



UNIVERSIDAD
DE MÁLAGA



Unit Testing

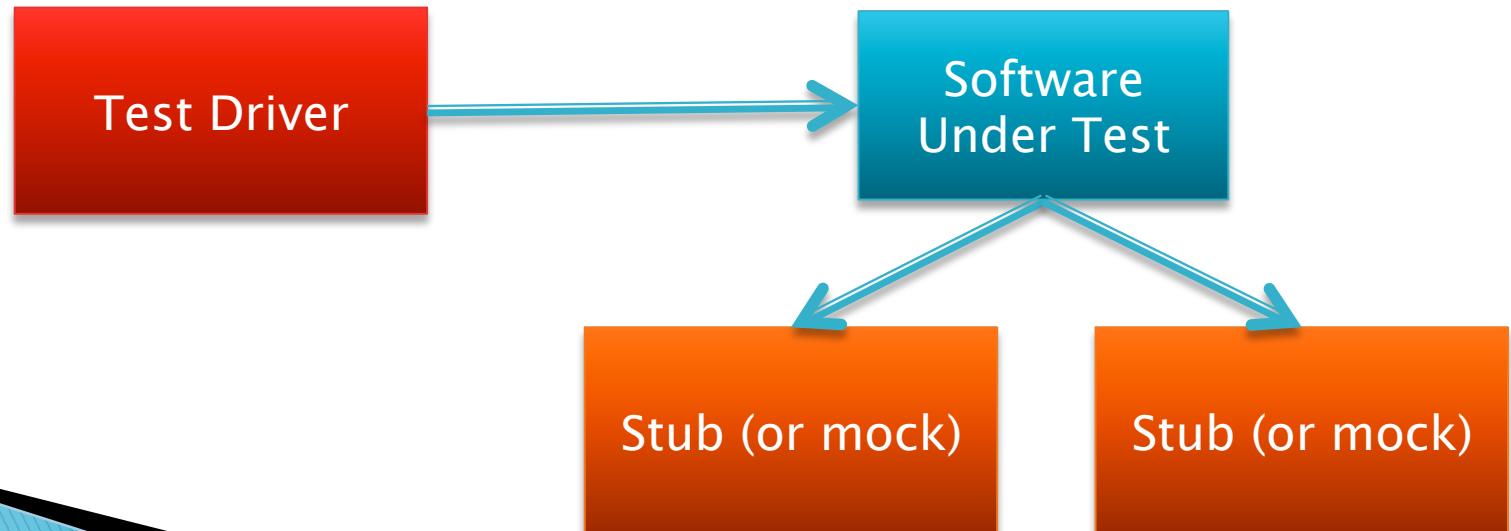
MÓDULO 5

Ingeniería del Software Software Testing

Unit testing

Background

- ▶ Unit testing consists in testing one single module or class **in an isolated way**.
- ▶ A **white-box** approach is frequently used.
- ▶ Drivers and stubs (or mocks) are required.



Unit testing

Background

- ▶ In object-oriented software
 - Classes are tested in isolation
 - Each method of the class is tested separately
 - Take into account the state of the object
- ▶ How to test:
 - Constructors
 - Getters and Setters
 - Conventional methods
 - Without (parameters/return value)
 - Exceptions
 - Non-public methods

Unit testing

Background

- ▶ Testing constructors (white-box).
 1. Call the constructor with adequate parameters
 2. Check that the fields have been properly filled (using get methods or direct access to the fields).
- ▶ Testing get/set methods.
 1. Create a new object.
 2. Set a property using a **set method**.
 3. Read the property using the **get method**.
 4. Compare the obtained value with the original one.

Unit testing

Background

- ▶ Testing conventional methods.
 - Execution depends on (determine the test data):
 1. Input parameters
 2. Internal state of the object
 3. State of collaborator objects
 4. External entities (database, file system, network...)
 - Effects of execution are (determine the check code) :
 1. Return value
 2. Changes in the parameters
 3. Exceptions
 4. Changes in the internal state of the object
 5. Changes in the external environment
 - We need to take into account the elements that influence the method and the effects of its execution in order to build suitable test cases for the method.

Unit testing

Background

▶ Testing conventional methods (cont.).

- How can we influence on:
 1. Input parameters
 2. Internal state of the object
 3. State of collaborator objects
 4. External entities
- Effects of execution:
 1. Return value
 2. Changes in the parameters
 3. Exceptions
 4. Changes in the internal state of the object
 5. Changes in the external environment

Arguments of the method

Sequence of methods call

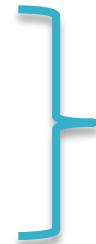
Tested objects,
stubs and mocks

Unit testing

Background

▶ Testing conventional methods (cont.).

- How can we influence on:
 1. Input parameters
 2. Internal state of the object
 3. State of collaborator objects
 4. External entities
- Effects of execution:
 1. Return value
 2. Changes in the parameters
 3. Exceptions
 4. Changes in the internal state of the object
 5. Changes in the external environment



Methods with no-arg



Methods
without
return value

Unit testing

Background

- ▶ Testing expected exceptions.
 1. Include the method in a try/catch block
 2. Notice a fail after the method (in case the exception is not raised)
 3. In the catch block check the parameters of the exception object.
- ▶ Testing unexpected exceptions.
 - Notice a fail if any other exception is raised.
 - Nothing to check because they are unexpected!!

Unit testing

Background

- ▶ Testing non-public methods (private, protected...)
- ▶ Several approaches:
 - Change the visibility of the method (**intrusive ↓**)
 - Use internal test classes (**intrusive ↓**)
 - Place the TC in the same package as CUT
(**protected and package only ↓**)
 - Use the Reflection API (**non intrusive ↑**)
 - Use Accessor class (**non intrusive ↑**)

Unit testing

JUnit



JUnit

Unit testing

JUnit

- ▶ JUnit is a framework for automating unit testing in Java applications.
- ▶ Developed by Gamma and Beck in 1997.
- ▶ It is Open Source and distributed under CPL 1.0.
- ▶ Freely available at GitHub
 - <https://github.com/junit-team/junit/wiki>
- ▶ Latest version: JUnit 4.11
- ▶ Web: <http://www.junit.org>

