

2 Introduction to unit testing with C#

Table of Contents

2 Introduction to unit testing with C#.....	1
2.0 Introduction.....	1
2.1 Unit testing in Visual Studio with C# and MSTest V2.....	2
2.2 Unit testing frameworks in .NET Core with C#.....	2
2.3 Unit testing in C# on your own.....	3
2.4 Refactoring task in C# (voluntary).....	4
2.5 Lab feedback.....	5
2.6 Appendix 1 – Create a test project in VS.....	5

2.0 Introduction

This lab will introduce the unit testing frameworks in C#.

Prerequisites

Before you begin make sure you got Visual Studio 2022 Enterprise Edition installed. You must use the Enterprise Edition since some testing capabilities are missing in the Community Edition as Code coverage for example.

Get VS 2022 EE from Microsoft Azure Dev Tools for Teaching (former Microsoft Imagine program): <https://aka.ms/devtoolsforteaching>.

The 2.1 – 2.2 tasks must be done to understand the available frameworks for unit testing in C# and how to use them with Visual Studio.

Problems and hints

If you got errors running the test project you may have to force the processor architecture to x86 or x64 if there is a mismatch in the projects “Platform Target” and/or “Any CPU” is selected in the project properties build configuration.

To fix this from the main menu set: Test > Test Settings > Default Processor Architecture > X64 for all your projects.

Sometimes newly added tests or changed tests do not update or populate the Test Explorer view correctly. I currently have no good solution for this more than to close and open the VS solution again. A solution may be available if you search for it:

<https://www.google.com/search?q=visual+studio+test+explorer+refresh>

Test framework documentation and samples:

- NUnit
 - <https://github.com/nunit>
 - <https://docs.nunit.org/articles/nunit/intro.html>
 - <https://github.com/nunit/nunit-csharp-samples>
- MSTest V2
 - <https://github.com/Microsoft/testfx-docs> and further links on the page
 - <https://docs.microsoft.com/en-us/dotnet/api/>

- <https://github.com/Microsoft/testfx/tree/master/src/TestFramework/MSTest.Core>
- xUnit
 - <https://xunit.net/>
 - <https://xunit.net/docs/getting-started/netfx/visual-studio>
 - <https://github.com/xunit/samples.xunit>

See appendix 1 for more hints and help. One can also have a look at the “Rövarspråket in Csharp” example at: <http://users.du.se/~hjo/cs/gmi2j3/lectures/> > code_rovarsprak.

2.1 Unit testing in Visual Studio with C# and MSTest V2

Report:

Hand in the full source code of your MS BankAccount solution.

Task:

Navigate to the Visual Studio Guide at:

<https://docs.microsoft.com/en-us/visualstudio/test/improve-code-quality> > Unit testing.

Read and browse thru the Get started, Unit test basics, ... and the other relevant unit test articles in the left-hand menu until you reach the “**Write unit tests for managed code**” section.

Perform the “Walkthrough: Create and run unit tests for managed code” tutorial. You should use the “MSTest V2” framework: <https://github.com/microsoft/testfx>. MSTest V2 is hereafter refereed only as MSTest since MSTest V1 is to old to be relevant. In appendix 1 there is some helpful VS-screenshots for how to create test projects. Remember to add a project reference to the tested source code.

2.2 Unit testing frameworks in .NET Core with C#

Report:

Hand in the working VS projects with unit tests and the tested source. All the three (NUnit, MSTest and xUnit) test projects for C# must have been implemented.

Task:

From the “.NET documentation” at: <https://docs.microsoft.com/en-us/dotnet/> > Open source .NET > .NET tutorials, click on the left-hand menu “**Testing > Overview**” a bit down in the list so you arrive at: <https://docs.microsoft.com/en-us/dotnet/core/testing/>. Then read the “Overview” and the “Unit testing best practices” section.

