

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 Eclipse) offers tools to support agile programming. In particular we will examine refactoring and JUnit testing tools.

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 JUnit test cases.
- Understand the claims made for Test driven Development



www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA 



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) JUnit
- 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.

Eclipse provides sophisticated automatic support for this activity.

Don't confuse this with a simple text editor find/replace – Eclipse understands the Java syntax and works intelligently...e.g. if you have local variables with the same name in two different methods and rename one of them, Eclipse 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 packages as methods from these classes may access and use this variable.

Rearranging existing classes within a package structure is also a refactoring activity

Eclipse makes this very easy to do. As a system design evolves, just as we may sometimes decide a class has become too large and complex and decide to split it into two or more separate classes, so we might decide to split a package. Less often we might merge packages or move classes between existing packages.

Eclipse will allow us to drag a class from one package to another. When this happens the package statement at the start of this class is changed and import statements within other classes are automatically adjusted to reflect the fact that the class concerned now resides in a new package.

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



CHALLENGING PERSPECTIVES

Internship opportunities

EADS unites a leading aircraft manufacturer, the world's largest helicopter supplier, a global leader in space programmes and a worldwide leader in global security solutions and systems to form Europe's largest defence and aerospace group. More than 140,000 people work at Airbus, Astrium, Cassidian and Eurocopter, in 90 locations globally, to deliver some of the industry's most exciting projects.

An **EADS internship** offers the chance to use your theoretical knowledge and apply it first-hand to real situations and assignments during your studies. Given a high level of responsibility, plenty of learning and development opportunities, and all the support you need, you will tackle interesting challenges on state-of-the-art products.

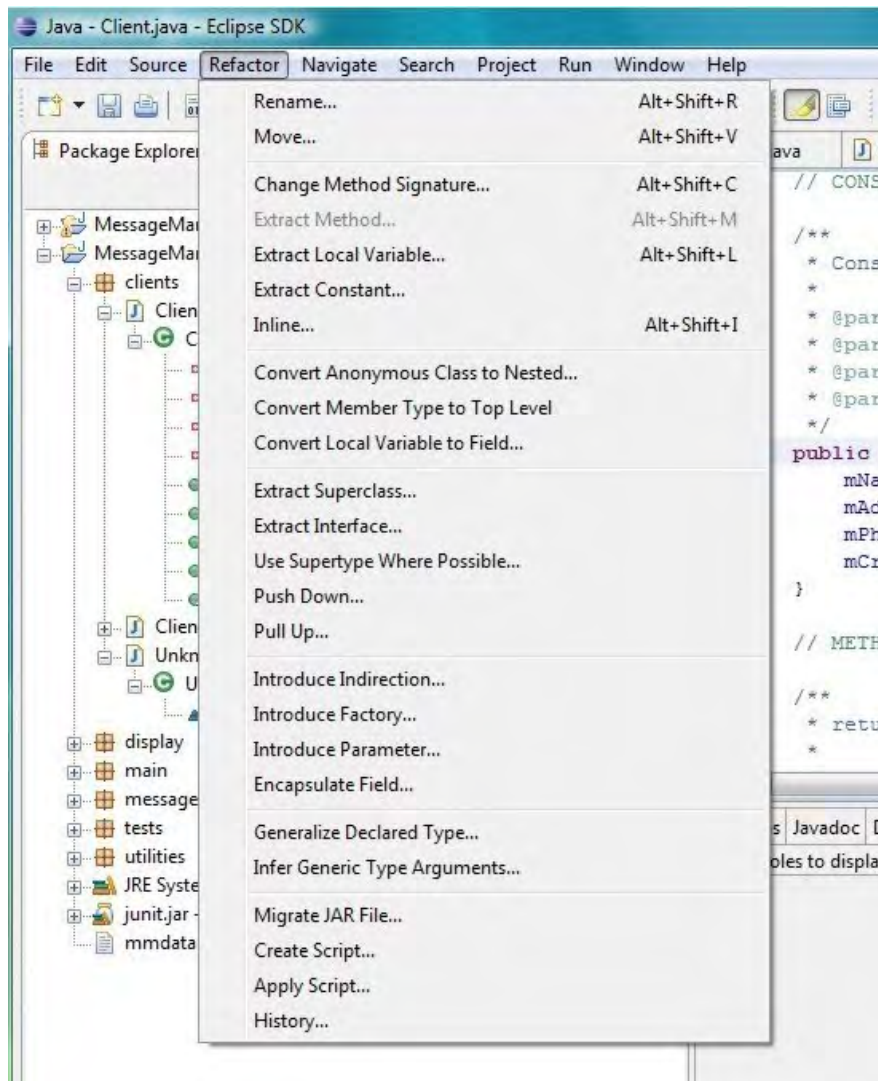
We welcome more than 5,000 interns every year across disciplines ranging from engineering, IT, procurement and finance, to strategy, customer support, marketing and sales. Positions are available in France, Germany, Spain and the UK.

