Kostenloses eBook

LERNEN unit-testing

Free unaffiliated eBook created from
Stack Overflow contributors.

#unit-testing

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit dem Komponententest

Bemerkungen

Unit-Tests beschreibt den Prozess, bei dem einzelne Codeeinheiten isoliert von dem System geprüft werden, zu dem sie gehören. Was eine Einheit ausmacht, kann von System zu System variieren und von einer einzelnen Methode bis zu einer Gruppe eng verwandter Klassen oder eines Moduls reichen.

Die Einheit wird von ihren Abhängigkeiten mit Hilfe von Testverdopplungen bei Bedarf isoliert und in einen bekannten Zustand versetzt. Ihr Verhalten als Reaktion auf Reize (Methodenaufrufe, Ereignisse, simulierte Daten) wird dann anhand des erwarteten Verhaltens getestet.

Unit-Tests für ganze Systeme können mit benutzerdefinierten schriftlichen Test-Kabeln durchgeführt werden. Es wurden jedoch viele Test-Frameworks geschrieben, um den Prozess zu rationalisieren und einen Großteil der Aufgaben im Bereich Sanitär-, Wiederholungs- und Alltagsaufgaben zu erledigen. Dadurch können sich die Entwickler auf das konzentrieren, was sie testen möchten.

Wenn für ein Projekt genügend Unit-Tests vorhanden sind, können Änderungen am Hinzufügen neuer Funktionen oder an einem Code-Refactoring problemlos durchgeführt werden, indem am Ende überprüft wird, dass alles wie zuvor funktioniert.

Code Coverage , normalerweise in Prozent ausgedrückt, ist die typische Metrik, die verwendet wird, um zu zeigen, wie viel Code in einem System von Unit Tests abgedeckt wird. Beachten Sie, dass es keine strenge Regel gibt, wie hoch dies sein sollte. Es wird jedoch allgemein akzeptiert, dass je höher, desto besser.

Test Driven Development (TDD) ist ein Prinzip, das vorsieht, dass ein Entwickler mit dem Codieren beginnen soll, indem er einen fehlerhaften Unit-Test schreibt und erst dann den Produktionscode schreibt, der den Test bestanden hat. Beim Üben von TDD kann gesagt werden, dass die Tests selbst der erste Benutzer des erstellten Codes sind. Sie helfen daher, das Design des Codes zu überprüfen und voranzutreiben, damit er so einfach zu verwenden und so robust wie möglich ist.

Versionen

Unit Testing ist ein Konzept, das keine Versionsnummern hat.

Examples

Ein grundlegender Unit-Test

Im einfachsten Fall besteht ein Komponententest aus drei Stufen:

- Bereiten Sie die Umgebung für den Test vor
- Führen Sie den zu testenden Code aus
- Überprüfen Sie, ob das erwartete Verhalten mit dem beobachteten Verhalten übereinstimmt

Diese drei Phasen werden häufig als Arrange-Act-Assert oder Gegeben-Wann-Dann-Zustand bezeichnet.

Unten ist ein Beispiel in C #, das das NUnit- Framework verwendet.

```
[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);
  }
}
```

Bei Bedarf wird eine optionale vierte Reinigungsstufe aufgeräumt.

Ein Unit-Test mit abgestumpfter Abhängigkeit

Gute Unit-Tests sind unabhängig, aber Code hat oft Abhängigkeiten. Wir verwenden verschiedene Arten von Testverdopplungen , um die Abhängigkeiten für das Testen zu entfernen. Einer der einfachsten Test-Doubles ist ein Stub. Dies ist eine Funktion mit einem hart codierten Rückgabewert, der anstelle der realen Abhängigkeit aufgerufen wird.

```
// 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
```

In Produktionscode `oneDayFromNow` nennen würde Date.now (), aber das wäre für widersprüchlich und unzuverlässig Tests machen. Also hier stummeln wir es raus.

Ein Unit-Test mit einem Spion (Interaktionstest)

Bei klassischen Einheitentests wird der *Status der* Tests geprüft. Es kann jedoch unmöglich sein,

Methoden richtig zu testen, deren Verhalten von anderen Klassen durch den Status abhängt. Wir testen diese Methoden durch *Interaktionstests* , mit denen sichergestellt wird, dass das getestete System seine Mitarbeiter korrekt aufruft. Da die Mitarbeiter ihre eigenen Komponententests haben, ist dies ausreichend und eigentlich ein besserer Test der tatsächlichen Verantwortung der getesteten Methode. Wir testen nicht, ob diese Methode bei einer Eingabe ein bestimmtes Ergebnis zurückgibt, sondern ruft die entsprechenden Mitarbeiter korrekt auf.

```
// 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
```

Einfacher Java + JUnit-Test

JUnit ist das führende Testframework zum Testen von Java-Code.

Die getestete Klasse modelliert ein einfaches Bankkonto, das eine Strafe verlangt, wenn Sie überzogen werden.

```
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
        }
    }
}
```

Diese `BankAccount` überprüft das Verhalten einiger öffentlicher Methoden von `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);
    }
}
```

Komponententest mit Parametern mit NUnit und 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);
        }
    }
}
```

Ein grundlegender Python-Unit-Test

```
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)
```

Nun zum Testteil:

```
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()
```

Ein XUnit-Test mit Parametern

```
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;
    }
}
```

Erste Schritte mit dem Komponententest online lesen: https://riptutorial.com/de/unit-testing/topic/570/erste-schritte-mit-dem-komponententest

Kapitel 2: Abhängigkeitsspritze

Bemerkungen

Ein Ansatz, der zum Schreiben von Software verwendet werden kann, besteht darin, Abhängigkeiten so zu erstellen, wie sie benötigt werden. Dies ist eine sehr intuitive Art, ein Programm zu schreiben, und die meisten Leute werden dazu neigen, unterrichtet zu werden, auch weil sie leicht zu verstehen sind. Eines der Probleme bei diesem Ansatz ist, dass er schwer zu testen sein kann. Stellen Sie sich eine Methode vor, die einige Verarbeitungsschritte basierend auf dem aktuellen Datum ausführt. Die Methode kann folgenden Code enthalten:

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

Der Code hängt direkt vom aktuellen Datum ab. Diese Methode kann schwer zu testen sein, da das aktuelle Datum nicht einfach manipuliert werden kann. Eine Möglichkeit, den Code besser testbar zu machen, besteht darin, den direkten Verweis auf das aktuelle Datum zu entfernen und stattdessen das aktuelle Datum an die Methode zu übergeben (oder zu injizieren), die die Verarbeitung durchführt. Diese Abhängigkeitsinjektion kann das Testen von Code-Aspekten durch Verwendung von Test-Doubles vereinfachen, um den Einrichtungsschritt des Komponententests zu vereinfachen.

IOC-Systeme

Ein weiterer zu berücksichtigender Aspekt ist die Lebensdauer von Abhängigkeiten. Wenn die Klasse selbst ihre eigenen Abhängigkeiten (auch als Invarianten bezeichnet) erstellt, ist sie für deren Beseitigung verantwortlich. Abhängigkeitsinjektion invertiert dies (weshalb wir eine Injektionsbibliothek häufig als "Inversion of Control" -System bezeichnen) und bedeutet, dass anstelle der Klasse, die für das Erstellen, Verwalten und Bereinigen ihrer Abhängigkeiten verantwortlich ist, ein externer Agent (in diesem Fall) In diesem Fall übernimmt das IoC-System dies.

Dies macht es viel einfacher, Abhängigkeiten zu haben, die von Instanzen derselben Klasse gemeinsam genutzt werden. Stellen Sie sich beispielsweise einen Dienst vor, der Daten von einem HTTP-Endpunkt abruft, damit eine Klasse sie konsumieren kann. Da dieser Dienst zustandslos ist (dh keinen internen Status hat), benötigen wir in der gesamten Anwendung nur eine einzige Instanz dieses Dienstes. Es ist zwar möglich (z. B. durch Verwendung einer statischen Klasse), dies manuell durchzuführen, es ist jedoch viel einfacher, die Klasse zu erstellen und dem IoC-System mitzuteilen, dass es als *Singleton erstellt werden soll* , wobei nur eine Instanz der Klasse vorhanden ist.

Ein anderes Beispiel wären Datenbankkontexte, die in einer Webanwendung verwendet werden, wobei ein neuer Kontext pro Anforderung (oder Thread) und nicht pro Instanz eines Controllers erforderlich ist. Dadurch kann der Kontext in jede von diesem Thread ausgeführte Ebene

eingefügt werden, ohne dass manuell weitergegeben werden muss.