Finally locate the three tutorial unit testing tasks in the left-hand menu:

- C# unit testing with xUnit
 - Unit testing C# in .NET Core using dotnet test and xUnit
- C# unit testing with NUnit
 - Unit testing C# with NUnit and .NET Core
- C# unit testing with MSTest

- Unit testing C# with MSTest and .NET Core

Perform the NUnit, MSTest and xUnit tutorial tasks. Only the test code (one more .csharp file per framework or one more project per framework) is needed when you add multiple tests into your solution. They can reference to the same source project.

You do not have to use the console creating the project files as described in the guides, just choose a .NET Core project in Visual Studio. In appendix 1 there is some helpful VS-screenshots for how to create test projects. Remember to add a project reference to the tested source code.

2.3 Unit testing in C# on your own

Report:

Hand in the source code to your solution for roman numerals.

- You should have implemented at least 11 different test cases depending on the chosen test framework.
- Make a small driver, eg.: “static void Main(String[] args) {menu}” for your code so one can run and test the conversion library manually.
- Also report your code coverage. MS help for code coverage:
<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-code-coverage>

Task:

Choose one of the unit test frameworks for C# which you experienced in task 2.2. A unit test framework comparison is available here: <https://xunit.net/docs/comparisons>.

Then perform the task of writing unit tests for the roman numerals problem in C#. I.e. perform the same number conversions (roman to int and vice versa) tests as you did in lab 1 with Python.

A roman numerals conversion solution is given in the folder “roman numerals” which you may have to perform some minor adjustments and refactoring to for your test code to work. If you want you can write some tests for the overloaded operator methods (math operations as +, -, etc.) and add more from the part2 article with Roman numerals as well.

If you want to use some other roman numerals conversion solution or implement your own roman numerals conversion it is OK as well.

I have corrected a small error in the original code.

```
public static string operator -(string r1, RomanNumeral r2)
{
    var r = RomanNumeral.Parse(r1) - r2; // org. error + r2;
    return r.ToString();
}
```

Note that all of the test cases that were performed in lab 1 is not possible or meaningful to implement in C#. This because C# is a more strongly typed language compared to Python.

You should however have implemented at least 11 different tests depending on how your selected test framework choose to count the tests cases. Note that some test methods may have many tests if they are data-driven (the test method use the: [InlineData], [TestCase] or [DataRow] attribute etc.) so the number of tests can easily be > 30.

Example: my NUnit, xUnit and MsTest Test Explorer result for RomanNumerals:

✓ RomanNumeralsNuTests (29)	223 ms
✓ RomanNumeralsNuTests (29)	223 ms
✓ FromRomanBadInput (4)	18 ms
✓ TestMalformedAntecedentsReturnArgumentException	18 ms
✓ TestNonRomanNumeralReturnArgumentException	< 1 ms
✓ TestRepeatedPairsReturnArgumentException	< 1 ms
✓ TestToManyRepeatedNumeralsReturnArgumentException	< 1 ms
✓ RoundtripCheck (1)	196 ms
✓ TestRoundtripReturnEqual	196 ms
✓ TestKnownValues (2)	4 ms
✓ TestFromRomanKnownValuesReturnEqual	4 ms
✓ TestToRomanKnownValuesReturnEqual	< 1 ms
✓ ToRomanBadInput (22)	5 ms
✓ TestForBadInputsStrings (17)	5 ms
✓ TestForNullAndEmptyInputsStrings (2)	< 1 ms
✓ TestIntegerReturnArgumentOutOfRangeException (3)	< 1 ms

