Бесплатная электронная книга

УЧУСЬ

unit-testing

Free unaffiliated eBook created from
Stack Overflow contributors.

#unit-testing

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unit-testing

It is an unofficial and free unit-testing ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unit-testing.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с модульным тестированием

замечания

В модульном тестировании описывается процесс тестирования отдельных блоков кода отдельно от системы, частью которой они являются. То, что составляет единицу, может варьироваться от системы к системе, начиная от индивидуального метода и заканчивая группой тесно связанных классов или модуля.

Блок изолирован от своих зависимостей, используя тестовые удваивает, когда необходимо, и настраивается в известное состояние. Его поведение в реакции на стимулы (вызовы методов, события, смоделированные данные) затем проверяется на ожидаемое поведение.

Единичное тестирование целых систем может быть выполнено с использованием специальных письменных тестовых жгутов, однако многие тестовые структуры были написаны для упрощения процесса и ухода за большой частью сантехнических, повторяющихся и мирских задач. Это позволяет разработчикам сосредоточиться на том, что они хотят проверить.

Когда проект имеет достаточно модульных тестов, любая модификация добавления новых функций или выполнения рефакторинга кода может быть легко выполнена путем проверки в конце, что все работает по-прежнему.

Кодовое покрытие , обычно выраженное в процентах, является типичной метрикой, используемой для отображения того, какая часть кода в системе покрывается Unit Tests; обратите внимание, что нет жесткого и быстрого правила о том, как высоко это должно быть, но общепризнано, что чем выше, тем лучше.

Test Driven Development (TDD) - это принцип, который указывает, что разработчик должен начать кодирование, написав неудачный модульный тест и только потом написать производственный код, который пропустит тест. При практике TDD можно сказать, что сами тесты являются первым потребителем создаваемого кода; поэтому они помогают проводить аудит и управлять дизайном кода, чтобы он был прост в использовании и максимально надежным.

Версии

Модульное тестирование - это концепция, которая не имеет номеров версий.

Examples

Базовый единичный тест

В своем простейшем, единичный тест состоит из трех этапов:

- Подготовьте среду для тестирования
- Выполнить проверенный код
- Проверка ожидаемого поведения соответствует наблюдаемому поведению

Эти три этапа часто называются «Arrange-Act-Assert» или «Given-When-Then».

Ниже приведен пример на C #, который использует структуру NUnit .

```
[TestFixture]
public CalculatorTest
{
   [Test]
   public void Add_PassSevenAndThree_ExpectTen()
   {
       // Arrange - setup environment
       var systemUnderTest = new Calculator();

       // Act - Call system under test
       var calculatedSum = systemUnderTest.Add(7, 3);

       // Assert - Validate expected result
       Assert.AreEqual(10, calculatedSum);
  }
}
```

Там, где это необходимо, добавляется дополнительная четвертая ступень очистки.

Единичный тест с заглубленной зависимостью

Хорошие модульные тесты независимы, но код часто имеет зависимости. Мы используем различные типы тестов для удаления зависимостей для тестирования. Один из простейших двойных тестов - заглушка. Это функция с жестко закодированным значением возврата, называемым вместо реальной зависимости.

```
// Test that oneDayFromNow returns a value 24*60*60 seconds later than current time

let systemUnderTest = new FortuneTeller()       // Arrange - setup environment
systemUnderTest.setNow(() => {return 10000})    //   inject a stub which will
                                                //   return 10000 as the result

let actual = systemUnderTest.oneDayFromNow()    // Act - Call system under test

assert.equals(actual, 10000 + 24 * 60 * 60)     // Assert - Validate expected result
```

В производственном коде `oneDayFromNow` будет вызывать Date.now (), но это приведет к

непоследовательным и ненадежным тестам. Итак, здесь мы это исключаем.

Единичный тест со шпионом (тест взаимодействия)

Классический блок тестирует *состояние* теста, но может быть невозможно правильно проверить методы, поведение которых зависит от других классов через состояние. Мы тестируем эти методы с помощью *тестов взаимодействия* , которые проверяют, что тестируемая система правильно называет своих сотрудников. Поскольку у коллаборационистов есть свои собственные модульные тесты, этого достаточно и на самом деле лучше проверить фактическую ответственность тестируемого метода. Мы не проверяем, что этот метод возвращает конкретный результат с учетом ввода, но вместо этого он правильно называет его соавтор (ы).

```
// Test that squareOfDouble invokes square() with the doubled value

let systemUnderTest = new Calculator()          // Arrange - setup environment
let square = spy()
systemUnderTest.setSquare(square)               //   inject a spy

let actual = systemUnderTest.squareOfDouble(3)  // Act - Call system under test

assert(square.calledWith(6))                    // Assert - Validate expected interaction
```

Простой тест Java + JUnit

JUnit - это ведущая платформа тестирования, используемая для тестирования кода Java.

Класс под тестированием моделирует простой банковский счет, который взимает штраф, когда вы переходите на более высокий уровень.

```
public class BankAccount {
    private int balance;

    public BankAccount(int i){
        balance = i;
    }

    public BankAccount(){
        balance = 0;
    }

    public int getBalance(){
        return balance;
    }

    public void deposit(int i){
        balance += i;
    }

    public void withdraw(int i){
        balance -= i;
        if (balance < 0){
            balance -= 10; // penalty if overdrawn
```

```
        }
    }
}
```

Этот тестовый класс проверяет поведение некоторых общедоступных методов `BankAccount` .

```
import org.junit.Test;
import static org.junit.Assert.*;

// Class that tests
public class BankAccountTest{

    BankAccount acc;

    @Before                          // This will run **before** EACH @Test
    public void setUptestDepositUpdatesBalance(){
        acc = new BankAccount(100);
    }

    @After                          // This Will run **after** EACH @Test
    public void tearDown(){
    // clean up code
    }

    @Test
    public void testDeposit(){
       // no need to instantiate a new BankAccount(), @Before does it for us

        acc.deposit(100);

        assertEquals(acc.getBalance(),200);
    }

    @Test
    public void testWithdrawUpdatesBalance(){
        acc.withdraw(30);

        assertEquals(acc.getBalance(),70); // pass
    }

    @Test
    public void testWithdrawAppliesPenaltyWhenOverdrawn(){

        acc.withdraw(120);

        assertEquals(acc.getBalance(),-30);
    }
}
```

Unit Test с параметрами с использованием NUnit и C

```
using NUnit.Framework;

namespace MyModuleTests
{
    [TestFixture]
    public class MyClassTests
    {
```

```
        [TestCase(1, "Hello", true)]
        [TestCase(2, "bye", false)]
        public void MyMethod_WhenCalledWithParameters_ReturnsExpected(int param1, string
param2, bool expected)
        {
        //Arrange
        var foo = new MyClass(param1);

        //Act
        var result = foo.MyMethod(param2);

        //Assert
        Assert.AreEqual(expected, result);
        }
    }
}
```

Базовый тестовый блок python

```
import unittest

def addition(*args):
    """ add two or more summands and return the sum """

    if len(args) < 2:
        raise ValueError, 'at least two summands are needed'

    for ii in args:
        if not isinstance(ii, (int, long, float, complex )):
            raise TypeError

    # use build in function to do the job
    return sum(args)
```

Теперь тестовая часть:

```
class Test_SystemUnderTest(unittest.TestCase):

    def test_addition(self):
        """test addition function"""

        # use only one summand - raise an error
        with self.assertRaisesRegexp(ValueError, 'at least two summands'):
            addition(1)

        # use None - raise an error
        with self.assertRaises(TypeError):
            addition(1, None)

        # use ints and floats
        self.assertEqual(addition(1, 1.), 2)

        # use complex numbers
        self.assertEqual(addition(1, 1., 1+2j), 3+2j)

if __name__ == '__main__':
    unittest.main()
```

Тест XUnit с параметрами

```
using Xunit;

public class SimpleCalculatorTests
{
    [Theory]
    [InlineData(0, 0, 0, true)]
    [InlineData(1, 1, 2, true)]
    [InlineData(1, 1, 3, false)]
    public void Add_PassMultipleParameters_VerifyExpected(
        int inputX, int inputY, int expected, bool isExpectedCorrect)
    {
        // Arrange
        var sut = new SimpleCalculator();

        // Act
        var actual = sut.Add(inputX, inputY);

        // Assert
        if (isExpectedCorrect)
        {
            Assert.Equal(expected, actual);
        }
        else
        {
            Assert.NotEqual(expected, actual);
        }
    }
}

public class SimpleCalculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Прочитайте Начало работы с модульным тестированием онлайн: https://riptutorial.com/ru/unit-testing/topic/570/начало-работы-с-модульным-тестированием

глава 2: Внедрение зависимости

замечания

Один из подходов, который можно использовать для написания программного обеспечения, - это создание зависимостей по мере необходимости. Это довольно интуитивный способ написать программу, и это способ, которым большинство людей будут учиться, отчасти потому, что легко следовать. Одна из проблем, связанных с этим подходом, заключается в том, что ее трудно проверить. Рассмотрим метод, который выполняет некоторую обработку на основе текущей даты. Метод может содержать некоторый код, например:

```
if (DateTime.Now.Date > processDate)
{
    // Do some processing
}
```

Код имеет прямую зависимость от текущей даты. Этот метод трудно проверить, потому что текущую дату нельзя легко манипулировать. Один из подходов к обеспечению большей проверки кода - это удалить прямую ссылку на текущую дату и вместо этого предоставить (или ввести) текущую дату методу, который выполняет обработку. Эта инъекция зависимостей может значительно облегчить проверку аспектов кода с помощью тестовых удвоений, чтобы упростить шаг настройки модульного теста.

Системы MOK

Еще один аспект, который следует рассмотреть, - время жизни зависимостей; в случае, когда сам класс создает свои собственные зависимости (также известные как инварианты), тогда он ответственен за их удаление. Инъекция зависимостей инвертирует это (и поэтому мы часто ссылаемся на библиотеку инъекций как на систему «Инверсия контроля») и означает, что вместо класса, ответственного за создание, управление и очистку его зависимостей, внешний агент (в этом случай, система IoC) делает это вместо этого.

Это делает гораздо проще иметь зависимости, которые совместно используются экземплярами одного и того же класса; например, рассмотрим службу, которая извлекает данные из конечной точки HTTP для потребляемого класса. Поскольку эта служба не имеет статуса (т. Е. Она не имеет внутреннего состояния), поэтому нам действительно нужен только один экземпляр этой службы в нашем приложении. Хотя это возможно (например, используя статический класс), чтобы сделать это вручную, гораздо проще создать класс и сообщить системе IoC, что он должен быть создан как *Singleton* , в котором существует только один экземпляр класса.

Другим примером могут быть контексты базы данных, используемые в веб-приложении, в результате чего требуется новый Контекст для каждого запроса (или потока), а не для экземпляра контроллера; это позволяет контексту вводить в каждый слой, выполняемый этим потоком, без необходимости вручную переносить его.

Это освобождает классы-потребители от необходимости управлять зависимостями.

Examples

Инъекция конструктора

Инъекция конструктора - самый безопасный способ инъекции зависимостей, от которых зависит весь класс. Такие зависимости часто называют *инвариантами* , поскольку экземпляр класса не может быть создан без их доставки. Требуя, чтобы зависимость была введена при построении, гарантируется, что объект не может быть создан в несогласованном состоянии.

Рассмотрим класс, который необходимо записать в файл журнала в условиях ошибки. Он имеет зависимость от `ILogger` , который можно вводить и использовать, когда это необходимо.

```
public class RecordProcessor
{
    readonly private ILogger _logger;

    public RecordProcessor(ILogger logger)
    {
        _logger = logger;
    }

    public void DoSomeProcessing() {
        // ...
        _logger.Log("Complete");
    }
}
```

Иногда при написании тестов вы можете заметить, что конструктор требует больше зависимостей, чем это действительно необходимо для проверяемого случая. Чем больше таких тестов у вас, тем больше вероятность того, что ваш класс нарушит *принцип единой ответственности* (SRP). Вот почему это не очень хорошая практика, чтобы определить поведение по умолчанию для всех маков вложенных зависимостей на этапе инициализации класса тестирования, поскольку оно может маскировать потенциальный предупреждающий сигнал.

Unittest для этого будет выглядеть следующим образом:

```
[Test]
public void RecordProcessor_DependencyInjectionExample()
```

```
{
    ILogger logger = new FakeLoggerImpl(); //or create a mock by a mocking Framework

    var sut = new RecordProcessor(logger); //initialize with fake impl in testcode

    Assert.IsTrue(logger.HasCalledExpectedMethod());
}
```

Инъекция недвижимости

Включение свойств позволяет обновлять зависимости классов после их создания. Это может быть полезно, если вы хотите упростить создание объекта, но все же разрешите переопределения зависимостей вашими испытаниями с двойным тестированием.

Рассмотрим класс, который необходимо записать в файл журнала в состоянии ошибки. Класс знает , как построить по умолчанию `Logger` , но позволяет ему быть перекрыт через инъекцию собственности. Однако стоит отметить, что при использовании вложения свойств таким образом вы тесно `ILogger` этот класс с точной реализацией `ILogger` который является `ConcreteLogger` в данном примере. Возможным обходным решением может быть фабрика, которая возвращает необходимую реализацию ILogger.

```
public class RecordProcessor
{
    public RecordProcessor()
    {
        Logger = new ConcreteLogger();
    }

    public ILogger Logger { get; set; }

    public void DoSomeProcessing()
    {
        // ...
        _logger.Log("Complete");
    }
}
```

В большинстве случаев Constructor Injection предпочтительнее Injection Property, потому что она обеспечивает лучшие гарантии состояния объекта сразу после его построения.

Метод инъекции

Метод инъекции представляет собой мелкозернистый способ инъекции зависимостей в обработку. Рассмотрим метод, который выполняет некоторую обработку на основе текущей даты. Текущую дату трудно изменить из теста, поэтому гораздо проще передать дату в метод, который вы хотите проверить.

```
public void ProcessRecords(DateTime currentDate)
{
    foreach(var record in _records)
    {
```

```
        if (currentDate.Date > record.ProcessDate)
        {
            // Do some processing
        }
    }
}
```

Контейнеры / DI Frameworks

В то время как извлечение зависимостей из вашего кода, чтобы они могли быть введены, делает ваш код более легким для тестирования, это еще больше увеличивает проблему иерархии и может также привести к сложным конструкциям. Для преодоления этой проблемы были написаны различные схемы инъекций зависимостей / инверсия контрольных контейнеров. Они позволяют регистрировать типы. Эти регистрации затем используются для разрешения зависимостей, когда контейнеру предлагается создать объект.

Рассмотрим эти классы:

```
public interface ILogger {
    void Log(string message);
}

public class ConcreteLogger : ILogger
{
    public ConcreteLogger()
    {
        // ...
    }
    public void Log(string message)
    {
        // ...
    }
}
public class SimpleClass
{
    public SimpleClass()
    {
        // ...
    }
}

public class SomeProcessor
{
    public SomeProcessor(ILogger logger, SimpleClass simpleClass)
    {
        // ...
    }
}
```

Чтобы построить `SomeProcessor` , необходимы как экземпляр `ILogger` и `SimpleClass` . Контейнер, подобный Unity, помогает автоматизировать этот процесс.

Сначала необходимо сконструировать контейнер, а затем регистрировать его. Обычно это

делается только один раз в приложении. Область системы, в которой это происходит, обычно известна как *Корень Композиции*

```
// Register the container
var container = new UnityContainer();

// Register a type mapping.  This allows a `SimpleClass` instance
// to be constructed whenever it is required.
container.RegisterType<SimpleClass, SimpleClass>();

// Register an instance.  This will use this instance of `ConcreteLogger`
// Whenever an `ILogger` is required.
container.RegisterInstance<ILogger>(new ConcreteLogger());
```

После того, как контейнер сконфигурирован, его можно использовать для создания объектов, при необходимости автоматически определяющих зависимости:

```
var processor = container.Resolve<SomeProcessor>();
```

Прочитайте Внедрение зависимости онлайн: https://riptutorial.com/ru/unit-testing/topic/597/внедрение-зависимости

глава 3: Единичное тестирование циклов (Java)

Вступление

Петли считаются одной из важных структур управления на любом языке программирования. Существуют различные способы, с помощью которых можно достичь охвата петлей.

Эти методы различаются в зависимости от типа цикла.

Одиночные петли

Вложенные петли

Конкатенированные петли

Examples

Тест с одним циклом

Это петли, в которых их тело цикла не содержит других циклов (самый внутренний цикл в случае вложенности).

Чтобы охватить петлю, тестеры должны провести тесты, приведенные ниже.

Тест 1: спроектировать тест, в котором тело цикла не должно выполняться вообще (т.е. нулевые итерации)

Тест 2: спроектировать тест, в котором переменная управления контуром будет отрицательной (отрицательное число итераций)

Тест 3: спроектировать тест, в котором цикл повторяется только один раз

Тест 4: спроектировать тест, в котором цикл повторяется дважды

Тест 5: спроектируйте тест, в котором цикл повторяется определенное количество раз, скажем m, где m <максимальное число возможных итераций

Тест 6: спроектируйте тест, в котором цикл повторяет одно меньше максимального числа итераций

Тест 7: спроектировать тест, в котором цикл выполняет итерацию максимального количества итераций

Тест 8: спроектируйте тест, в котором цикл повторяет более, чем максимальное количество итераций

Рассмотрим приведенный ниже пример кода, который применяет все указанные условия.

открытый класс SimpleLoopTest {

private int [] numbers = {5, -77,8, -11,4,1, -20,6,2,10};

