COMP2501
Assignment 2

JUnit

Group code: W2E

Group members:

Charles Brooking	40116336
Andrew Crouch	40117267
Craig Daniels	40090315
Cameron Smith	40114378

Date due: 19/9/2002

Table of contents:

1 Introduction....... 1
2 Design and source code description....... 2
2.1 General design....... 2
2.2 The ‘framework’ package....... 2
2.2.1 Test cases....... 2
2.2.2 Errors and failures....... 4
2.2.3 Recording results....... 4
2.2.4 Test suites....... 5
2.2.5 Recording results (returned)....... 6
2.3 The ‘runner’ package....... 7
2.3.1 Structure....... 7
2.3.2 BaseTestRunner....... 7
2.3.3 Collectors....... 8
2.3.4 TestSuiteLoader....... 9
2.3.5 Miscellaneous....... 10
2.4 The ‘textui’ package....... 11
3 Source code management.......**Error! Bookmark not defined.**
3.1 Version control with CVS.......**Error! Bookmark not defined.**
3.2 Build evidence.......**Error! Bookmark not defined.**
4 Future research.......**Error! Bookmark not defined.**
4.1 Design documentation.......**Error! Bookmark not defined.**
4.2 Proposed extensions.......**Error! Bookmark not defined.**
4.3 Test plan.......**Error! Bookmark not defined.**
5 Learning experience.......**Error! Bookmark not defined.**
6 References....... 16

1 Introduction

This report presents a case study in software design investigation based on JUnit, a prominent open-source regression-testing framework for Java. JUnit was originally written by Erich Gamma and Kent Beck, two very famous exponents of software patterns [1], and is a strong example of how pattern-driven design can be applied to real-life systems. It also offers the opportunity to reverse engineer a large body of code and develop standard UML diagrams to describe it.

The first section of this document contains a description of the overall design of JUnit with reference to important aspects of the source code. Following this description, details are provided of the practical task of source code management using a version control system and building tools. The possibility of future research is considered in some detail with plans for creating more documentation on the current design, a plan for extending the source and the associated tests required for those extensions. The team journal and a reflection on the learning process involved with this task is also presented.

2 Design and source code description

There are a number of interesting features of the source for JUnit both at the level of its overall design and its specific programs. This section of the report seeks to highlight interesting factors in these areas as means for providing an understanding of the internal structure and behaviour of JUnit. From its initial stages, JUnit was developed using a pattern-driven design approach and thus is best understood in terms of the software patterns involved [2]. References to patterns are made frequently in this report and an understanding of how they are implemented affords considerable insight into how JUnit was composed.

2.1 General design

The programs for JUnit are divided into a set of Java packages that partition the overall functionality. The logical structure of these packages lends itself well to an outline of the design, and so the following sections will each deal with a particular package and outline the role of its programs.

JUnit packages:

Package	Description
framework	Provides the basic framework for writing reusable test classes and collecting results.
runner	Provides a basis for executing test classes and responding to results.
textui	Provides a concrete test runner with a textual user interface.
awtui	Provides a concrete test runner with a windows-style user interface.
swingui	Provides a concrete test runner with a Java swing user interface.
extensions	Allows for classes to be added that extend the functionality of existing objects. Based on the Decorator pattern [3] to allow extensions to test classes.
tests	A collection of test classes, implemented in JUnit, that test JUnit itself!
samples	Some sample tests for educational purposes.

Important

This report describes only the ‘framework’, ‘runner’ and ‘textui’ packages. The motivation for this is that ‘framework’ and ‘runner’ reflect the most important and fundamental aspects of JUnit’s design and hence, given the limited amount of time granted for preparing this document, are presented without the details of the other packages that build on them. The ‘textui’ package is included, however, to illustrate how the features of JUnit’s structure carry through to interaction with the user.

2.2 The ‘framework’ package

The framework package contains classes that define the most basic structures for writing test cases and recording test results. This package relies heavily on software patterns – a fact described well by documents provided by the authors of JUnit [2]. These will be outlined.

2.2.1 Test cases

The most basic element of a testing framework is a test case. JUnit implements this as a class by applying the Command pattern, which suggests that a request be encapsulated as an object [3]. In this case the request consists of the steps necessary to perform a test, which are: establish the conditions or the *‘fixture’* needed for test execution; run the test; clean up the fixture. The structure of this algorithm has been represented by applying the Template pattern [3] to create an abstract class with key methods that get overridden to specify each step in the algorithm. This class is called TestCase, and is shown in the UML diagram below:

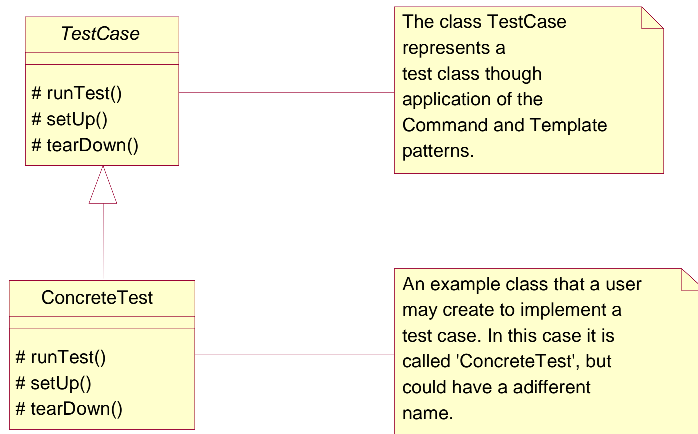

Figure 2.2.1 – The basic design of the TestCase class.

Within each test class there may be a number of methods defined by the user that apply to specific aspects of the unit under test. For instance, in a concrete class MathTest there may be methods such as testAdd() and testMultiply() which are implemented as private methods within the test class. In the Junit framework, a particular instance of a TestCase only considers one of these internal test methods. For this reason, each TestCase has an attribute name, which corresponds to the name of the test method to be executed. This is called the Pluggable Selector pattern[2], and relies on the Java reflection API to select the appropriate method at run-time.

This may seem confusing at first. Remember:

- A test class extends TestCase and may contain a number of test methods. Example: MathTest.
- An actual test case is characterised by both its test class and its name. Example: new MathTest("testAdd").