✓ RomanNumeralsXuTests (29)	245 ms
✓ RomanNumeralsXuTests (29)	245 ms
✓ FromRomanBadInput (4)	24 ms
✓ TestMalformedAntecedentsReturnArgumentException	< 1 ms
✓ TestNonRomanNumeralReturnArgumentException	10 ms
✓ TestRepeatedPairsReturnArgumentException	< 1 ms
✓ TestToManyRepeatedNumeralsReturnArgumentException	14 ms
✓ RoundtripCheck (1)	169 ms
✓ TestRoundtripReturnEqual	169 ms
✓ TestKnownValues (2)	21 ms
✓ TestFromRomanKnownValuesReturnEqual	16 ms
✓ TestToRomanKnownValuesReturnEqual	5 ms
✓ ToRomanBadInput (22)	31 ms
✓ TestForBadInputsStrings (17)	31 ms
✓ TestForNullAndEmptyInputsStrings (2)	< 1 ms
✓ TestIntegerReturnArgumentOutOfRangeException (3)	< 1 ms

Test	Duration
✓ RomanNumeralsMsTests (11)	240 ms
✓ RomanNumeralsMsTests (11)	240 ms
✓ FromRomanBadInput (5)	82 ms
✓ test_blank	69 ms
✓ test_malformed_antecedents	13 ms
✓ test_null	< 1 ms
✓ test_repeated_pairs	< 1 ms
✓ test_too_many_repeated_numerals	< 1 ms
✓ RoundtripCheck (1)	149 ms
✓ test_roundtrip	149 ms
✓ TestKnownValues (2)	4 ms
✓ test_from_roman_known_values	3 ms
✓ test_to_roman_known_values	1 ms
✓ ToRomanBadInput (3)	5 ms
✓ ReturnArgumentOutOfRangeException	4 ms
✓ TestForBadInputsStrings	1 ms
✓ TestForNullAndEmptyInputsStrings	< 1 ms

2.4 Refactoring task in C# (voluntary)

In the refactor-lab folder there is a refactoring challenge in C#. Read the RefactoringChallenge.pdf for further instructions. I have updated the solution to net6.0 and



net6.0-windows. You need a SQL-database to perform the task. Use either the Visual Studio built-in SQL Server Express LocalDB or a Microsoft SQL Server container via Docker. You can manage the DB with: <https://docs.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15>. Below is a docker-compose.yml file for both the x64 and arm64 Azure SQL Edge DB. Consult the teacher if you need help to get it started.

```
# https://hub.docker.com/_/microsoft-azure-sql-edge
# docker-compose up -d
# docker-compose down
#####
version: '3.3'
services:
  db:
    image: 'mcr.microsoft.com/azure-sql-edge:latest'
    user: root
    container_name: sqledge
    ports:
      - '1433:1433'
    volumes:
      - '////c/vm/db/mssql/data_edge:/var/opt/mssql/data'
      - '////c/vm/db/mssql/log:/var/opt/mssql/log'
      - '////c/vm/db/mssql/secrets:/var/opt/mssql/secrets'
    environment:
      MSSQL_SA_PASSWORD: 'mypassword123' # username: sa
      ACCEPT_EULA: 'Y'
      MSSQL_PID: 'Developer' # or Premium
    cap_add:
      - SYS_PTRACE
    stdin_open: true
    tty: true
    restart: unless-stopped
```

2.5 Lab feedback

- a) Were the labs relevant and appropriate and what about length etc.?
- b) What corrections and/or improvements do you suggest for these labs?

