Object-Oriented Programming with C#

Unit Testing in Visual Studio

By Per Laursen
24-08-2025

Introduction

We have a couple of times discussed various aspects of code quality. One aspect of code quality – which is obviously quite important – is **correctness**. By correctness, we more specifically mean: does the code behave in accordance with the requirements specified for the code. The activity of determining this is called **testing**, and is in itself a very large topic in software development. Testing can be performed on a number of levels[1], ranging from **system testing** (testing the functionality of a system as a whole) to **unit testing** (testing the functionality of a single code "unit", e.g. a class).

We will not go into details about testing as such, but primarily focus on facilities for creating unit tests in Visual Studio. In that context, a unit test will typically be a test of a single class, i.e. testing the functionality of each method/property in the class.

The typical approach to creating a unit test is to create a unit test class for each class under test. If we are developing unit tests for an entire application, the unit tests will usually be defined within a single unit test project, which in a sense mirrors the application project itself.

---

[1] https://en.wikipedia.org/wiki/Software_testing#Testing_levels

Benefits of automated Unit Testing

Testing is often percieved as a somewhat tedious and repetitive activity, and since a thorough unit test often involves calling a specific method with a large number of parameter combinations, it can also become a very labor-intensive task to perform manually. The consequence is that testing can become an under-prioritised task, since the investment of effort can seem disproportional to the gain. However, if the unit tests can be specified in the form of code, it becomes possible to execute unit tests with very little effort. We still need to specify the test cases and create the unit test code, but that will then become a one-off effort.

Once you have a solid set of unit tests in place, it also becomes much safer to make changes in the code. We have a couple of times mentioned the concept of **refactoring**, which is the activity of improving the structure of the code, <u>without</u> changing the functionality. The latter constraint can easily be checked, if you have defined a solid set of automatic unit tests for the code. Once a small change has been made, you simply execute the unit tests, and check if all tests still pass. If not, you know that the small change you just made must be the change that introduced the error(s), and it should then be fairly easy to track down the problem and fix it.

The idea of relying on unit tests for verifying correctness can be taken even further. A software development process named **Test-Driven Development** (TDD) promotes unit tests to the most important process artifact, and lets them be the driving force in the process. An outline of the process is as follows:

1. **Write a unit test based on requirements**: The assumption is that requirements should be so clear, that it is possible to write a corresponding test <u>up-front</u>, i.e. before any code has been created at all. If this is not possible, it is seen as a symptom of inadequate requirements. Further work must then be done on detailing the requirements.
2. **Write code**: With the unit tests in order, we can now begin to write the actual application code. The code should be written with the goal of passing the tests, <u>not</u> creating perfectly designed code.
3. **Run and evaluate tests**: If some of the tests fail, we must go back to step 2. If they all pass, we can proceed to step 4.
4. **Refactor code**: Since all the tests pass, we can assume that the functionality is correct. However, since we have not coded with code quality as an explicit goal, the code structure may need to be improved by refactoring. The refactoring should be done according to the quality and design standards set for the code. Since we have unit tests in place, we can safely refactor.

Such a process puts testing at the center stage, and will in practice only be feasible if unit testing can be automated. This kind of development process is also knovn as **Red-Green-Refactor**
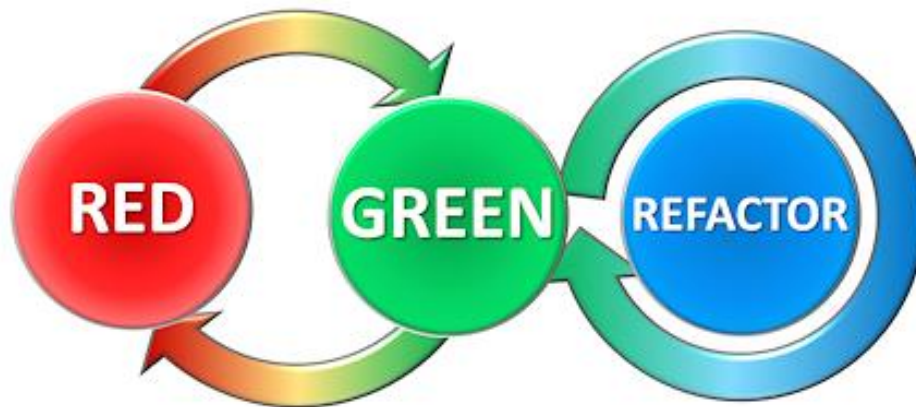
As we will see later, development environments like Visual Studio usually indicate a failed unit test with red (and passed unit tests with green), so you will usually start out with most unit tests being red, and then gradually turning more and more unit tests into green. Once they are all green, you can start refactoring, while keeping all the unit tests green.