Unit testing

JUnit: getting started



1. Add the JUnit jar file to the classpath.
2. Add the `@Test` annotation to the test methods.
3. Import the static methods of `org.junit.Assert`.

```
package riatec.unit.example;

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

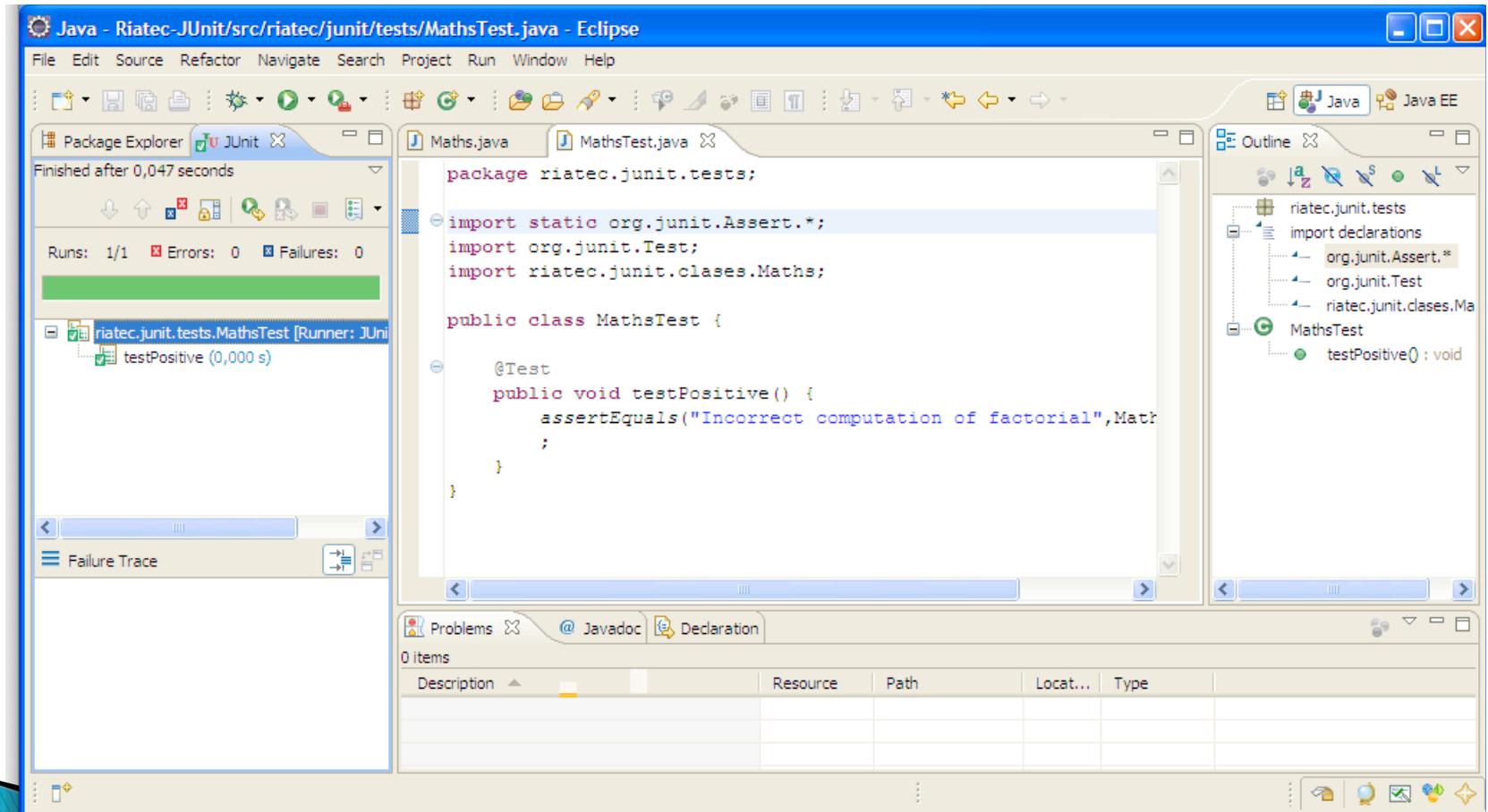
public class MathsTest {

    @Test
    public void testPositive()
    {
        assertEquals("Incorrect computation of factorial", 120, Maths.f(5));
    }
}
```

Unit testing

JUnit: getting started

- ▶ Easy support for JUnit test cases in Eclipse



Unit testing

JUnit: methodology

1. Create a test class, TC, (the test driver). The name should recall the CUT (class under test)
 - For example: for class Maths -> MathsTest
2. Create a test method in TC for each method to test in CUT (annotated with `@Test`).
3. Use the `assert...` methods to report errors.
4. Use descriptive messages for the failures.

Unit testing

JUnit: assert methods

- ▶ Defined in `org.junit.Assert`

- `assertEquals`
- `assertArrayEquals`
- `assertNull`
- `assertNotNull`
- `assertTrue`
- `assertFalse`
- `assertSame`
- `assertNotSame`
- `fail`

Examples

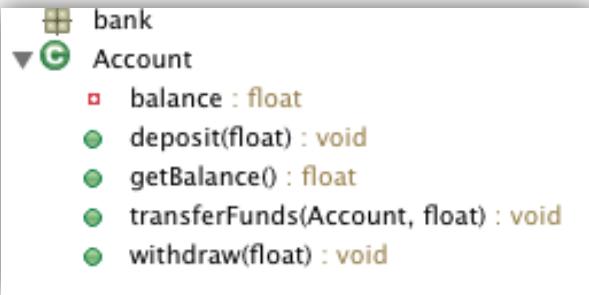
```
assertEquals(200, source.getBalance());  
assertEquals("Withdrawal error", 200, source.getBalance());
```

- ▶ Parameters:
 - `(expected, actual)`
 - `(message, expected, actual)`

Unit testing

JUnit: one simple example

▶ Example



```
package bank;

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

public class BankTest {

    @Test
    public void TransferFunds()
    {
        Account source = new Account();
        source.deposit(200.00f);
        Account destination = new Account();
        destination.deposit(150.00f);

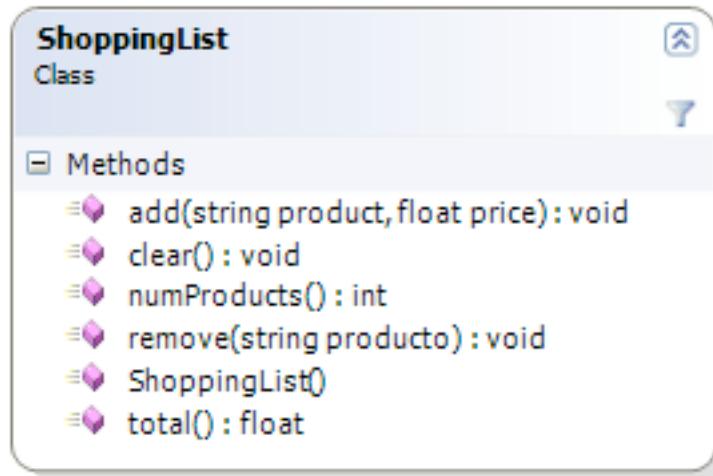
        source.transferFunds(destination, 100.00f);
        assertEquals(250.00f, destination.getBalance(),0.0);
        assertEquals(100.00f, source.getBalance(),0.0);

    }
}
```

Unit testing

JUnit: one exercise

- ▶ Book shop Web application.
 - Test the following class (in package bookshop).



- Create a new project.
- Include **bookshop.jar** in the java build path.

Unit testing

JUnit: @Before and @After

- ▶ Before running some tests some initialization might be required (fixture).
- ▶ `@Before` annotates the methods that do this initialization.
- ▶ All the “`@Before` methods” are run **before** each test method.
- ▶ `@After` annotates the methods that release the resources allocated with `@Before`
- ▶ If a per-class initialization and release is required use the annotations `@BeforeClass` and `@AfterClass` (on static no-arg methods).