```
/** Compute total of  positive numbers in the array
 *  @param numItems number of items to total.
 */
public int findSum(int numItems)
{
    int total = 0;
    if (numItems <= 10)
    {
        for (int count=0; count < numItems; count = count + 1)
        {
          if (numbers[count] > 0)
             {
                total = total + numbers[count];
             }
        }
    }
    return total;
}
```

}

открытый класс TestPass расширяет TestCase {

```
public void testname() throws Exception {

    SimpleLoopTest s = new SimpleLoopTest();
    assertEquals(0, s.findSum(0));    //Test 1
    assertEquals(0, s.findSum(-1));   //Test 2
    assertEquals(5, s.findSum(1));    //Test 3
    assertEquals(5, s.findSum(2));    //Test 4
    assertEquals(17, s.findSum(5));   //Test 5
    assertEquals(26, s.findSum(9));   //Test 6
    assertEquals(36, s.findSum(10));  //Test 7
    assertEquals(0, s.findSum(11));   //Test 8
}
```

}

Тест вложенных циклов

Вложенный цикл представляет собой цикл внутри цикла.

Внешний контур изменяется только после полного завершения / прерывания внутреннего цикла.

В этом случае тестовые примеры должны быть сконструированы таким образом, чтобы

Начните с самого внутреннего цикла. Установите все внешние петли на их минимальные значения. Выполните Простые проверки циклов в самом внутреннем цикле (Test3 / Test4 / Test5 / Test6 / Test7). Продолжайте до тех пор, пока все петли не пройдут проверку

Тест конкатенированных петель

Две петли конкатенированы, если можно достичь одного после выхода из другого на том же пути от входа до выхода. Иногда эти две петли независимы друг от друга. В этих случаях мы можем применять методы проектирования, указанные как часть однотактного тестирования.

Но если значения итерации в одном цикле прямо или косвенно связаны с итерационными значениями другого цикла, и они могут встречаться на одном пути, то мы можем рассматривать их как вложенные циклы.

Прочитайте Единичное тестирование циклов (Java) онлайн: https://riptutorial.com/ru/unit-testing/topic/10116/единичное-тестирование-циклов--java-

глава 4: Испытательные пары

замечания

При тестировании иногда полезно использовать двойной тест для управления или проверки поведения тестируемой системы. Дублисты передаются или вводятся в тестируемый класс или метод вместо экземпляров производственного кода.

Examples

Использование заглушки для подачи консервированных ответов

Штук - это двойной тест с легким весом, который предоставляет консервированные ответы при вызове методов. В тех случаях, когда тестируемый класс использует интерфейс или базовый класс, альтернативный класс «заглушка» может быть реализован для тестирования, который соответствует интерфейсу.

Итак, предполагая следующий интерфейс,

```
public interface IRecordProvider {
    IEnumerable<Record> GetRecords();
}
```

Если бы был протестирован следующий метод

```
public bool ProcessRecord(IRecordProvider provider)
```

Класс заглушки, который реализует интерфейс, может быть записан для возврата известных данных тестируемому методу.

```
public class RecordProviderStub : IRecordProvider
{
    public IEnumerable<Record> GetRecords()
    {
        return new List<Record> {
            new Record { Id = 1, Flag=false, Value="First" },
            new Record { Id = 2, Flag=true, Value="Second" },
            new Record { Id = 3, Flag=false, Value="Third" }
        };
    }
}
```

Эта реализация заглушки может быть предоставлена тестируемой системе, чтобы влиять на ее поведение.

```
var stub = new RecordProviderStub();
```

```
var processed = sut.ProcessRecord(stub);
```

Используя насмешливую структуру как заглушку

Термины Mock и Stub часто могут смущаться. Одной из причин этого является то, что многие издевательские рамки также обеспечивают поддержку для создания Stubs без этапа проверки, связанного с Mocking.

Вместо того, чтобы писать новый класс для создания заглушки, как в примере «Использование заглушки для подачи консервированных ответов», вместо этого можно использовать насмешливые фреймворки.

Использование Moq:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

Это приводит к тому же поведению, что и ручная кодировка, и может быть поставлена в тестируемую систему аналогичным образом:

```
var processed = sut.ProcessRecord(stub.Object);
```

Использование фальшивой структуры для проверки поведения

Mocks используются, когда необходимо проверить взаимодействие между тестируемой системой и тестовыми двойниками. Необходимо проявлять осторожность, чтобы не создавать слишком хрупкие тесты, но издевательство может быть особенно полезно, когда метод тестирования просто организует другие вызовы.

Этот тест проверяет , что , когда метод испытуемый называется (`ProcessRecord`), что метод обслуживания (`UseValue`) вызывается для `Record` , где `Flag==true` . Для этого он устанавливает заглушку с законсервированными данными:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

Затем он устанавливает макет, который реализует интерфейс `IService` :

```
var mockService = new Mock<IService>();
mockService.Setup(service => service.UseValue(It.IsAny<string>())).Returns(true);
```

Затем они передаются в тестируемую систему и вызывается метод, подлежащий тестированию.

```
var sut = new SystemUnderTest(mockService.Object);

var processed = sut.ProcessRecord(stub.Object);
```

Затем макет может быть опрошен, чтобы проверить, что к нему был приложен ожидаемый вызов. В этом случае вызов `UseValue` с одним параметром «Second», который является значением из записи, где `Flag==true` .