Structure of a Unit Test case

A single unit test case will usually consist of a single method call, involving a specific combination of parameters. In addition to this, it may also be necessary to set the state of the unit under test to a specific state, in order to conduct the test in a meaningful way. Once the method call has been performed, we will need to evaluate if the result was as expected. In general, we therefore have three phases in a unit test:

- **Arrange**: Setting up the test "scenario", such that the test itself can be performed. You can also describe this as arranging the **preconditions** of the test.
- **Act**: Performing the actual action under test; this is typically a single method call (or using a property), but could also be a specific sequence of method calls. In general, the actions under test should be as atomic as possible.
- **Assert**: Once the testable action has been performed, a comparison between the expected and the actual outcome of the test must be made. The comparison is usually a *true/false* comparison; either the actual outcome matches the expected outcome completely, or it doesn't. Such a comparison is known as an **assertion**. You can also describe this as comparing the actual **postconditions** of the test to the expected postconditions.

The outcome of the Assert part is thus a yes/no answer to the question: did the test pass? There is no middle ground. This answer is often used to indicate the outcome of a unit test by color, as mentioned above. Red for failed, green for passed. This makes it very easy to get an overview of the outcome of a (large) set of unit tests.

Unit Testing in Visual Studio

Visual Studio supports unit testing directly, in the sense that you can create unit tests as described above, without having to install any third-party packages. For completeness, it should be mentioned that third-party frameworks for unit testing do exist, so the description here should only be seen as an example of how to create unit tests in practice. At the time of writing, the default unit test framework (for testing C# code) is named **MSTest**, and we will use that framework in the following.

We start out with an existing class – so we are not adopting the TDD process here – with a single method. The class is named **MediCare**, and contains a single method **SubsidisedExpense**.

```
public double SubsidisedExpense(double expense)
```

In Denmark, medical expenses for an individual are subsidised on a progressive scale, as given below (as of 2023):

Medical expenses (kr.)	Subsidy (%)
0 – 1.045	0
1.045 – 1.750	50
1.750 – 3.795	75
3.795 – 20.636	85
above 20.636	100

The functionality of the method **SubsidisedExpense** can then be described in terms of the expected outcome for various values of **expense**:

expense	Subsidy (%)
less than 0	throw an exception of type **ArgumentException**
0 or greater	return a value in accordance with the Subsidy table above

This looks fairly simple, but note that this does <u>not</u> translate into just two test cases. The subsidy table specifies five subsidy intervals, so a covering set of test cases should at least involve one test case within each interval. Deriving a covering set of test cases is a non-trivial discipline in itself, and we will not discuss it in detail here. It is, however, worth noting that there seems to be two categories of outcomes here; returning a correctly calculated value, or throwing an exception. We should be able to specify and execute unit tests for both categories.

Given the **MediCare** class – which is part of an ordinary C# application project – we now create a new C# project, with two features:

- It will be part of the <u>same</u> solution as the application project
- It will be of the type **MSTest Test Project**

The new unit test project is created as follows:

- Highlight the solution (<u>not</u> the project) in the **Solution Explorer** window
- Right-click, and choose **Add | New Project…** in the context menu
- In the dialog box, search for the **MSTest** project template (for C#), and choose it (click the **Next** button).
- Give the unit test project a name; this is completely up to you, so you can e.g. just call it **UnitTestProject**.
- Click the **Create** button. This will create the test project in the same solution as the application project.

After these steps, the new project be created. It will initially contain a single class called **UnitTest1**, which will look something like this:

```csharp
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

Since the purpose of this class is to test the methods (in our case just one method) in **MediCare**, we rename the class to **MediCareTest**; this is a typical naming convention for a unit test class:

```csharp
[TestClass]
public class MediCareTest
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

A feature to take note of are the so-called **attributes** [**TestClass**] and [**TestMethod**]. The attribute [**TestClass**] indicates to Visual Studio that this class contains unit test methods, and these methods will be executed when the unit test as a whole is executed. It is possible to define classes in a test project which are not as such part of a unit test – but maybe act to support a unit test in some way – which is why the attribute is needed. Likewise, a test class may contain methods that are not as such unit tests. Only the methods marked with [**TestMethod**] are actual unit tests.

