Test Driven Development Workshop (.Net)

Exercise Guide

Exercise Notes

- Use the 'Class Library' project type for all projects
- Ensure the project framework type is set to .NET3.5 in the project properties.
- Best practice is not to reference DLL's in the GAC. Create your own Windows folder and reference the required DLLs by browsing to this folder.

Exercise 1: Unit Testing

a. Phone Numbers

Objectives

The principal objective of this exercise is to get you to think about defining a suitable range of tests for some code, *before* you code the solution. You need to think of a good range of both positive and negative test cases, to ensure that the functionality you write both accepts the positive cases and rejects the negatives.

Problem statement. You will be receiving telephone numbers input by ordinary users, as e.g. entered through a web form. A ***valid*** phone number, as we need to define it here, must start with a '9', and be followed by exactly 10 digits. Assume that users may use spaces, hyphens, parentheses or commas when typing the number in. These are irrelevant characters which are to be discarded. Thus given e.g. "9 (608) 555-1212" as input, the function you are to define must return "96085551212". For this first implementation of a solution, for a number which does not meet the conditions of validity, "invalid" is to be returned.

Steps

1. You can use whatever combination of IDE, unit testing framework and plugins is your preferred tool set; for simplicity, the lab will be described in terms of plain Visual Studio and NUnit. Start Visual Studio and create a solution for this lab – e.g. UnitTestingIntro.
2. Specify a suitable default namespace for the project, e.g. TDD.UnitTestingIntro. Within that, create a class for the application code, e.g. PhoneUtils. Give it the stub of a method ValidatePhoneNumber(), which just returns null.
3. If your first class is PhoneUtils, add another which is PhoneUtilsTest. In the project references, Add Reference… to the .NET component nunit.framework. You will need one using directive for your test class, for NUnit.Framework. Define a test method with the single statement Assert.Fail("Not yet implemented"). Ensure that your testing code is appropriately annotated with the [TestFixture] and [Test] attributes. Build and Save All.
4. Fire up the NUnit GUI. With the File menu Open Project… command, open the UnitTestingIntro.dll in the Release directory of your Solution. You should see your test class with its single test method in the tree view of tests on the left. If you don't, go back and fix your test class (are the attributes, method signature correct?), then rebuild the project. NUnit will automatically refresh its view of the tests in the project. Click the Run button, and observe the red bar for your failing test.
5. Create a series of test methods, to test your as-yet-unwritten method. Each test method should contain one sample phone number string. It is up to you to decide what a suitable range of test cases will be. As you add each test, rebuild the project, then verify that it is displayed in NUnit, and that all your tests fail.

6. Now return to your application code. The easiest way to implement it is to avail yourself of the Regex class in the System.Text.RegularExpressions namespace. Note that if you are not familiar with this API, or are a bit rusty on it, you can actually use unit tests as a way of learning it. That is, create a separate test class where the test methods test your understanding, your predictions, of how the regular expression API behaves.
7. Some points to keep in mind in writing good testing code. Factor out any set-up common to all your tests to a SetUp method. Ensure your test methods all have fully self-describing method names. Use variables like string expected, string computed, and ensure you get them the right way round. Use the overloaded version of Assert.That() which takes a string message, and write informative messages for each test. Try the Category attribute.

Exercise 1: Unit Testing
b. Date Arithmetic

Objectives

The principal objective of this exercise is to get you to think about defining a suitable range of tests for some code, *before* you code the solution. You need to think of a good range of both positive and negative test cases, to ensure that the functionality you write both accepts the positive cases and rejects the negatives.

Problem Statement. Suppose there is a Date class which is just a wrapper around three ints, for day, month and year. You are tasked with writing a class for providing date calculations. Firstly, you must provide a method DateByDays(), which takes a Date and an int, and returns a new Date which represents the date from the input date by that number of days. In this first iteration, ignore the complexities of leap years (and even, if necessary, the differences in the number of days in a month). Follow the same methodology as described above. First, define the stub of your new class. Then create an NUnit test class for it. Devise a range of test cases that will test your method with different int delta days. (Assume the Date argument will be passed a Date with three values representing a valid date.) Then implement DateByDays() to pass your tests.