Unit testing

JUnit: @Before and @After example

- ▶ One example with @Before

This method is executed
before each test



```
package bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class BankTest {

    Account source;
    Account destination;

    @Before
    public void initialize()
    {
        source = new Account();
        source.deposit(200.00f);
        destination = new Account();
        destination.deposit(150.00f);
    }

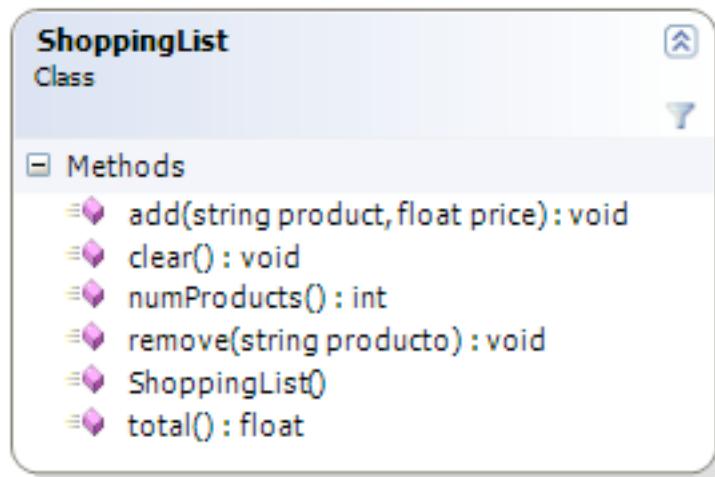
    @Test
    public void TransferFunds()
    {
        source.transferFunds(destination, 100.00f);
        assertEquals(250.00f, destination.getBalance(),0.0);
        assertEquals(100.00f, source.getBalance(),0.0);
    }

    @Test
    public void Withdrawal()
    {
        source.withdraw(50.00f);
        assertEquals(150.00f, source.getBalance(),0.0);
    }
}
```

Unit testing

JUnit: another exercise

- ▶ Book shop Web application.
 - Test the following class and use `@Before`.



- Create a new project.
- Include `bookshop.jar` in the java build path.

Unit testing

JUnit: exceptions

- ▶ Test of expected exceptions
 - `@Test(expected=...Exception.class)`
 - The exception must be declared using throws
 - Very simple
 - Drawback: we cannot check the exception object
 - Surround the code with try–catch
 - The most general approach (less elegant)
 - We can check the exception object
- ▶ Test of unexpected exceptions
 - Nothing to do: JUnit reports a fail if an unexpected exception occurs.

Unit testing

JUnit: non-public methods

- ▶ How to test protected, private or package methods?
- ▶ Use the Reflection API (non intrusive ↑)

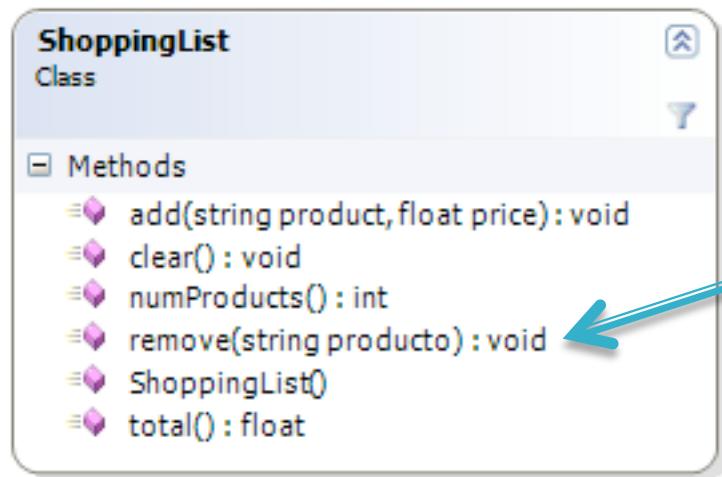
```
@Test
public void testPrivate() throws Exception
{
    Class cl = sl.getClass();
    Method m = cl.getDeclaredMethod("privateMethod", int.class);
    m.setAccessible(true);
    Object o = m.invoke(sl, 6);
    assertEquals(12,o);

}
```

Unit testing

JUnit: exceptions exercise

- ▶ Book shop Web application.
 - Now, the remove method can throw an exception.



Throws a NoSuchElementException if the product is not found

The message of the exception should be: "Product not found"

- Check that the exception is thrown.
- Check that the message is correct.
- Upload the .java file of the TestClass.

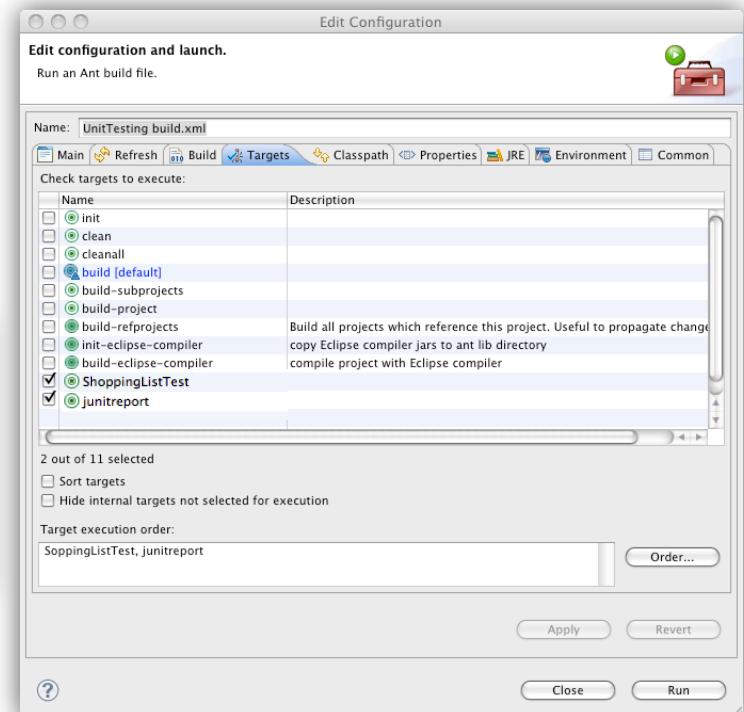
Unit testing

JUnit: data-driven tests and reports



- ▶ Sometimes we want to test a method in the same way using different parameters (data).
- ▶ They are called **data-driven** tests.
- ▶ **JTestCase** can be used together with JUnit for this.
- ▶ It is possible to get junit reports.
 1. Export the project as an Ant buildfile
 2. Run the tests in ant followed by the **junitreport** target
 3. HTML reports will be obtained

<input type="checkbox"/>	build-refprojects	Build all projects
<input type="checkbox"/>	init-eclipse-compiler	copy Eclipse com
<input type="checkbox"/>	build-eclipse-compiler	compile project w
<input checked="" type="checkbox"/>	ShoppingListTest	
<input checked="" type="checkbox"/>	junitreport	



Unit testing

JUnit: data-driven tests and reports

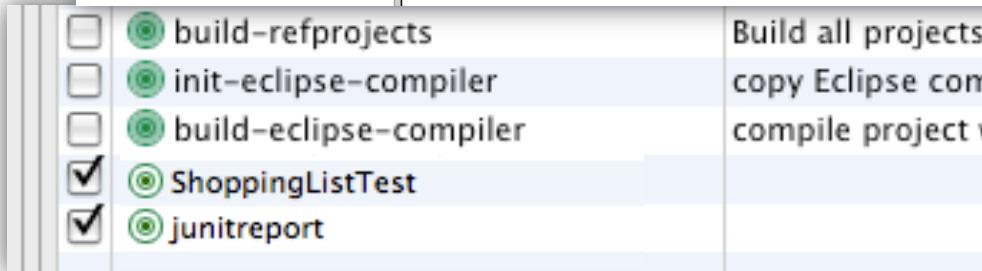


- Sometimes we want to test a method in the same way using different parameters (data).