Before starting on the first unit test, note that the test project needs to have a reference to the application project, before it can use classes and methods in that project. For the test project, select **Dependencies**, right-click, and choose **Add Project Reference…** In the **Reference Manager** window, select **Projects | Solution**, set a checkmark in the checkbox for the application project, and click **OK**. Now the test project knows the application project.

Let us now consider how to write a single unit test. Suppose it has been determined that a test case with an amount equal to 1.145 kr. is needed. The expected outcome is 1.095 kr., according to the subsidy table. We then create one test method for this specific case. The name of this method is not in itself significant, but – just as for ordinary code – we should of course choose a name that helps us understand the purpose of the method. Naming convensions for unit test methods are not as well-established as for ordinary methods; one suggestion is a convention along these lines:

MethodUnderTest_StateUnderTest_ExceptionOrOutcome

In our example, we could e.g. name a test case method like:

```
void SubsidisedExpense_1145kr_1095kr()
```

This example also illustrates another important feature of test case methods: they are parameterless, and do not return any value. All establishment of preconditions must thus be done inside the method itself. We now have an outline of our test method:

```
[TestMethod]
public void SubsidisedExpense_1145kr_1095kr()
{
    // Arrange
    // Act
    // Assert
}
```

What remains is to fill in code for the three stages. The **Arrange** part is fairly simple: it involves creating a **MediCare** object, and setting up variables for parameters and expected return values (it can be debated if the last part truly belongs to **Arrange**, but the most important point is to use your convention consistently):

```
[TestMethod]
public void SubsidisedExpense_1145kr_1095kr()
{
        // Arrange
        MediCare mCare = new MediCare();
        double expense = 1145.0;
        double expectedResult = 1095.0;

        // Act

        // Assert
}
```

The **Act** part consists of a single method call:

```
[TestMethod]
public void SubsidisedExpense_1145kr_1095kr()
{
        // Arrange
        MediCare mCare = new MediCare();
        double expense = 1145.0;
        double expectedResult = 1095.0;

        // Act
        double actualResult = mCare.SubsidisedExpense(expense);

        // Assert
}
```

The **Assert** part involves using the **Assert** class, which is part of the *MSTest* test framework. The **Assert** class contains a lot of methods for comparision between expected and actual results. In this particular case, we want to compare the value of two variables of type **double**: **expectedResult** and **actualResult**. This can be done with the method **AreEqual**:

```
[TestMethod]
public void SubsidisedExpense_1145kr_1095kr()
{
    // Arrange
    MediCare mCare = new MediCare();
    double expense = 1145.0;
    double expectedResult = 1095.0;

    // Act
    double actualResult = mCare.SubsidisedExpense(expense);

    // Assert
    Assert.AreEqual(expectedResult, actualResult, 0.01, "Fail 1145 kr");
}
```

The **AreEqual** method is available for a lot of types (**int**, **bool**, etc.), and usually has the structure: **AreEqual(valueA, valueB, message)**. The intention is that if **valueA** is equal to **valueB**, the assertion is considered successful. Otherwise, the assertion is failed, and the text in **message** can be used to display some additional information about the test case, if needed. In this particular case, an extra parameter is included. You might recall that care should be taken when trying to compare **double** values, due to possible rounding errors. Therefore, you can specify a maximal acceptable difference – here set to 0.01 – between the values.

