10 Agile Programming

While a detailed discussion of Agile development methods is beyond the scope of this book, this chapter will explain the claims made by proponents of agile programming methods and show how modern IDE's (such as Visual Studio) offers tools to support agile programming. In particular we will examine refactoring and the automatic testing framework within Visual Studio.

Objectives

By the end of this chapter you will be able to

- Appreciate the importance of the claims made for Agile development
- Understand the need for refactoring and how a modern IDE supports this
- Understand the advantages of Unit testing and
- Understand how to create automated test cases.
- Understand the claims made for Test driven Development

This chapter consists of fifteen sections :-

1) Agile Approaches
2) Refactoring
3) Examples of Refactoring
4) Support for Refactoring
5) Unit Testing
6) Automated Unit Testing
7) Regression Testing
8) Unit Testing in Visual Studio
9) Examples of Assertions
10) Several Test Examples
11) Running Tests
12) Test Driven Development (TDD)
13) TDD Cycles
14) Claims for TDD
15) Summary

10.1 Agile Approaches

Traditional development approaches emphasized detailed advance planning and a linear progression through the software lifecycle *Code late, get it right first time* (Really??)

Recent 'agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution *Code early, fix and improve it as you go along.*

This is a very hot topic in Software Engineering circles at the moment, and as with all such developments it has its share of zealots and ideologues!

Is the waterfall lifecycle model really successful in enabling large, complex projects to proceed from start to finish without ever looking back? Advocates of agile approaches contend that these better fit the reality of software development.

However agile programming requires tools that will enable software to change and evolve. Two specific tools provided by modern IDEs that support agile programming are refactoring and testing tools.

10.2 Refactoring

A key element of 'agile' approaches is 'refactoring'. This technique accepts that some early design and implementation decisions will turn out to be poor, or at least less than ideal.

Refactoring means changing a system to improve its design and implementation quality without altering its functionality (in traditional development such work was termed 'preventive maintenance').

Although the idea of structurally improving existing software is not new, the difference is as follows. In traditional development it was seen as a remedial action taken when the software design quality had degraded, usually as a result of phases of functional modification and extension. In agile methodologies refactoring is regarded as a natural healthy part of the development process.

10.3 Examples of Refactoring

During the development process a programmer may realise that a variable within a program has been badly named. However changing this is not a trivial task.

Changing a local variable will only require changes in one particular method – if a variable with the same name exists in a different method this will not require changing.

Alternatively changing a public class variable could require changes throughout the system (one reason why the use of public variables are not encouraged).

Thus implementing a seemingly trivial change requires an understanding of the consequences of that change.

Other more complex changes may also be required. These include..

- Renaming an identifier everywhere it occurs
- Moving a method from one class to another
- Splitting out code from one method into a separate method
- Changing the parameter list of a method
- Rearranging the position of class members in an inheritance hierarchy

10.4 Support for Refactoring

Even the simplest refactoring operation, e.g. renaming a class, method, or variable, requires careful analysis to make sure all the necessary changes are made consistently.

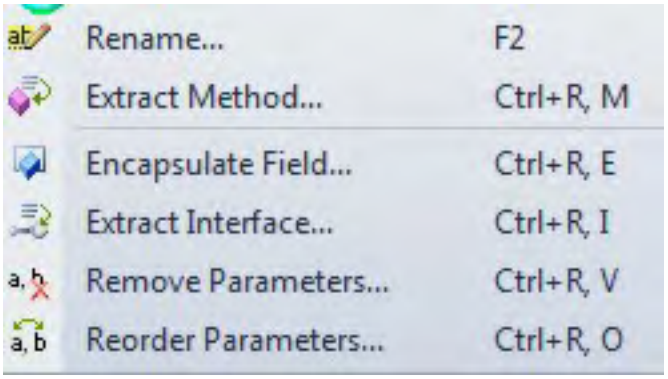
Visual Studio and many other IDEs provide sophisticated automatic support for this activity.

Don't confuse this with a simple text editor find/replace – Visual Studio understands the C# syntax and works intelligently… e.g. if you have local variables with the same name in two different methods and rename one of them, Visual Studio knows that the other is a different variable and does not change it. However if you rename a public instance variable this may require changes in other classes and even in other namespaces as methods from these classes may access and use this variable.

Changing methods to reorder or remove parameters are also refactoring activities. Doing this will require changes to be made wherever these methods are called throughout the system. Visual Studio will automatically change the method calls as appropriate.