Dies befreit die konsumierenden Klassen von der Verwaltung der Abhängigkeiten.

Examples

Konstruktorinjektion

Konstruktorinjektion ist der sicherste Weg, Abhängigkeiten zu injizieren, von denen eine ganze Klasse abhängt. Solche Abhängigkeiten werden oft als *Invarianten bezeichnet* , da eine Instanz der Klasse nicht erstellt werden kann, ohne sie anzugeben. Durch die Anforderung, dass die Abhängigkeit bei der Konstruktion eingefügt wird, ist sichergestellt, dass ein Objekt nicht in einem inkonsistenten Zustand erstellt werden kann.

Stellen Sie sich eine Klasse vor, die unter Fehlerbedingungen in eine Protokolldatei schreiben muss. Es hängt von einem `ILogger` , der injiziert und bei Bedarf verwendet werden kann.

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

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

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

Beim Schreiben von Tests stellen Sie möglicherweise fest, dass der Konstruktor mehr Abhängigkeiten erfordert, als für einen getesteten Fall tatsächlich erforderlich sind. Je mehr solcher Tests Sie durchführen, desto wahrscheinlicher ist es, dass Ihre Klasse gegen das *Single Responsibility Principle* (SRP) verstößt. Aus diesem Grund ist es nicht empfehlenswert, das Standardverhalten für alle Mocks injizierter Abhängigkeiten in der Testklasseninitialisierungsphase zu definieren, da dies das potenzielle Warnsignal maskieren kann.

Das Einzige, was dafür aussieht, würde wie folgt aussehen:

```
[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());
}
```

Immobilieninjektion

Durch die Eigenschaftsinjektion können Klassenabhängigkeiten nach der Erstellung aktualisiert werden. Dies kann nützlich sein, wenn Sie die Objekterstellung vereinfachen möchten, aber dennoch zulassen, dass die Abhängigkeiten von Ihren Tests mit Testverdopplungen überschrieben werden.

Stellen Sie sich eine Klasse vor, die unter einer Fehlerbedingung in eine Protokolldatei geschrieben werden muss. Die Klasse weiß, wie ein Standard- `Logger` kann jedoch durch die Eigenschaftsinjektion überschrieben werden. Es lohnt sich jedoch zu beachten, dass durch die Verwendung der Eigenschaftsinjektion auf diese Weise diese Klasse eng mit einer genauen Implementierung von `ILogger` wird, in diesem Beispiel `ConcreteLogger` . Eine mögliche Problemumgehung könnte eine Fabrik sein, die die erforderliche ILogger-Implementierung zurückgibt.

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

    public ILogger Logger { get; set; }

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

In den meisten Fällen ist Konstruktorinjektion der Eigenschaftsinjektion vorzuziehen, da dadurch der Zustand des Objekts unmittelbar nach seiner Konstruktion besser gewährleistet werden kann.

Methode Injection

Die Methodeninjektion ist eine feinkörnige Methode, um Abhängigkeiten in die Verarbeitung einzubringen. Stellen Sie sich eine Methode vor, die einige Verarbeitungsschritte basierend auf dem aktuellen Datum ausführt. Das aktuelle Datum kann von einem Test nur schwer geändert werden. Daher ist es viel einfacher, ein Datum in die Methode zu übernehmen, die Sie testen möchten.

```
public void ProcessRecords(DateTime currentDate)
{
    foreach(var record in _records)
    {
        if (currentDate.Date > record.ProcessDate)
        {
            // Do some processing
        }
    }
}
```

Container / DI-Frameworks

Durch das Extrahieren von Abhängigkeiten aus Ihrem Code, sodass sie injiziert werden können, ist der Code einfacher zu testen, doch das Problem wird in der Hierarchie weiter nach oben gerückt und kann dazu führen, dass Objekte schwer zu erstellen sind. Verschiedene Frameworks zur Abhängigkeitsinjektion / Inversion von Kontrollcontainern wurden geschrieben, um dieses Problem zu lösen. Diese ermöglichen die Registrierung von Typzuordnungen. Diese Registrierungen werden dann verwendet, um Abhängigkeiten aufzulösen, wenn der Container aufgefordert wird, ein Objekt zu erstellen.

Betrachten Sie diese Klassen:

```
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)
    {
        // ...
    }
}
```

Um `SomeProcessor` zu `SomeProcessor` , sind sowohl eine Instanz von `ILogger` als auch `SimpleClass` erforderlich. Ein Container wie Unity hilft, diesen Prozess zu automatisieren.

Zuerst muss der Container erstellt werden und dann werden Mappings damit registriert. Dies erfolgt normalerweise nur einmal innerhalb einer Anwendung. Der Bereich des Systems, in dem dies auftritt, wird im Allgemeinen als *Kompositionswurzel bezeichnet*

```
// 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());
```

Nach der Konfiguration des Containers können Objekte erstellt und Abhängigkeiten automatisch aufgelöst werden:

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

Abhängigkeitsspritze online lesen: https://riptutorial.com/de/unit-testing/topic/597/abhangigkeitsspritze

Kapitel 3: Assertion-Typen

Examples

Überprüfen eines zurückgegebenen Werts

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

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

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

State Based Testing

In Anbetracht dieser einfachen Klasse können wir testen, ob die `ShaveHead` Methode ordnungsgemäß funktioniert, indem der Zustand der `HairLength` Variablen nach `ShaveHead` der `ShaveHead` Methode auf Null `ShaveHead` wird.

```
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);
}
```

Überprüfen, ob eine Ausnahme ausgelöst wurde

Manchmal ist es notwendig zu behaupten, wann eine Ausnahme ausgelöst wird. Verschiedene Einheitentest-Frameworks haben unterschiedliche Konventionen, um zu behaupten, dass eine Ausnahme ausgelöst wurde (z. B. die Assert.Throws-Methode von NUnit's). In diesem Beispiel werden keine rahmenspezifischen Methoden verwendet, die nur in die Ausnahmebehandlung integriert sind.

```
[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);
}
```

Assertion-Typen online lesen: https://riptutorial.com/de/unit-testing/topic/6330/assertion-typen

Kapitel 4: Die allgemeinen Regeln für Unit-Tests für alle Sprachen

Einführung

Beim Start mit Unit-Testing tauchen alle möglichen Fragen auf:

Was ist Unit-Testing? Was ist ein SetUp und TearDown? Wie gehe ich mit Abhängigkeiten um? Warum überhaupt Unit-Testing? Wie mache ich gute Komponententests?

In diesem Artikel werden alle diese Fragen beantwortet, sodass Sie mit Unit-Tests in jeder gewünschten Sprache beginnen können.

Bemerkungen

Was ist Unit-Test?

Unit-Tests sind das Testen von Code, um sicherzustellen, dass er die Aufgabe ausführt, die er ausführen soll. Es testet Code auf einer möglichst niedrigen Ebene - den individuellen Methoden Ihrer Klassen.

Was ist eine Einheit?

Jedes diskrete Codemodul, das isoliert getestet werden kann. Die meisten Zeitklassen und ihre Methoden. Diese Klasse wird im Allgemeinen als "Class Under Test" (CUT) oder "System Under Test" (SUT) bezeichnet.

Der Unterschied zwischen Unit-Tests und Integrationstests

Komponententest ist der Vorgang, bei dem eine einzelne Klasse unabhängig von ihren tatsächlichen Abhängigkeiten getestet wird. Integrationstests sind das Testen einer einzelnen Klasse zusammen mit einer oder mehreren ihrer tatsächlichen Abhängigkeiten.

Das SetUp und TearDown

Wenn die SetUp-Methode erstellt ist, wird sie vor jedem Komponententest und TearDown nach jedem Test ausgeführt.

Im Allgemeinen fügen Sie alle erforderlichen Schritte in SetUp und alle Bereinigungsschritte in TearDown hinzu. Sie erstellen diese Methode jedoch nur, wenn diese Schritte für jeden Test erforderlich sind. Wenn nicht, werden diese Schritte innerhalb der spezifischen Tests im Abschnitt "Anordnen" ausgeführt.

Umgang mit Abhängigkeiten

Oftmals hängt eine Klasse von anderen Klassen ab, um ihre Methoden auszuführen. Um sich nicht auf diese anderen Klassen verlassen zu können, müssen Sie diese fälschen. Sie können diese Klassen selbst erstellen oder ein Isolations- oder Mockup-Framework verwenden. Ein Isolationsframework ist eine Sammlung von Code, mit der gefälschte Klassen einfach erstellt werden können.

