JUnit 3.8 Documented Using Collaborations

Dirk Riehle, dirk@riehle.org, www.riehle.org

Abstract

This technical report describes the design of the unit testing framework JUnit in its version 3.8. The documentation technique employed is an enhanced version of collaboration-based design, also known as role modeling. In collaboration-based design, objects are viewed as playing multiple roles in different contexts, called collaborations. A collaboration, in turn, organizes several usually distinct objects for a specific purpose, for example to provide a basic service or to maintain a state dependency between two objects.

The documentation presented here accounts for every method in the original JUnit 3.8 framework and assigns it to at least one role. This way, collaboration-based documentation finds a middle ground between the more coarse-grained class-based view and the more detailed method-by-method view of objects. The report uses UML packages and interfaces to capture collaborations and roles rather than the evolving UML collaboration specification notation. For the purposes of this document this is sufficient.

The general purpose of this documentation form is to ease framework comprehension; an additional purpose that motivated this document is to investigate whether collaborations can serve as basic units of functionality provided by a design like a framework. Such a measure of functionality can serve multiple purposes, for example estimating implementation efforts or measuring complexity.

1 Collaboration-Based Design

Role modeling (in the context of object-oriented software design) was invented over 20 years ago [1]. It is related to the CRC (Class Responsibility Collaboration) cards approach to software design [2]. In recent years, role modeling has made its way into UML where it is called collaborations. My dissertation about role modeling for framework design [3] showed that role modeling makes designing, documenting, and using frameworks easier than possible with the traditional class-based approach alone.

UML 2.0 provides support for developers to document objects as playing roles and engaging in collaborations [4]. However, it is still an evolving specification and is limited when it comes to full-blown role modeling. For this reason, this document uses a lightweight UML-based approach to describe the JUnit 3.8 design.

> **Definition: A collaboration is the specification of how objects, playing roles, work together to achieve a single focused purpose.**

Figure 1 shows an example collaboration with three roles (the TestResultObserver collaboration from JUnit 3.8). The UML concept of interface is used to specify a role, and the UML concept of package is used to scope a collaboration.

 Printed March 20, 2007

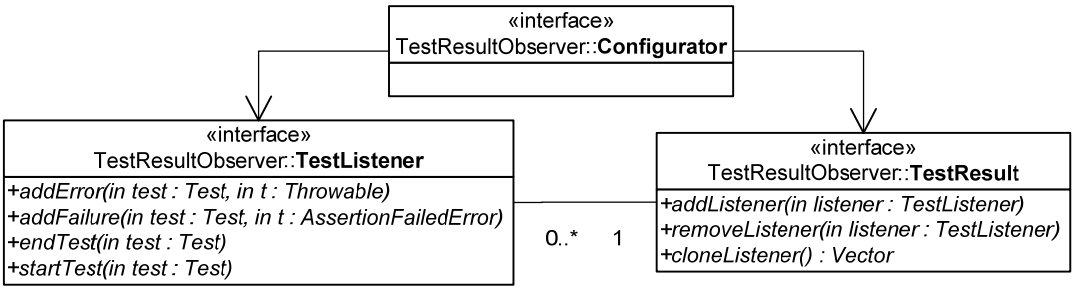


Figure 1: The TestResultObserver collaboration

What's important here is that collaborations consist of roles, and that a collaboration has *one* purpose, not many. This prepares the road for orthogonal composition of class-based designs from collaborations.

With this definition, a collaboration is a model. Its enactment at runtime is called a collaboration instance in this document. While a collaboration instance has identity, for most practical matter this is irrelevant (unless you start tracking transactions etc.)

Definition: A role is the specification of the behavior of an object within a given collaboration.

Technically, that specification may simply be a set of methods that define how an object plays the role. Please note that a role does not stand alone, it is always part of a collaboration. It has no right of existence outside the scope of a collaboration and for that purpose is bound to the collaboration.

Figure 2 shows the details of the TestResultObserver::TestResult role.