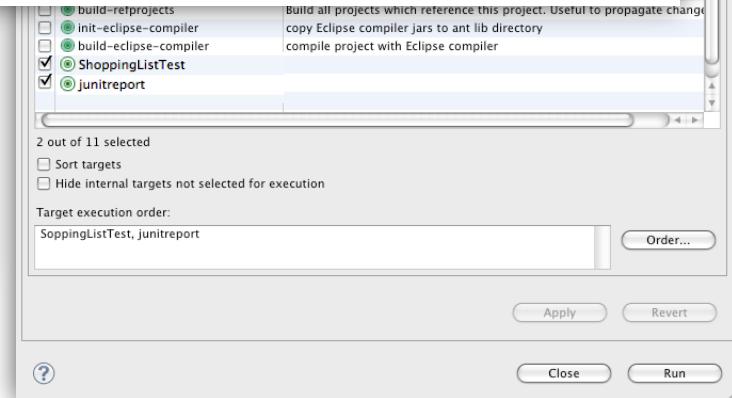


The screenshot shows two windows side-by-side. On the left is the JUnit Test Results window titled 'Unit Test Results.' It displays a table of test results for 'All Tests' under the 'ShoppingListTest' class. The table includes columns for Class, Name, Status, Type, and Time(s). The 'testRemove' test is listed as a Failure, and 'testRemoveException' is listed as an Error. On the right is the Eclipse 'Build Target' dialog, which lists various build targets like 'build-refprojects' and 'init-eclipse-compiler'. The 'ShoppingListTest' and 'junitreport' targets are selected. The dialog also shows the target execution order as 'SoppingListTest, junitreport'.

Class	Name	Status	Type	Time(s)
ShoppingListTest	testAdd	Success		0.004
ShoppingListTest	testClear	Success		0.000
ShoppingListTest	testNumProducts	Success		0.001
ShoppingListTest	testTotal	Success		0.001
ShoppingListTest	testRemove	Failure	expected:<1> but was:<0> junit.framework.AssertionFailedError: expected:<1> but was:<0> at ShoppingListTest.testRemove(ShoppingListTest.java:49)	0.009
ShoppingListTest	testRemoveException	Error	expected:<Product[] not found> but was:<Product[o] not found> at ShoppingListTest.testRemoveException(ShoppingListTest.java:61)	0.002



This screenshot shows the 'Build Target' dialog from Eclipse. It lists several build targets on the left, with 'ShoppingListTest' and 'junitreport' checked. The right side of the dialog contains buttons for 'Build all projects', 'copy Eclipse com...', and 'compile project v...'. Below these buttons is a list of selected targets: 'build-refprojects', 'init-eclipse-compiler', 'build-eclipse-compiler', 'ShoppingListTest', and 'junitreport'. A note at the top right says: 'Build all projects which reference this project. Useful to propagate changes'.



This screenshot shows the 'Build Target' dialog from Eclipse. It lists several build targets on the left, with 'ShoppingListTest' and 'junitreport' checked. The right side of the dialog contains buttons for 'Build all projects', 'copy Eclipse com...', and 'compile project v...'. Below these buttons is a list of selected targets: 'build-refprojects', 'init-eclipse-compiler', 'build-eclipse-compiler', 'ShoppingListTest', and 'junitreport'. A note at the top right says: 'Build all projects which reference this project. Useful to propagate changes'.

Unit testing

Visual Studio



Visual Studio

Unit testing

Visual Studio: getting started

1. Import the namespace:
`Microsoft.VisualStudio.TestTools.UnitTesting`
2. Add the `[TestClass]` attribute to the test class.
3. Add the `[TestMethod]` attribute to the test methods.
4. Use the static methods of `Assert`.



Simplified thanks to
the testing
integration

```
namespace Bank
{
    using Microsoft.VisualStudio.TestTools.UnitTesting;
    [TestClass]
    class AccountTest
    {

        [TestMethod]
        public void TransferFunds()
        {
            Account source = new Account();
            source.Deposit(200.00f);
            Account destination = new Account();
            destination.Deposit(150.00f);

            source.TransferFunds(destination, 100.00f);
            Assert.AreEqual(250.00f, destination.Balance);
            Assert.AreEqual(100.00f, source.Balance);
        }
    }
}
```

Unit testing

Visual Studio: methodology

1. Create a test class, TC, (the test driver) and assign the attribute [TestClass]. The name should recall the CUT (class under test).
 - For example: for class Maths -> MathsTest
2. Create a test method in TC for each method to test in CUT (with attribute [TestMethod]).
3. Use the Assert static methods to report failures.
4. Use descriptive messages for the failures.

Unit testing

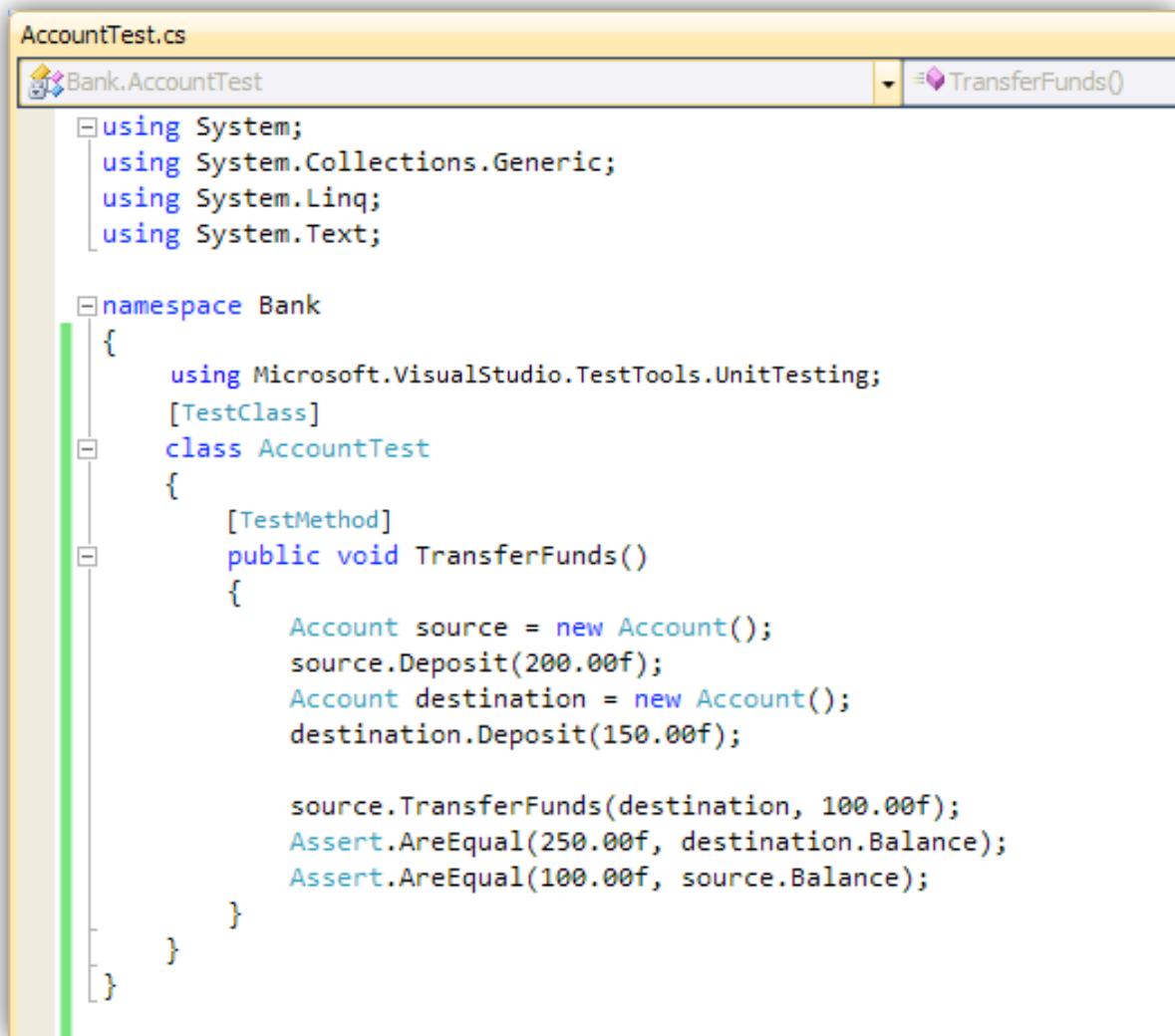
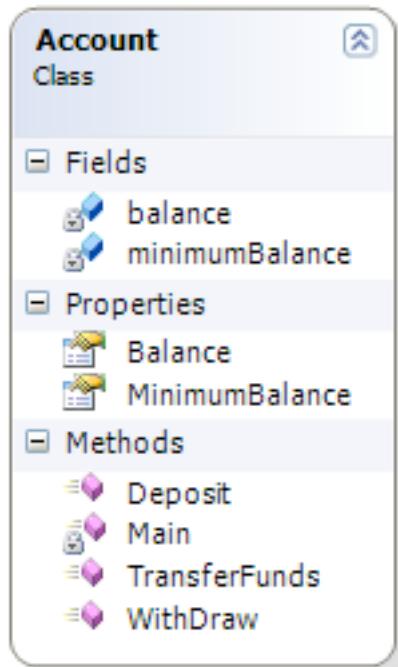
Visual Studio: Assert methods