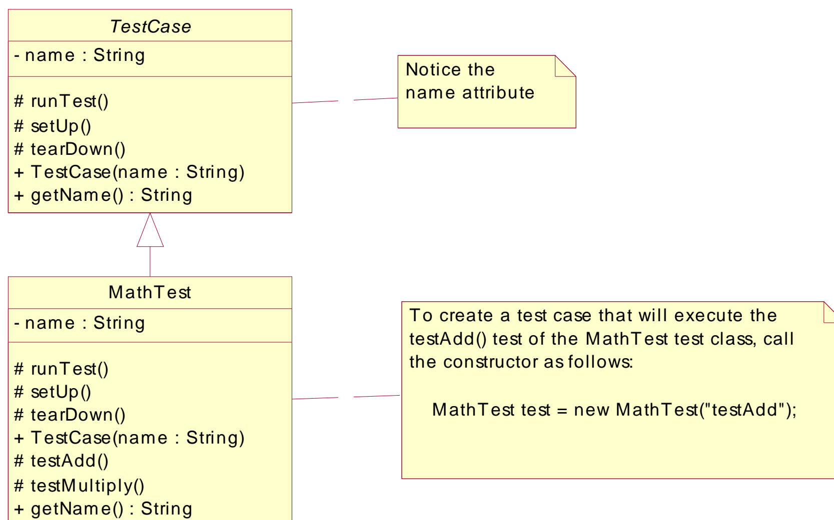

Figure 2.2.2 – Extending TestCase to create a test case MathTest for a particular unit.

Each of these test methods uses a special class within the framework package named Assert. This class simply provides a set of static methods that perform tasks such as comparing actual results with those expected. The class TestCase extends Assert, as shown in the diagram below:

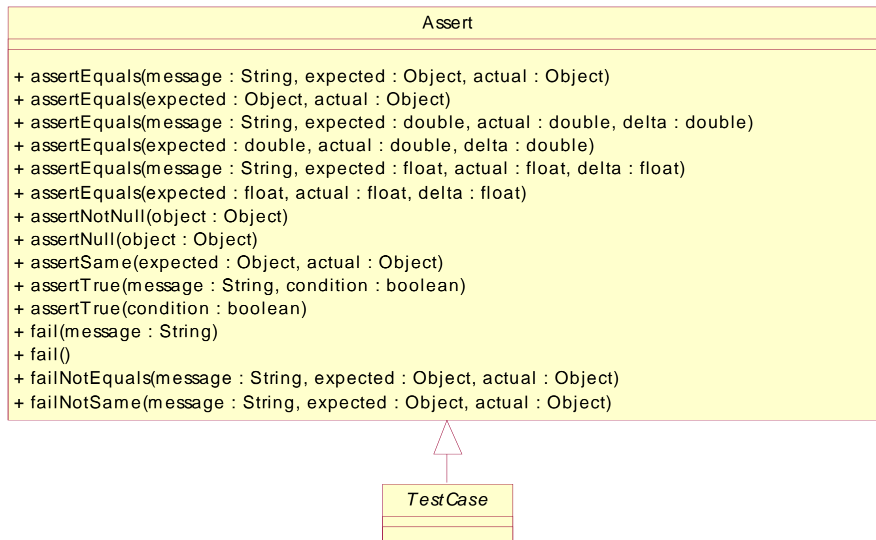

Figure 2.2.3 – The Assert class.

The methods signatures shown above are reasonably self-explanatory. The assert methods take arguments for results that are tested for validity and the fail methods are used to manually induce a test failure.

2.2.2 Errors and failures

In the case where a false assertion is made, it is required that an error signal be generated. Each of the methods in Assert are capable of throwing an AssertionFailedError, which is a JUnit class that extends the Error class in the java.lang package. This is in fact termed as a 'failure' in testing vocabulary due to the fact that a test has failed. This concept of a failure is distinct from an 'error', which is an unexpected exception within a program and thus a subclasses of Throwable.

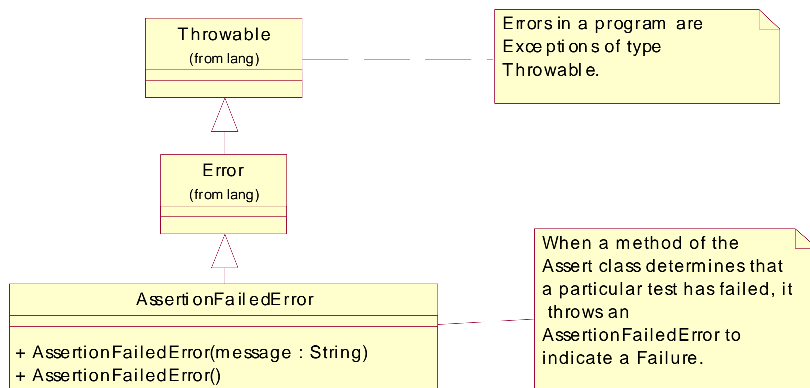

Figure 2.2.4 – Classes involved with errors and failures.

2.2.3 Recording results

Recording the results of test cases is another basic element of a testing framework. JUnit has a class TestResult that is used for this purpose according to the Collecting Parameter pattern. This pattern, according to JUnit design documentation, suggests that "when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you" [2]. Thus, in running a TestCase, we do so by passing a TestResult object that will record the result of that test.

Figure 2.2.5 – A TestResult recording a TestCase.

With only this very general concept of the TestResult, the additional constructs for test cases will now be discussed. Following that description, TestResult will be explained in more detail.

2.2.4 Test suites

JUnit also offers the ability to create a ‘test suite’ which contains a collection of test cases. This is implemented in a very particular way, so as to have the same interface as an individual TestCase. This is an example of the Composite pattern and will form a tree structure containing both test suites and test cases in which the leaf nodes are TestCase objects [2]. Test suites are modelled using the class TestSuite in the framework package.