```
mockService.Verify(service => service.UseValue("Second"));
```

Прочитайте Испытательные пары онлайн: https://riptutorial.com/ru/unit-testing/topic/615/испытательные-пары

глава 5: Общие правила модульного тестирования для всех языков

Вступление

При запуске с модульным тестированием возникают всевозможные вопросы:

Что такое модульное тестирование? Что такое SetUp и TearDown? Как я могу работать с зависимостями? Зачем вообще тестировать единицы измерения? Как сделать хорошие модульные тесты?

Эта статья ответит на все эти вопросы, поэтому вы можете начать модульное тестирование на любом желаемом языке.

замечания

Что такое модульное тестирование?

Единичное тестирование - это проверка кода, чтобы гарантировать выполнение задачи, которую она должна выполнять. Он проверяет код на самом низком уровне - индивидуальные методы ваших классов.

Что такое единица?

Любой дискретный модуль кода, который может быть протестирован изолированно. Большинство классов времени и их методы. Этот класс обычно называют «тест класса» (CUT) или «тест системы» (SUT)

Разница между тестированием модулей и интеграционным тестированием

Единичное тестирование - это испытание отдельного класса, полностью отличное от любых его зависимостей. Интеграционное тестирование - это испытание одного класса вместе с одной или несколькими его фактическими зависимостями.

SetUp и TearDown

Когда метод SetUp запускается перед каждым тестированием устройства и TearDown после каждого теста.

В общем случае вы добавляете все необходимые шаги в SetUp и все этапы очистки в TearDown. Но вы только делаете этот метод, если эти шаги необходимы для каждого теста. Если нет, то эти шаги будут предприняты в рамках конкретных тестов в разделе «Оформить».

Как работать с зависимостями

Много раз у класса была зависимость других классов от выполнения его методов. Чтобы быть в состоянии не зависеть от этих других классов, вы должны подделывать их. Вы можете сами создавать эти классы или использовать среду изоляции или макета. Рамка изоляции представляет собой набор кода, который позволяет легко создавать поддельные классы.

Поддельные классы

Любой класс, предоставляющий функциональность, достаточную для того, чтобы притворяться, что это зависимость, необходимая CUT. Есть два типа подделок: Stubs и Mocks.

- Штук: подделка, которая не влияет на прохождение или неудачу теста и что существует только для того, чтобы позволить запустить тест.
- Макет: подделка, которая отслеживает поведение CUT и передает или не проходит тест на основе этого поведения.

Почему модульное тестирование?

1. В модульном тестировании будут найдены ошибки

> Когда вы пишете полный набор тестов, которые определяют, каково ожидаемое поведение для данного класса, раскрывается все, что не ведет себя так, как ожидалось.

2. Тестирование модулей будет содержать ошибки

> Внесите изменения, которые вводят ошибку, и ваши тесты могут выявить ее в следующий раз, когда вы запустите свои тесты.

3. Тестирование устройства экономит время

> Написание модульных тестов помогает гарантировать, что ваш код работает с самого начала. Единичные тесты определяют, что должен делать ваш код, и поэтому вы не будете тратить время на написание кода, который делает то, чего он не должен делать. Никто не проверяет код, который они не считают, и вы должны что-то сделать, чтобы заставить себя думать, что он работает.

Проведите это время, чтобы написать модульные тесты.

4. Модульное тестирование дает покой

Вы можете запустить все эти тесты и знать, что ваш код работает так, как предполагается. Зная состояние вашего кода, что он работает, и что вы можете обновлять и улучшать его без страха, это очень хорошо.

5. Удостоверяющие документы проверяют правильное использование класса

Модульные тесты становятся простыми примерами того, как работает ваш код, что он должен делать и как правильно использовать тестируемый код.

Общие правила модульного тестирования

1. Для структуры единичного теста следуйте правилу AAA

Упорядочить:

Настройте предмет для тестирования. Как переменные, поля и свойства, позволяющие запустить тест, а также ожидаемый результат.

Действие: На самом деле вызовите метод, который вы тестируете

Утверждают:

Вызовите структуру тестирования, чтобы убедиться, что результат вашего «действия» - это то, что ожидалось.

2. Проверяйте одно в то время в изоляции

Все классы должны тестироваться изолированно. Они не должны зависеть от чего-либо другого, кроме как насмешек и окурков. Они не должны зависеть от результатов других тестов.

3. Сначала напишите простые тесты «прямо вниз по середине»

Первые тесты, которые вы пишете, должны быть простейшими. Они должны быть теми, которые в основном и легко иллюстрируют функциональность, которую вы пытаетесь написать. Затем, как только эти тесты пройдут, вы должны начать писать более сложные тесты, которые проверяют границы и границы вашего кода.

4. Напишите тесты, проверяющие края

После того, как основы будут протестированы, и вы знаете, что ваша базовая функциональность работает, вы должны проверить края. Хороший набор тестов

будет исследовать внешние границы того, что может случиться с данным методом.

Например:

- Что произойдет, если произойдет переполнение?
- Что делать, если значения равны нулю или ниже?
- Что делать, если они идут в MaxInt или MinInt?
- Что делать, если вы создаете дугу 361 градуса?
- Что произойдет, если вы пройдете пустую строку?
- Что произойдет, если строка имеет размер 2 ГБ?

5. Испытание через границы

Единичные тесты должны проверять обе стороны заданной границы. Перемещение через границы - это места, где ваш код может выйти из строя или выполнить непредсказуемым образом.

6. Если вы можете, проверьте весь спектр

Если это практично, проверьте весь набор возможностей для своей функциональности. Если он включает перечисляемый тип, проверьте функциональность с каждым из элементов в перечислении. Это может быть нецелесообразно проверять каждую возможность, но если вы можете проверить все возможности, сделайте это.

7. Если возможно, покройте каждый путь кода

Это тоже сложно, но если ваш код предназначен для тестирования, и вы используете инструмент покрытия кода, вы можете убедиться, что каждая строка вашего кода покрывается модульными тестами хотя бы один раз. Покрытие каждого пути кода не гарантирует, что ошибок нет, но он, безусловно, дает вам ценную информацию о состоянии каждой строки кода.

8. Напишите тесты, которые показывают ошибку, затем исправьте ее

Если вы обнаружите ошибку, напишите тест, который показывает это. Затем, вы легко можете исправить ошибку, отлаживая тест. Затем у вас есть хороший регрессионный тест, чтобы убедиться, что если ошибка вернется по какой-либо причине, вы сразу узнаете. Очень легко исправить ошибку, когда у вас есть простой, прямой тест для запуска в отладчике.

Боковое преимущество здесь в том, что вы протестировали свой тест. Поскольку вы видели, что тест терпит неудачу, а затем, когда вы видели его, вы знаете, что тест действительно в том смысле, что было доказано, что он работает правильно. Это делает его еще лучшим регрессионным тестом.

9. Сделайте каждый тест независимым друг от друга

Тесты никогда не должны зависеть друг от друга. Если ваши тесты должны выполняться в определенном порядке, вам нужно изменить тесты.

10. Напишите одно утверждение за тест

Вы должны написать одно утверждение за тест. Если вы не можете этого сделать, то преломляйте свой код, чтобы ваши события SetUp и TearDown использовались для правильного создания среды, чтобы каждый тест можно запускать индивидуально.

11. Назовите свои тесты четко. Не бойтесь длинных имен

Поскольку вы делаете одно утверждение за каждый тест, каждый тест может оказаться очень специфичным. Таким образом, не бойтесь использовать длинные полные имена тестов.

Длинное полное имя позволяет сразу узнать, какой тест не удалось и что именно пытался выполнить этот тест.

Длинные, четко названные тесты также могут документировать ваши тесты. Тест под названием «DividedByZeroShouldThrowException» документирует то, что делает код, когда вы пытаетесь разделить на ноль.

12. Испытайте, что каждое поднятое исключение действительно поднято

Если ваш код вызывает исключение, тогда напишите тест, чтобы убедиться, что каждое исключение, которое вы поднимаете, фактически поднимается, когда оно предполагается.

13. Избегайте использования CheckTrue или Assert.IsTrue

Избегайте проверки булевского состояния. Например, вместо проверки, если две вещи равны с CheckTrue или Assert.IsTrue, вместо этого используйте CheckEquals или Assert.IsEqual. Зачем? Из-за этого:

CheckTrue (Ожидаемый, Фактический). Это будет сообщать о чем-то вроде: «Некоторое испытание не выполнено: ожидалось, что True, но фактический результат был False».

Это ничего не говорит.

CheckEquals (ожидается, актуально)

Это скажет вам что-то вроде: «Некоторое испытание не выполнено: ожидалось 7, но фактический результат был 3.»

Используйте только CheckTrue или Assert.IsTrue, когда ваше ожидаемое значение на самом деле является булевым.

14. Постоянно проводите тесты

Запускайте свои тесты во время написания кода. Ваши тесты должны выполняться быстро, позволяя запускать их после незначительных изменений. Если вы не можете запускать свои тесты как часть вашего обычного процесса разработки, тогда что-то идет не так. Модульные тесты должны запускаться почти мгновенно. Если это не так, вероятно, потому, что вы не используете их изолированно.

15. Запустите свои тесты как часть каждой автоматической сборки

Так же, как вы должны проходить тест во время разработки, они также должны быть неотъемлемой частью непрерывного процесса интеграции. Сбой теста должен означать, что ваша сборка нарушена. Не позволяйте провальным испытаниям задерживаться. Считайте это сбоем сборки и исправьте его немедленно.

Examples

Пример простого модульного теста в C

В этом примере мы проверим метод суммирования простого калькулятора.

В этом примере мы протестируем приложение: ApplicationToTest. У этого есть класс под названием Calc. Этот класс имеет метод Sum ().

Метод Sum () выглядит так:

```
public void Sum(int a, int b)
{
    return a + b;
}
```

Единичный тест для проверки этого метода выглядит следующим образом:

```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

        //Act
```

```
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Прочитайте Общие правила модульного тестирования для всех языков онлайн: https://riptutorial.com/ru/unit-testing/topic/9947/общие-правила-модульного-тестирования-для-всех-языков

глава 6: Тестирование блока в Visual Studio для C

Вступление

Как создать единичный тестовый проект и модульные тесты и как запустить модульные тесты и инструмент покрытия кода.