- ▶ Defined in
[Microsoft.VisualStudio.TestTools.UnitTesting.Assert](#)
(non-exhaustive list)
 - [AreEqual](#)
 - [AreNotEqual](#)
 - [AreSame](#)
 - [AreNotSame](#)
 - [IsNull](#)
 - [IsNotNull](#)
 - [IsTrue](#)
 - [IsFalse](#)
 - [Fail](#)
 - ...
- ▶ Parameters:
 - [\(expected, actual\)](#)
 - [\(expected, actual, message\)](#)
 - [\(expected, actual, message, arguments\)](#)

Unit testing

Visual Studio: one simple example

Example



The screenshot shows the `AccountTest.cs` file in Visual Studio. The code defines a `Bank` namespace containing a `AccountTest` class. The `TransferFunds()` method is annotated with `[TestMethod]` and `[TestMethod]`. The method initializes two `Account` objects, performs a transfer of 100.00f from the source to the destination, and then asserts that the destination's balance is 250.00f and the source's balance is 100.00f.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

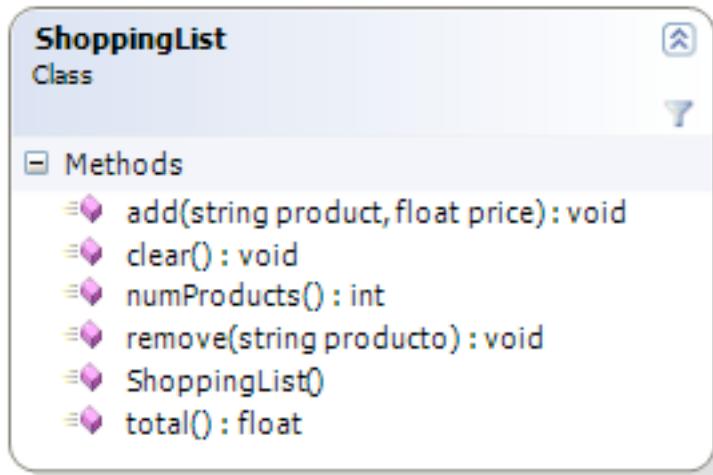
namespace Bank
{
    using Microsoft.VisualStudio.TestTools.UnitTesting;
    [TestClass]
    class AccountTest
    {
        [TestMethod]
        public void TransferFunds()
        {
            Account source = new Account();
            source.Deposit(200.00f);
            Account destination = new Account();
            destination.Deposit(150.00f);

            source.TransferFunds(destination, 100.00f);
            Assert.AreEqual(250.00f, destination.Balance);
            Assert.AreEqual(100.00f, source.Balance);
        }
    }
}
```

Unit testing

Visual Studio: one exercise

- ▶ Book shop Web application.
 - Test the following class (in namespace BookShop).



- Create a new project.
- Include **BookShop.dll** in your references.

Unit testing

VS: [TestInitialize] and [TestCleanup]

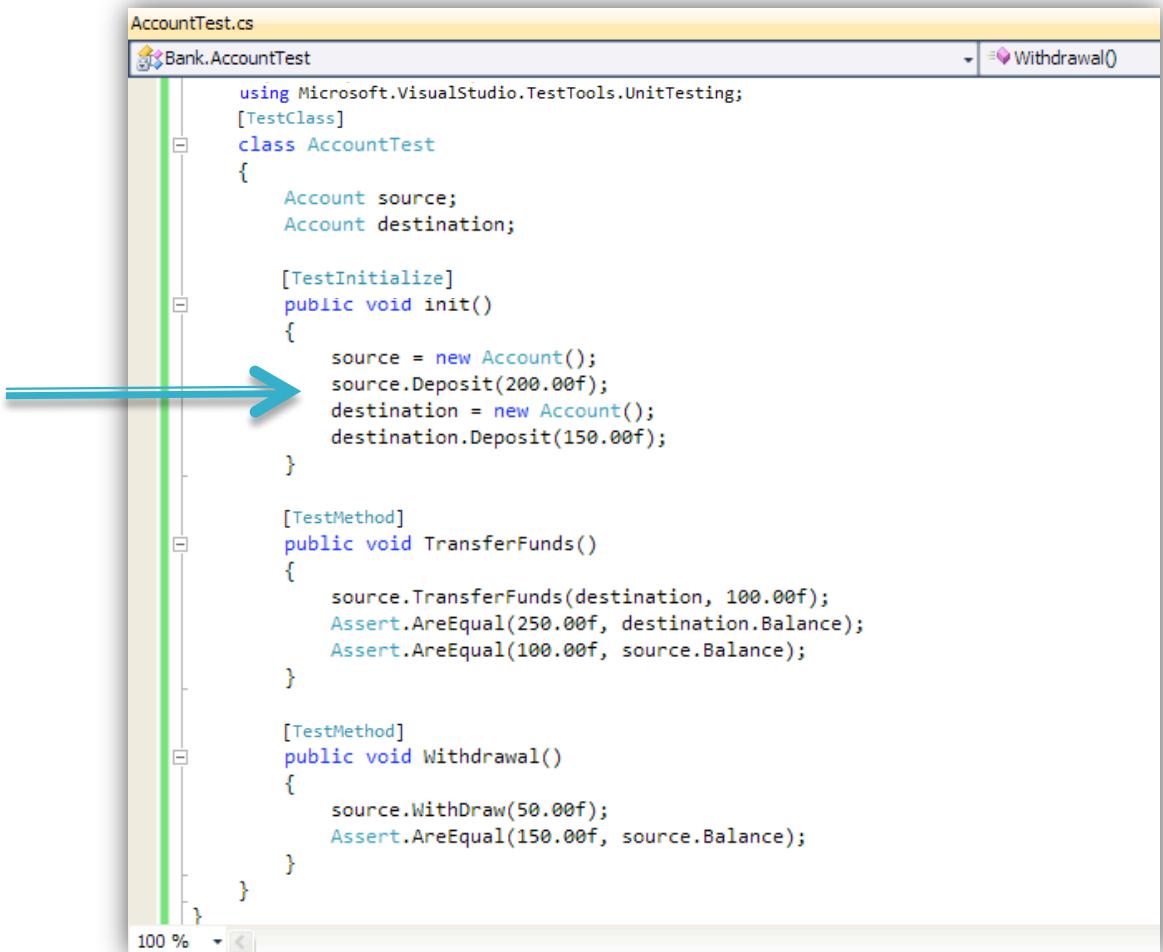
- ▶ Before running some tests some initialization might be required (fixture).
- ▶ **[TestInitialize]** annotates the methods that do this initialization.
- ▶ All the “[**TestInitialize**] methods” are run **before each** test method.
- ▶ **[TestCleanup]** annotates the methods that release the resources allocated with **[TestInitialize]**
- ▶ If a per-class (fixture) initialization and release is required use the attributes:
 - **[ClassInitialize]** and **[ClassCleanup]**.

Unit testing

VS: [TestInitialize] and [TestCleanup] example

- ▶ One example with [TestInitialize]

This method is executed
before each test