Visual Studio provides automated support for the creation of an Interface. To do this it extracts an Interface from a class i.e. given a class it can define an interface based on specified features of that class it will then automatically change that class to define the fact that it implements that new interface. This is a significant restructuring activity that allows us then to create new classes which implement the same interface!

The screen shot below shows the refactoring options provided by the Visual Studio IDE.



Another essential facility provided by modern IDEs is automated testing tools.

10.5 Unit Testing

Testing the individual methods of a class in isolation from their eventual context within the system under construction is known as Unit Testing.

This testing is generally 'wrapped into' the implementation process:

- Write class
- Write tests
- Run tests, debugging as necessary

A unit test should be independent i.e. it should not require other methods or classes within the system to work.

10.6 Automated Unit Testing

To save time we want to automate unit tests but this will not work if the tests require a human to type in test data or if we need a human to check the programs output. Hence to enable automatic testing we need to set up the test data and we need an automated method for checking the program outputs.

Test cases follow a similar anatomy, no matter which testing framework is used, referred to as the arrange-act-assert cycle. Firstly we 'arrange' the test by setting up appropriate test data. Then we 'act' i.e. we run the test. Finally we 'assert' what the expected output should be. The computer can then compare the actual output against the expected output. If there is a difference then the test indicates that the code is faulty.

Tools and frameworks are available to automate the unit testing process. Using such a tool generally requires a little more effort than running tests once manually and significant benefits arise from the ability to re-run the tests as often as desired just by pushing a button.

This is a great aid to the 'regression testing' which must be undertaken whenever previously tested code is modified.

Regression testing was developed long before agile methods were proposed and automated unit testing supports this. However automated unit testing also supports Test Driven Development processes and these play a major role in agile software development methods.

We will explore the Test Driven Development processes within this chapter but before doing so we will first explore conventional regression testing and the support offered for this via the unit testing framework within Visual Studio Professional edition. While this support is not available in the Express edition other tools do support unit testing… though the mechanics will be different from those described here the principles will be the same.

10.7 Regression Testing

All large software systems need to be adapted to meet changing business needs. Many large systems may have been in use for a decade or more. Over the years the software will need to be updated and improved many times to meet the ever changing needs of the client. For this reason regression testing is required. By running regression tests we want to ensure that changes to the code to not 'break' existing functionality. To do this as we write classes we must also write test cases that demonstrate these classes work. Thus we follow a process of …

- Write class
- Write tests
- Run tests, debugging as necessary
- Write more classes
- …

As we decide its necessary to change some of the earlier classes (or classes which they depend on) due to bugs, changing user requirements, or refactoring, we need to re-run all previous tests to check the new code still passes the older tests. Without regression testing any modification of existing code is extremely hazardous!

As we regularly need to re-run sets of test cases it is helpful, and hugely timesaving, to have automated testing facilities such as those that exist within the professional edition of Visual Studio.

10.8 Unit Testing in Visual Studio

Visual Studio includes a widely used unit testing framework. This framework requires us to set up the test cases – however once these have been set up a thousand test cases can be run at the push of a button – and the same set of test cases can be re-run every time the program is amended.

The system will run the tests and highlight which tests pass and which fail.

Note that tests that which have 'failed' actually indicate a failure in the program being tested…. You could argue that in showing this failure the test has in fact been successful.

A summary of the process if explained here ..

1) Firstly set up a project to store all of the tests for a system
2) Within this create a test fixture for each of the classes we are testing
3) Within each test fixture create multiple tests. Several are probably required for each method being tested.
4) Setup the tests i.e. define any initialisation that must be done before the tests are run
5) Teardown the tests i.e. define anything that should be done after the tests have been performed.
6) Run the tests

A full treatment of this framework and how to use it for unit testing can be found at:-
http://msdn.microsoft.com/en-us/library/ms182515%28v=VS.90%29.aspx

There is a naming convention that makes it easy to relate the tests to the code being tested :-

1) The project used to store the tests is named after the system being tested (e.g. for example a project storing tests for a bank system could be called BankSystemTests)
2) The test fixture is named using the name of the class being tested (e.g. assuming within the bank system there is a class called 'Client' a test fixture would be called TestFixture_ClientTests).
3) The tests are named using the method name being tested _ the test being applied _ the expected result (e.g. a method called AddMoney may have a test as follows…AddMoney_TestAdd5Pounds_BalanceEquals10)

In Visual Studio following the menu Test \ New Test \ Basic Test will set up a testing suite capable of testing C# code. Firstly The name of the test fixture will be requested and if a test project has not previously been set up a name for this will then be requested. Individual tests will each need to be named as they are added to the test fixture.

