)	0	0,00%	2	100,00%



# Demo

The screenshot displays the Visual Studio IDE with the Test Explorer on the left and the code editor on the right.

**Test Explorer (Left):**

- Search bar: [Search]
- Streaming Video: Configure continuous integration
- Run All | Run... | Playlist: All Tests
- Passed Tests (7)**
  - Remove\_ElementDeleteFromNotEmptyQueue\_FalseRet... 369 ms
  - Remove\_ElementDeleteFromQueueWithOneElement\_El... < 1 ms
  - Remove\_FirstElementDeleteFromNotEmptyQueue\_FirstE... < 1 ms
  - Remove\_LastElementDeleteFromNotEmptyQueue\_LastEl... < 1 ms
  - Remove\_MiddleElementDeleteFromNotEmptyQueue\_Mid... 6 ms
  - Remove\_NotExistElementDeleteFromNotEmptyQueue\_F... < 1 ms
  - Remove\_NotExistElementDeleteFromQueueWithOneEle... < 1 ms

**Test Results (Bottom Left):**

Remove\_ElementDeleteFromQueueWithOneElement\_ElementDeletedTrueReturned()

Source: TwoEndsQueueTests.cs line 121

Test Passed - Remove\_ElementDeleteFromQueueWithOneElement\_ElementDeletedTrueReturned()

Elapsed time: < 1 ms

**Code Editor (Right):**

Program.cs TwoEndsQueue.cs TwoEndsQueueTests.cs

UnitTestProject TCD\_Lab\_3.Tests.TwoEndsQueueTests Remove\_ElementDeleteFromQueueWithOneElement\_ElementDeletedTrueReturned()

```
{
    [TestClass()]
    0 references
    public class TwoEndsQueueTests
    {
        [TestMethod()]
        0 references
        public void Remove_ElementDeleteFromNotEmptyQueue_FalseReturned()...

        [TestMethod()]
        0 references
        public void Remove_MiddleElementDeleteFromNotEmptyQueue_MiddleElementDeletedTrueReturned()...

        [TestMethod()]
        0 references
        public void Remove_FirstElementDeleteFromNotEmptyQueue_FirstElementDeletedTrueReturned()...

        [TestMethod()]
        0 references
        public void Remove_LastElementDeleteFromNotEmptyQueue_LastElementDeletedTrueReturned()...

        [TestMethod()]
        0 references
        public void Remove_NotExistElementDeleteFromNotEmptyQueue_FalseReturned()...

        [TestMethod()]
        0 references
        public void Remove_ElementDeleteFromQueueWithOneElement_ElementDeletedTrueReturned()...

        [TestMethod()]
        0 references
        public void Remove_NotExistElementDeleteFromQueueWithOneElement_FalseReturned()...
    }
}
```

## МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

### Ресурсы:

1. **Рой Ошероув Искусство автономного тестирования с примерами на C# (Roy Osherove The Art of Unit Testing With Examples in .NET)**
2. <https://msdn.microsoft.com/ru-ru/library/ms182532.aspx>
3. <https://habrahabr.ru/post/169381/>