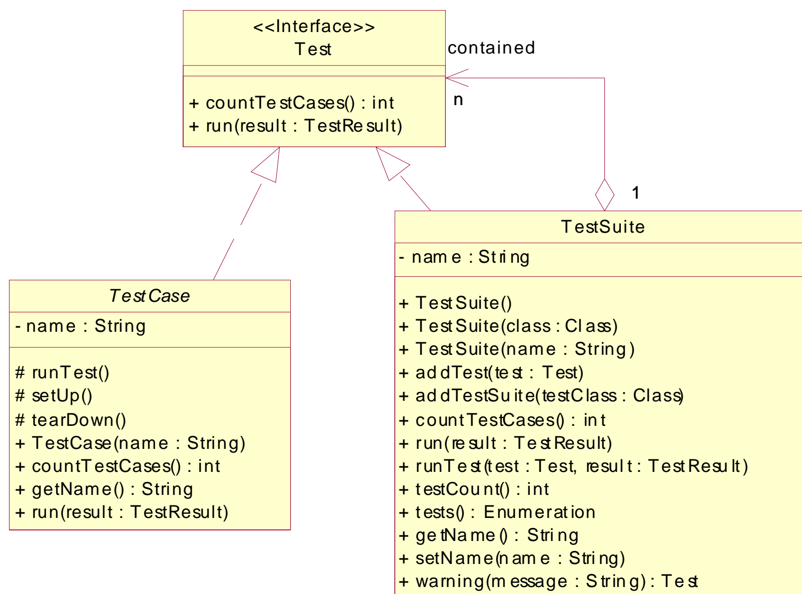

Figure 2.2.6 – The Composite pattern tree of tests.

The diagram above introduces Test, which is defined as the common interface that allows both TestCase and TestSuite to be dealt with in the same way. It contains two methods: countTestCases(), which returns the number of Tests contained; and run(result : TestResult) which runs the test with the supplied TestResult instance. The number of tests contained in a TestCase is always one, so the implementation of countTestCases() there is trivial, however in TestSuite the count is more significant.

The diagram also shows methods contained in the TestSuite class, which provide the following functionality:

Method	Description
TestSuite()	Creates an empty TestSuite.
TestSuite(name : String)	Creates a TestSuite with the given name.
TestSuite(class : Class)	Creates a TestSuite and adds all tests from a given TestCase class.
addTest(test : Test)	Adds the supplied test. Note this can be either a TestCase or TestSuite!
addTestSuite(testClass: Class)	Adds a new TestSuite object corresponding to the supplied class.
countTestCases()	Return number of Test objects contained in the TestSuite.
warning(message: String)	Returns a Test that will fail and gives the supplied message.

2.2.5 Recording results (returned)

We now return to the issue of collecting results. When a given Test is run, failures or errors may occur and these must be recorded in order to provide feedback to the tester. For this purpose, a small class named TestFailure exists to record the pairing between the Test and the failure/error within that test. This is shown on the diagram below. Two collections of TestFailure instances are stored in a single TestResult object – one for *errors* and a second for *failures*.

In addition to these is another collection that stores associations with TestListener objects, which arise from an application of the Observer pattern [2]. The TestListener is important at the level of running tests, which is performed by classes in the runner package. A TestListener registers its interest with the TestResult, which then notifies it whenever a test is started or ended and when failures or errors occur in any test. This is illustrated in the diagram below.

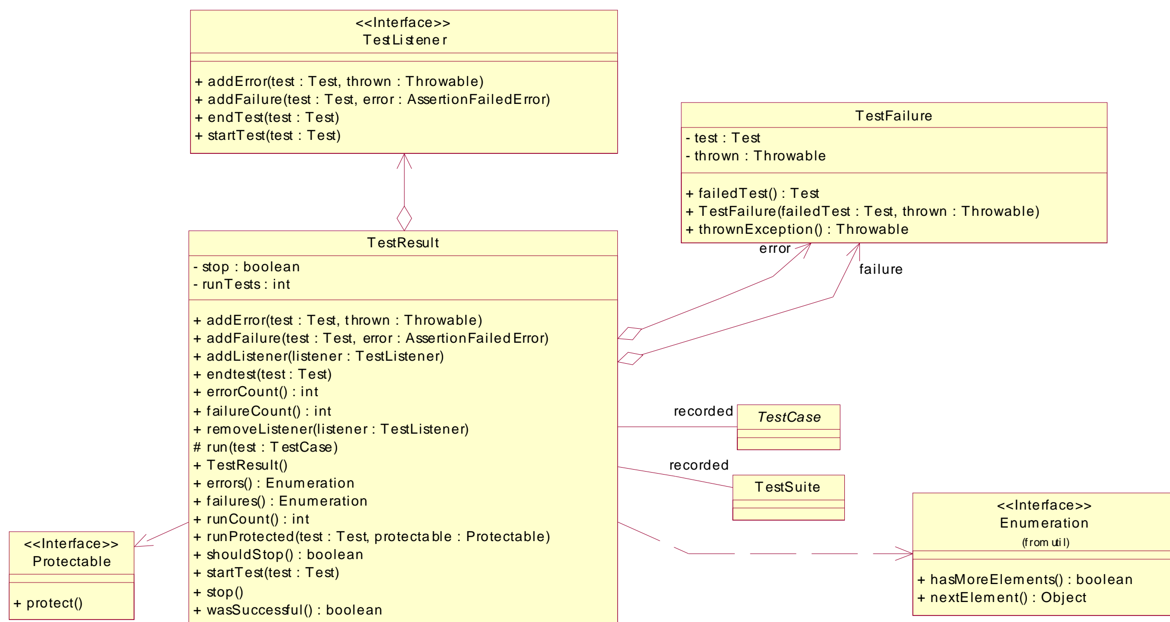

Figure 2.2.7 – The TestResult class shown with TestListener.

This diagram also shows a number of additional details about the TestResult class.

Errors and failures

The methods addError, addFailure, startTest and endTest perform precisely what their names suggest, and also have the effect of notifying the TestListener. The methods errorCount and failureCount return the total number of errors and failures that have been added. Methods errors and failures return an Enumeration of their respective collections.

Adding and removing listeners

TestListener objects can be attached and removed from the TestResult without affecting it. This is performed using the addListener and removeListener methods.

Running a test

When a test is run, it is performed such that any errors or failures will be caught and recorded by the TestResult object. The TestResult class also includes a boolean variable stop, and several methods such as stop() and shouldStop() which are used in controlling the flow of test execution when run.