To find out more and apply, visit www.jobs.eads.com. You can also find out more on our **EADS Careers Facebook page**.

AIRBUS **ASTRIUM** **CASSIDIAN** **EUROCOPTER**

EADS





Another essential tool provided by modern IDEs are 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

10.6 Automated Unit Testing

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. However the main benefit arises 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 JUnit a testing framework within Java.

10.7 Regression Testing

Regression testing is required as a software is adapted to meet changing business needs. By running regression tests we want to ensure that changes to the code do 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 they still pass. 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 Java.

10.8 JUnit

JUnit is a very widely used unit testing framework for Java. It has been developed as a suite of open-source classes, and is integrated into popular Java IDEs such as Eclipse and NetBeans.

While JUnit provides an automated testing framework we still need 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 a program is amended.

JUnit test classes are constructed as subclasses of `junit.framework.TestCase`

The correct behaviour of code being tested is checked using `assert...()` methods which must be true for the test to pass

Currently version 3.8 and version 4 of JUnit are widely used but there are significant differences between them. Here we are using the more widely established JUnit 3.8

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...

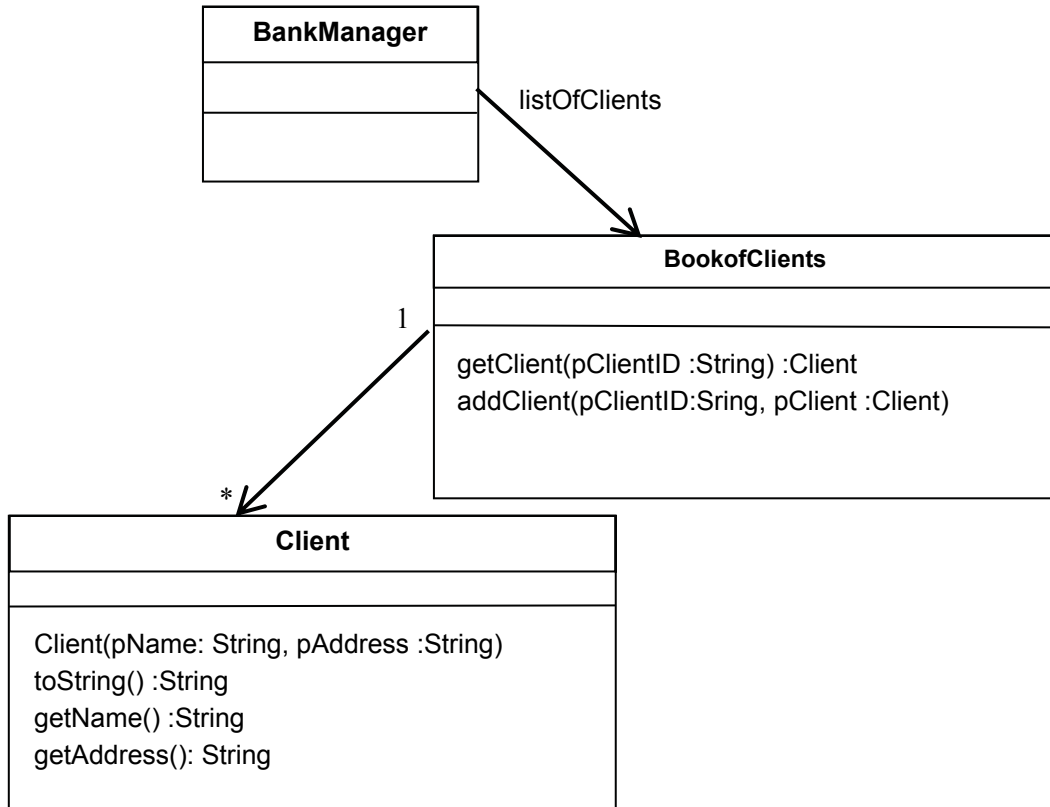
- `assertTrue(...)`
- `assertFalse(...)`
- `assertEquals(...)`
- `assertNotEquals(...)`
- `assertNull(...)`
- `assertNotNull(...)`
- `fail()`

`assertEquals()` and `fail()` will be adequate for many testing purposes, as we will see here.