Having created a test fixture we need to set up the tests and write the tests themselves. As part of this we need to specify the correct behaviour of the code being tested. We do this using Assert…() methods which must be true for the test to pass.

We also make use of attributes to provide essential information to Visual Studio so that it can run the tests….

- The [TestClass] attribute indicates that the test class will indeed act as a test fixture. Thus this attribute must be placed at the start of each test fixture.

```
[TestClass]
public class TestFixture_ClientTests
{
.
.
.
```

- The [TestInitialize] attribute declares a method that is run before every test case is executed.
- Test cases are methods that must be preceded by the [TestMethod] attribute.
- The [TestCleanup] attribute declares a method that is run after every test case is executed.
- The [ExpectedException(typeof(…))] attribute allows us to specify that we are expecting the code being tested to generate an exception.

We will see examples of each of these shortly.

10.9 Examples of Assertions

When setting up test cases we make assertions. An assertion is a statement which should be true if the code has functioned correctly. Example of assertion include …

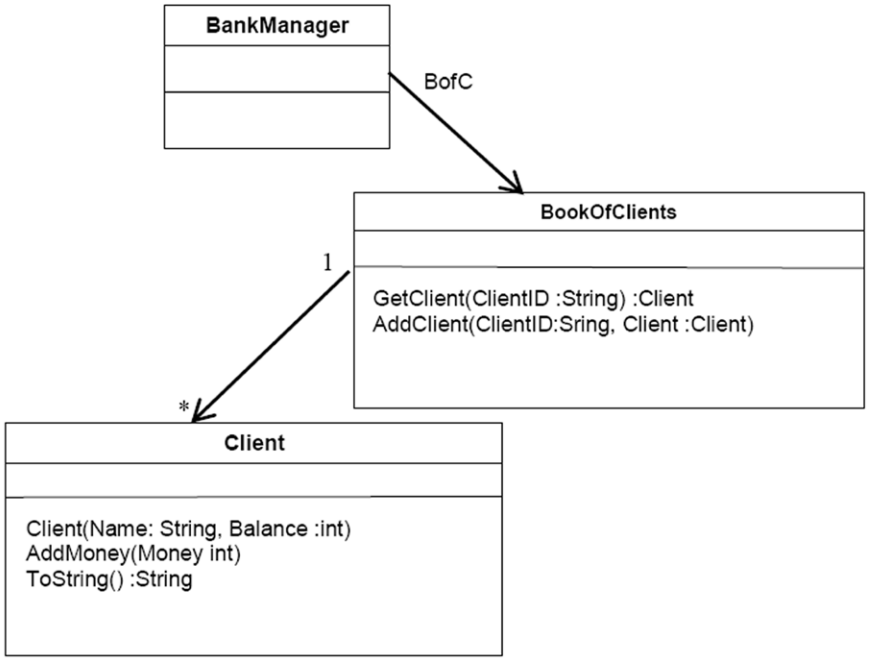
- Assert.AreEqual(…)
- Assert.AreNotEqual(…)
- Assert.Fail()
- Assert.IsTrue()
- Assert.IsFalse(…)

Assert.AreEqual() and Assert.Faill() will be adequate for many testing purposes, as we will see here.

Assert.Faill() indicates that having reached this line means the test has failed! While we say the test has failed this is not strictly true. It is not the test that has failed but our code that has failed. The test successfully found an error in our code.

10.10 Several Test Examples

To illustrate the testing framework we will create several test cases to test the functionality of a BankManagment system as represented below :-



In this system a BankManager class maintains a book of clients (BofC). Client objects can be added by the AddClient() method which requires an ID for that client and the client object to be added. Clients can be retrieved via the GetClient() method which requires a ClientID as a parameter and returns a client object (if one exists) or generates an exception (if a client with the specified ID does not exist).

The Client class has a constructor that requires the name of the client and the clients balance .

For simplicity sake we are assuming each client of the bank only has one account (obviously this is not a realistic system) and we do not need an account number as each client will have a unique ID.

We will specifically test the ability to…

- Add a client to the BookOfClients
- Trying to lookup a non-existent client
- Increase a Clients balance by invoking the AddMoney() method.

This is of course only a small fraction of the test cases which would be needed to thoroughly demonstrate the correct operation of all classes and all methods within the system.

Testing adding a client

To test a client can be added we need to

1) Set up the test by creating a new empty BookOfClients
2) Create a new client and add this to the BookOfClients
3) Check that the client has been added successfully by trying to retrieve the client just added (this of course should work) and finally
4) We need to test that the client retrieved has the same attributes as the client we just added – to ensure it was not corrupted in the process.