The final result

The method wasSuccessful() can be called upon to see if all tests were a success. It simply checks that the errorCount and the failureCount are both zero.

2.3 The 'runner' package

The design of the runner package is both elegant and effective. It provides an abstract runner class to tie together a number of supporting mechanisms that transform a textual description of a test class into any number of tests to be run.

While the runner is closely integrated with the framework, it does delegate responsibility in a logical way. Where helper classes are used, the runner is designed to allow the programmer to use as little or as much as has been provided. Generic interface classes are numerous, and are well used throughout the code, with little or no reliance on concrete classes.

An interesting point to note is that the implementing classes of the runner do not actually run any tests. They delegate that responsibility to the framework, which enumerate, run and document the test. The framework also informs any output classes the status of the tests, as defined by the runner.

2.3.1 Structure

The runner package comprises 13 classes that can be broadly grouped in the following way:

Breakdown of the runner package

SubSection	Description
BaseTestRunner	The base class for all test runners
Collectors	Classes that store a list of valid test class names
TestSuiteLoaders	Classes that generate class objects from test class names
Miscellaneous	A class that returns the JUnit version number An interface that describes extra operations that can be performed on failed tests Custom string sorter.

2.3.2 BaseTestRunner

All Test Runners, graphical or text-based, inherit the BaseTestRunner. Due to the fact that most of the test related operations are based in the framework, the BaseTestRunner leaves most, if not all testing, result handling and output to the implementing classes. It does however focus a lot of attention around class loading, test validation, runner control and extensions.

Control and extension information is contained in a properties file that is parsed on invocation. This largely controls whether a test is valid for inclusion and loading. It also allows modification to the behaviour of the runner, by forcing the BaseTestRunner to use various standard and user defined helper classes.

To a large extent the runner package in general relies on interface realisation. Concrete classes that conform to defined interfaces are seamlessly used throughout with little or no code modification. For example, referring to Figure 2.3.1 the BaseTestRunner creates a Class object via a call to a TestSuiteLoader. As we shall see shortly, two concrete classes with slightly difference semantics implement the TestSuiteLoader. The way that the BaseTestRunner is designed is that it will use any valid TestSuiteLoader, giving the programmer complete freedom over implementation. A simple update to the properties file will mean that it is available to the programmer or user at both design and runtime without any need for the underlying code to be modified.

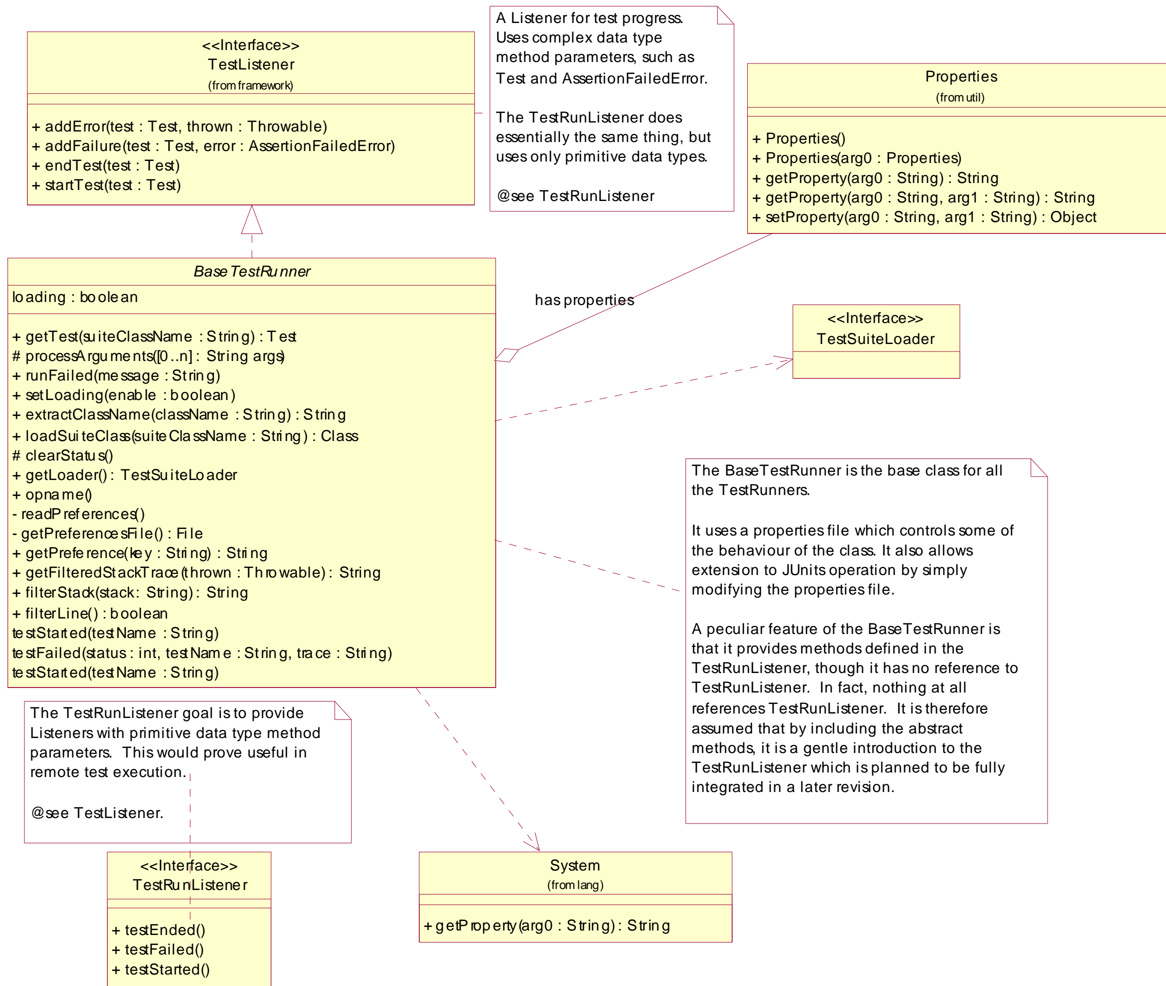

Figure 2.3.1 – BaseTestRunner and supporting classes.