```
AccountTest.cs
Bank.AccountTest
using Microsoft.VisualStudio.TestTools.UnitTesting;
[TestClass]
class AccountTest
{
    Account source;
    Account destination;

    [TestInitialize]
    public void init()
    {
        source = new Account();
        source.Deposit(200.00f);
        destination = new Account();
        destination.Deposit(150.00f);
    }

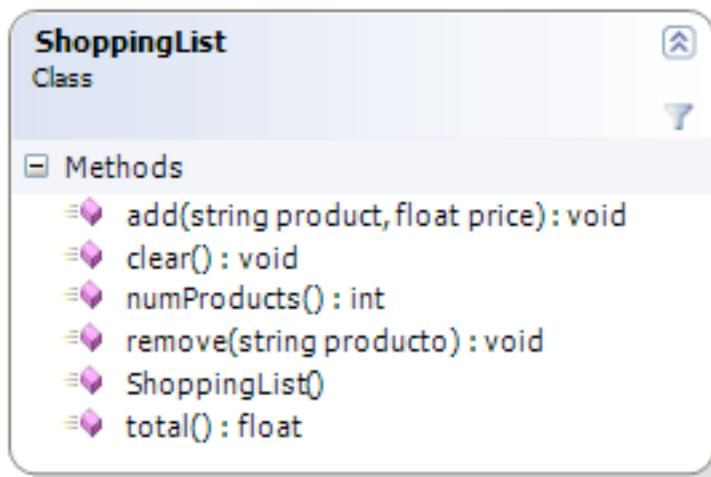
    [TestMethod]
    public void TransferFunds()
    {
        source.TransferFunds(destination, 100.00f);
        Assert.AreEqual(250.00f, destination.Balance);
        Assert.AreEqual(100.00f, source.Balance);
    }

    [TestMethod]
    public void Withdrawal()
    {
        source.WithDraw(50.00f);
        Assert.AreEqual(150.00f, source.Balance);
    }
}
```

Unit testing

Visual Studio: another exercise

- ▶ Book shop Web application.
 - Test the following class and use [TestInitialize] methods.



- Create a new project.
- Include **BookShop.dll** in your references.

Unit testing

Visual Studio: exceptions

- ▶ Test of expected exceptions
 - [TestMethod] [ExpectedException(typeof(... Exception))]
 - Very simple
 - Drawback: we cannot check the exception object
 - Surround the code with try–catch
 - The most general approach (less elegant)
 - We can check the exception object
- ▶ Test of unexpected exceptions
 - Nothing to do: Visual Studio reports a fail if an unexpected exception occurs.

Unit testing

Visual Studio: non-public methods



- ▶ How to test protected, private or package methods?
- ▶ Use the Reflection API (non intrusive ↑)

```
[TestMethod]
public void TestPrivate()
{
    Type t = sl.GetType();
    MethodInfo mi = t.GetMethod("PrivateMethod", BindingFlags.NonPublic | BindingFlags.Instance);
    object o = mi.Invoke(sl, new object[] { 6 });
    Assert.AreEqual(12, o);
}
```

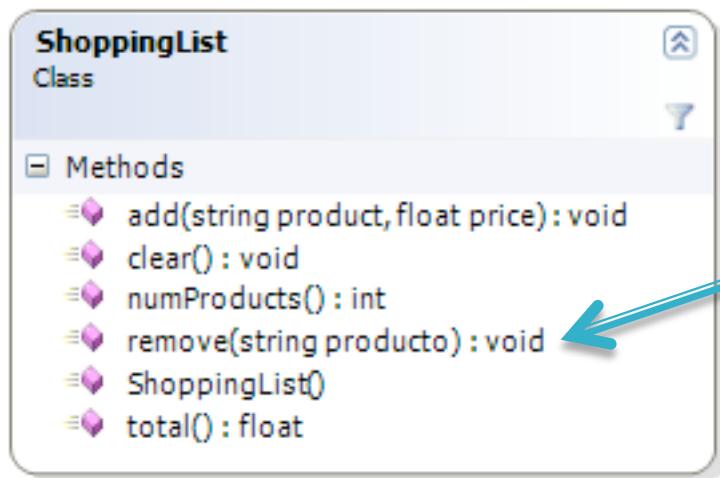
- ▶ Use of Accessor classes (non intrusive ↑)