Firstly we initialise the test by creating a new empty BookOfClients object (BofC) and we clean up the test by setting this to null at the end. The code for this is given below:-

```
[TestInitialize]
public void TestInitialize()
{
    BofC = new BankManager.BookOfClients();
}

[TestCleanup]
public void TestCleanup()
{
    BofC = null;
}
```

Next we create our first test method. This method will work by trying to add a client to the empty object BoFC. If the system generates an exception because this client already exists then we know there is a fault in our code… hence under these circumstances we assert that the test has failed. See the test for this below…

```
[TestMethod]
public void AddClient_TestNewClient_ExceptionShouldNotBeGenerated()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room 1234", "x200", 10);
     try
     {
          BofC.AddClient(1, c);
     }
     catch(BankManager.ClientAlreadyExistsException)
     {
          Assert.Fail("ClientAlreadyExists exception should not be
          thrown for new clients");
     }
}
```

If the test above passes then we know our code has not generated an exception however this does not prove that the client has been added successfully. We need to create other tests to show that we can retrieve the client just added and we need a test to show that in adding / retrieving the client object the attributes have not been corrupted. Multiple tests are required before we can have confidence that the method being tested will actually work!

Note the name of the test above indicates the name of the method being tested followed by a short description of the test being performed and a description of the expected output. Test names like this, while long, are important. By reading a long list of test names we can work out which parts of our system have been adequately tested and which parts have been missed.

In addition to the tests described above we should also try to add a client with the same ID twice. Our system should of course not allow this to happen. Hence the test succeeds if the code prevents a client being added twice.

Conversely if the system does not generate an exception after adding the same client for the second time then the test should indicate that the code has failed. A test for this is given below…

```
[TestMethod]
public void AddClient_TestAddExistingClient()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room
     1234", "x200", 10);
     try
     {
          BofC.AddClient(1, c);
          BofC.AddClient(1, c);
          Assert.Fail("ClientAlreadyExists exception should be thrown
          if client added twice");
     }
     catch (BankManager.ClientAlreadyExistsException)
     {
     }
}
```

An alternative way of writing the test above is to indicate that the test expects an exception to be generated by using the [ExpectedException(…)] attribute. If this exception is generated and interrupts the test then the test has passed. See an alternative version of the test code below…

```
[TestMethod]
[ExpectedException(typeof(BankManager.ClientAlreadyExistsException))]
public void AddClient_TestAddExistingClient()
{
     BankManager.Client c = new BankManager.Client("Simon", "Room
     1234", "x200", 10);
     BofC.AddClient(1, c);
     BofC.AddClient(1, c);
     Assert.Fail("ClientAlreadyExists exception should be thrown if
     client added twice");
}
```

Testing the GetClient method

We must of course test all of the methods in our system including the GetClient() method.

One test case we need to perform is to test that an exception is thrown if we try to retrieve a client that does not exist. To test this we create an new empty BookofClients and try to retrieve a client from this – any client!

In this instance we would hope our system generates an unknown client exception. Since we have initialised our test by creating an empty client book we should not be able to retrieve any clients unless we first add them.

Activity 1

Look at the test code below and decide at which point in this code we could assert the test has shown our system has failed (point A, B, C or D).

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException()
{    //point A
     try
     {
          // point B
          BofC.GetClient(1);
          // point C
     }
     catch (BankManager.UnknownClientException)
     {
          // point D
     }
}
```

Feedback 1

We cannot determine if the GetClient() method has failed before we have invoked it … hence we cannot place an Assert.Fail() at point A or B.

When we do invoke this method an exception should be generated as our client book is empty. The try block should be terminated at this point and the catch block initiated hence if the code reaches point D the test has succeeded not failed. However if the code reaches point C it indicates that the expected exception was not generated and hence we can assert that the GetClient() method has failed at this point.

The complete code for this test is given below….

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException ()
{
     try
     {
          BofC.GetClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
     }
     catch (BankManager.UnknownClientException)
     {
     }
}
```

As well as testing our BookOfClients class we should of course test the other classes in our system.

Testing the Client Class

As well as testing the BookOfClients class we should test our Client class. The tests for this class should be in a different test fixture and will need to initialise these tests as well….

```
[TestClass]
public class TestFixture_ClientTests
{

     private MessageManagerSystem.Clients.Client c;

     [TestInitialize]
     public void TestInitialize()
     {
          c = new BankManager.Client("Simon", 10);
     }

     [TestCleanup]
     public void TestCleanup()
     {
          c = null;
     }
```

In the initialisation we have created a new client with an opening balance of 10.