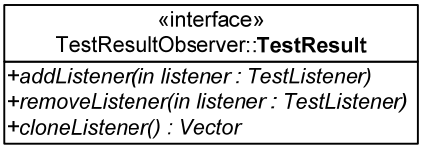


Figure 2: The TestResultObserver::TestResult role

Please note that while this document uses the UML concept of interface to specify a role's behavior, this does not imply the need for a technical (Java) interface. A UML interface may simply map into a set of methods in some larger Java interface or class.

We are adding to the diagrammatic notation simple pseudocode to list a role's methods. For example, the pseudocode for TestResultObserver::TestResult looks like this:

```
role TestResult {
    public synchronized void addListener(TestListener listener);
    public synchronized void removeListener(TestListener listener);
    private synchronized Vector cloneListeners();
}
```

Sometimes, a role is qualified as "free" as in "free role TestListener […]". This is to show that the user is free to assign this role to some classes. Non-free roles like TestResult above are typically assigned to classes in a fixed form, most notably by simply being part of the class interface. (Remember: The TestResult role specification above was taken out of the TestResult class, which provides several roles.) Free roles in Java are typically interfaces---otherwise the free assignment would not be possible. We discuss the importance of free roles for framework and component coupling elsewhere [5, 3].

The definition of role given above also implies that a collaboration specification can never have just one role; there must at least be two roles. If you can't find that second role: It may well be an implicit client role with no callbacks and hence no methods. It is in fact quite common to have client roles (of some basic service) that specify no meth-

ods at all. This doesn't mean there is no behavior attached to the client role. For example, in the TestResultObserver collaboration, the Configurator shouldn't call removeListener if it hasn't previously called addListener. Traditionally, such constraints were specified on the service side, that is, the TestResult role. What they really are, however, are constraints on the client side and are better specified on the client side as well.

Figure 3 shows the most common collaboration, the provision of a service to a client. This diagram really is a schematic illustration, not a specific collaboration, because each collaboration needs to list the domain-specific methods of the service role.

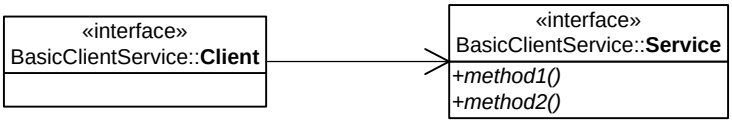


Figure 3: Illustration of most common client/service collaboration.

Another common type of collaboration adds a configuration role to the basic client/service collaboration. Here, the Configurator role configures the client with the service to use. In the basic client/service collaboration, the Client usually knows where to get the Service from. In a configurable client/service collaboration, this knowledge is moved to a separate Configurator object. Figure 4 makes the additional role explicit.

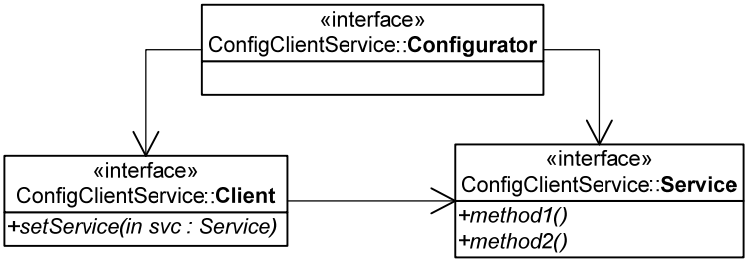


Figure 4: Illustration of a configurable client/service collaboration.

Beyond these domain-specific collaborations, most collaborations I have found were instances of design patterns. In fact, one can already argue that the configurable client/service collaboration is a pattern, as benign as it may appear. It is a hypothesis of this work (to be validated) that except for the basic domain-specific service provision through client/service collaborations, all collaborations and hence all other functionality are instances of design patterns.

In the following documentation we omit some of the complexity of a full-blown specification. What the documentation does, however, is to account for every single method in JUnit 3.8 and how it is part of a role in a collaboration that contributes one aspect of JUnit's overall functionality.

2 The Class View of JUnit 3.8

JUnit is a framework for writing unit tests (a.k.a. component tests) in Java. It is being developed by Kent Beck and Erich Gamma since 1997 and is based on SUnit, a Smalltalk testing framework. It has seen rapid and wide-spread adoption since its inception. It currently is in its version 4.0. This document focuses on the most widely used release, which is version 3.8. JUnit 3.8 is also the last major (and stable) release that provides the "old" design before the switch to Java annotations that happened with release 4.0.