```
[TestMethod()]
[DeploymentItem("Gramatica.dll")]
public void setTerminalTest()
{
    Grammar_Accessor target = new Grammar_Accessor(); // TODO: Initialize to an appropriate value
    int v = 4; // TODO: Initialize to an appropriate value
    string t = "hola"; // TODO: Initialize to an appropriate value
    target.setTerminal(v, t);
    Assert.AreEqual("hola", target.getTerminales(v), "Error al insertar terminal");
}
```

Unit testing

Visual Studio: exceptions exercise

- ▶ Book shop Web application.
 - Now, the remove method can throw an exception.



Throws an ApplicationException if the product is not found

The message of the exception should be: "No product found"

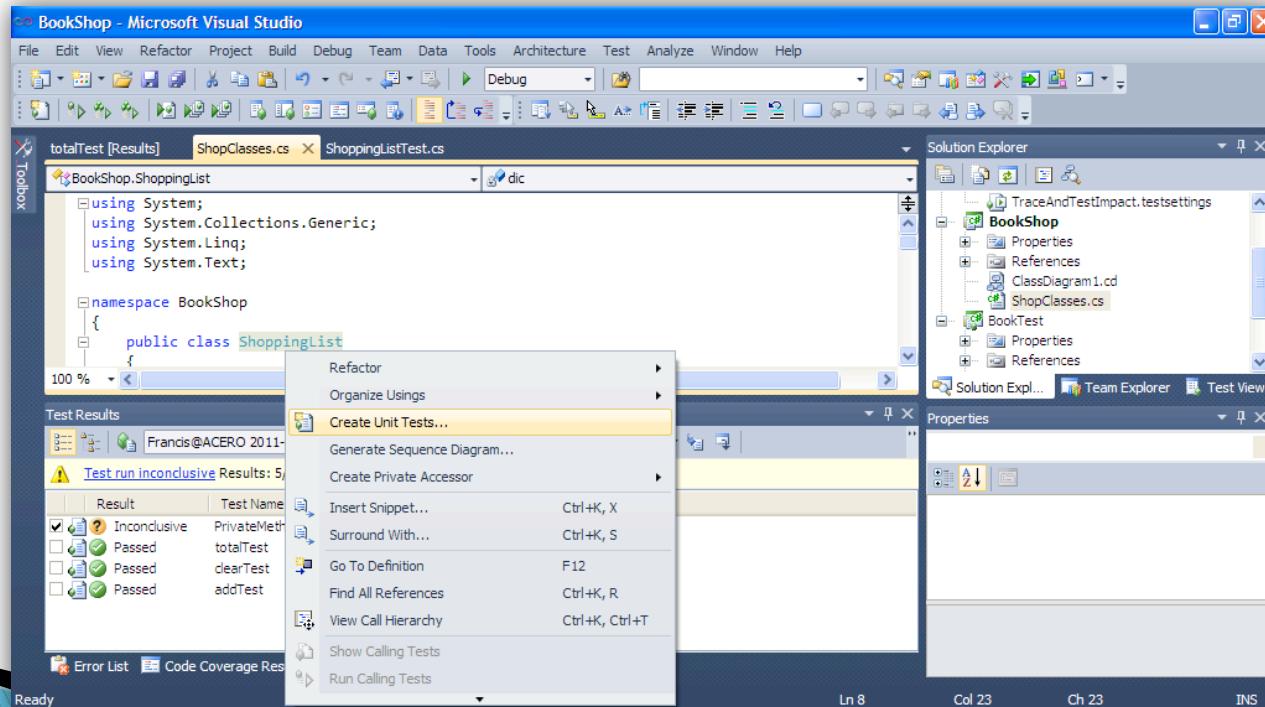
- Check that the exception is thrown.
- Check that the message is correct.
- Upload the .cs file of the TestClass.

Unit testing

Visual Studio: other features



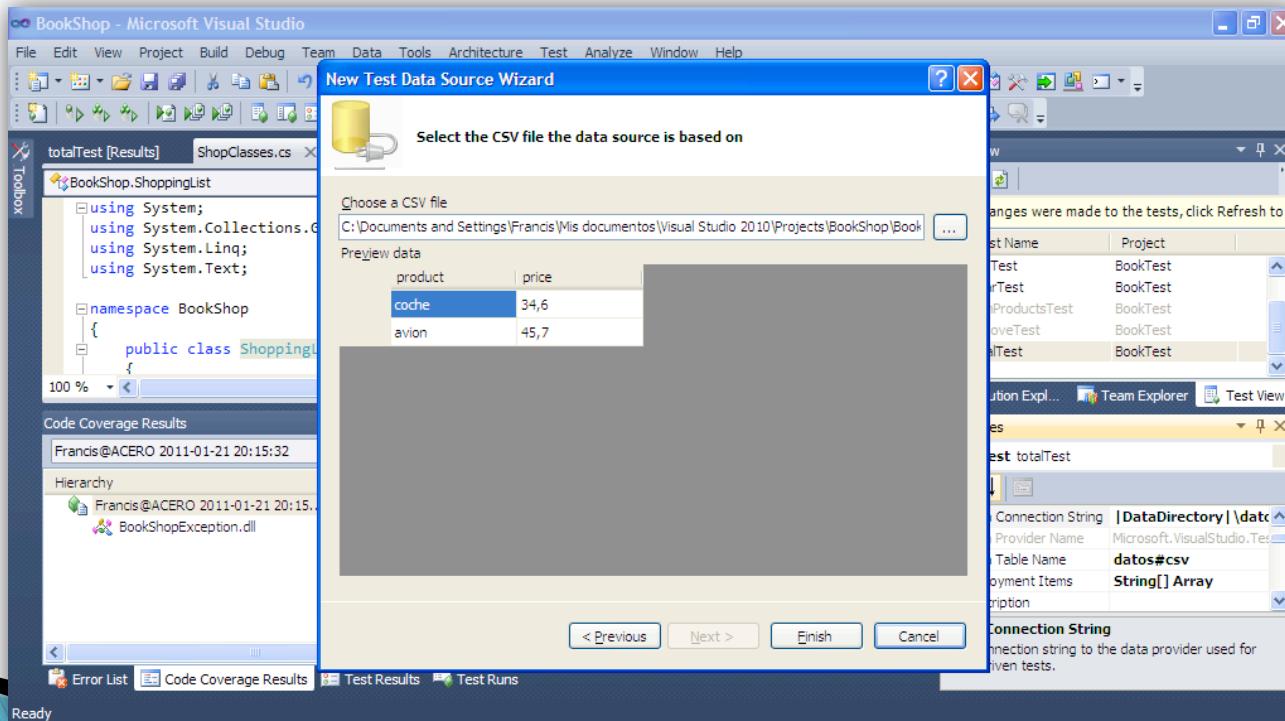
- ▶ Other features of Visual Studio:
 - Easy to generate test class (in VS 2010)
 - Easy support for data-driven tests
 - Code coverage analysis



Unit testing

Visual Studio: other features

- ▶ Other features of Visual Studio:
 - Easy to generate test class (in VS 2010)
 - Easy support for data–driven tests
 - Code coverage analysis



Unit testing

Visual Studio: other features

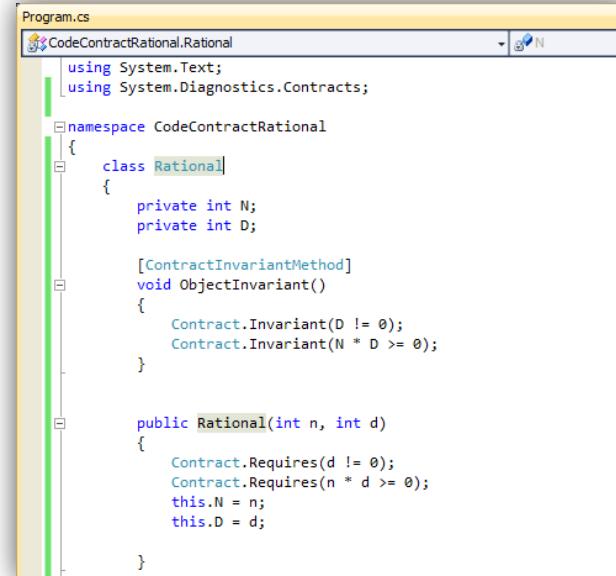
- ▶ Other features of Visual Studio:
 - Easy to generate test class (in VS 2010)
 - Easy support for data–driven tests
 - Code coverage analysis

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Francis@ACERO 2011-01-21 20:15...	5	19,23 %	21	80,77 %
BookShopException.dll	5	19,23 %	21	80,77 %

Unit testing

Code Contracts

- ▶ Allows for **preconditions**, **postconditions** and **invariants** to be specified and dynamically and **statically** checked.
- ▶ Supports the “design by contract” approach.
- ▶ Introduction of formal tools in software implementation.
- ▶ Example:

A screenshot of a code editor showing a C# file named Program.cs. The code defines a namespace CodeContractRational and a class Rational. It includes using statements for System.Text and System.Diagnostics.Contracts. The Rational class has a private int N and a private int D. A constructor Rational(int n, int d) uses Contract.Requires(d != 0) and Contract.Requires(n * d >= 0). An ObjectInvariant method uses Contract.Invariant(D != 0) and Contract.Invariant(N * D >= 0).

Unit testing

Code Contracts

▶ Preconditions