As the diagram above also shows, the interface that defines the ability to be notified by the TestResult (framework) is TestListener shown at the top of the hierarchy.

It is interesting that a TestRunner realises the TestListener interface as it doesn't generally handle any test ouput. It is usually the case that an output object is created by TestRunner which is added to the list of objects that TestResult informs about test behaviour.

It is also interesting to note that the BaseTestRunner provides methods defined in the TestRunListener, though it has no reference to TestRunListener. It is therefore assumed that by including these abstract methods, it is a gentle introduction to the TestRunListener which is planned to be fully integrated in a later revisions of JUnit.

2.3.3 Collectors

The TestCollector is at present, seldom used though it shows promise of continued future use. It shows a good use of inheritance to minimise potential code duplication. It is currently used only in the swingui package where its primary function is to gather a collection of valid test case class names.

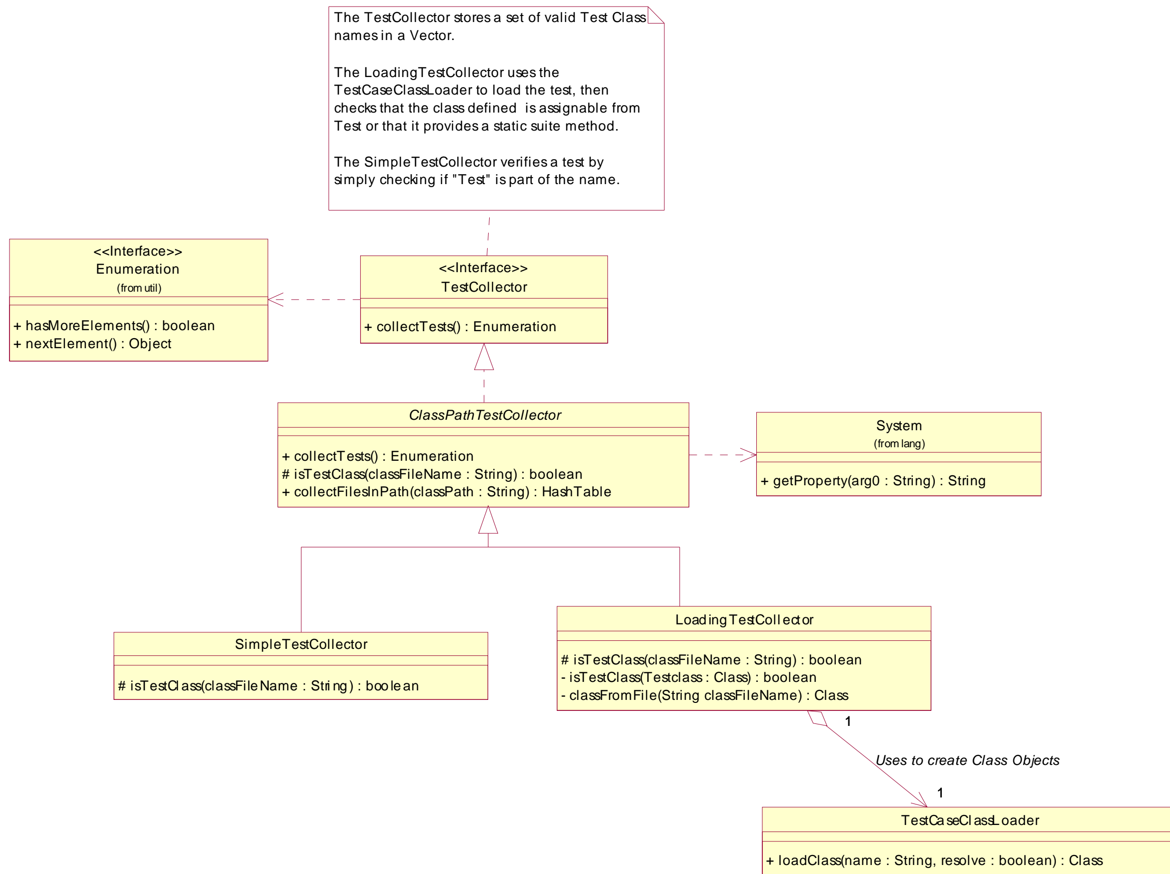

Figure 2.3.2 TestCollector Realisation and Extension.

The class names are extracted in the same way, making good sense to put an abstract class implementing this functionality up the hierarchy. The only observable difference between the implementing classes is in the way they check a class for inclusions using the isTestClass() boolean method.

The class names are determined by first finding all the classes in the class path. An exclusion file is consulted as to whether or not the class should be excluded. Such exclusions could be something like “sun.java.*”, where anything in the *sun.java* hierarchy is ignored. Following this, the class is checked to see if it is a test class.

This is where the SimpleTestCollector and LoadingTestCollector differ. The SimpleTestCollector deems a class a valid test class if its name contains the string “Test” such as in the previous example class MathTest. The LoadingTestCollector gets a Class object from the TestCaseClassLoader and checks, using Java reflection, that either a test or suite method is defined. Each test that passes is added to the collection.

The swingui uses this information to build a list of tests that can be run. From there, the test can be loaded via a “point and click” system and run as usual.

2.3.4 TestSuiteLoader

The SuiteLoaders are responsible for returning Class objects based on a given string. They are, for the most part, delegating the request to the TestCaseClassLoader, or the system ClassLoader to achieve their functionality.