Gefälschte Kurse

Jede Klasse, die über ausreichend Funktionalität verfügt, um vorzugeben, dass es sich um eine Abhängigkeit handelt, die von einer CUT benötigt wird. Es gibt zwei Arten von Fälschungen: Stubs und Mocks.

- Ein Stub: Eine Fälschung, die keine Auswirkung auf das Bestehen oder Nichtbestehen des Tests hat und die nur existiert, um den Test auszuführen.
- Ein Mock: Eine Fälschung, die das Verhalten der CUT verfolgt und den Test aufgrund dieses Verhaltens besteht oder nicht besteht.

Warum Unit-Tests durchführen?

1. Unit-Tests werden Fehler finden

Wenn Sie eine vollständige Testreihe schreiben, die das erwartete Verhalten für eine bestimmte Klasse definiert, wird alles angezeigt, was sich nicht wie erwartet verhält.

2. Unit-Tests halten Fehler fern

Nehmen Sie eine Änderung vor, die einen Fehler einführt, und Ihre Tests können ihn schon beim nächsten Ausführen Ihrer Tests aufdecken.

3. Gerätetests sparen Zeit

Durch das Schreiben von Komponententests können Sie sicherstellen, dass Ihr Code von Anfang an wie geplant funktioniert. Unit-Tests definieren, was Ihr Code tun soll, und Sie werden keine Zeit damit verbringen, Code zu schreiben, der Dinge tut, die er nicht tun sollte. Niemand überprüft im Code, dass er nicht glaubt, dass er funktioniert, und Sie müssen etwas tun, damit Sie glauben, dass es funktioniert. Verbringen Sie diese Zeit, um Komponententests zu schreiben.

4. Unit-Tests geben Sicherheit

Sie können all diese Tests ausführen und wissen, dass Ihr Code wie erwartet funktioniert. Es ist eine sehr gute Sache, den Status Ihres Codes zu kennen, dass er funktioniert und dass Sie ihn ohne Angst umstellen und verbessern können.

5. Unit-Tests dokumentieren die ordnungsgemäße Verwendung einer Klasse

Komponententests sind einfache Beispiele dafür, wie Ihr Code funktioniert, was er tun soll und wie Sie den getesteten Code richtig verwenden.

Allgemeine Regeln für Komponententests

1. Befolgen Sie für den Aufbau eines Komponententests die AAA-Regel

Ordnen:

Richten Sie das zu testende Ding ein. Wie Variablen, Felder und Eigenschaften, um den Test sowie das erwartete Ergebnis auszuführen.

Act: Rufen Sie tatsächlich die Methode auf, die Sie testen

Behaupten:

Rufen Sie das Testframework auf, um zu überprüfen, ob das Ergebnis Ihrer "Aktion" das ist, was erwartet wurde.

2. Testen Sie eine Sache zur Zeit isoliert

Alle Klassen sollten isoliert getestet werden. Sie sollten sich nicht auf etwas anderes als die Spötter und Stummel verlassen. Sie sollten sich nicht auf die Ergebnisse anderer Tests verlassen.

3. Schreiben Sie zuerst einfache Tests in die Mitte

Die ersten Tests, die Sie schreiben, sollten die einfachsten Tests sein. Sie sollten die sein, die die Funktionalität, die Sie schreiben möchten, im Grunde und auf einfache Weise veranschaulicht. Sobald diese Tests bestanden sind, sollten Sie mit dem Schreiben der komplizierteren Tests beginnen, die die Kanten und Grenzen Ihres Codes testen.

4. Schreiben Sie Tests, die die Kanten testen

Nachdem Sie die Grundlagen getestet haben und wissen, dass Ihre Basisfunktionen funktionieren, sollten Sie die Kanten testen. Eine gute Reihe von Tests untersucht die äußeren Ränder der möglichen Methode.

Zum Beispiel:

- Was passiert bei Überlauf?
- Was ist, wenn die Werte auf Null oder darunter gehen?
- Was ist, wenn sie zu MaxInt oder MinInt gehen?
- Was ist, wenn Sie einen Bogen von 361 Grad erstellen?
- Was passiert, wenn Sie eine leere Zeichenfolge übergeben?
- Was passiert, wenn eine Zeichenfolge 2 GB groß ist?

5. Testen Sie über Grenzen hinweg

Unit-Tests sollten beide Seiten einer bestimmten Grenze testen. Wenn Sie sich über Grenzen hinweg bewegen, gibt es Bereiche, an denen der Code möglicherweise nicht funktioniert oder auf unvorhersehbare Weise funktioniert.

6. Wenn möglich, testen Sie das gesamte Spektrum

Wenn es praktisch ist, testen Sie die gesamten Möglichkeiten für Ihre Funktionalität. Wenn es sich um einen Aufzählungstyp handelt, testen Sie die Funktionalität mit jedem der Elemente in der Aufzählung. Es kann unpraktisch sein, jede Möglichkeit zu testen, aber wenn Sie jede Möglichkeit testen können, tun Sie es.

7. Decken Sie nach Möglichkeit jeden Codepfad ab

Dies ist ebenfalls eine Herausforderung, aber wenn Ihr Code zum Testen vorgesehen ist und Sie ein Code Coverage-Tool verwenden, können Sie sicherstellen, dass jede Zeile Ihres Codes mindestens einmal durch Komponententests abgedeckt wird. Wenn Sie jeden Codepfad abdecken, kann dies nicht garantieren, dass keine Fehler vorhanden sind. Sie erhalten jedoch wertvolle Informationen zum Status jeder Codezeile.

8. Schreiben Sie Tests, die einen Fehler enthüllen, und beheben Sie ihn dann

Wenn Sie einen Fehler finden, schreiben Sie einen Test, der ihn aufdeckt. Dann können Sie den Fehler leicht beheben, indem Sie den Test debuggen. Dann haben Sie einen schönen Regressionstest, um sicherzustellen, dass der Fehler sofort erkannt wird, wenn der Fehler aus irgendeinem Grund zurückkommt. Es ist wirklich einfach, einen Fehler zu beheben, wenn Sie einen einfachen, direkten Test im Debugger haben.

Ein Nebeneffekt hier ist, dass Sie Ihren Test getestet haben. Da Sie gesehen haben, dass der Test fehlgeschlagen ist und wenn Sie gesehen haben, dass der Test erfolgreich ist, wissen Sie, dass der Test gültig ist, da er sich als richtig erwiesen hat. Dies macht es zu einem noch besseren Regressionstest.

9. Machen Sie jeden Test unabhängig voneinander

Tests sollten niemals voneinander abhängen. Wenn Ihre Tests in einer bestimmten Reihenfolge ablaufen müssen, müssen Sie die Tests ändern.

10. Schreiben Sie einen Test pro Test

Sie sollten einen Test pro Test schreiben. Wenn Sie dies nicht tun können, refraktorieren Sie Ihren Code, sodass die SetUp- und TearDown-Ereignisse zum korrekten Erstellen der Umgebung verwendet werden, sodass jeder Test einzeln ausgeführt werden kann.

11. Benennen Sie Ihre Tests eindeutig. Fürchte dich nicht vor langen Namen

Da Sie eine Prüfung pro Test durchführen, kann jeder Test sehr spezifisch sein.

Scheuen Sie sich also nicht, lange, vollständige Testnamen zu verwenden.

Ein langer vollständiger Name informiert Sie sofort darüber, welcher Test fehlgeschlagen ist und was genau der Test versucht hat.

Lange, klar benannte Tests können Ihre Tests auch dokumentieren. Ein Test mit dem Namen "DividedByZeroShouldThrowException" dokumentiert genau, was der Code tut, wenn Sie versuchen, durch Null zu teilen.

12. Testen Sie, dass jede angehobene Ausnahme tatsächlich ausgelöst wird

Wenn Ihr Code eine Ausnahme auslöst, schreiben Sie einen Test, um sicherzustellen, dass jede Ausnahme, die Sie auslösen, tatsächlich ausgelöst wird, wenn sie beabsichtigt ist.

13. Vermeiden Sie die Verwendung von CheckTrue oder Assert.IsTrue

Vermeiden Sie die Prüfung auf einen booleschen Zustand. Wenn Sie beispielsweise prüfen, ob zwei Dinge mit CheckTrue oder Assert.IsTrue übereinstimmen, verwenden Sie stattdessen CheckEquals oder Assert.IsEqual. Warum? Deswegen:

CheckTrue (Expected, Actual) Dies wird in etwa Folgendes berichten: "Einige Tests sind fehlgeschlagen: Erwartete war wahr, aber das tatsächliche Ergebnis war False."