`fail()` indicates that having reached this line means the test has failed! (We will see an example).

10.10 Several Test Examples

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



360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.



In this system a BankManager class maintains a 'map' of clients. 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 address and provides associated accessor methods.

We will specifically test the ability to...

- Add a client to the BookofClients
- Trying to lookup a non-existent client
- The string representation yielded by the toString() method of the Client class.

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 create a JUnit test case that

- 1) Creates a new empty BookofClients
- 2) Creates a new client and adds this to the BookofClients
- 3) Try's to retrieve a client with the same ID as 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.

The code for this is given below...

```
public void testAddClient() {
    BookofClients bofc = new BookofClients();
    Client c = new Client("Simon",
                        "No 5, Main St., Sunderland, SR1 0DD");
    Client c2 = null;
    bofc.addClient("SK001", c);
    try {
        c2 = bofc.getClient("SK001");
    } catch (UnknownClientException uce) {
        fail();
    }
    assertEquals("Simon", c2.getName());
    assertEquals("No 5, Main St., Sunderland, SR1 0DD",
                 c2.getAddress());
}
```

Note that a test method name always begins test...()

If an `UnknownClientException` is thrown the test fails because we should have found the “SK001” client which was added

If we successfully retrieve the client we assert that the value from `getName()` should equal “Simon” and the value from `getAddress()` should equal “No 5, Main St., Sunderland, SR1 0DD”. If either of these is not the case then the test will fail.

Note that the two parameters of the `assertEquals()` method are first the **expected** value and second the **actual** value. Although the outcome of the assertion will be the same if these are reversed, it will result in JUnit giving misleading reports if the test fails.

If the test method ends without failing any assertions then the test is passed.

Note that for this test to work we need accessor methods `getName()` and `getAddress()` in the `Client` class. It is common practice in designing classes to create accessor methods which were intended purely to support unit testing even when these are not required in the actual system.

Testing for Unknown client

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!

The code for this is given below...

```
public void testGetUnknownClient() {
    BookofClients bofc = new BookofClients();
    try {
        bofc.getClient("SK001");
        fail();
    } catch (UnknownClientException uce) {
    }
}
```

In this case we expect an `UnknownClientException` to be thrown (as we haven’t added the client!) and therefore if the line after the call to `getClient()` is reached the test fails.

If the exception is caught the `fail()` statement is skipped, the catch block has no action to take and the test then completes successfully.

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 returned by every accessor method.

Activity 1

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: " + mName + "\nAddress: " + mAddress);  
}
```

Hint: Firstly create a new Client object with specified attributes then test the attributes of this object are as expected by using an assertion to test the string returned by the toString() method.

Feedback 1

One solution to this exercise is given below however clearly the attributes of the client created are arbitrary.

```
public void testClientToString() {  
    Client c = new Client("Simon", "5 Main St., Sunderland");  
    assertEquals("Client name: Simon\nAddress: 5 Main St.,  
                Sunderland", 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 is useful.

10.11 Running Tests

Having designed a batch of test cases we need to set these up so that they can be run as often as required at the push of a button.

- To do this in Eclipse we
- create new package “tests”
- in “tests” create new JUnit test case “ClientTests”
- add in the three test methods
- Run As... JUnit test
- edit code to cause one of the tests to fail and observe result

10.12 Test Driven Development (TDD)

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 simpler?


- Teach someone everything you know about a subject and then decide how to test their knowledge or
- Decide specifically what it is they need to learn (i.e. decide what to test) and then teach the person just what they need to know in order to pass that test.

SIMPLY CLEVER

ŠKODA



We will turn your CV into an opportunity of a lifetime



Do you like cars? Would you like to be a part of a successful brand? We will appreciate and reward both your enthusiasm and talent. Send us your CV. You will be surprised where it can take you.

Send us your CV on www.employerforlife.com



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 it the test itself 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.

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

An automated unit testing framework (e.g. JUnit) 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.