Exercise: 2a Stubs and Mocks – The Lottery Number Generator

Objective

The purpose of this lab is to allow you to walk through some code to simply gain a better understanding of the difference between a Stub and Mock.

1. Open the project **Lottery_Stub_and_Mocking**.
2. Examine the classes in the solution **Lottery_Stub_Mocking_BOs_CS**
3. Look for the Program.cs file under the **LotteryDriver** C# solution
4. Step through it using the debugger to see what happens
5. Once you're confident it makes sense, step through test `testStubGenerateRandomSetWithMocks()` in the file LotteryTest.cs which can be found in the solution **Lottery_Stub_Mocking_BOs_CS**

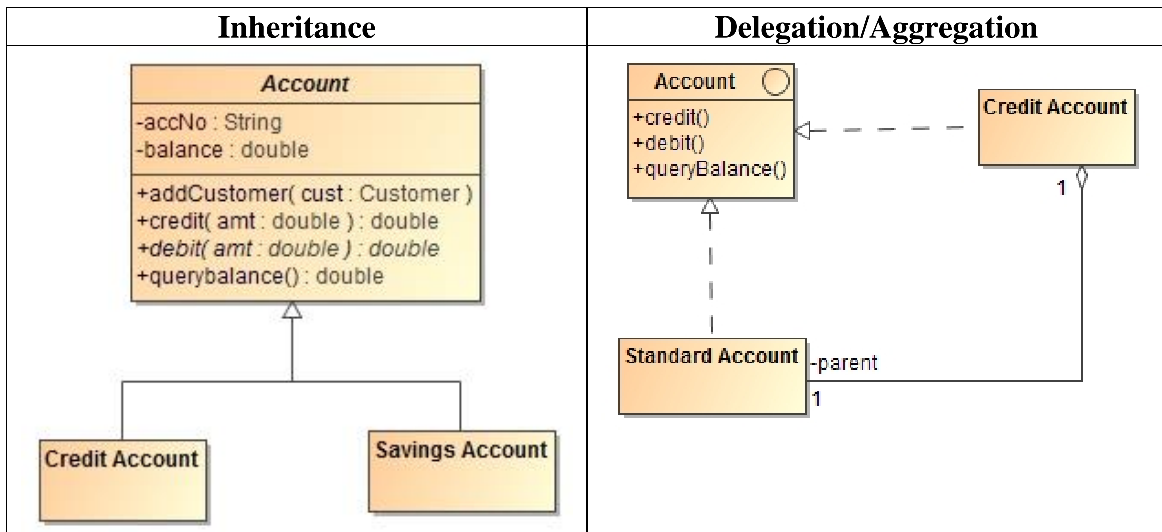
Exercise: 2b Mocking – Bank Account with IoC

Objective

Separate out a classes' parent (inheritance) and use delegation/aggregation instead. Once you understand the model, you will mock the parent.

Using normal inheritance in your code leads to strongly coupled systems. By converting that relationship into delegation, you decouple the classes and allow for flexibility and better maintenance of your code.

Consider the following example of Bank Account and a Credit Account. Because inheritance has been used, any changes made to the parent are automatically propagated to the children. This would mean potentially large amounts of regression testing and the inability to change the type and instance of the parent at runtime, it could only be done at design time.



In the Delegation model, the StandardAccount will play the role of a Parent (the delegate) to the CreditAccount (the aggregate). When you use this pattern it is important to note that both delegate and aggregate implement the same root interface but, the aggregate will contain an object of the interface type and should not be aware the delegates type.

1. Begin by opening the project MockAccountExample
2. Examine the source files and run it to make sure you understand its design
3. Rename the Account class to AccountImpl
4. Make a copy the AccountImpl Class and call it BankAccount, change its language type to that of an interface, be sure to remove anything that cannot be placed on an interface such as method bodies and attributes
5. Make AccountImpl implement the BankAccount interface
6. Change CreditAccount's relationship to BankAccount from inheritance to an interface implementation
7. Add a new data member to CreditAccount – `private BankAccount itsParent`
8. Change CreditAccount's constructor so you must pass a parent object of type BankAccount to it, initialise the `itsParent` with this parameter
9. Modify CreditAccount so all the calls that are made using the default inheritance features of the language to base member methods must be done like this `itsParent.methodName()`
10. Fix any other errors that occur in the code
11. Run the application using `Program.cs`
12. If all works, you have implemented inheritance using delegation and the tests should still work (you may need to alter how the objects are constructed)