Strictly speaking we should have written the Client tests before the BookOfClient tests as we need client objects to test the BookOfClients. In VisualStudio it is possible to ensure these tests are run first.

Below is a simple test to test the AddMoney() method…

```
[TestMethod]
public void AddMoney_TestAdd10ToTheBalance_FinalBalShouldBe20()
{
     c.AddMoney(10);
     Assert.AreEqual(20, c.Money, "Balance after adding 10 is not as
     expected. Expected: 20 Actual: "+c.Money);
}
```

The test above uses the Assert.AreEqual() method. If the first two parameters are equal then our code has passed the test. If they are not equal then our code has failed and the third parameter is the error message to be displayed. To be as helpful as possible the error message specified the balance we expected and the actual balance after the AddMoney() method was invoked.

In all of the tests we have written if the test method ends without failing any assertions then the test is passed.

Testing the ToString() method

One way of implicitly testing that ALL the attributes a client have been stored correctly is to test the ToString() method returns the value expected. This is a little tricky because the format of the string must match exactly including every space, punctuation symbol, and newline.

The alternative however is to test the value of every property.

[Activity 2

Assuming the ToString() method of the Client class is defined as below create a test method to test the value returned by the ToString() method is as expected.

```
public String ToString() {
        return ("Client name: " + Name + "\nBalance: " + Balance);
}
```

Hint: We have already initialised client tests by creating a client with a name "Simon" and a balance of 10.

Feedback 2

One solution to this exercise is given below.

```
[TestMethod]
public void ToString_TestClientValues()
{
     Assert.AreEqual ("Client name: Simon\nBalance: 10", c.ToString(),
     "String returned is not as expected. Expected: Client name:
     Simon\nBalance: 10 Actual: " + c.ToString());
}
```

Here we assert that the string returned from c.ToString() is equal to the string we are expecting. This is quite tricky because the format of the string must match **exactly** including every space, punctuation symbol, and newline.

This test is of course implicitly testing that ALL the attributes have been stored correctly which would be particularly useful if we used it to check a client has been retrieved from the book of clients correctly.

10.11 Running Tests

Having designed a batch of test cases these can be run as often as required at the push of a button.

To do this in Visual Studio we run all of the tests via the Test menu.

10.12 Test Driven Development (TDD)

While very useful for regression testing automated unit testing also supports Test Driven Development (TDD) which is a technique mainly associated with 'agile' development processes. This has become a hot topic in software engineering

The Test Driven Development approach is to

1) Write the tests (before writing the methods being tested).
2) Set up automated unit testing, which fails because the classes haven't yet been written!
3) Write the classes and methods so the tests pass

This reversal seems strange at first, but many eminent contributors to software engineering debates believe it is a powerful 'paradigm shift'

The task of teaching can be used as an analogy. Which of the following is more focused and leads to better teaching?

- Teach someone everything they should know about a subject and then decide how to test their knowledge or
- Decide specifically what a student should be capable of doing after you have taught them, then decide how to test the student to ensure they are capable of performing this task and finally decide just what you must teach so that they can perform this task and hence pass the test.

It can be argued that the second approach leads to more focussed teaching. In the same way it is argued by some that test driven development leads to simpler code that focuses just on achieving the functionality required.

10.13 TDD Cycles

When undertaking test driven development the test will initially cause a compilation error as the method being tested doesn't exist!

Creating a stub method enables the test to compile but the test will 'fail' because the actual functionality being tested has not been implemented in the method.

We then implement the correct functionality of the method so that the test succeeds.

For a complex method we might have several cycles of: write test, fail, implement functionality, pass, extend test, fail, extend functionality, pass… to build up the solution.

10.14 Claims for TDD

Among the advantages claimed for TDD are:

- testing becomes an intrinsic part of development rather than an often hurried afterthought.
- it encourages programmers to write simple code directly addressing the requirements
- a comprehensive suite of unit tests is compiled in parallel with the code development
- a rapid cycle of "write test, write code, run test", each for a small developmental increment, increases programmer confidence and productivity.

In conventional software lifecycles if a software project is running late financial pressures often result in the software being rushed to market not having been fully tested and debugged. With Test Driven Development this is not possible as the tests are written before the system has been implemented.

10.15 Summary

'Agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution.

Refactoring tools help agile development methods.

Unit testing is an important part of software engineering practice whatever style of development process is adopted.

An automated unit testing framework allows unit tests to be regularly repeated as system development progresses.

Test Driven Development reverses the normal sequence of code and test creation, and plays a major part in 'agile' approaches.