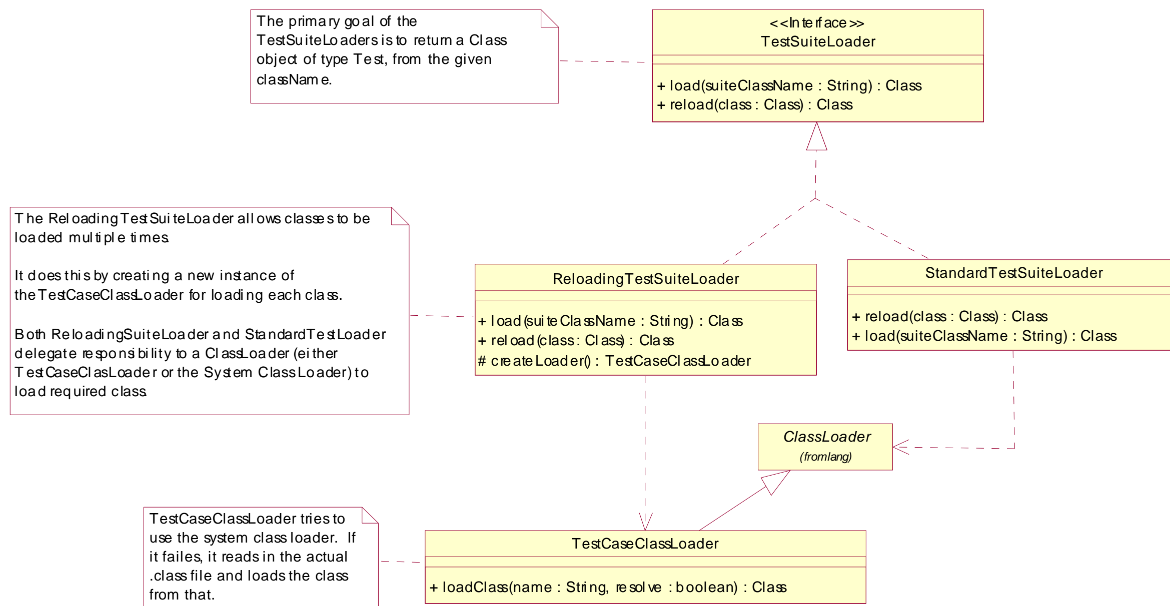

Figure 2.3.3 TestSuiteLoaders delegate responsibility.

A point of interest is the ReloadingTestSuiteLoader. On each request it creates a new TestCaseClassLoader that reloads the test class, whereas the more standard ClassLoader just returns a reference to the previously loaded Class object rather than loading a new copy. This is an important design decision, because it prevents the usage of stale copies of test objects used by TestRunners that run a test multiple times.

In the textui, invoking a new textui runner will run the specified tests then exit. To run the test over the same class, a new runner is started. Where the above design consideration comes into play is when the TestRunner is not closed once the test is finished. By reloading the classes, it allows a "fresh copy" of the test classes to be used, instead of the cached copies that would be returned by the system ClassLoader. For example, it allows you to update your code without having to open and close graphical test windows to test each modification or update.

2.3.5 Miscellaneous

These classes are grouped together as they provide no direct functionality to any other runner package classes, but serve concrete TestRunner classes.

The Version class, simply returns the version number. This is used in all the runners in an 'About' message or dialogue.

The Sorter is used and extended only in the swingui package to order the test case class names returned by the TestCollector.

The FailureDetailView is to provide further information about a Test Failure. The interface can be realised to provide extra functionality when a Failure is discovered. The realisation is only applicable in the swingui.

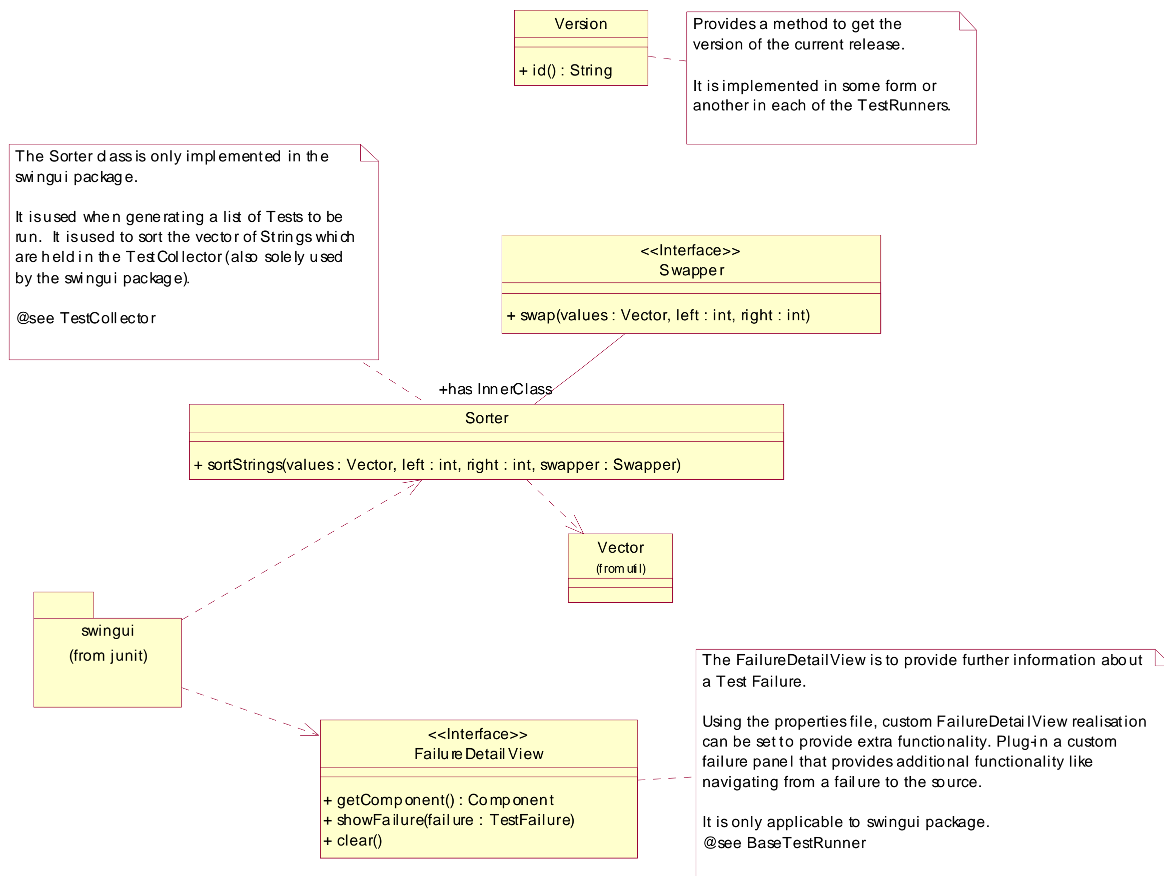

Figure 2.3.4 Miscellaneous Classes

2.4 The ‘textui’ package

Given a fairly complete, yet disjoint description of both the framework and the runner, it is the goal of this section to show a simple implementation of a TestRunner that demonstrates the most fundamental operations that the framework and runner provide.