Moving to Mocking

1. Write a new test that will mock the BankAccount interface (as you saw in the **Lottery_Stub_and_Mocking** lab), so you will not supply to the CreditAccount object an instance of AccountImpl but a mocked object of the BankAccount interface
2. Define the operations you want to mock, such BankAccount.getBalance() and BankAccount.debit()
3. Use `mock.AssertAll()` to verify that the methods were called

Exercise 2c Mocking

Objectives

The objective of this lab is to continue on from the previous exercise, in writing NUnit tests which drive the development of an implementation that passes them. Here we will be developing a class which interacts with a relational database, such as SQL Server. You can use whatever mocking framework, and syntax within that framework, you prefer. Rhino Mocks with the using() syntax is a good choice for a first iteration. If time permits, experiment with alternative syntaxes within the framework, and even a completely separate framework such as Moq.

Let's suppose that we already have some of this class already written:

```
using System;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace TDD.Mocking
{
    public class SqlSvrConnect
    {
        public IDbConnection OpenConnection(string connectString)
        {
            IDbConnection ds = new SqlConnection();
            ds.ConnectionString = connectString;
            ds.Open();
            return ds;
        }

        public IDataReader GetResults(IDbConnection con, string sql)
        {
            IDbCommand command = con.CreateCommand();
            command.CommandType = CommandType.Text;
            command.CommandText = sql;
            IDataReader reader = command.ExecuteReader();
            return reader;
        }
```

One of the benefits of TDD with mocks is that it naturally encourages a style of breaking an app down into lots of small, testable methods. Here we are not going to test the method OpenConnection() – to do that would be to write an integration test, since it would involve actually invoking SqlConnection(). Instead we are going (in the second part of the exercise) to mock the instance of IDbConnection which it creates for us.

1. Let's start with a relatively straightforward bit of mocking. We have been tasked to develop a generic method

```
public string DisplayResultsAsString(IDataReader reader, int numCols)
```

to iterate through the result set of an IDataReader and return them in a string of a certain format.

How this app would typically work would be that first the method OpenConnection() gets called with a specific connection string, such as

```
"Data Source=(local);Initial Catalog=UserDB;Integrated Security=SSPI"
```

The connection object this returns (if the connect to the database succeeds) is then passed to GetResults() shown above, with a SQL string such as

```
SELECT Uid, Lastname FROM Users
```

The IDataReader that this produces will be passed to the method we are going to write, DisplayResultsAsString(), with a number of columns to display set e.g. to 2. A typical implementation of the method would be to call the IDataReader's Read() method inside a while loop, getting the data back from the first two columns as strings and appending them to a StringBuilder.

Write a TestDisplayResultsAsString() method which mocks an IDataReader containing two records. At the end of the test method we will want to test the output that DisplayResultsAsString() returns, for equality with a string such as

```
fred001 : Foggs : \nbill100 : Boggs : \n
```

I.e. this shows the format of the string which the method is required to produce: each record in the IDataReader is expressed on its own line, and each item of data within a record concatenated with the next with a " : ". Much of the work in the test method will be involved with setting up the expectations on the mock IDataReader necessary for our method to convert it to such a string. Notice that since the methods in the SqlSvrConnect class are instance methods, you will need a (real) instance of that class to invoke DisplayResultsAsString() on.

2. The next method we have been asked to write is

```
public bool ValidateUser(IDbConnection con, string user)
```

This method is going to validate a given user id by looking it up in the relevant table (a more realistic example might take an additional parameter, string password). The method will contain the relevant SQL needed for this validation:

```
SELECT 1 FROM Users WHERE Uid=@user
```

The method is going to need to create an IDbCommand from the IDbConnection passed to it, an IDbDataParameter from the IDbCommand, on which it will set the parameter name (to "@user") and the parameter value to the user string passed in. It will then execute the command and obtain a DataReader, which it can interrogate to discover the boolean status of the SQL query.