Figure 3 shows the class diagram of the core JUnit framework. This is the contents of the Java package junit.framework. Other packages like junit.extensions and junit.runner are not considered further. Please note that

the top four classes in Figure 3 are classes defined in java.lang and presented here only for reasons of completeness. For another documentation of JUnit please see the original “cook’s tour” [6].

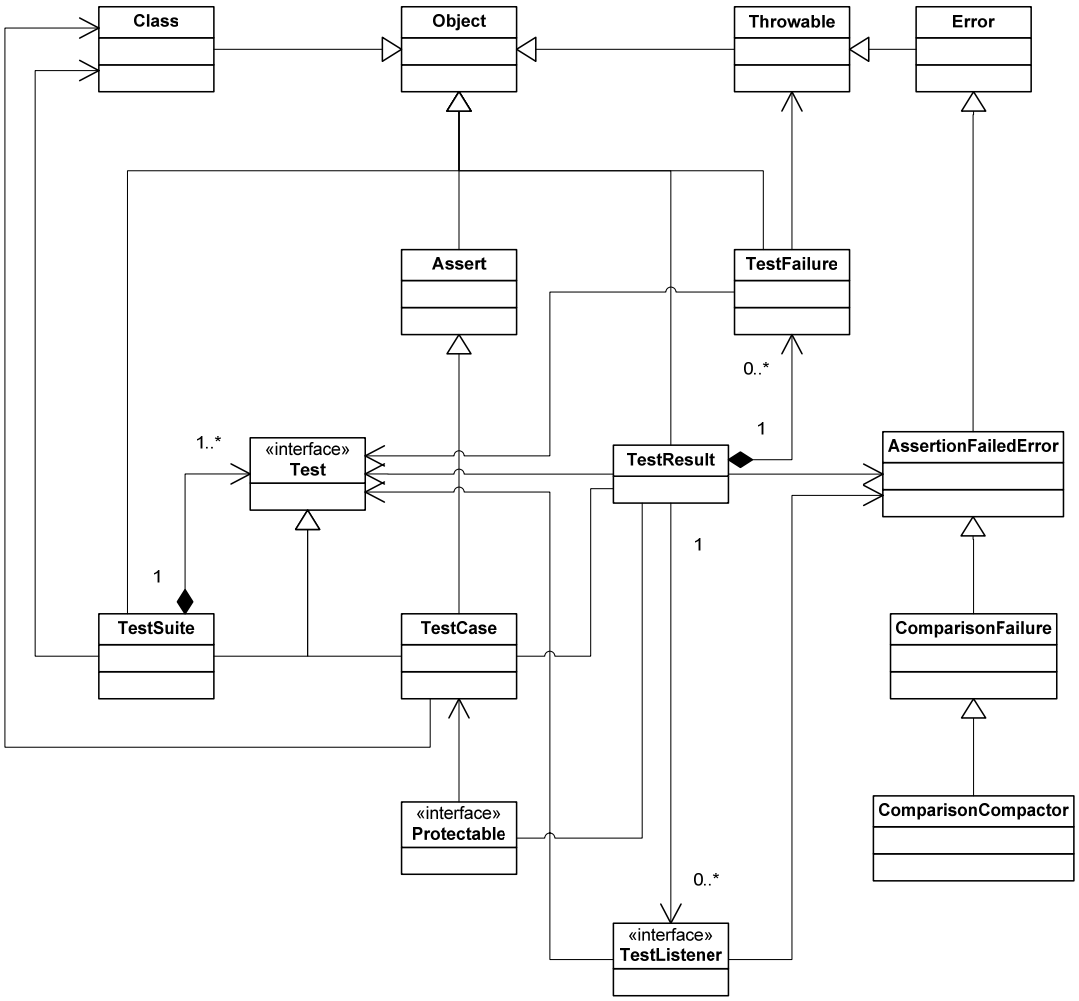


Figure 3: The class diagram of the JUnit 3.8 junit.framework classes.

In a nutshell, JUnit works the following way.