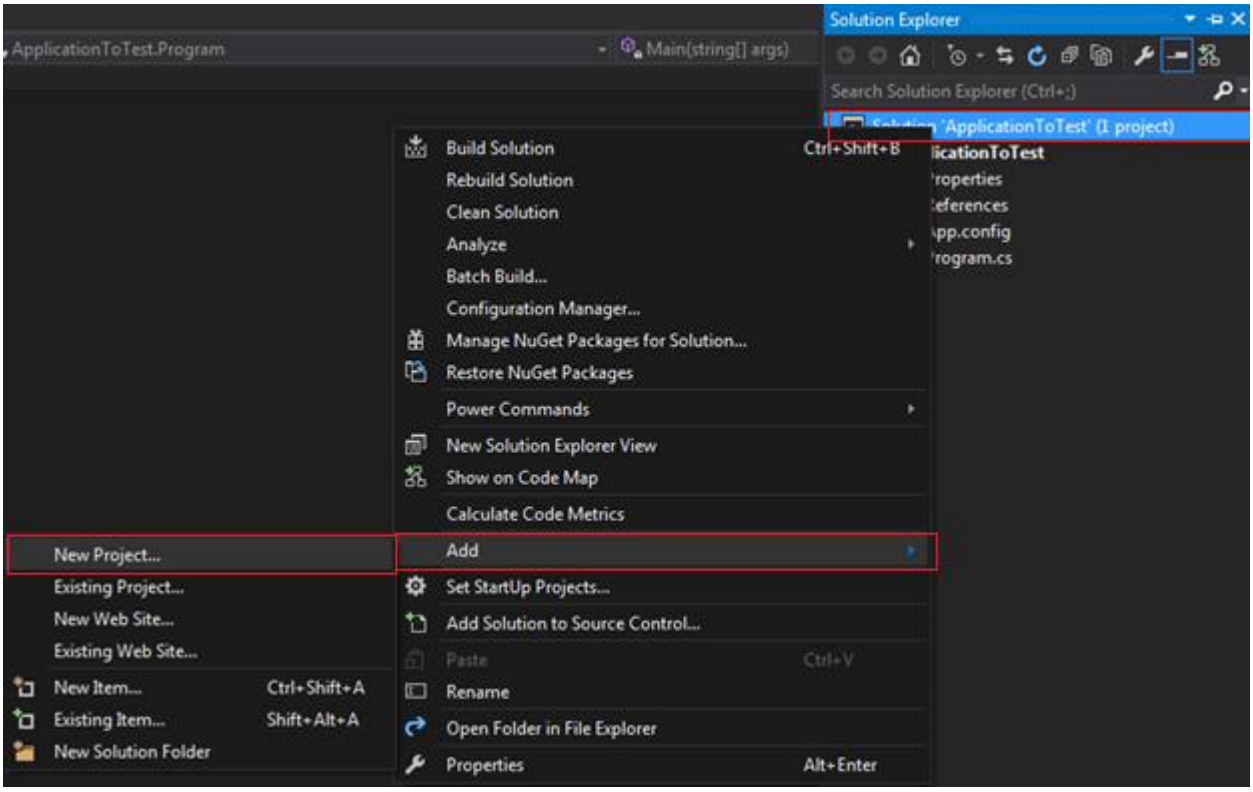
В этом руководстве будет использоваться стандартная среда MSTest и стандартный инструмент анализа кода, который доступен в Visual Studio.

Руководство было написано для Visual Studio 2015, поэтому, возможно, в других версиях разные вещи.

Examples

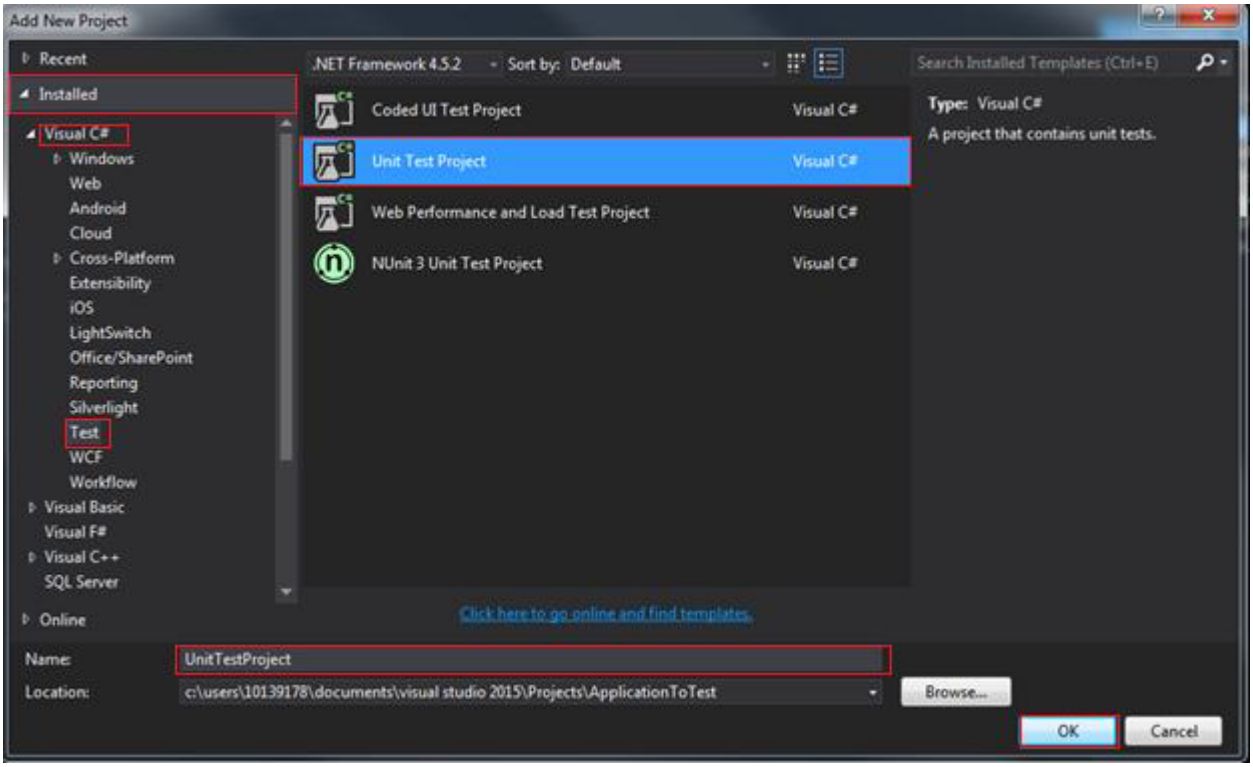
Создание единичного тестового проекта

- Откройте проект C #
- Щелкните правой кнопкой мыши по решению -> Добавить -> Новый проект ...
- (Рисунок 1)

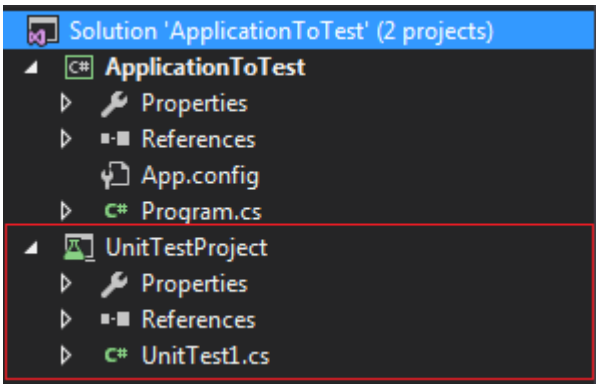


- Перейти к Installed -> Visual C # -> Test

- Нажмите «Проект тестирования единицы»
- Дайте ему имя и нажмите «OK».
- (Фигура 2)

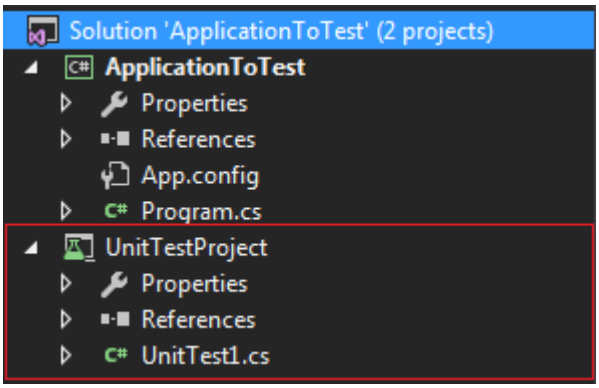


- Проект блока тестирования добавлен в решение
- (Рисунок 3)