In the TestValidateUser() method you write, you will probably need to declare five mock instances of interfaces (IDataParameterCollection being the fifth). Much of the work in the expectation setting phase will be taken up with declaring how one of these mock objects comes as the return value from a method call on another. Unless you

are steeped in this API, you will probably find it most natural to write a line in the ValidateUser() method, then write the line or lines in the Test method to set up the mock for it, and its expectations.

Exercise 3: TDD Intro
a. Seconds To Words

Objectives

The objective of this lab is to develop code which will convert a given integer number of seconds into a string representation in English words. So for instance 3723 is to be converted into "1 hour, 2 minutes and 3 seconds".

What is critical now is that the code should be developed in a test-first way. That is, you will work in small steps, at each step specifying what you are going to develop in the form of a unit test. You will gradually build up a suite of tests, and you will be re-running these tests very frequently.

Step 1
Create a new C# Class Library project Seconds2WordsApp in a new Solution Seconds2Words. Name the class in it Seconds2Words. Then from the Test menu, pull down the command New Test… With the Basic Unit Test template highlighted, create a test class Seconds2WordsTest in a new C# test project Seconds2WordsTests (in the same overall Solution Seconds2Words, and in the same namespace as the application class.) Ensure this second project has a reference to your application code project.
You need to create a "getting things set up test"; the obvious candidate in this case is for 0 seconds. So define a TestMethod ConvertZeroSeconds(), and make an Assert about what you expect – that is, when your convert method is passed the integer 0, what string should it return? At this point, of course, your test class will not compile, since it will be making an assertion about a method in your application class which does not yet exist.
Do the minimum to get your test to compile: put into your application class a convert method that returns the empty string. At this point it is critical that you now run the test, to verify that you get the "failing test" outcome.
Do the minimum to get your test to pass. Remember at all times not to let your application development run away from what is specified by the tests; don't attempt to develop the next bit of functionality before you have specified the requirement in the form of a test.

Step 2
Deal with simple seconds (below one minute), handling the English pluralisation corner case ("1 second").

Step 3
Deal with simple minutes (below one hour). Suggested sequence of tests: 122 seconds. 61 seconds. Ensure that your test method names are as informative as possible (e.g. ConvertPluralMinsSingularSecs().)

Step 4
Deal with whole minutes (below one hour). Suggested sequence of tests: 180 seconds. 60 seconds.

Remember DRY. You should find you have used the same logic to determine the pluralisation suffix both for “minute(s)” and “second(s)”. So now we can see a simple illustration of the way in which having a suite of unit tests gives us confidence to undertake a refactoring of our code. Assuming you have green bar/all tests passing, highlight one section where the code has been duplicated, and use the Refactor menu’s Extract Method… command to pull this out into a separate method. Replace the other section of duplicated code by a call to this method, and ensure that in the method definition, the method parameter is named in a way which is neutral between minutes and seconds. Crucially, re-run your tests to verify that they still all pass.

Step 5
Handle basic hours (less than a day). So for instance 11045 should be translated into “3 hours, 4 minutes, 5 seconds”.

Step 6
Handle the remaining cases for hours: singular hour, minute and second; whole number of hours; multiple hours plus whole number of minutes; multiple hours, seconds but no minutes, etc.

Step 7
Convert digits to words. A common style convention is to express all single digits by their English equivalents, and all numbers greater than nine by numerals. So convert e.g. 14467 into “four hours, one minute, seven seconds”, and 21612 into “six hours, 12 seconds”.

Step 8
Handle days and weeks.

Exercise 3: TDD Intro
b. Money

Objectives

The objective of this exercise is to work through a development task in the TDD style, a sequence of short steps.

Problem statement. An application for managing portfolios of equities, priced only in US Dollars, has been successfully developed. A request has come in, to extend it to handle securities priced in other currencies. So, to take a simple scenario, we want to be able to handle a situation like this:

Share	Quantity	Price	Value
IBM	1000	25 USD	25000 USD
BP	2000	5.50 GBP	11000 GBP
			47000 USD

Somewhere, exchange rates will have to be represented, as here assumed £1 = $2. We can see from this table that we need to represent at least two calculations:

i) In the rows: multiply a price in a given currency by an amount
   $5 * 2 = $10
ii) In the last column: add amounts in two different currencies (given an exchange rate)
   $5 + £5 = $15 (if £1 = $2).