Looking at Figure 2.4.1, it can be seen that the TestRunner is responsible for

- Creating an OutputStream (ResultPrinter [TestListener])
- Creating a TestSuiteLoader for obtaining a Class object from the string on the command line
- Creating a TestSuite
  - when the TestSuite is created, the TestCases and/or TestSuites contained within are enumerated and stored for later testing.
- Creating a TestResult to store a summary of all the tests and to inform any TestListeners about the test status.

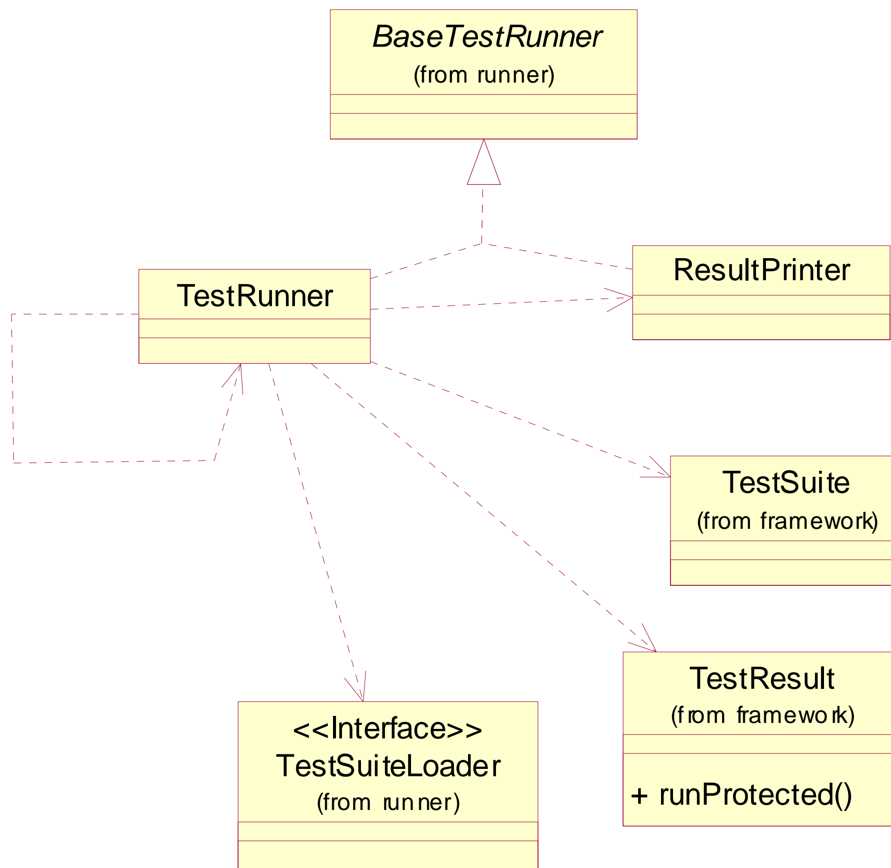

Figure 2.4.1 Class Diagram of the TestRunner

Referring to the sequence diagram, Figure 2.4.2, a very brief description on how the different JUnit components interact are described. A complete sequence diagram shows nearly 40 method calls to a variety of depths to run but a single test and output to a single TestListener. To make the interactions clearer, a textual description is also provided below.

The textui begins by creating a new instance of itself called from the main() method.

```
TestRunner aTestRunner= new TestRunner()
```

It does allow for various output classes via a collection of public constructors which eventually instantiate a ResultPrinter using a given PrintStream. (Step 1, Figure 2.4.2)

```
public TestRunner()
public TestRunner(PrintStream writer)
public TestRunner(ResultPrinter printer)
```