2.6 Appendix 1 – Create a test project in VS

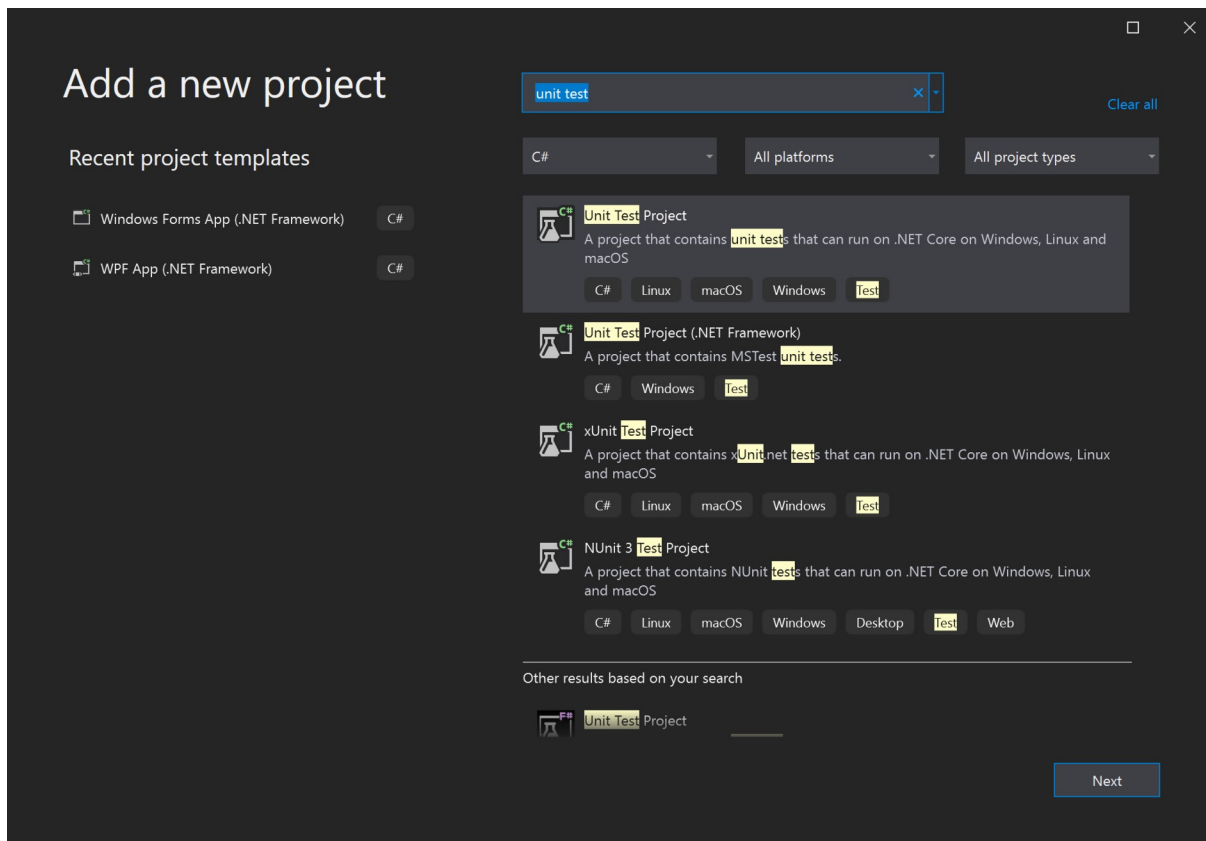
Select Unit Test Projects of type .NET 5.x or higher. The course exam project will use .NET 5.x or higher: <https://devblogs.microsoft.com/dotnet/announcing-net-5-0/> or <https://devblogs.microsoft.com/dotnet/announcing-net-6/>

This is preferable if you want to explicitly choose the test framework, use command line, use newer API standard, need better language compatibility or build your application for .NET 5.x or 6.x.

- Overview of porting from .NET Framework to .NET: <https://docs.microsoft.com/en-us/dotnet/core/porting/>
- Porting projects to .NET 5: <https://www.youtube.com/watch?v=bvmd2F11jpA>