Maintain a paper-and-pencil *To Do* list, starting with these two calculations as mnemonics for what needs to be achieved. As we go along, inevitably new tasks will occur to us, and need to be added; conversely, as issues get resolved/implemented, we can cross them off.
If you want to keep a record of the progress of your TDD, you can create a series of packages, qa.tdd.money.iteration00, qa.tdd.money.iteration01, etc. As you go along, you can qualify each preceding [TestFixture] with the [Ignore] attribute.
The exercise is a relatively “hand-holding” one, in which the development steps are suggested in detail. It is adapted from material in the first part of Kent Beck’s *Test Driven Development By Example.*

Iteration 0
Starting with task (i), write a MoneyTest class with a method to test the multiplication task.

a) Assume there is going to be a class Dollar, with a constructor that takes an int, and create an object to represent $5.
b) Assume that this class has an instance method which allows you to times it by an int amount like 2
c) Finish the test with an Assert() statement, with an exception message like “$5 * 2: amount should be 10”, 10 as the expected value, and some way of finding the actual value now in your dollar object. For the latter, the quickest solution for now: a property *Amount*.

Various issues may strike you about this test: using integer values for monetary amounts; the side effect that multiplying a five-dollar object has changed it into a ten-dollar object. But the point is we are getting a quick initial test up and running. Already it is prompting us to think of further things that will need to be done. As Beck says, "We'll make a note of the stinkiness and move on". In other words, add any issues you foresee with this to the paper *To Do* list, and get on with the task in hand, of getting NUnit to go red, then green.
Do the absolute minimum necessary to get the test to compile:

a) Create the class being tested
b) Give it (the stub of) the relevant constructor
c) Give it (the stub of) the relevant multiplication method
d) Give it the property.

The test now compiles. We know it's going to fail, but - very important – run it. Koskela makes the point that he always runs the test at this point, because red confirms he is running the right test (*Test Driven*, p. 53). Inevitably it happens sometimes that you run an old test, which would be very misleading. So always check for red first.

Iteration 1
If you are working in the same solution and wish to keep a record of its evolution, without using version control, you could at this stage copy MoneyTest.cs and Dollar.cs to MoneyTest1.cs and Dollar1.cs. Update the namespaces in the two new files, and continue on with them.
Make the *smallest step* necessary to get the test to pass. This will be the hardest part for most developers. If you're not sure what this should be, check the TDD Worked Example on the slides. When you come to generalise your solution in a moment, leave that initial implementation in the source file – but commented out – so that your instructor can see that you did take that small step. Run the test, and confirm we now have green.
Now we have a passing test, we move to the other principal injunction of the TDD methodology: remove duplication. Again, try to move in very small steps, smaller than you would normally take. The point is to get a feel for the possibility of moving in tiny steps when you need to. The idea is that there is some flexibility in the TDD process: when you are confident, you can take bigger steps; when you hit difficulties you can move back to smaller steps.
At the end of Iteration 1 you should have an implementation that you are satisfied correctly passes the test. Add another multiplication test, with different values, to confirm that your solution generalises. Remove duplication in the tests by factoring out the construction of the test fixture object to a SetUp() method. Cross the multiplication task off the list.

Iteration 2
Address the issue that multiplying the $5 object had the side-effect of changing it. This is wrong. Dollar is a Value Object; once a Dollar's state has been set when it was constructed, that should not be capable of being subsequently changed. Re-write the test so that your multiplication method returns a new Dollar object. Clearly, we are making a refinement to the interface of Dollar. (Question: now, for any multiplication test, there are two Dollar objects: the original one, and the new product. What Asserts should any multiplication test have?)

Change the application code so the tests compile. Run and fail. Then refactor to make the test pass.

Iteration 3
Value objects need a notion of equality – one $5 bill is equals-to another. This fact suggests the test we need to write for Dollar.Equals(). Add a new test to your test class, for Dollar.Equals(). Add another test for non-equality (hint: the assertion message might be "$5 should not equal $10".) C Sharp 101: are these tests going to compile? run? pass or fail?
Now back to application code: override Equals() in the Dollar class. (There's nothing tricky about this; the essential information you need is contained in the preceding paragraph. To be correct, you can add checks to the effect that the other should not be null, and should be a Dollar.) Re-run the test class until all tests pass satisfactorily. (The rules of C Sharp, like Java, say that if you override Equals() for a class, you must also override GetHashCode(). Either, put the latter onto the *To Do* list, or, if you're uneasy about unresolved warnings in your projects, be sure as always to write the *test* for this method before the method itself.)