Das sagt dir nichts.

CheckEquals (Voraussichtlich, Ist)

Dies sagt Ihnen etwas wie: "Einige Tests sind fehlgeschlagen: Erwartete 7, aber das tatsächliche Ergebnis war 3."

Verwenden Sie CheckTrue oder Assert.IsTrue nur, wenn Ihr erwarteter Wert tatsächlich eine boolesche Bedingung ist.

14. Führen Sie ständig Ihre Tests durch

Führen Sie Ihre Tests aus, während Sie Code schreiben. Ihre Tests sollten schnell ablaufen, sodass Sie sie auch nach geringfügigen Änderungen ausführen können. Wenn Sie Ihre Tests nicht als Teil Ihres normalen Entwicklungsprozesses ausführen können, läuft etwas schief. Unit-Tests sollen fast sofort ablaufen. Wenn dies nicht der Fall ist, liegt dies wahrscheinlich daran, dass Sie sie nicht isoliert ausführen.

15. Führen Sie Ihre Tests als Teil jedes automatisierten Builds aus

So wie Sie einen Test während der Entwicklung ausführen sollten, sollten sie auch ein integraler Bestandteil Ihres kontinuierlichen Integrationsprozesses sein. Ein fehlgeschlagener Test sollte bedeuten, dass Ihr Build defekt ist. Lassen Sie die fehlgeschlagenen Tests nicht verweilen. Betrachten Sie es als Build-Fehler und beheben Sie es sofort.

Examples

Beispiel für einen einfachen Komponententest in C

In diesem Beispiel testen wir die Summenmethode eines einfachen Rechners.

In diesem Beispiel testen wir die Anwendung: ApplicationToTest. Dieser hat eine Klasse namens Calc. Diese Klasse hat eine Methode Sum ().

Die Methode Sum () sieht folgendermaßen aus:

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

Der Komponententest zum Testen dieser Methode sieht folgendermaßen aus:

```
[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);
    }
}
```

Die allgemeinen Regeln für Unit-Tests für alle Sprachen online lesen: https://riptutorial.com/de/unit-testing/topic/9947/die-allgemeinen-regeln-fur-unit-tests-fur-alle-sprachen

Kapitel 5: Test-Doubles

Bemerkungen

Beim Testen ist es manchmal nützlich, ein Test-Double zu verwenden, um das Verhalten des getesteten Systems zu manipulieren oder zu überprüfen. Die Doubles werden in die Klasse oder Methode, die getestet wird, anstelle von Produktionscode übergeben oder eingefügt .

Examples

Verwenden eines Stubs zur Bereitstellung von Dosenantworten

Bei einem Stub handelt es sich um ein Test-Double mit geringem Gewicht, das beim Aufruf von Methoden vorgefertigte Antworten liefert. Wenn eine getestete Klasse von einer Schnittstelle oder Basisklasse abhängt, kann eine alternative 'Stub'-Klasse zum Testen implementiert werden, die der Schnittstelle entspricht.

Also, unter der Annahme der folgenden Schnittstelle,

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

Wenn die folgende Methode getestet werden sollte

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

Eine Stub-Klasse, die die Schnittstelle implementiert, kann geschrieben werden, um bekannte Daten an die getestete Methode zurückzugeben.

```
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" }
        };
    }
}
```

Diese Stub-Implementierung kann dann dem getesteten System zur Verfügung gestellt werden, um dessen Verhalten zu beeinflussen.

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

Verwenden eines spöttischen Rahmens als Stichleitung

Die Begriffe Mock und Stub können oft verwirrt werden. Ein Grund dafür ist, dass viele Mocking-Frameworks auch die Erstellung von Stubs ohne den mit Mocking verbundenen Überprüfungsschritt unterstützen.

Anstatt eine neue Klasse zu schreiben, um einen Stub wie im Beispiel "Verwenden eines Stubs zum Bereitstellen von Antworten aus Dosen" zu implementieren, können stattdessen Mock-Frameworks verwendet werden.

Moq verwenden:

```
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" }
});
```

Dadurch wird dasselbe Verhalten wie mit dem handcodierten Stub erreicht und auf ähnliche Weise an das zu testende System geliefert:

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

Verwendung eines Mocking-Frameworks zur Überprüfung des Verhaltens

Mocks werden verwendet, wenn die Wechselwirkungen zwischen dem getesteten System und Testverdoppelungen überprüft werden müssen. Es muss sorgfältig darauf geachtet werden, dass nicht zu spröde Tests erstellt werden. Spottungen können jedoch besonders nützlich sein, wenn die zu testende Methode einfach andere Anrufe koordiniert.

Dieser Test überprüft , dass , wenn das Verfahren im Test genannt wird (`ProcessRecord`), dass die Service - Methode (`UseValue`) für die aufgerufen wird `Record` wo `Flag==true` . Dazu wird ein Stub mit Daten aus der Datenbank eingerichtet:

```
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" }
});
```

Dann wird ein Mock eingerichtet, der die `IService` Schnittstelle implementiert:

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

Diese werden dann an das zu testende System geliefert und die zu testende Methode wird aufgerufen.

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

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

Das Mock kann dann abgefragt werden, um zu überprüfen, ob der erwartete Anruf angefordert wurde. In diesem Fall ein Aufruf von `UseValue` mit einem Parameter "Second", dem Wert aus dem Datensatz, bei dem `Flag==true` .

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

Test-Doubles online lesen: https://riptutorial.com/de/unit-testing/topic/615/test-doubles

Kapitel 6: Testen von Einheiten in Visual Studio für C

Einführung

Erstellen von Komponententestprojekten und Komponententests sowie Ausführen des Komponententests und des Code-Coverage-Tools.

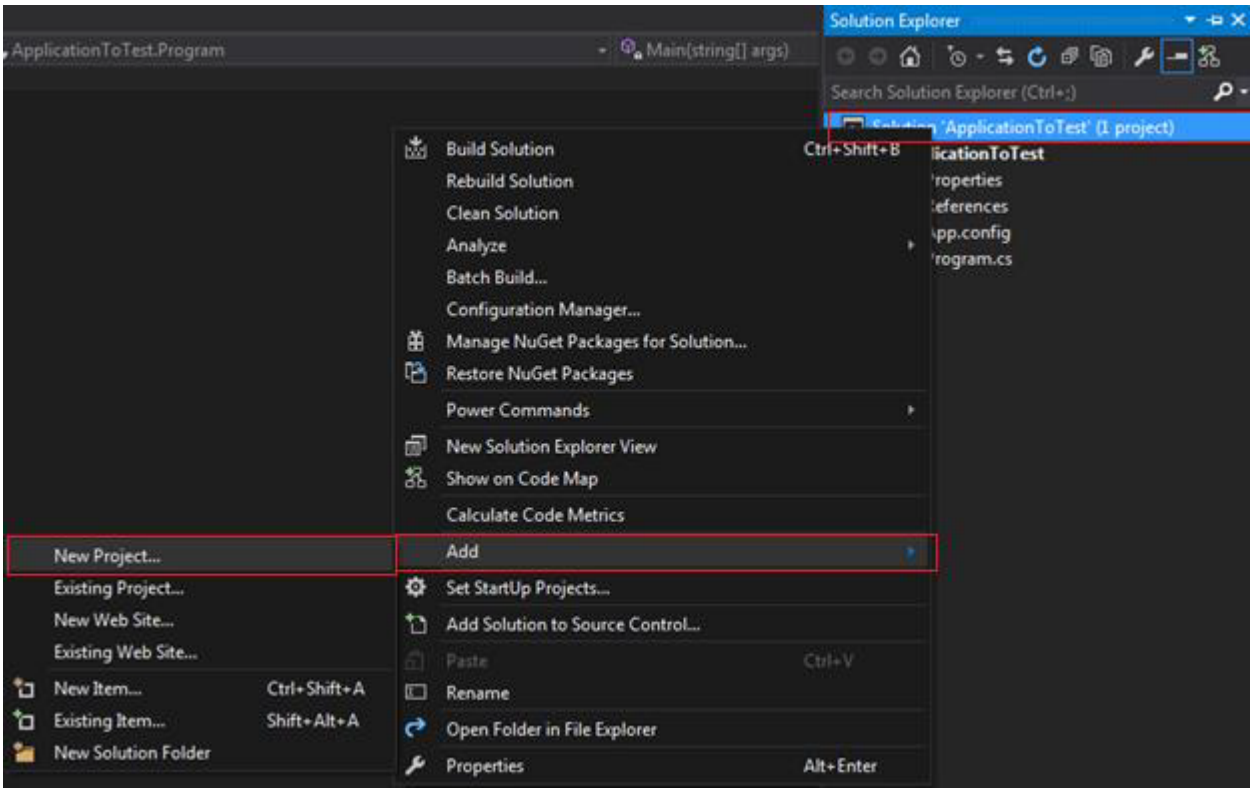
In diesem Handbuch wird das Standard-MSTest-Framework und das Standard-Tool zur Codeabdeckungsanalyse verwendet, die in Visual Studio verfügbar sind.