The **Assert** class is an integral part of the *MSTest* test framework, and the **AreEqual** method should (almost, see later) always be used for comparing expected and actual outcomes. Now that we have created a single unit test, we can run it! You can open the **Test Explorer** window by choosing **Test | Test Explorer** from the main menu. This should produce something like this (note that this and the subsequent screenshots are taken from a different project, where ten unit tests have been defined):
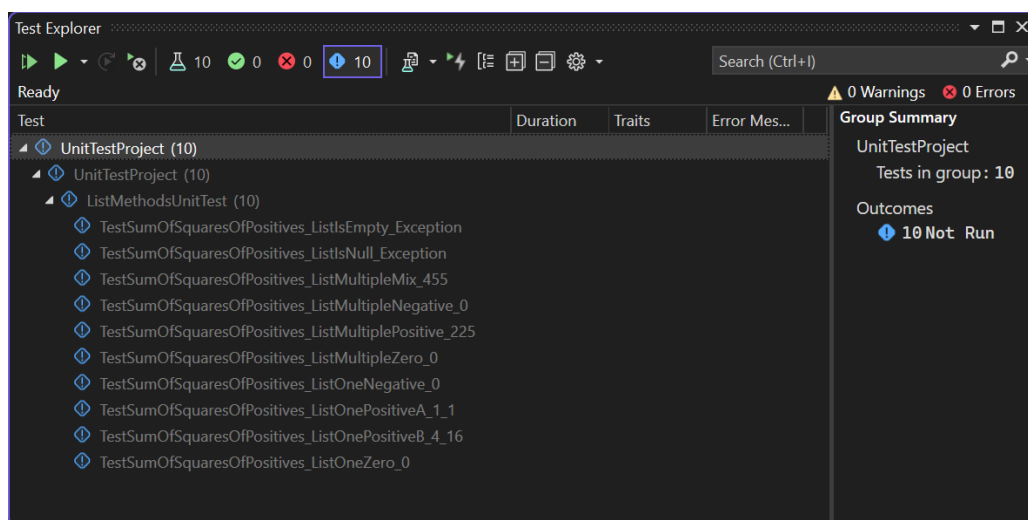
In the current state, this windows indicates that ten unit tests have been defined, but none of them have been run yet. Running the tests is done simply by cliking the **Run All Tests in View** icon (the leftmost triangle in the toolbar). Suppose that two of the ten unit tests fail. This would be displayed like this:
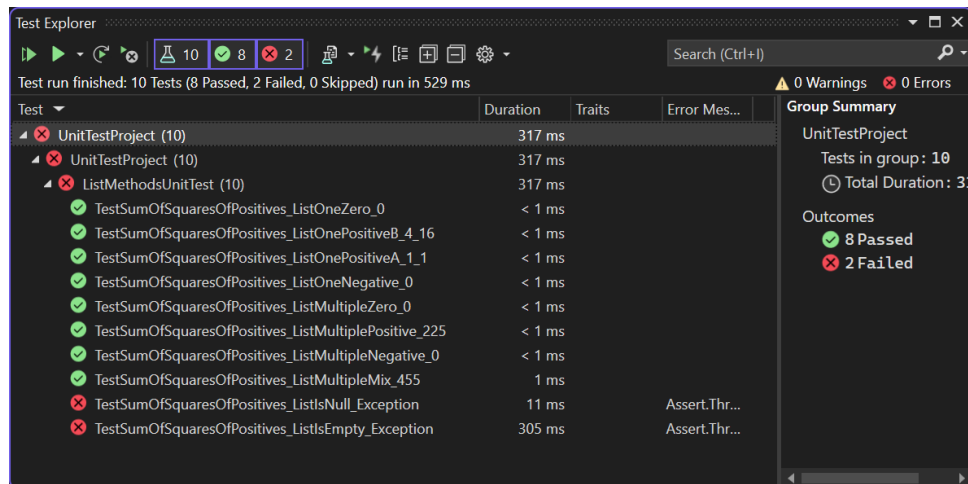
By selecting one of the failed tests, more details about the reason for the failure will be displayed in the right-hand window:
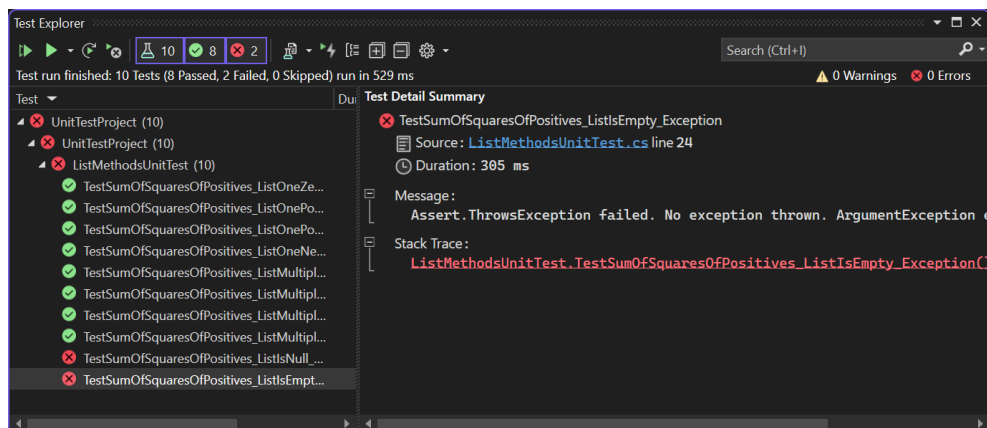
A failed test can of course indicate that the code being tested contains an error, but it could also be an error in the test code itself! We should obviously be quite careful when writing test code, but just as for ordinary code, errors tend to sneak in anyway.