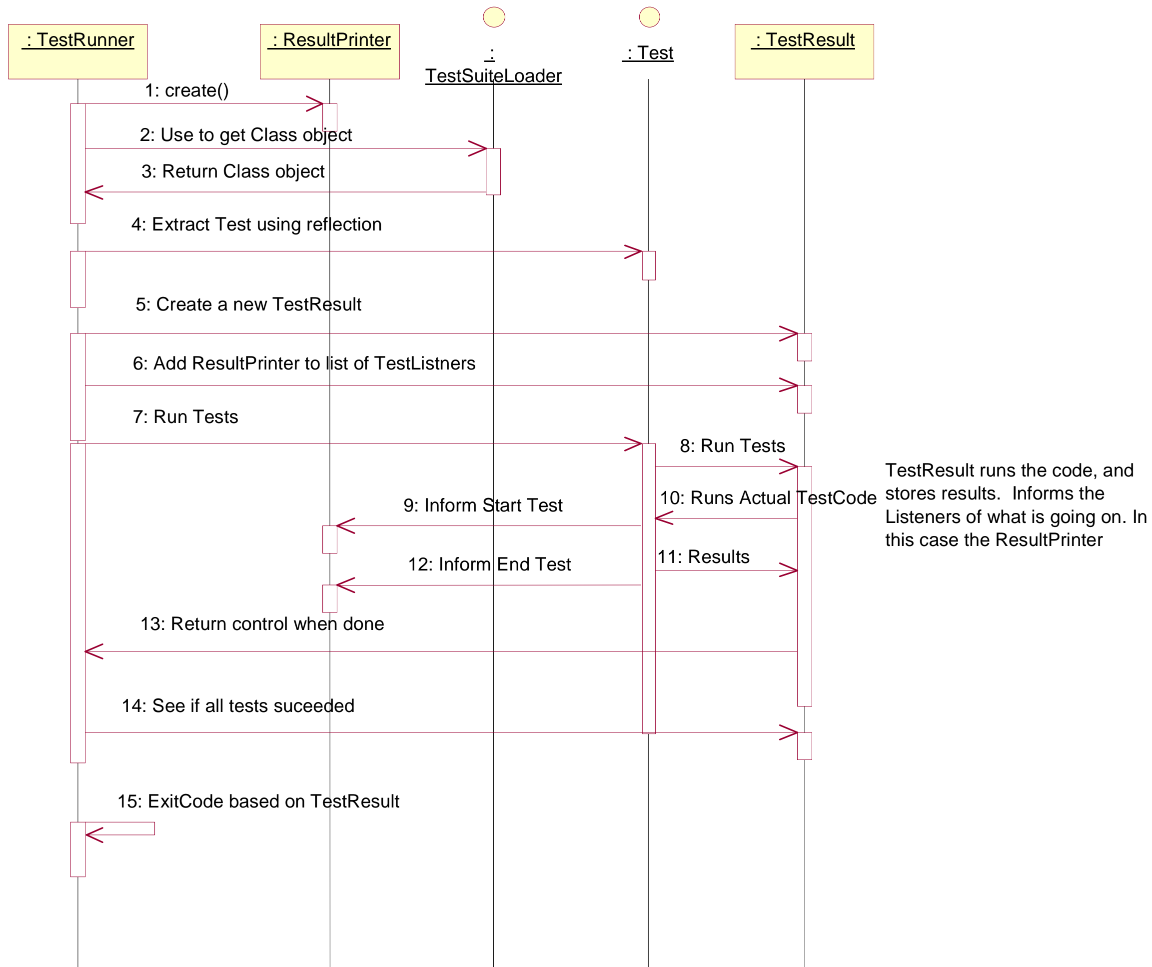

Figure 2.4.2 - A brief sequence diagram of how a test runner runs the tests.

The newly created TestRunner is given the responsibility to run the tests and place the output in a new TestResult object.

```
TestResult r= aTestRunner.start(args);
```

It does this via a number of nested function calls that become more and more specific. The `Start()` method is responsible for parsing the command line arguments and extracting the Test Class name.

```
testCase= args[i];
```

The test in question is extracted via a call to the inherited `getTest(String classname)` method.

```
Test suite= getTest(testCase);
```

If we follow the hierarchy of nested function calls, it can be shown that the `getTest()` calls a SuiteLoader to return a Class object which is then used to extract either a TestSuite or TestCase based on the information available using Java reflection . A new Test object is instantiated, which enumerates any underlying TestCases and TestSuites. (Step 2,3,4 Figure 2.4.2)

The `Start()` method then calls the `doRun()` method which will eventually return a TestResult object that summarizes the tests performed.

//in the start() method

```
…
return doRun(suite, wait); //Returns a TestResult
```

The doRun() method creates a new TestResult object and informs it that the ResultPrinter is the ouput object it should inform about test status. (Step 5,6 Figure 2.4.2)
//in the doRun()

```
TestResult result= createTestResult();
result.addListener(fPrinter);
…
return result;
```

Following this, the Test that was extracted previously is run. (Step 7 Figure 2.4.2)

```
suite.run(result);
```

Following the hierarchy outside of the TestRunner class, we assume that the Test class is a TestCase. If it were a TestSuite, the following would be run on each of the TestCases inside the TestSuite.

//TestCase.java

```
public void run(TestResult result) {
     result.run(this);
}
```

This code calls the `run()` method of the TestResult created previously. It parse `this` which is the Test instance. (Step 8 Figure 2.4.2). This makes sense, as there must be a controller class to run and catalogue multiple tests.

The TestResult class informs all the specified listeners (from Step 6 Figure 2.4.2) and increments the TestCount. (Step 8 Figure 2.4.2)

//TestResult.java

```
startTest(test);


public void startTest(Test test) {
….
     for (Enumeration e= cloneListeners().elements();
          e.hasMoreElements(); ) {
          ((TestListener)e.nextElement()).startTest(test);
     }
}
```

The TestResult then gets the Test object to run the Test code. The TestResult catches any exceptions thrown and stores them if a more detailed analysis is required. (Steps 10,11 Figure 2.4.2)

//TestResult.java

```
runProtected(test, p);

public void runProtected(final Test test, Protectable p) {
     try {
          p.protect();
     }
     catch (AssertionFailedError e) {
          addFailure(test, e);
```

```
        }
        catch (ThreadDeath e) { // don't catch ThreadDeath by
                                   accident
             throw e;
        }
        catch (Throwable e) {
             addError(test, e);
        }
   }
```

Note: Because each TestCase calls the TestResult, to run multiple tests it is the responsibility of the TestSuite to run all the TestCases it defines.

//TestSuite.java

```
   /**
    * Runs the tests and collects their result in a TestResult.
    */
   public void run(TestResult result) {
        for (Enumeration e= tests(); e.hasMoreElements(); ) {
             if (result.shouldStop() )
                  break;
             Test test= (Test)e.nextElement();
             runTest(test, result);
        }
   }

   public void runTest(Test test, TestResult result) {
        test.run(result);
   }
```

Once the Test has been run, all listeners are informed that the Test has completed. (Step 12 Figure 2.4.2). Note: Only one test is ever run at the one time.

The `doRun()` method from the TestRunner class then returns and the results can be extrapolated.

// doRun() in TestRunner.java

```
        TestResult result= createTestResult();
        …
        suite.run(result);  //The method returns
        …
        return result; // This is returned to the start() method
```

//start() from TestRunner.java

```
        return doRun(suite, wait); // which returns the TestResult to
                                        the main() method
```

// main() from TestRunner.java

```
        TestResult r= aTestRunner.start(args);

        if (!r.wasSuccessful())
             System.exit(FAILURE_EXIT);
        System.exit(SUCCESS_EXIT);
```

The TestRunner exit code is determined by the result of the tests. All test output is handled by the ResultPrinter, which is called by the TestResult class.

It can be seen from this example how closely integrated the runner and the framework are. It should also be noted that this has been done in such a way as to provide extensibility, with TestListeners and scalability in the design and usage of the TestSuites, TestCases and the TestResult.

3 References

1. Coplien J., *'A brief history of design patterns'*, http://www.bell-labs.com/user/cope/Patterns/ICSE96/node3.html, Aug 1996.

2. Gamma E., *'JUnit: A Cook's Tour'*, http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

3. Vico, *'Patrones de diseño'*, http://www.vico.org/pages/PatronsDisseny.html, Oct 2000.