Der Leitfaden wurde für Visual Studio 2015 geschrieben. Es ist daher möglich, dass sich in anderen Versionen einige Unterschiede ergeben.

Examples

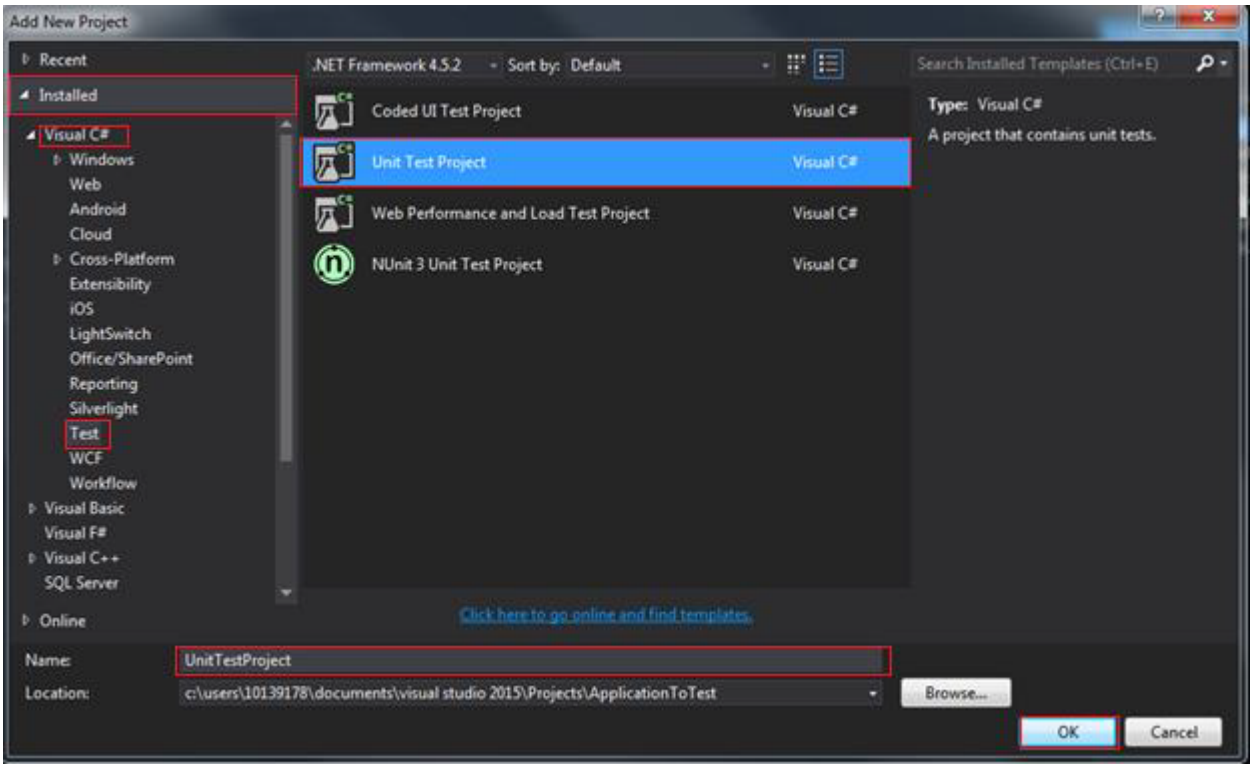
Erstellen eines Komponententestprojekts

- Öffnen Sie das C # -Projekt
- Klicken Sie mit der rechten Maustaste auf die Lösung -> Hinzufügen -> Neues Projekt…
- (Abbildung 1)

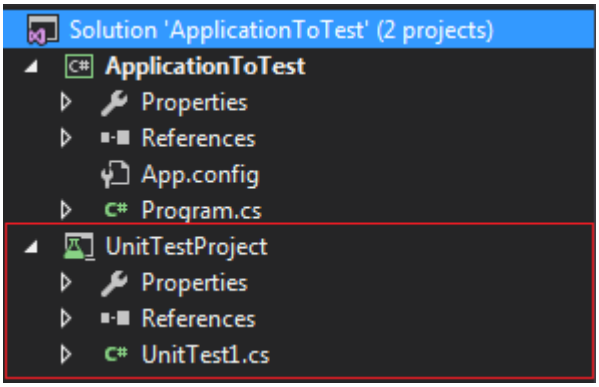


- Gehen Sie zu Installed -> Visual C # -> Test
- Klicken Sie auf Unit Test Project
- Vergeben Sie einen Namen und klicken Sie auf OK

- (Figur 2)

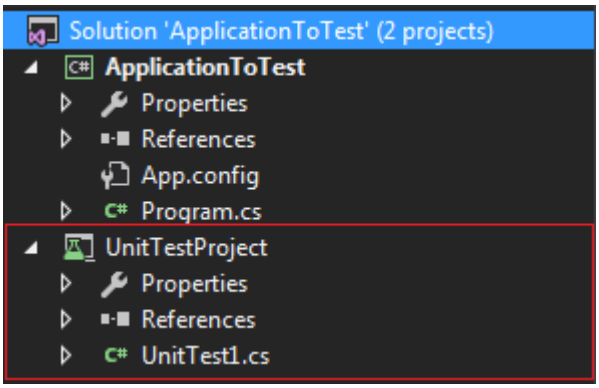


- Das Unit-Testprojekt wird der Lösung hinzugefügt
- (Figur 3)