Another useful feature is the ability to start a debugging session directly from the **Test Explorer** window. If you select the failed test, right-click and choose **Debug**, a debug session is started for that particular test case. You can then debug your code as usual.

The majority of test cases will tend to follow the pattern illustrated above: create an object of the class under test (here **MediCare)**, call the method being tested (here **SubsidisedExpense**), and compare the expected and actual outcome. But what about the case where an exception should be thrown (say, for negative amounts in the **MediCare** example)? This can also be tested within the framework, but the structure of the test method will look a bit different:

```
[TestMethod]
public void SubsidisedExpense_NegativeAmount_Exception()
{
        // Arrange
        MediCare mCare = new MediCare();
        double expense = -1.0;

        // Act & Assert
        Assert.ThrowsExactly<ArgumentException>(() =>
        {
                mCare.SubsidisedExpense(expense);
        });
}
```

Note the use of the **ThrowsExactly** method. This method is a Generic method (it takes a type parameter, i.e. the type of exception we expect to be thrown), and the parameter to the method is a function of the type **Action**, i.e. no parameters and no return type. This is why we need to "wrap" the method call into a function definition, like **() => { code to test...}**. This is a bit convoluted, but enables proper testing of this case. This also illustrates an important aspect of testing: it is not enough to test that valid cases are successful; you should also test that invalid cases fail in the expected manner!
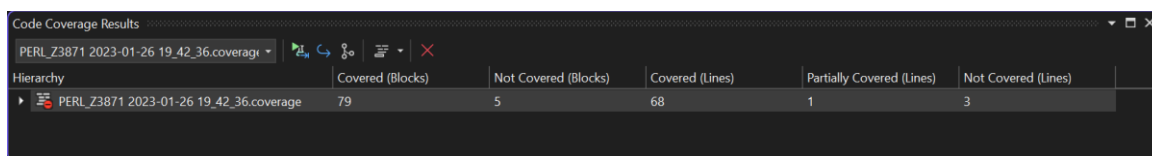
Live Unit Testing

The setup described above enables you to execute unit tests by a simpe click, but it is still up to you to activate the tests. This can perhaps be compared to the state of affairs for syntax checking many years ago. You would write your code without any help from the development environment, and then "activate" syntax checking by e.g. trying to compile the program. Modern development environments like Visual Studio now offer "live" checking of syntax, highlighting syntax errors as soon as you type. In Visual Studio, you can now also perform "live" unit testing. When activated, the set of unit tests runs continuously, and you get instant feedback with regards to the status of your unit tests.

This is how it's supposed to be... but to be fair, Live Unit Testing has yet to become a widespread success, and is also currently only available in the Enterprise version of Visual Studio. Right from the outset in 2017, many developers experienced Live Unit Testing as being quite resource-intensive, and this – perhaps undeserved – perception seem to linger still. At the time of writing, Microsoft has supposedly reworked Live Unit Testing to be more operational, but the jury is still out. We mention it here for completeness, so if you are curious, you should try your hand yourself by switching it on (go to **Test | Live Unit Testing | Start**) and get a feel for its current state.
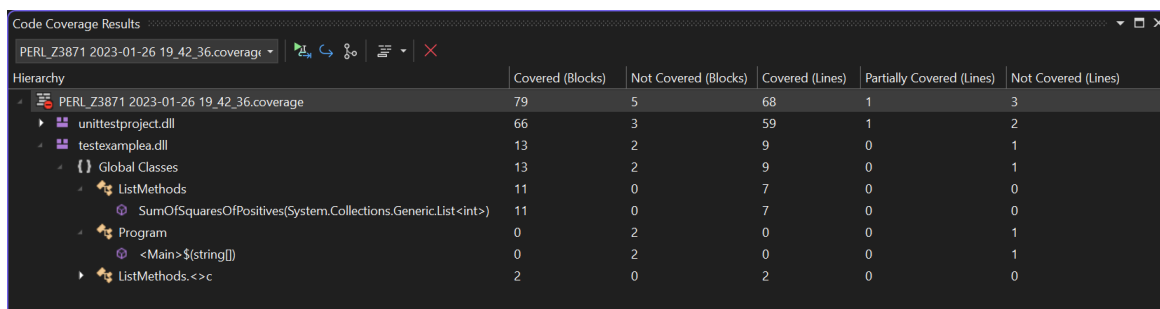
Code Coverage