Iteration 4
Now that we have a workable value object Dollar, we can recast the original test asserts in a more properly OO format. Instead of an expected int 10 compared to the amount that comes from doubling 5 Dollar, we can rather have an expected new Dollar(10) compared against the result of that multiplication. Re-write the test, and run it.
The whole purpose of this development exercise is to handle multiple currencies, and now we are in a position to start. The first step is of course not to copy the Dollar *class* for a new currency, but to copy the dollar multiplication *test* for a new currency – call it TestFrancMultiplication() or some such. Now take the smallest step necessary to get this to compile – copy and paste the Dollar class (perhaps Dollar4.cs now) to a class for your new currency.
Make sure you always have tests for all the code. So introduce tests for the other methods in your new class, i.e. for equality/non-equality of Francs. Re-run.

Iteration 5
Remove duplication! In this iteration, we're going to see how having a suite of unit tests will give us confidence to start a substantial refactoring of the code. At *each* of the following steps, re-run the tests. Suggested sequence:

a) Introduce Money class
b) Make the classes for specific currencies subclass it
c) Move Amount to the base class, with appropriate adjustments
d) Define an Equals() method for the base class, generalising it as appropriate (and GetHashCode(), if you have implemented this.)
e) Delete Equals() (and GetHashCode()) from the subclasses

Iteration 6
We've introduced Dollar and Franc, but there is one set of equals/non-equals tests which we have glaringly failed to include. Rectify this now.
These tests fail. We expect 5 Dollars not to be equals to 5 Francs, and *vice versa*, but there is nothing in the generic Equals() method to rule this out. A quick fix is to add an extra condition using the GetType() method on the two objects being compared.

This, as Beck says, "is a bit smelly" – we are using meta-data from the world of programming rather than domain knowledge from the world of finance. Make a note of this on the *To Do* list.

Iteration 7

There is still a lot of duplication. The two currency classes are essentially isomorphic; there doesn't seem enough of a need in this application to warrant retaining them. There are a number of ways the design could go from here. If we want to remove references to the Dollar and Franc classes, and do so in small TDD steps, a first small step would be to replace constructor calls to Dollar(5) etc. with factory methods in the base class. Again, we start by specifying this with a test; replace the original creation of a $5 object (step 0a above) with a call to Money.Dollar(5). Now introduce this static factory method into Money and run the tests.

The next step to removing Dollar from the test code: change the type of this $5 object from Dollar to Money. Once this is done, any lines in the test class invoking the multiplication method Times(2) on this Money object will now throw up a compilation error, since there is no such method in the base class. A quick fix to get this to compile is to introduce an abstract Times() method into the base class. With suitable minor adjustments (e.g. signatures of Times() in the derived classes), the tests should still all pass.

This should give us confidence to remove all references to Dollar and Franc (as types) from the test class. Go ahead and do this, re-running the tests with each small adjustment to MoneyTest. This is an important achievement: the client code no longer has any knowledge of the subclasses. We will have a suite of tests which potentially allow us to refactor away completely the Dollar and Franc classes.

Keep up your maintenance of the *To Do* list.

Iteration 8

A design decision which may already have occurred to you, if we wish to have a single Money class but distinguish currencies, is to introduce a representation of currency – USD, GBP, CHF, etc. For simplicity, suppose this will be a string. So how can we test this? A quick and dirty initial idea, by analogy with *amount*, would be to assume that a Money object will have a property *currency*, which, depending on the factory method used to create it, will evaluate to "USD", "CHF", etc. Implement tests for this.

To get the tests to compile, introduce the property into the Money class. Run and fail. The factory methods in the base class still call the constructors in the subclasses. So it is a simple step to change the constructors to set a value for *currency*. Run and go green.