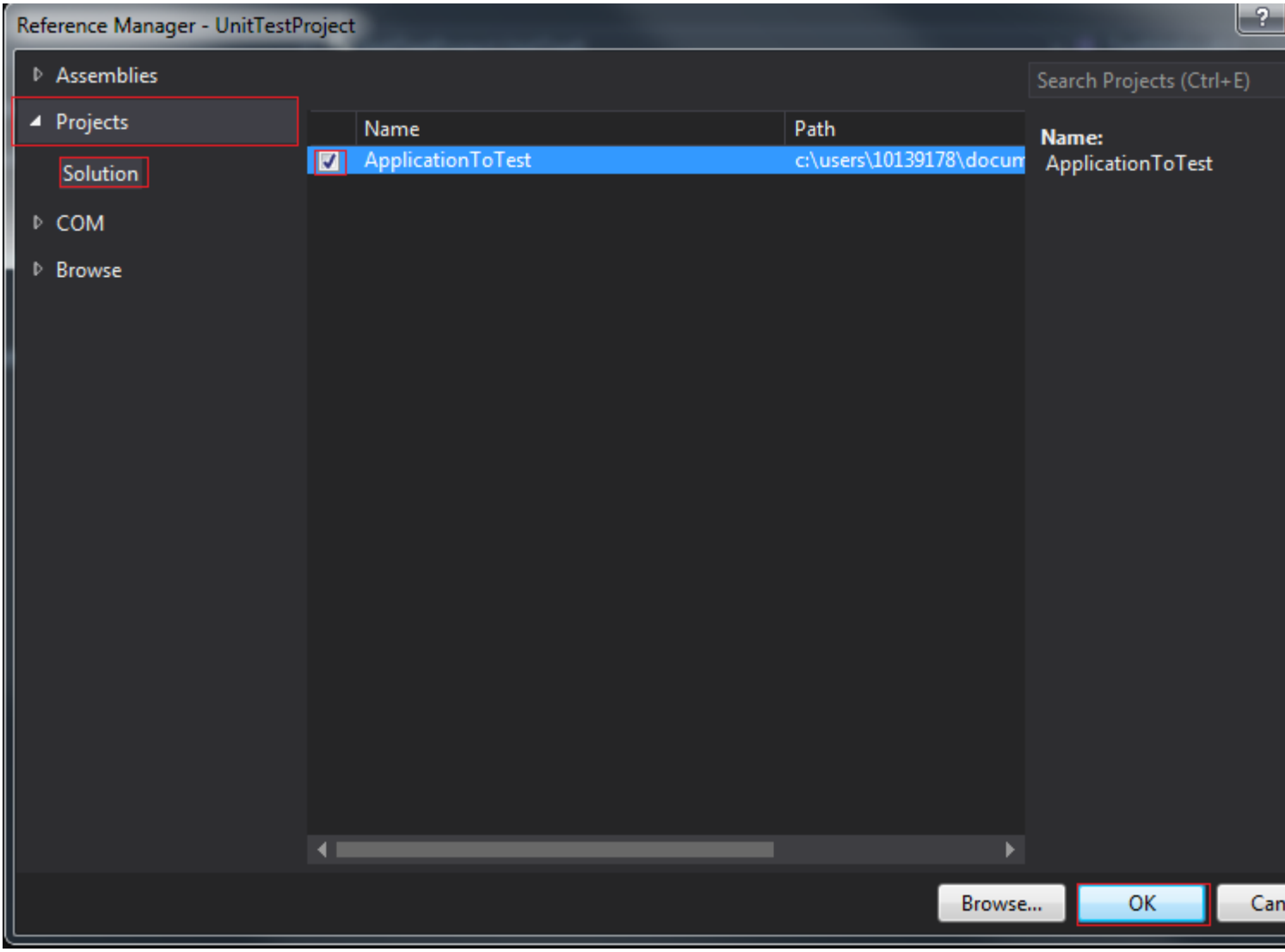


Hinzufügen des Verweises zu der Anwendung, die Sie testen möchten

- Fügen Sie im Komponententestprojekt einen Verweis auf das Projekt hinzu, das Sie testen möchten
- Klicken Sie mit der rechten Maustaste auf Referenzen -> Referenz hinzufügen…
- (Figur 3)



- Wählen Sie das Projekt aus, das Sie testen möchten
- Gehen Sie zu Projekte -> Lösung
- Aktivieren Sie das Kontrollkästchen des Projekts, das Sie testen möchten -> klicken Sie auf OK
- (Figur 4)



Zwei Methoden zum Erstellen von Komponententests

Methode 1

- Wechseln Sie zu Ihrer Komponententestklasse im Komponententestprojekt
- Schreiben Sie einen Komponententest