As we suggested above, it is definitely not a trivial task to write a set of proper unit tests. Such a set should ideally test our code by subjecting it to any sorts of use that might occur in practice. But how do we determine that we have achieved this? One criterion – which may not in itself be a sufficient criterion – could be that the unit tests must activate all lines of application code at least once. If a set of unit tests does not achieve that, they must either be incomplete, or we have unnecessary code in our application… However, determining if a set of test cases activates all lines of code is a quite complex task to perform manually, once our code grows beyond the trivial. This is where the **code coverage** tool can be applied.

The code coverage tool is activated by choosing **Test | Analyze Code Coverage for All Tests** from the main menu. After a short while, a **Code Coverage Results** window should open, looking like this:

As indicated by the small triangle to the far left, it is possible to expand the result in a tree-like manner, which displays a more detailed breakdown of the code coverage:


We have now "drilled down" to the finest level of detail, which is the method level. In the example shown here, we are testing a single method **SumOfSquares** on the class **ListMethods**, and it seems our tests are doing quite well w.r.t. code coverage. That is, all lines of code are covered by at least one unit test. You should – as we mentioned before – of course be aware that 100 % code coverage is <u>not</u> the same as being sure that your program is now proven error-free! The code coverage tool can be used to track down places in your code that are not yet covered by unit tests, and is thus a tool to aid you in creation of additional unit tests.

Testing in more complex scenarios

In the above discussion, we have not shown any of the actual code in the **Subsidised-Expense** method, since that code is not as such relevant. It is the <u>functionality</u> of the code we are testing. The code itself is fairly straightforward, and only uses elements already present in C#, like the **List** class and the **for** and **if** control statements. Why is this important to note? When using e.g the **List** class – which is a part of the .NET class library – we tacitly assume that it is a well-tested and error-free class. So, if our unit tests reveal any errors, we assume that the error must originate from our own code. This is a reasonable assumption, when using classes from the .NET class library.

Suppose now that our **MediCare** class relied an another class that we ourselves have defined. It could make perfect sense to define a class **SubsidyTable**, which manages the subsidy intervals defined in the table above. This class could then depend solely on elements from the class library, i.e. a dependency like:
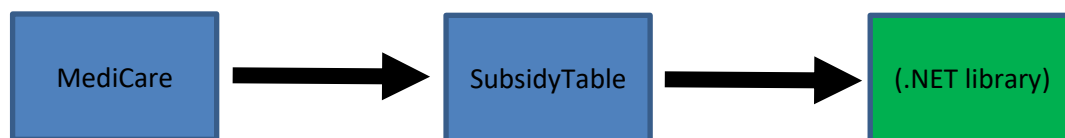
How should this change affect our unit tests? First of all, we ought to create separate unit tests for the **SubsidyTable** class, to verify its functionality. Once these tests have been added – and are successful – we need to consider the **MediCare** units tests. Can we simply keep the existing unit tests? An argument in favor could be that since we have added unit tests for **SubsidyTable**, we can now rely on this class in the same way as we rely on a class from the .NET class library. An argument against could be that even though we have successful unit tests for **SubsidyTable**, it is not unthinkable that it contains errors still. Another – more general – argument against is that if we allow classes under test to be dependent on "real" classes, it will become increasingly difficult to test classes, the deeper the chain of class dependencies become.