A complete test run has two phases: The first phase is configuration, second phase is test case execution. This may be followed by analysis and visual display of the results to a user.

During configuration, an object hierarchy of test cases is built, where one object represents a test case. Any such object is an instance of a subclass of TestCase. Programmers implement such subclasses, and every method in a subclass that starts with “test” (or is configured otherwise as a test case) leads to an object in the hierarchy. These “test” methods contain the actual (domain and application-specific) test code. Such TestCase subclass instances are always leaf objects of the hierarchy. The intermediate nodes and the root node are instances of TestSuite, which mostly provides the grouping functionality necessary for the tree structure. Both TestCase and TestSuite implement the Test interface to allow homogenous treatment of each node in the tree.

The actual test execution is started by calling run on the root object of the test hierarchy. The tree is traversed depth first, and the order of nodes is determined by the order in which they were added to the tree. On each and every leaf node, that is on each and every test case object, run will eventually be called. Thus, as many test cases will be run as there are instances of subclasses of TestCase. The results of each test case execution are recorded in a collecting

parameter object that is passed on from test to test as the execution is working its way through the test object hierarchy. The collecting parameter is the TestResult object.

JUnit distinguishes between failures and errors. Failures are instances of AssertionFailureError and are created by test case code as some assertion about the test code doesn't hold, that is, the test fails. An error occurs when something else goes wrong. Both are handled as exceptions in Java and are collected by the TestResult object being passed around. When storing a test failure or an error, the TestResult object wraps the failure in a TestFailure object.

When a test case object is first asked to execute the actual test code, it doesn't do so directly, but rather delegates this job to the TestResult object it receives as an argument. The TestResult object will prepare a few things and then call back on the test case for the actual test code execution. This callback is mediated by an inner subclass of Protectable that defines the test failure protocol. The TestResult's preparation consists of notifying registered TestListeners about the beginning and end of a test. This is used, for example, by a UI to display progress to a user.

In a test case, the actual test code is wrapped by calls to the methods setUp and tearDown that initialize the test case object respectively de-initialize it for test case execution. The framework method run finds the actual test method to execute by name matching using reflection and then executes it. If the test succeeds, the execution will run right through the code and return from the method call without problems. Programmers implement test code by making assertions about the state of the program under testing using calls to static "assert" methods inherited from class Assert. If an assertion fails an exception will be thrown to indicate a test failure. As mentioned above, this exception will be caught by the TestResult object and stored as a test failure before progressing to the next test case.

After the tree traversal finished and hence the test execution completed (or was interrupted), the client who started the test run can inspect the results as collected by the TestResult object and display them to a user or use them programmatically in a different way.

As a final comment, if you are wondering where the design patterns are [6]: Rather than saying Composite or Observer or Collecting Parameter, these patterns are recognizable by the streamlined prose they come with as well as their language specific adaptation. For example, you recognize Composite by the words tree, node, and leaf or Observer by callback, notifying, and listeners.

3 Collaboration Specifications

First, two general collaborations (inherited from the java.lang Object framework):

- Printable
- Throwable

Then, the set of collaboration specifications that make up the junit.framework classes:

- TestCase
- TestSuite
- TestSuiteTestCreation
- TestRun (Command pattern)
- TestCaseTestRun
- TestSuiteTestRun
- TestHierarchy (Composite pattern)
- TestResult (Collecting Parameter pattern)
- TestResultController
- TestResultObserver (Observer pattern)
- CollectingTestRun (Command pattern)

- ProtectedTestRun (Adapter pattern)
- TestRunMethod (Template method)
- Assertions
- TestFailure
- AssertionFailedError
- ComparisonFailure
- ComparisonCompactor (Strategy pattern)
- CompactMethod (Composed Method pattern)

They are discussed in order.