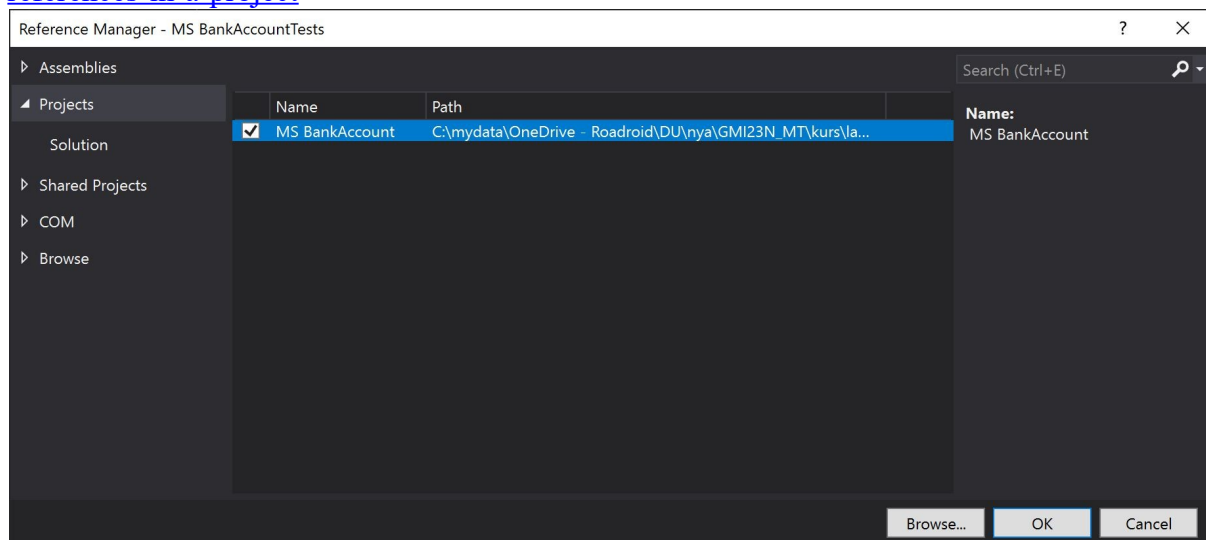
More resources are available in my porting resource folder:

http://users.du.se/~hjo/cs/gmi2j3/lectures/Porting_Projects_to_.NET5/









Add a reference to your BankAccount project.

Manage references in a project: <https://docs.microsoft.com/en-us/visualstudio/ide/managing-references-in-a-project>









If you want the latest MSTest version of the framework upgrade the Nuget packages. For MSTest .NET Core projects you have to install the Microsoft.NET.Test.Sdk package as well.








	Microsoft.NET.Test.Sdk by Microsoft	v16.9.1
	The MSbuild targets and properties for building .NET test projects.	
	MSTest.TestAdapter by Microsoft	v2.2.3
	The adapter to discover and execute MSTest Framework based tests.	
	MSTest.TestFramework by Microsoft	v2.2.3
	This is MSTest V2, the evolution of Microsoft's Test Framework.	

If you created a NUnit project you should have the Nuget packages described below installed and they may also be upgraded via the Nuget package manager.

NuGet Documentation: <https://docs.microsoft.com/en-us/nuget/>

	Microsoft.NET.Test.Sdk by Microsoft	v16.9.1
	The MSbuild targets and properties for building .NET test projects.	
	NUnit by Charlie Poole, Rob Prouse	v3.13.1
	NUnit is a unit-testing framework for all .NET languages with a strong TDD focus.	
	NUnit3TestAdapter by Charlie Poole, Terje Sandstrom	v3.17.0
	NUnit 3 adapter for running tests in Visual Studio and DotNet. Works with NUnit 3.x, use the NUnit 2 adapter for 2.x tests.	

If you created a xUnit project you should have the Nuget packages described below installed and they may be upgraded via the Nuget package manager.

	Microsoft.NET.Test.Sdk by Microsoft	v16.9.1 
	The MSbuild targets and properties for building .NET test projects.	
	xunit by James Newkirk, Brad Wilson	v2.4.1
	xUnit.net is a developer testing framework, built to support Test Driven Development.	
	xunit.runner.visualstudio by .NET Foundation and Contributors	v2.4.3
	Visual Studio 2017 15.9+ Test Explorer runner for the xUnit.net framework. Capable of running xUnit.net v1.9.2 and v2.0+ tests. Support...	