Suppose that the subsidy intervals managed by **SubsidyTable** were read from a database or through a web service. In order to test the **MediCare** class, we would then need to get a fully functional **SubsidyTable** class up-and-running for each test, maybe including a time-consuming connection to a database. This is clearly not optimal, and would prevent us from doing any testing before a fully functional **SubsidyTable** has been implemented…

What then? The usual approach to this problem is to use some kind of substitute class, when testing classes depending on other classes. There are different categories of such substitute classes, like Fake, Stub or Mock[2], but they are all classes that in some way try to mimic the functionality of the real class, while being much simpler with regards to implementation. A substitute class for the **SubsidyTable** class could be a class with exactly the same methods, but only containing some hard-coded values that are sufficient for testing classes depending on **SubsidyTable**.

This is as mentioned a very common approach, but how is it done in practice? Suppose that in the current implementation of the **MediCare** class, the class looks like:

```
public class MediCare
{
        private SubsidyTable _subsidyTable;

        public MediCare()
        {
                _subsidyTable = new SubsidyTable();
        }

        // Rest of class omitted
}
```

The class thus contains an explicit reference to the **SubsidyTable** class. This makes it difficult to reconfigure the class to use a substitute class, since we would need to rewrite the code to refer to the substitute class. Can we instead redesign the code to enable such reconfiguration? Indeed we can, by using interfaces and Dependency Injection! We said above that a substitute class should contain exactly the same methods as the original class. Another way to express this is to require the substitute class and the original class to implement the same interface. An interface for a class managing subsidy intervals could e.g. be:

```
public interface ISubsidyTable
{
        List<int> GetSortedPercentages();
        double GetIntervalLow(int percentage);
        double GetIntervalHigh(int percentage);
}
```

[2] https://www.martinfowler.com/articles/mocksArentStubs.html

The original **SubsidyTable** class can now implement this interface, but we can also create a substitute class **SubsidyTableMock**, which implements the same interface, but has a very simple implementation based on hard-coded values. With this interface in place, we can then update the implementation of **MediCare**:

```csharp
public class MediCare
{
        private ISubsidyTable _subsidyTable;

        public MediCare(ISubsidyTable subsidyTable)
        {
                _subsidyTable = subsidyTable;
        }

        // Rest of class omitted
}
```

The **MediCare** implementation is now unaware of the specific implementation of **ISubsidyTable** injected into it in the constructor, which makes it quite easy to update the unit tests:

```csharp
        public void SubsidisedExpense_1145kr_1095kr()
        {
                // Arrange
                MediCare mCare = new MediCare(new SubsidyTableMock());
                double expense = 1145.0;
                double expectedResult = 1095.0;

                // Act
                double actualResult = mCare.SubsidisedExpense(expense);

                // Assert
                Assert.AreEqual(expectedResult, actualResult, 0.01, "Fail 1145 kr");
        }
```

By introducing the **ISubsidyTable** interface, we have thus made the **MediCare** class as such more versatile, but also made it more testable!

Since this idea of using substitute classes in testing is so common, a number of third-party frameworks exist which can aid you in producing such substitute classes. We will not mention any specific frameworks here – some of then are commercial, others are open-source – but encourage you to investigate further on your own, if this looks like something that might be useful to you.

Testing – closing remarks

Testing is as mentioned earlier a very large topic in its own right, and this chapter only provides a very brief introduction to one aspect of testing. We have intentionally only focused on the mechanics of <u>how</u> to define and execute a unit test, and evaded the question of <u>if</u> you should define a specific test. The perfectionist view on testing would be that everything should be tested to the highest possible degree; in practice, you are seldom allocated resources to achieve this.

Testing can <u>never</u> provide you with a 100 % bullet-proof guarantee for code correctness, but should rather be seen as a tool for <u>increasing confidence</u> in your code. Try to identify parts of your code where the benefits of unit testing are most obvious (e.g. complex logic or high-risk code), and concentrate the initial effort there. Just as for many other aspects of software development, testing will always be a tradeoff between effort and benefit, and the exact balance needs to be found individually for each scenario.