```
Contract.Requires(p >= 0 && p <= this.GetSize());
```

▶ Postconditions

```
Contract.Ensures(Contract.Result<int>() >= 0);
```

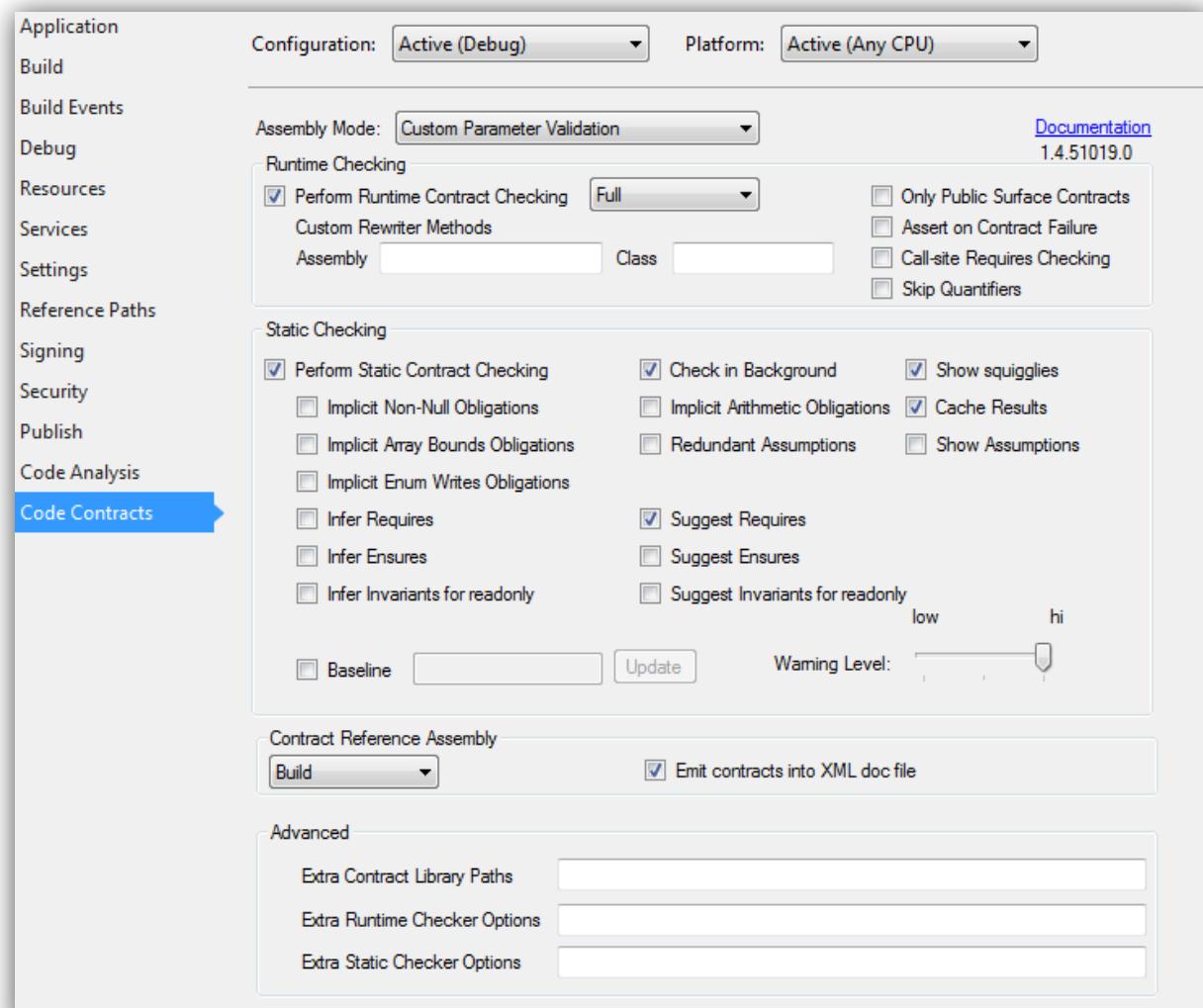
▶ Invariants

```
[ContractInvariantMethod]
[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Performance",
    "CA1822:MarkMembersAsStatic", Justification = "Required for code contracts.")]
private void ObjectInvariant()
{
    Contract.Invariant(n >= 0);
}
```

Unit testing Code Contracts



▶ Configuration



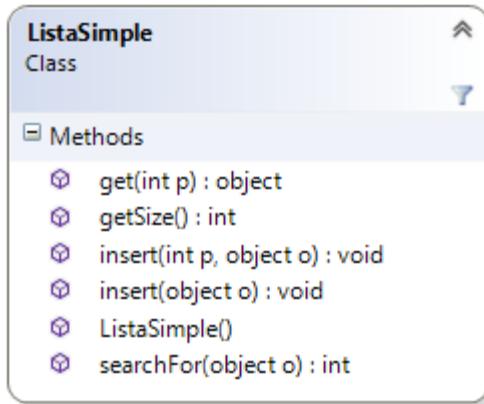
The screenshot shows the 'Code Contracts' configuration dialog in Visual Studio. The left sidebar lists various project settings, with 'Code Contracts' highlighted in blue. The main area contains several sections:

- Configuration:** Active (Debug) | Platform: Active (Any CPU)
- Assembly Mode:** Custom Parameter Validation | Documentation: 1.4.51019.0
- Runtime Checking**:
 - Perform Runtime Contract Checking (Full dropdown)
 - Custom Rewriter Methods
 - Assembly: [text box] Class: [text box]
 - Only Public Surface Contracts
 - Assert on Contract Failure
 - Call-site Requires Checking
 - Skip Quantifiers
- Static Checking**:
 - Perform Static Contract Checking
 - Implicit Non-Null Obligations
 - Implicit Array Bounds Obligations
 - Implicit Enum Writes Obligations
 - Infer Requires
 - Infer Ensures
 - Infer Invariants for readonly
 - Check in Background
 - Implicit Arithmetic Obligations
 - Redundant Assumptions
 - Suggest Requires
 - Suggest Ensures
 - Suggest Invariants for readonly
 - Show squiggles
 - Cache Results
 - Show Assumptions
- Warning Level: A slider from 'low' to 'hi'.
- Contract Reference Assembly**:
 - Build dropdown
 - Emit contracts into XML doc file
- Advanced**:
 - Extra Contract Library Paths
 - Extra Runtime Checker Options
 - Extra Static Checker Options

Unit testing

Code Contracts: exercise

- ▶ Add contracts to the methods in a simple array-based list.



```
class MainClass
{
    public static void Main()
    {
        ListaSimple ls = new ListaSimple();
        ls.insert("hola");
        int p = ls.searchFor("hola");
        if (p >= 0)
            Console.WriteLine(ls.get(p));
    }
}
```

Unit testing

PEX

- ▶ PEX is a tool for automatically generating test cases.
- ▶ Available at: www.pexforfun.com
- ▶ Based on (decision) coverage



- ▶ Similar tools using genetic algorithms, evolutionary algorithms, and metaheuristics in general:
 - In our research group

Unit testing

PEX

- ▶ A hot topic in current research.