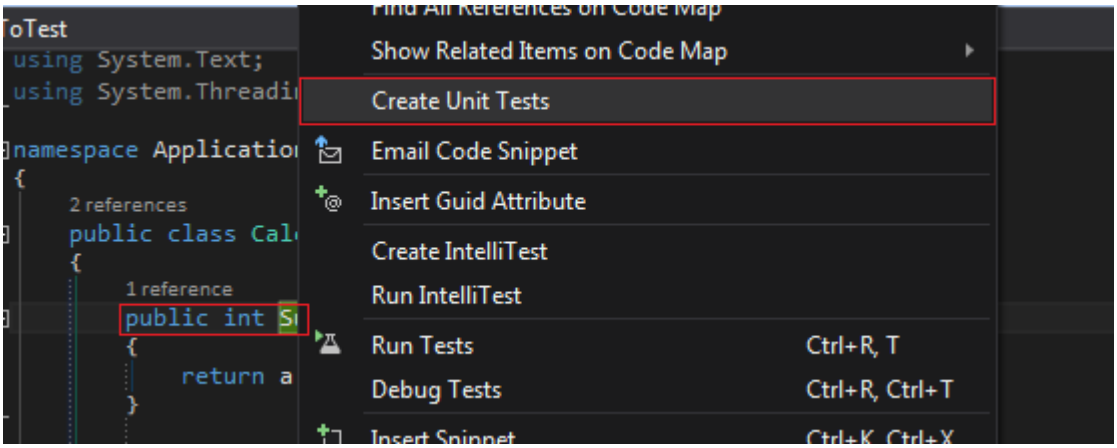
```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

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

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

Methode 2

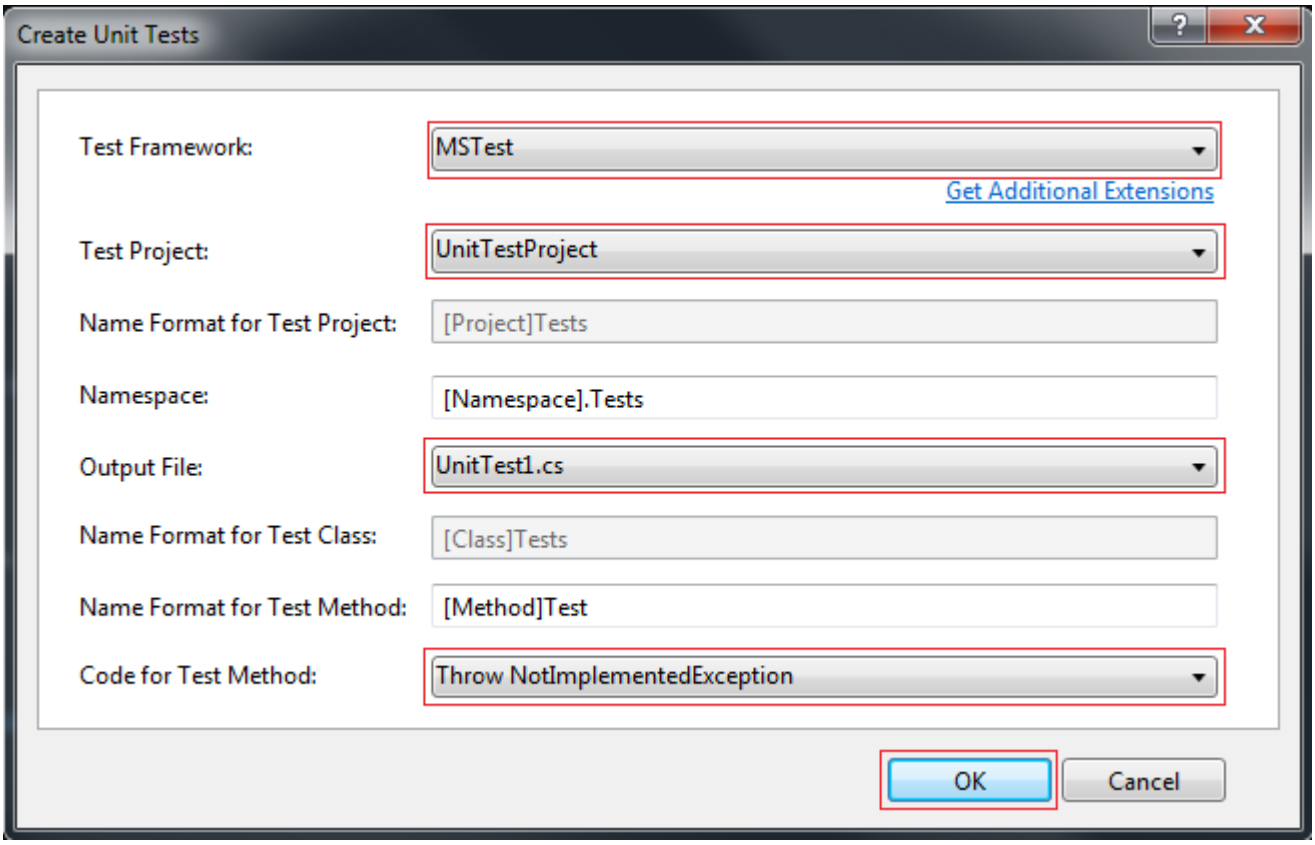
- Gehen Sie die Methode, die Sie testen möchten
- Klicken Sie mit der rechten Maustaste auf die Methode -> Unit-Tests erstellen
- (Figur 4)



- Legen Sie Test Framework auf MSTest fest
- Stellen Sie Testprojekt auf den Namen Ihres Gerätetestprojekts ein
- Legen Sie Ausgabedatei auf den Namen der Klasse der Komponententests fest
- Stellen Sie Code for Test Method auf eine der aufgelisteten Optionen ein
- Die anderen Optionen können bearbeitet werden, sind aber nicht erforderlich

(Tipp: Wenn Sie noch kein Unit-Test-Projekt erstellt haben, können Sie diese Option dennoch verwenden. Setzen Sie einfach Testprojekt auf und Ausgabedatei auf. Es erstellt das Unit-Test-Projekt und fügt die Referenz des Projekts zum hinzu Gerätetestprojekt)

- (Abbildung 5)

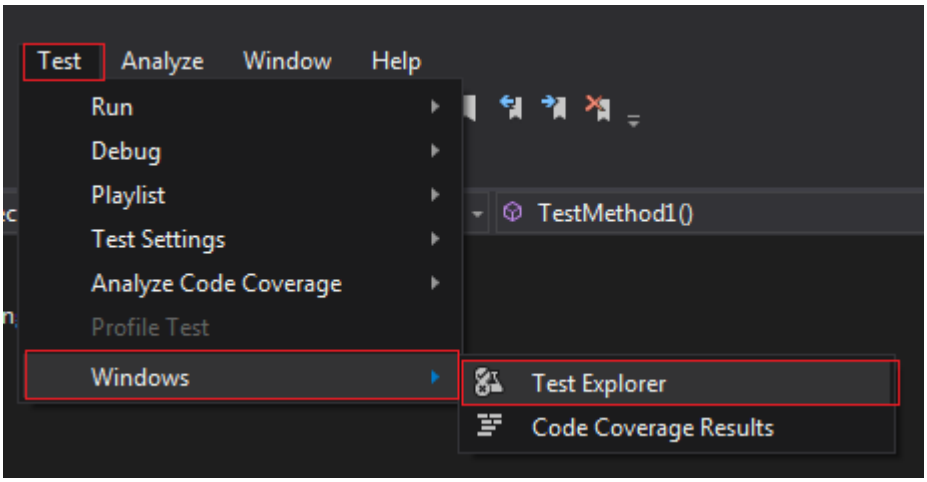


- Wie Sie unten sehen, wird die Basis des Komponententests erstellt, die Sie ausfüllen müssen
- (Figur 6)

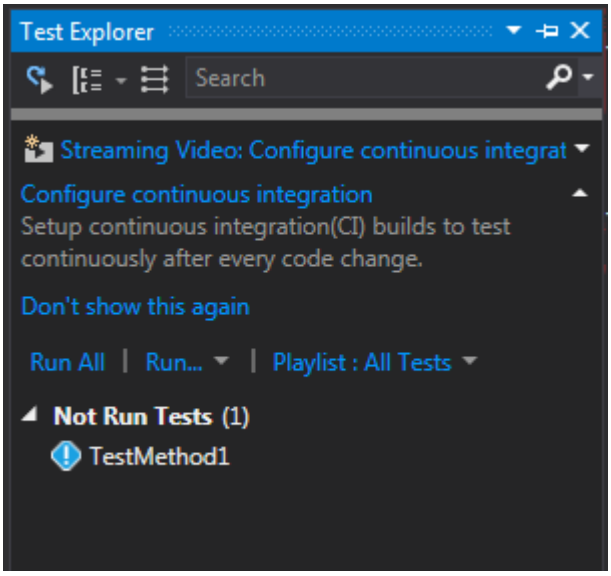
```
namespace ApplicationToTest.Tests
{
    [TestClass()]
    0 references
    public class UnitTest1
    {
        [TestMethod()]
        0 references
        public void SumTest()
        {
            throw new NotImplementedException();
        }
    }
}
```

Ausführen von Komponententests in Visual Studio

- Um Ihre Unit-Tests anzuzeigen, gehen Sie zu Test -> Windows -> Test Explorer
- (Abbildung 1)



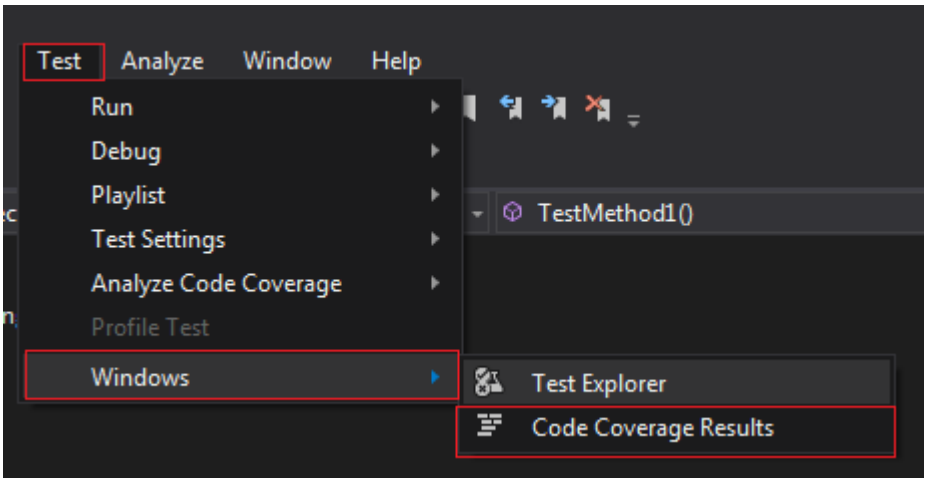
- Dadurch wird eine Übersicht aller Tests in der Anwendung geöffnet
- (Figur 2)



- In der Abbildung oben sehen Sie, dass das Beispiel nur einen Komponententest hat und noch nicht ausgeführt wurde
- Sie können auf einen Test doppelklicken, um zu dem Code zu gelangen, in dem der Komponententest definiert ist
- Sie können einzelne oder mehrere Tests mit dem Befehl Alles ausführen oder Ausführen ausführen.
- Sie können auch Tests ausführen und Einstellungen über das Testmenü ändern (Abbildung 1).

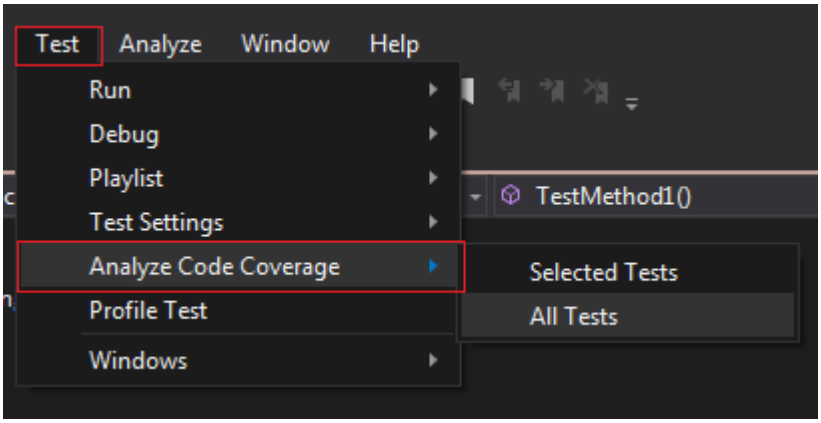
Ausführen der Codeabdeckungsanalyse in Visual Studio

- Um Ihre Unit-Tests anzuzeigen, gehen Sie zu Test -> Windows -> Code Coverage Results
- (Abbildung 1)

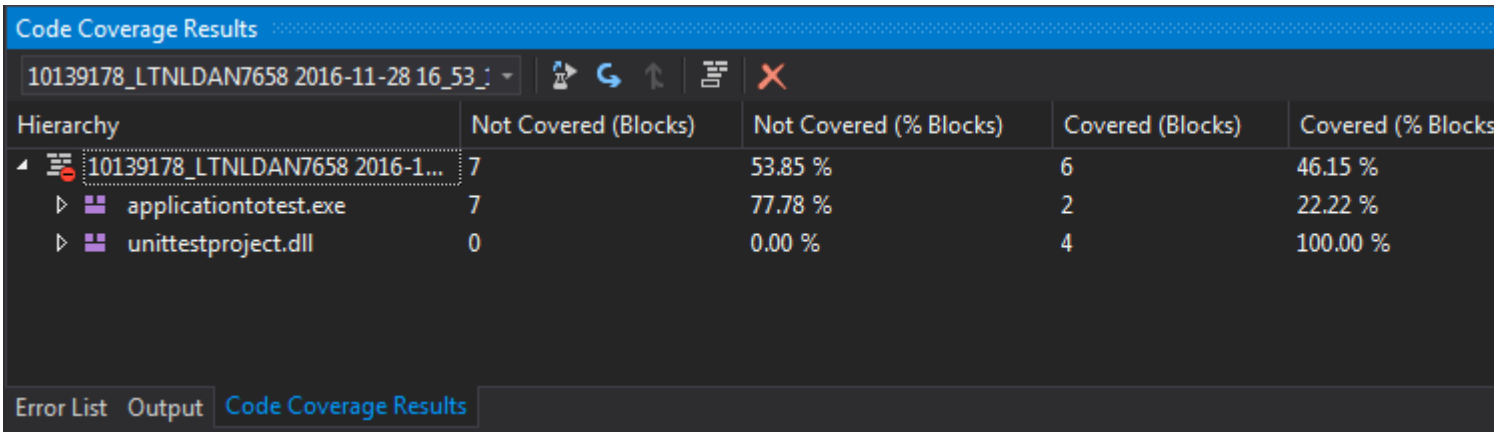


- Es öffnet sich folgendes Fenster
- (Figur 2)

- Das Fenster ist jetzt leer
- Gehen Sie zum Test-Menü -> Code Coverage analysieren
- (Figur 3)



- Die Tests werden nun ebenfalls ausgeführt (Siehe Ergebnisse im Test Explorer).
- Die Ergebnisse werden in einer Tabelle angezeigt, in der Sie sehen können, welche Klassen und Methoden von Komponententests abgedeckt werden und welche nicht
- (Figur 4)



Testen von Einheiten in Visual Studio für C # online lesen: https://riptutorial.com/de/unit-testing/topic/9953/testen-von-einheiten-in-visual-studio-fur-c-sharp

Kapitel 7: Unit Testing: Best Practices

Einführung

Ein Komponententest ist der kleinste überprüfbare Teil einer Anwendung wie Funktionen, Klassen, Prozeduren, Schnittstellen. Unit Testing ist eine Methode, mit der einzelne Einheiten des Quellcodes getestet werden, um zu bestimmen, ob sie für die Verwendung geeignet sind. Komponententests werden grundsätzlich von Software-Entwicklern geschrieben und ausgeführt, um sicherzustellen, dass der Code seinem Design und seinen Anforderungen entspricht und sich wie erwartet verhält.

Examples

Gute Benennung

Die Bedeutung einer guten Benennung lässt sich am besten anhand einiger schlechter Beispiele veranschaulichen:

```
[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?
```

Gute Tests brauchen gute Namen. Gute Tests testen keine Methoden, Testszenarien oder Anforderungen.

Eine gute Benennung liefert auch Informationen zum Kontext und zum erwarteten Verhalten. Wenn der Test auf Ihrer Build-Maschine fehlschlägt, sollten Sie im Idealfall entscheiden können, was falsch ist, ohne auf den Testcode zu achten.

Eine gute Benennung erspart Ihnen Zeit zum Lesen von Code und zum Debuggen:

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

Für Anfänger kann es hilfreich sein, den **Testnamen** mit **EnsureThat_** oder einem ähnlichen Präfix zu beginnen. Beginnen Sie mit einem "EnsureThat_", um über das Szenario oder die Anforderung nachzudenken, für die ein Test erforderlich ist:

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

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

Die Benennung ist auch für Testgeräte wichtig. Benennen Sie das Testgerät nach der getesteten Klasse:

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

Die endgültige Schlussfolgerung lautet:

Eine gute Benennung führt zu guten Tests, was zu einem guten Design im Produktionscode führt.

Von einfach bis komplex

Wie beim Schreiben von Klassen - beginnen Sie mit den einfachen Fällen und fügen Sie dann Anforderung (aka Tests) und Implementierung (auch Produktionscode) von Fall zu Fall hinzu:

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

Vergessen Sie nicht den Refactoring-Schritt, wenn Sie mit den Anforderungen fertig sind - zuerst den Code umgestalten und dann die Tests überarbeiten

Wenn Sie fertig sind, sollten Sie über eine vollständige, aktuelle und READABLE-Dokumentation Ihrer Klasse verfügen.

MakeSut-Konzept

Testcode hat die gleichen Qualitätsanforderungen wie der Produktionscode. MakeSut ()

- verbessert die Lesbarkeit
- kann leicht umgestaltet werden
- Unterstützt die Injektion von Abhängigkeiten perfekt.

Hier ist das Konzept:

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

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

Das einfachste MakeSut () gibt nur die getestete Klasse zurück:

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

Wenn Abhängigkeiten benötigt werden, können sie hier eingefügt werden:

```
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>();
    ...
}
```

Man könnte sagen, dass MakeSut nur eine einfache Alternative für Setup- und Teardown-Methoden ist, die von Testrunner-Frameworks bereitgestellt werden, und diese Methoden möglicherweise einen besseren Platz für testspezifische Setup- und Teardown-Methoden bieten.

Jeder kann selbst entscheiden, welchen Weg er verwenden soll. Für mich bietet MakeSut () eine bessere Lesbarkeit und mehr Flexibilität. Nicht zuletzt ist das Konzept unabhängig von einem Testrunner-Framework.

Unit Testing: Best Practices online lesen: https://riptutorial.com/de/unit-testing/topic/6074/unit-testing--best-practices

Kapitel 8: Unit-Test von Loops (Java)

Einführung

Schleifen gelten als eine der wichtigen Kontrollstrukturen in jeder Programmiersprache. Es gibt verschiedene Möglichkeiten, wie wir eine Schleifenabdeckung erreichen können.

Diese Methoden unterscheiden sich je nach Art der Schleife.

Einzelne Schleifen

Verschachtelte Loops

Verkettete Loops

Examples

Einzelschleifentest

Dies sind Schleifen, bei denen ihr Schleifenkörper keine anderen Schleifen enthält (die innerste Schleife, falls geschachtelt).

Um eine Schleifenabdeckung zu erzielen, sollten die Tester die unten angegebenen Tests durchführen.

Test 1: Entwerfen Sie einen Test, bei dem der Schleifenkörper überhaupt nicht ausgeführt werden soll (dh keine Iterationen).

Test 2: Entwerfen Sie einen Test, bei dem die Schleifensteuerungsvariable negativ ist (negative Anzahl von Iterationen).

Test 3: Entwerfen Sie einen Test, bei dem die Schleife nur einmal wiederholt wird

Test 4: Entwerfen Sie einen Test, bei dem die Schleife zweimal wiederholt wird

Test 5: Entwerfen Sie einen Test, bei dem die Schleife eine bestimmte Anzahl von Malen durchläuft, beispielsweise m mit m <maximal mögliche Anzahl von Iterationen

Test 6: Entwerfen Sie einen Test, bei dem die Schleife eine weniger als die maximale Anzahl von Iterationen durchläuft

Test 7: Entwerfen Sie einen Test, bei dem die maximale Anzahl der Iterationen durchlaufen wird

Test 8: Entwerfen Sie einen Test, bei dem eine Schleife mehr als die maximale Anzahl von Iterationen durchläuft

Betrachten Sie das folgende Codebeispiel, das alle angegebenen Bedingungen anwendet.

öffentliche Klasse SimpleLoopTest {

private int [] zahlen = {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;
}
```

}

öffentliche Klasse TestPass erweitert 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
}
```

}

Test für verschachtelte Schleifen

Eine verschachtelte Schleife ist eine Schleife innerhalb einer Schleife.

Die äußere Schleife ändert sich erst, nachdem die innere Schleife vollständig beendet / unterbrochen wurde.

In diesem Fall sollten Testfälle so gestaltet werden, dass

Beginnen Sie bei der innersten Schleife. Stellen Sie alle äußeren Schleifen auf ihre Mindestwerte ein. Führen Sie einen einfachen Schleifentest für die innerste Schleife durch (Test3 / Test4 / Test5 / Test6 / Test7). Fahren Sie fort, bis alle Loops getestet wurden

Verkettete Schleifen Test

Zwei Schleifen werden verkettet, wenn es möglich ist, eine nach dem Verlassen der anderen auf demselben Weg vom Eingang zum Ausgang zu erreichen. Manchmal sind diese beiden Schleifen unabhängig voneinander. In diesen Fällen können wir die Entwurfstechniken anwenden, die im Rahmen der Einzelschleifenprüfung angegeben wurden.

Wenn die Iterationswerte in einer Schleife direkt oder indirekt mit den Iterationswerten einer anderen Schleife zusammenhängen und auf demselben Pfad vorkommen können, können wir sie als verschachtelte Schleifen betrachten.

Unit-Test von Loops (Java) online lesen: https://riptutorial.com/de/unit-testing/topic/10116/unit-test-von-loops--java-

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit dem Komponententest	Andrey, Carl Manaster, Community, Farukh, forsvarir, Fred Kleuver, mahei, mark_h, Quill, silver, Stephen Byrne, Thomas Weller, zhon
2	Abhängigkeitsspritze	forsvarir, kayess, mrAtari, Pavel Voronin, Stephen Byrne
3	Assertion-Typen	Danny
4	Die allgemeinen Regeln für Unit-Tests für alle Sprachen	DarkAngel
5	Test-Doubles	forsvarir
6	Testen von Einheiten in Visual Studio für C #	DarkAngel
7	Unit Testing: Best Practices	mrAtari, RamenChef, Shrinivas Patgar, user2314737
8	Unit-Test von Loops (Java)	Remya