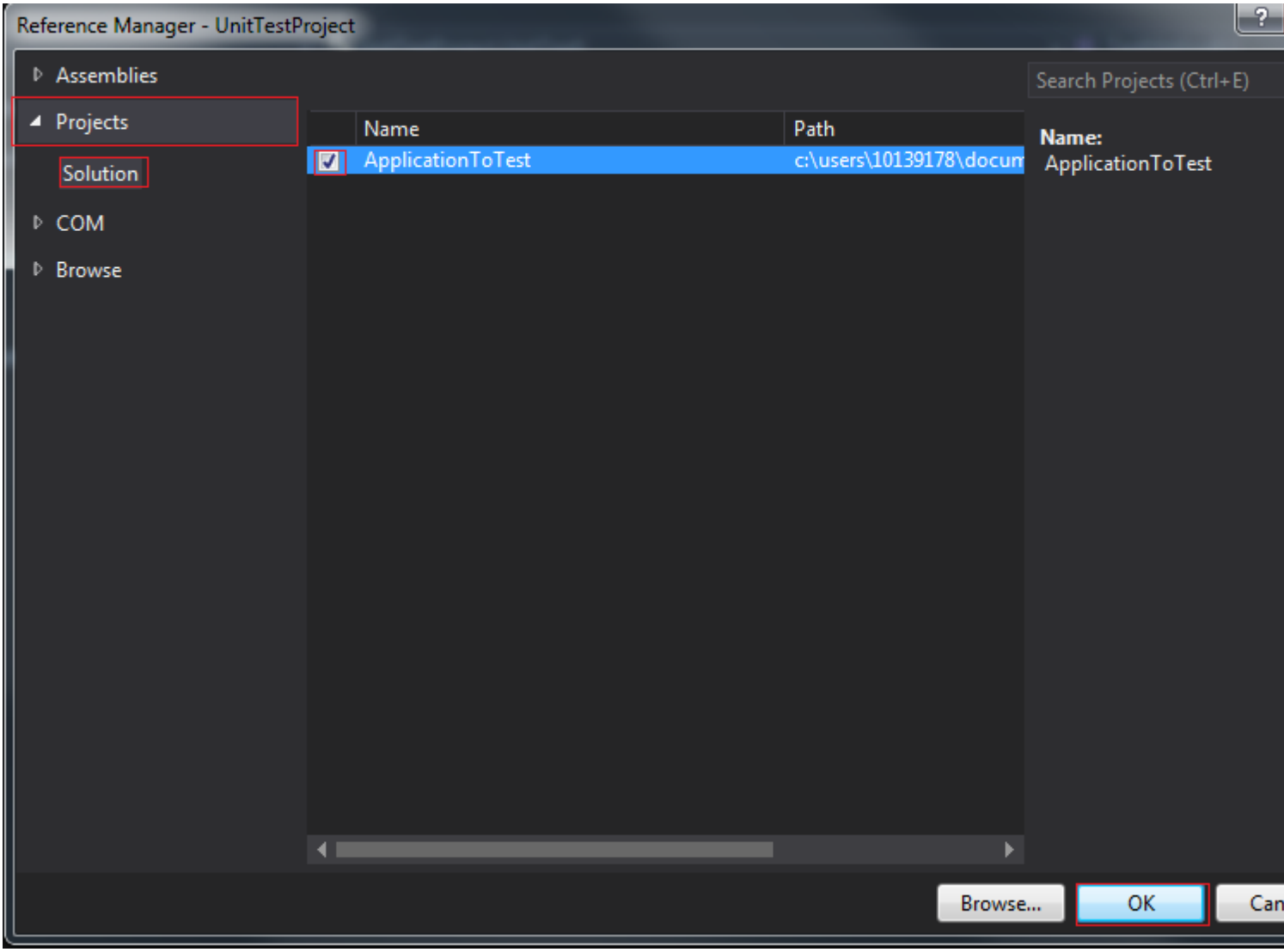


Добавление ссылки на приложение, которое вы хотите проверить

- В модульном тестовом проекте добавьте ссылку на проект, который вы хотите проверить
- Щелкните правой кнопкой мыши ссылку -> Добавить ссылку ...
- (Рисунок 3)



- Выберите проект, который вы хотите проверить.
- Перейти в Проекты -> Решение
- Установите флажок проекта, который вы хотите проверить, -> нажмите «OK».
- (Рисунок 4)



Два метода создания модульных тестов

Способ 1

- Перейдите на свой тестовый класс в модульном тестовом проекте
- Напишите единичный тест