Iteration 9

We now have a full suite of tests, incorporating a notion of currency. So now let's move towards refactoring away the subclasses. Keep re-running the tests at every opportunity.

a) Give Money a constructor which takes an amount and a currency
b) Change the factory methods Dollar() and Franc() to use this constructor
c) Remove *abstract* from the signature of Times(), and define it
d) Remove *abstract* from the signature of Money (you had to do this when you gave the class an abstract method)

e) Ensure any remaining references to the derived classes are removed from Money
f) Delete the derived classes
g) Adjust the definition of Equals() to use currency rather than GetType()

Iteration 10
One of the key requirements we are aiming to meet is to have the capacity to form the sum of two quantities of money, in different currencies. An obvious first step would be to first test adding two quantities of the same currency.
You know the routine by now: implement a new test method for this case, then make it compile and fail, then do the minimum to get it to pass. If you're not confident of the implementation, use the Fake It strategy, and then generalise; if you are confident that the implementation is obvious, go straight to that – with of course the option to back out and step more slowly if that doesn't work.

Discussion
What lessons do we learn from this exercise? What was good? What was bad?

Exercise 4: TDD Main
a. Amazonian

Overview

User stories have become an important part of Agile development, as a way of capturing requirements. But a user story is far less detailed than a traditional statement of a requirement; it is a short specification written in ordinary language about something which some user of the system to be developed wants to achieve. A good way of writing them is to follow the template:

As a <role>, I want to do <goal>, so that I can achieve <aim>

User stories are sometimes described as a "promise for a conversation" – the starting point for a discussion between developer and client. In these exercises, treat your instructor as the client: for anything you need clarifying, go to the client.

Your objective is to develop the core business tier functionality for a web site like amazon. Your team is not concerned with the front end – there is another team working on the user interface.

You are to develop a solution taking a wholly test-driven approach to it. You may want to read through all the user stories to get an idea of what it's all about, but you should tackle it one user story at a time, keeping in mind YAGNI at all times. So for user story 1, for instance, you will probably decide you need a Product class. Put only as much detail into that as you need for that user story; to get started, a product perhaps needs only a name and a price. Later on you may find you may need to add further detail to that class – but do so only when it's needed by a specific user story.

As always, keep in mind good coding practice for both your application and testing code. For instance, respect DRY:

a. If you find you are repeating the same test fixture set-up in multiple places, consider factoring this out to a common fixture repository
b. Depending on your implementation, by around story 3 you may find that two of your classes share some common functionality: consider refactoring out a common base class

Note: there are more user stories here than you are likely to have time to complete. If you get to story 5, you might like to explore the difference between specifying a test using a real .csv file versus mocking the interaction with the file system.

1. As a user
   I want to ask for a list of products
   So I can choose what I want to buy
2. As a user
   I want to be able to choose a product
   So I can put it in my shopping basket
3. As a user
   I want to be able to remove a product from my basket
   So that I can change my mind
4. As a user

I want to be able to order the contents of my basket
So that I can have them delivered *

5. As an administrator
   I want to be able to define the products in a flat file
   So that the catalogue can be loaded up from file
6. As a user
   I want to be able to save my basket
   So that I can come back at a later date and continue
7. As a user
   I want to be able to register for the site
   So that my address and credit card details will be saved

* Footnote to story 4. So that development can proceed in parallel between your team and the user interface team, your respective project managers have agreed a common interface. Your solution must be coded in terms of implementing the following two interfaces:

C#

```
public interface IOrder
{
    string GetCustomerId();
    IList<ILineItem> GetLineItems();
    int GetTotalCost();
    void Place();
    // ONLY AFTER Place() DOES THIS METHOD RETURN AN
    // ORDER NUMBER OTHER THAN 0:
    int GetOrderNumber();
}

public interface ILineItem
{
    Product GetProduct();
    int GetQuantity();
    int GetCost();
}
```

VB

```
Public Interface IOrder
    Function GetCustomerId() As String
    Function GetLineItems() As IList(Of ILineItem)
    Function GetTotalCost() As Integer
    Sub Place()
    ' ONLY AFTER Place() DOES THIS METHOD RETURN AN
    ' ORDER NUMBER OTHER THAN 0:
    Function GetOrderNumber() As Integer
End Interface

Public Interface ILineItem
    Function GetProduct() As Product
```

```
    Function GetQuantity() As Integer
    Function GetCost() As Integer
End Interface
```

Exercise 4: TDD Main
b. My Builder

Overview

User stories have become an important part of Agile development, as a way of capturing requirements. But a user story is far less detailed than a traditional statement of a requirement; it is a short specification written in ordinary language about something which some user of the system to be developed wants to achieve. A good way of writing them is to follow the template:

As a <role>, I want to do <goal>, so that I can achieve <aim>

User stories are sometimes described as a "promise for a conversation" – the starting point for a discussion between developer and client. In these exercises, treat your instructor as the client: for anything you need clarifying, go to the client.