Exercises

Exercise	UnitTest.1
Project	TestExampleA
Purpose	Implement a method, using existing unit tests as guidance.
Description	The solution contains two projects: • The project **TestExampleA** contains the class **ListMethods**, with the single method **SumOfSquaresOfPositives**. • The project **UnitTestProject** contains the class **ListMethods-UnitTest**, which contains test cases for testing the method **SumOfSquaresOfPositives**.
Steps	1. Study the **SumOfSquaresOfPositives** method. The method itself is initially empty, so focus on understanding what the method should do. This is described in the comments. 2. Study the test cases in **ListMethodsUnitTest**. Try to run the tests (open the **Test Explorer** window, by choosing **Test | Test Explorer**, and click on **Run All Tests in View** in the **Test Explorer** window). Note that some of the tests actually pass, even though the method is clearly not correctly implemented yet. 3. Implement **SumOfSquaresOfPositives** correctly, such that all test cases pass. 4. Do you find the existing test cases to be sufficient? Can you think of some test cases it would be useful to add?

Exercise	**UnitTest.2**
Project	TestExampleB
Purpose	Implement a unit test for an existing class.
Description	The solution contains two projects: - The project **TestExampleB** contains the class **Warrior**. - The project **UnitTestProject** contains the class **WarriorUnitTest**, which should contain test cases for testing the **Warrior** class. Initially, the unit test only contains a few test cases, and needs to be extended.
Steps	1. Study the **Warrior** class, until you have a detailed understanding of how it is intended to work. 2. Study the existing test cases in **WarriorUnitTest**. They are clearly insufficient… 3. Add new test cases to **WarriorUnitTest**, until you feel you have covered all aspects of the functionality. You can use the existing test cases for inspiration, with regards to how to structure test cases.

Exercise	UnitTest.3
Project	TestExampleC
Purpose	Implement both a class and associated unit test, given a requirement specification
Description	The solution contains two projects: • The project **TestExampleC** which contains the class **Currency-Exchange**. • The project **UnitTestProject** which contains the class **Currency-ExchangeTest**. None of the classes are complete, however. The starting point is the below requirement specification (see next page), from which the implementation of the **CurrencyExchange** class and the **Currency-ExchangeTest** class must be completed.
Steps	1. Implement the **CurrencyExchange** class and the **CurrencyExchangeTest** class, given the below requirement specification. If you are in doubt about a specific requirement detail, you must make a decision about how to interpret it, and work forward from that. Note that it is perfectly fine – even encouraged – to define interfaces, helper methods classes, etc. in order to create the implementation. 2. When you are done with the requirements specified below, you can try to extend the class (and the test class) along these lines: • If the currency cross **AAABBB** is specified, you can calculate exchanges from **AAA** to **BBB**. However, that should also make it possible to calculate exchanges from **BBB** to **AAA**. • Suppose you wish to make an exchange from **AAA** to **CCC**. This currency cross has not been specified, but **AAABBB** and **BBBCCC** have. This can be utilised to calculate the **AAACCC** exchange rate. • Exchange rates should be consistent. If you have specified **AAABBB** = 2 and **BBBCCC** = 3, it should not be possible to set **AAACCC** to e.g. 7 (it should be 6).

Exercise	UnitTest.3 (continued)
Requirement Specification	**CurrencyExchange – Definitions** • A **currency identifier** is defined as being a <u>three-character</u> acronym for a currency. Examples are **USD** (US Dollars), **EUR** (Euro), **DKK** (Danish Kroner), and so on. • A **currency cross** is a combination of <u>two different</u> currency identifiers. Examples are **EURUSD** (Euro to US Dollars), **DKKEUR** (Danish Kroner to Euro), and so on. • An **exchange rate** is a currency cross and a <u>positive</u> decimal number. Example: (**EURUSD**, 1.20), meaning that 1.00 Euro is worth 1.20 US Dollar. **CurrencyExchange – Requirement specifications** • It must be possible to <u>set</u> a number of exchange rates. • Trying to specify an <u>illegal</u> exchange rate (see Definitions), should cause an <u>exception</u> to be thrown. • It is permitted to <u>change</u> an existing exchange rate, simply by specifying it again. • Given a currency cross **AAABBB** and a positive amount of currency **AAA**, it must be possible to <u>calculate</u> the amount obtained by exchanging the amount to currency **BBB**. Example: Given **USDDKK** = 6.50 and an amount of 200 **USD**, the result should be 1300 **DKK**. • Trying to perform the calcuation with either an <u>illegal</u> (or <u>non-existing</u>) currency cross or <u>illegal</u> amount, should cause an <u>exception</u> to be thrown.