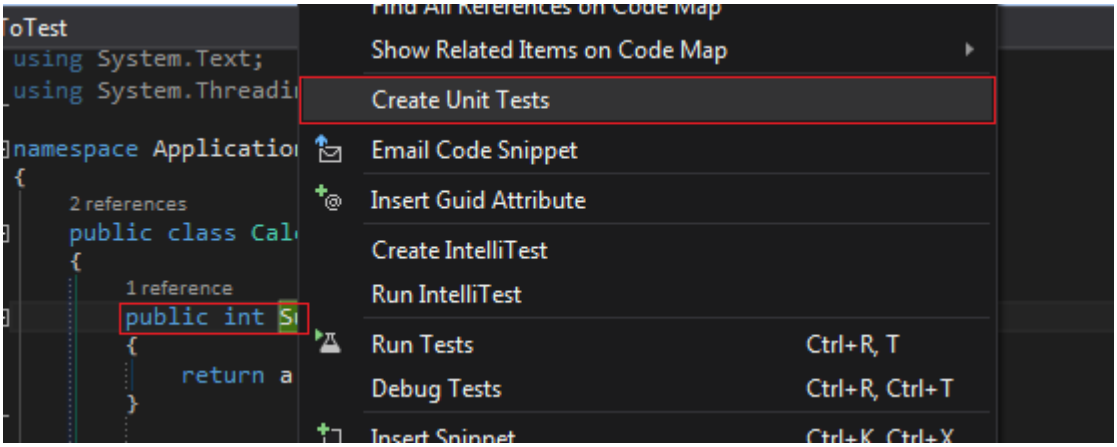
```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

        //Act
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Способ 2

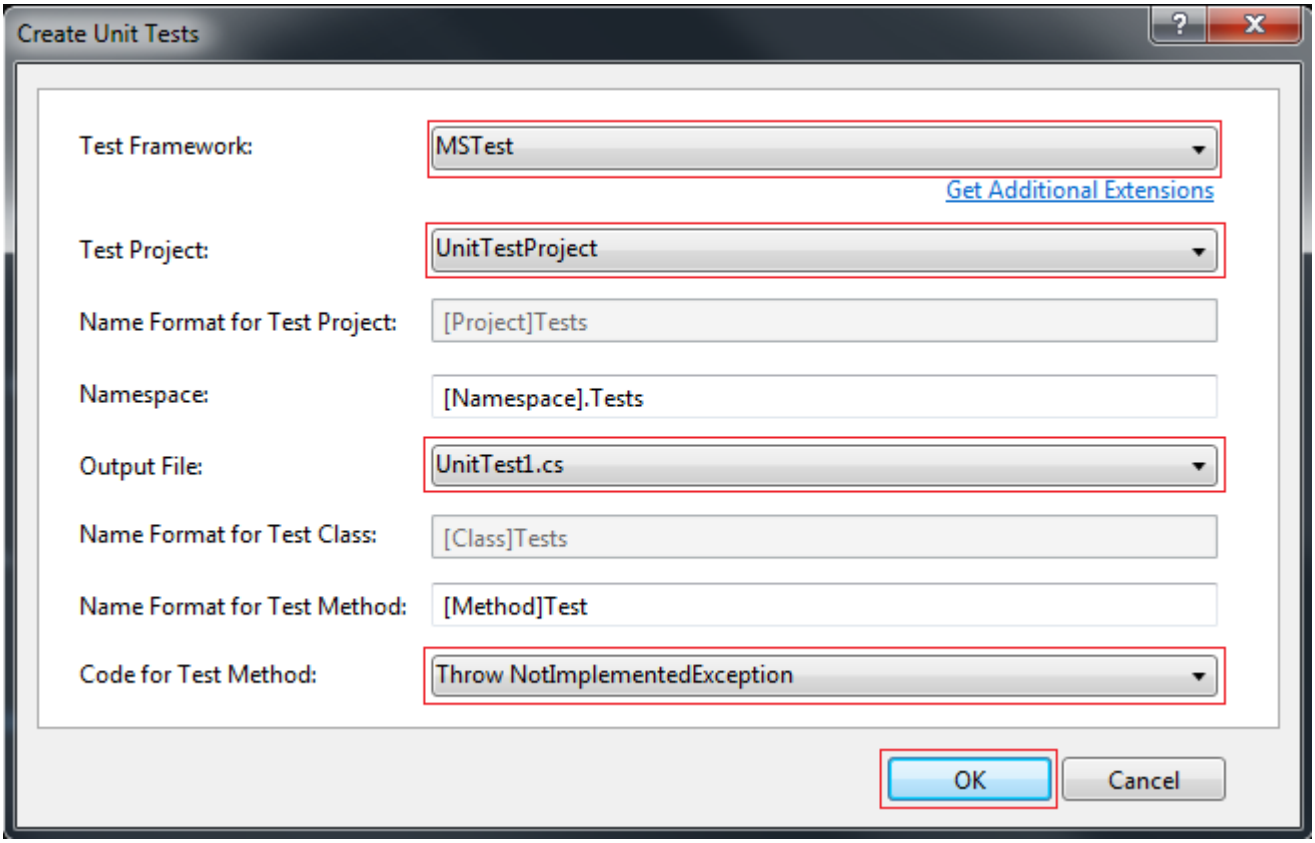
- Выполните метод, который хотите проверить
- Щелкните правой кнопкой мыши на методе -> Создать единичные тесты
- (Рисунок 4)



- Установить тестовую структуру в MSTest
- Задайте тестовый проект на название проекта тестирования устройства
- Установите выходной файл на имя класса модульных тестов
- Установите код для тестового метода на один из вариантов, которые вы предпочитаете
- Другие параметры можно редактировать, но это не обязательно

(Совет: если вы еще не сделали проект модульных тестов, вы все равно можете использовать эту опцию. Просто установите тестовый проект и выходной файл. Он создаст проект тестирования модулей, и он добавит ссылку проекта на единичный тестовый проект)

- (Рисунок 5)

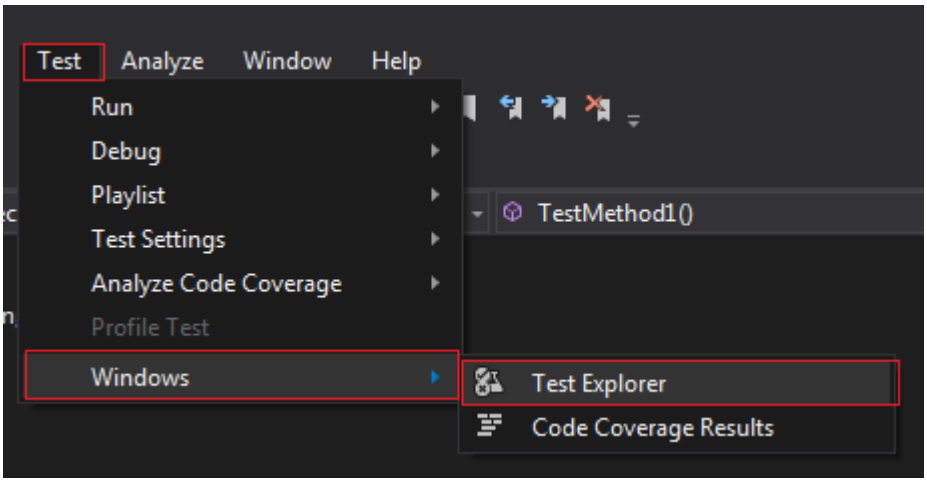


- Как вы видите ниже, он создает базу модульного теста для заполнения вами
- (Рисунок 6)

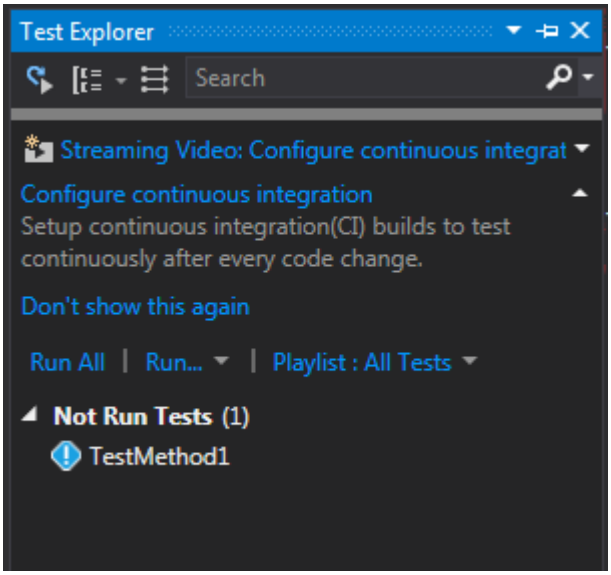
```
namespace ApplicationToTest.Tests
{
    [TestClass()]
    0 references
    public class UnitTest1
    {
        [TestMethod()]
        0 references
        public void SumTest()
        {
            throw new NotImplementedException();
        }
    }
}
```

Выполнение модульных тестов в Visual Studio

- Чтобы увидеть, что модульные тесты переходят в Test -> Windows -> Test Explorer
- (Рисунок 1)



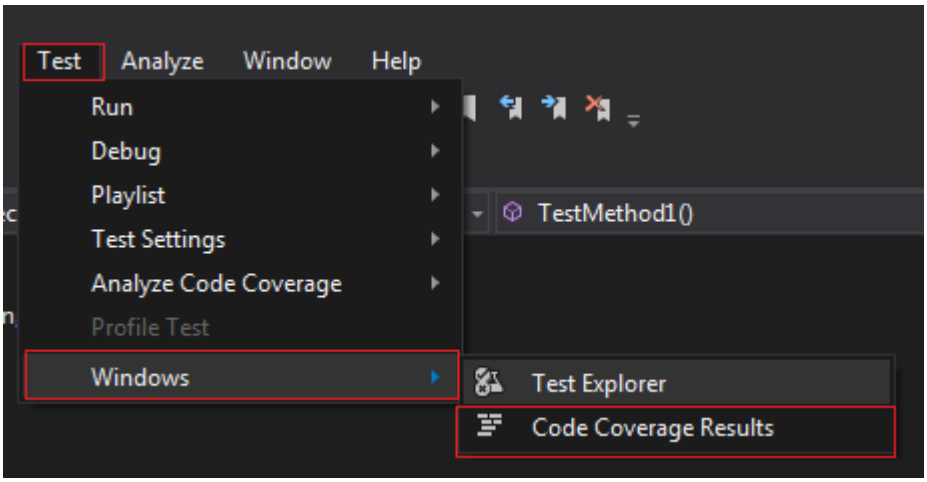
- Это откроет обзор всех тестов в приложении
- (Фигура 2)



- На рисунке выше вы можете видеть, что пример имеет один единичный тест, и он еще не запущен
- Вы можете дважды щелкнуть по тесту, чтобы перейти к коду, где определен единый тест
- Вы можете выполнить одно или несколько тестов с помощью «Запустить все» или «Запустить ...
- Вы также можете запускать тесты и изменять настройки в меню «Тест» (рисунок 1)

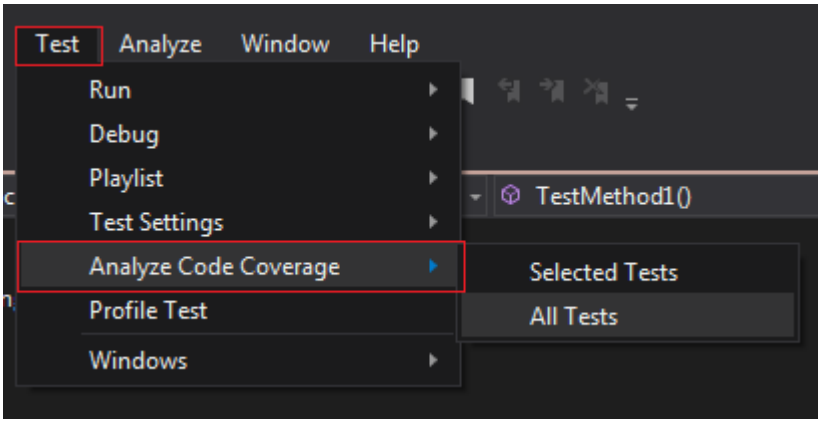
Анализ покрытия кода в Visual Studio

- Чтобы увидеть, что модульные тесты идут в Test -> Windows -> Результаты покрытия кода
- (Рисунок 1)

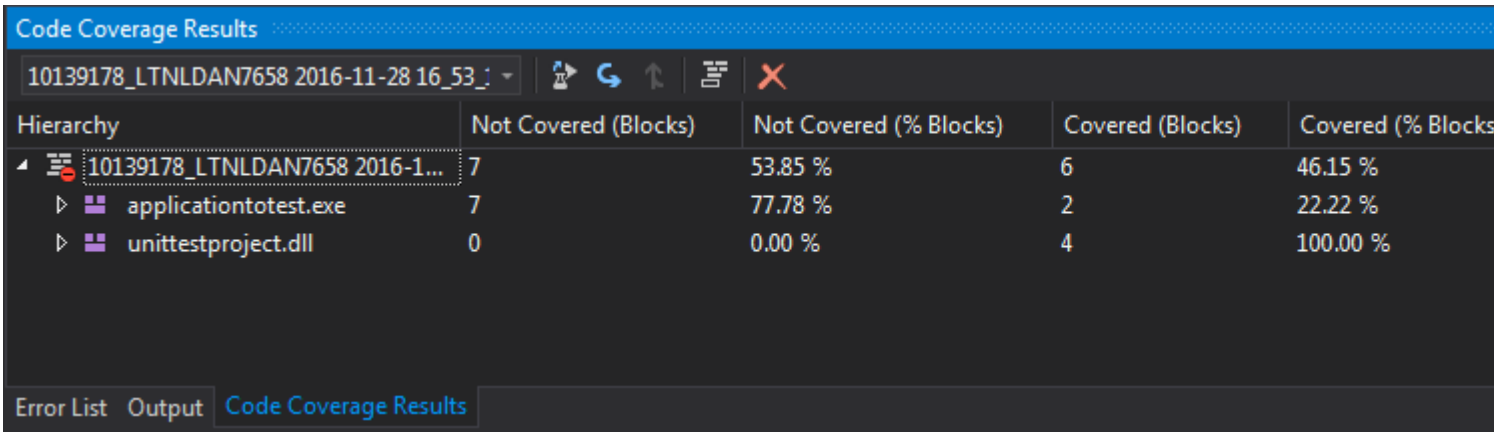


- Он откроет следующее окно
- (Фигура 2)

- Теперь окно пустое
- Перейдите в меню «Тест» -> «Анализ покрытия кода»
- (Рисунок 3)



- Тесты также будут запущены (см. Результаты в Test Explorer)
- Результаты будут показаны в таблице, в которой вы можете увидеть, какие классы и методы покрываются модульными тестами, а какие нет
- (Рисунок 4)



Прочитайте Тестирование блока в Visual Studio для C # онлайн: https://riptutorial.com/ru/unit-testing/topic/9953/тестирование-блока-в-visual-studio-для-c-sharp

глава 7: Тестирование единиц: наилучшая практика

Вступление

Единичный тест - это самая маленькая тестируемая часть приложения, такая как функции, классы, процедуры, интерфейсы. Модульное тестирование - это метод, с помощью которого тестируются отдельные единицы исходного кода, чтобы определить, подходят ли они для использования. Модульные тесты в основном написаны и выполняются разработчиками программного обеспечения, чтобы убедиться, что код соответствует его дизайну и требованиям и ведет себя так, как ожидалось.

Examples

Хорошее название

Важность хорошего именования может быть лучше всего проиллюстрирована некоторыми плохими примерами:

```
[Test]
Test1() {...} //Cryptic name - absolutely no information

[Test]
TestFoo() {...} //Name of the function - and where can I find the expected behaviour?

[Test]
TestTFSid567843() {...} //Huh? You want me to lookup the context in the database?
```

Хорошие тесты нуждаются в хороших именах. Хороший тест не тестирует методы, тестовые сценарии или требования.

Хорошее именование также предоставляет информацию о контексте и ожидаемом поведении. В идеале, когда тест не выполняется на вашей машине сборки, вы должны решить, что не так, не глядя на тестовый код или даже сложнее, имея необходимость отлаживать его.

Хорошее именование позволяет вам время для чтения кода и отладки:

```
[Test]
public void GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

Для новичков может оказаться полезным запустить тестовое имя с помощью **EnsureThat_**

или аналогичного префикса. Начните с «EnsureThat_», чтобы начать думать о сценарии или требовании, требующем теста:

```
[Test]
public void EnsureThat_GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void EnsureThat_GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

Именование важно для тестовых приборов. Назовите тестовое крепление после тестируемого класса:

```
[TestFixture]
public class OptionsTests //tests for class Options
{
    ...
}
```

Окончательный вывод:

Хорошее именование приводит к хорошим испытаниям, что приводит к хорошему дизайну в производственном коде.

От простого до сложного

То же самое, что и при написании классов - начните с простых случаев, затем добавьте требование (ака тесты) и реализацию (как производственный код) в каждом случае:

```
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf4() {...}
[Test]
public void EnsureThat_IsNOTLeapYearIfDecimalMultipleOf100 {...}
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf400 {...}
```

Не забудьте шаг рефакторинга, когда закончите с требованиями - сначала реорганизуйте код, а затем реорганизуйте тесты

Когда закончите, у вас должна быть полная, актуальная и READABLE документация вашего класса.

Концепция MakeSut

Тестовый код имеет те же требования к качеству, что и производственный код. MakeSut ()

- улучшает читаемость
- может быть легко реорганизован
- отлично поддерживает инъекцию зависимости.

Вот концепция:

```
[Test]
public void TestSomething()
{
    var sut = MakeSut();

    string result = sut.Do();
    Assert.AreEqual("expected result", result);
}
```

Простейший MakeSut () просто возвращает проверенный класс:

```
private ClassUnderTest MakeSUT()
{
    return new ClassUnderTest();
}
```

Когда нужны зависимости, их можно вводить здесь:

```
private ScriptHandler MakeSut(ICompiler compiler = null, ILogger logger = null, string
scriptName="", string[] args = null)
{
    //default dependencies can be created here
    logger = logger ?? MockRepository.GenerateStub<ILogger>();
    ...
}
```

Можно сказать, что MakeSut - это просто альтернатива методам настройки и удаления, предоставляемая платформами Testrunner, и, возможно, усугубляет эти методы лучше для тестовой настройки и разгона.

Каждый может решить самостоятельно, какой способ использовать. Для меня MakeSut () обеспечивает лучшую читаемость и большую гибкость. И последнее, но не менее важное: концепция не зависит от какой-либо рамки testrunner.

Прочитайте Тестирование единиц: наилучшая практика онлайн: https://riptutorial.com/ru/unit-testing/topic/6074/тестирование-единиц--наилучшая-практика

глава 8: Типы утверждений

Examples

Проверка возвращаемой стоимости

```
[Test]
public void Calculator_Add_ReturnsSumOfTwoNumbers()
{
    Calculator calculatorUnderTest = new Calculator();

    double result = calculatorUnderTest.Add(2, 3);

    Assert.AreEqual(5, result);
}
```

Государственное тестирование

Учитывая этот простой класс, мы можем проверить, что метод `ShaveHead` работает правильно, утверждая, что состояние переменной `HairLength` устанавливается равным нулю после `ShaveHead` метода `ShaveHead` .

```
public class Person
{
    public string Name;
    public int HairLength;

    public Person(string name, int hairLength)
    {
        this.Name = name;
        this.HairLength = hairLength;
    }

    public void ShaveHead()
    {
        this.HairLength = 0;
    }
}

[Test]
public void Person_ShaveHead_SetsHairLengthToZero()
{
    Person personUnderTest = new Person("Danny", 10);

    personUnderTest.ShaveHead();

    int hairLength = personUnderTest.HairLength;

    Assert.AreEqual(0, hairLength);
}
```

Подвергается проверке исключения

Иногда необходимо утверждать, когда генерируется исключение. В разных модульных модулях тестирования есть разные соглашения для утверждения о том, что было выбрано исключение (например, метод NUnit Assert.Throws). В этом примере не используются какие-либо специфичные для конкретной среды методы, только встроенные в обработку исключений.

```
[Test]
public void GetItem_NegativeNumber_ThrowsArgumentInvalidException
{
    ShoppingCart shoppingCartUnderTest = new ShoppingCart();
    shoppingCartUnderTest.Add("apple");
    shoppingCartUnderTest.Add("banana");

    double invalidItemNumber = -7;

    bool exceptionThrown = false;

    try
    {
        shoppingCartUnderTest.GetItem(invalidItemNumber);
    }
    catch(ArgumentInvalidException e)
    {
        exceptionThrown = true;
    }

    Assert.True(exceptionThrown);
}
```

Прочитайте Типы утверждений онлайн: https://riptutorial.com/ru/unit-testing/topic/6330/типы-утверждений

кредиты

S. No	Главы	Contributors
1	Начало работы с модульным тестированием	Andrey, Carl Manaster, Community, Farukh, forsvarir, Fred Kleuver, mahei, mark_h, Quill, silver, Stephen Byrne, Thomas Weller, zhon
2	Внедрение зависимости	forsvarir, kayess, mrAtari, Pavel Voronin, Stephen Byrne
3	Единичное тестирование циклов (Java)	Remya
4	Испытательные пары	forsvarir
5	Общие правила модульного тестирования для всех языков	DarkAngel
6	Тестирование блока в Visual Studio для C #	DarkAngel
7	Тестирование единиц: наилучшая практика	mrAtari, RamenChef, Shrinivas Patgar, user2314737
8	Типы утверждений	Danny