Your objective in this exercise is to develop the core business tier functionality for a web site like mybuilder, jobsorted or findatrade. At this stage of the project development, you are not concerned with issues such as either the user interface, or how data will be stored.

You are to develop a solution taking a wholly test-driven approach to it. You may want to read through all the user stories to get an idea of what it's all about, but you should tackle it one user story at a time, keeping in mind YAGNI at all times. So for user story 1, for instance, you will probably decide you need a Job class. Put only as much detail into that as you need for that user story; a job perhaps needs a title, a description, and its postcode location. Later on you may find you may need to add further detail to that class – but do so only when it's needed by a specific user story.

[Note: there are more user stories here than you are likely to have time to complete.]

1. As a user
   I want to list jobs that have been posted
   So I can get an idea of what to write for my job
2. As a user
   I want to list builders with a specific skill
   So I can get a list of their names
3. As a user
   I want to list builders with a specific skill
   So I can compare their reviews
4. As a user
   I want to be able to register for the site
   So that I can post a job
5. As a registered user
   I want to be able to post a job
   So that builders can tender for it
6. As a builder
   I want to be able to register for the site
   So that I can tender for jobs in my area which match my skills
7. As a registered builder

I want to be able to tender for a job
So that I have a chance of getting it

8. As a registered user
   I want to be able to list the builders who have tendered for my job
   So I can rank them by average rating
9. As a registered user
   I want to be able to send a message to a builder
   So I can ask for clarification or invite them for a site visit
10. As a registered user
    I want to be able to appoint a builder for my job
    So they can come and do the work, for the price agreed
11. As a registered user
    I want to be able to review a builder
    So others can see how good/bad they are
12. As an administrator
    I want to be able to load and save builders from/to persistent storage*
    So that the data can be recovered in the event of a system crash
13. As an administrator
    I want to be able to load and save jobs from/to persistent storage
    So that the data can be recovered in the event of a system crash
14. As an administrator
    I want to be able to review each job
    So that I can accept it for the site or reject it

*Note: “persistent storage” could be: relational database; CSV file; XML file. Discuss with your instructor. Builders should of course be saved with their *multiple* skills.

Exercise 4: TDD Main
c. Blackjack

Objectives

The objective with this exercise is to work with a much more loosely-specified problem. By thinking first about how to write tests, you will sharpen up the requirements, make design decisions, as well as driving the development process.

If you're working with an edition of Visual Studio with a Refactoring menu, take some time to acquaint yourself with some of the refactoring commands it offers. Try out some of the refactorings on some old code of your own. Try to use some of these operations during your coding of the exercise.

Problem statement. Develop a text-based application that will simulate the card game Blackjack (Vingt-et-un , Twenty One). Model a game of two players: the dealer and one other player. Each player will be dealt two cards, and can then decide to stick or twist (as many times as desired). So the output for typical run of the program, as shown in Nunit's Text Output tab, might look like this:

***** tdd.blackjack.GameTest.TestWinnerHasHighestScoreWithPlay
Player's hand was: Jack of Hearts, Eight of Diamonds
Dealer's hand was: Two of Hearts, Nine of Hearts, Five of Clubs, Ace of Diamonds
Player scored 18 Dealer scored 17 - Winner was Player

Your program should be capable of modelling the fact that an ace can either be high or low (as in this example output.) If time permits: an additional requirement is that each play of the game is to be recorded in a simple database table, with columns for the dealer's score and player's score. Develop this test-first by mocking the database connection.

Periodically your instructor will call everyone together to discuss how they are doing, what they are doing, what problems they have faced, and overcome. The first discussion should revolve around your test plans: what test classes you envisage writing, in what sequence, and what tests they will contain.