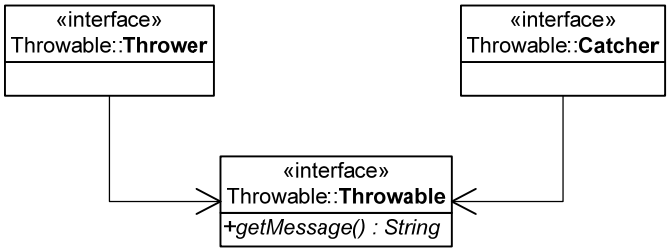
3.1 Printable

A Printable object provides the well-known toString method for providing a string representation of itself. The Client role is a free role because any object can assume the Client role in the Printable collaboration to call the method.


```
public collaboration Printable {

    free role Client {
        // no methods
    }

    role Printable {
        public String toString();
    }

}

// role to class assignments
public class Object provides Printable;
```

3.2 Throwable

The Throwable collaboration allows a Thrower to throw a Throwable and a Catcher to catch it. This collaboration captures the basic exception handling collaboration in Java; it is supported by the language through keywords throw and catch. Please note that creating and configuring an instance of a subclass of Throwable is viewed and handled as a separate collaboration.



```
public collaboration Throwable {

    free role Thrower {
        // no methods
    }

    free role Catcher {
        // no methods
    }

    role Throwable {
        public String getMessage();
    }

}

// role to class assignments
public class Throwable provides role Throwable;
```

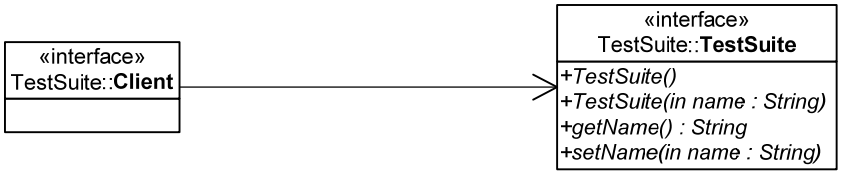
3.3 TestCase

This client/service collaboration lets a Client create a TestCase object and provide and retrieve basic information about it, namely its name. Theoretically, the constructor calls could be factored out into another collaboration (separate from the get/setName methods), however, as happens frequently on the code level, they are mixed together and for simplicity's sake are joined here in one collaboration.


```
public collaboration TestCase {

    free role Client {
        // no methods
    }

    role TestCase {
        public TestCase();
        public TestCase(String name);
        public String getName();
        public void setName(String name);
    }

}

// role to public class assignment
public class TestCase provides role TestCase;
```

3.4 TestSuite

Mirroring the basic TestCase collaboration, this client/service collaboration lets a Client create a TestSuite object and get and set its name.

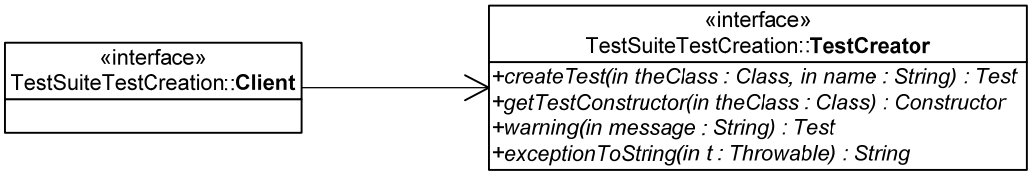


```
public collaboration TestSuite {

    free role Client {
        // no methods
    }

    role TestSuite {
        public TestSuite();
        public TestSuite(String name);
        public String getName();
        public void setName(String name);
    }

}

// role to class assignments
public class TestSuite provides role TestSuite;
```

Further constructors can be found in the TestHierarchy collaboration.

3.5 TestSuiteTestCreation

The TestSuiteTestCreation collaboration lets a Client use convenience methods for creating TestCase objects. These convenience methods are provided through the TestCreator role of the TestSuite class.




```
public collaboration TestSuiteTestCreation {

    free role Client {
        // no methods
    }

    role TestCreator {
        public static Test createTest(Class theClass, String name);
        public static Constructor getTestConstructor(Class theClass) throws NoSuchMethodException;
        public static Test warning(final String message);
        private static String exceptionToString(Throwable t);
    }

}

// role to public class assignment
public class TestSuite provides role TestCreator;
```

3.6 TestRun (Command Pattern)

Following the Command pattern [7] and according to [6], a Client calls run on a test to run that test. It provides a collecting parameter, a TestResult object, to track what's happening as the test run progresses.

```
public collaboration TestRun {

    free role Client {
        // no methods
    }

    role Command {
        public void run(TestResult result);
    }

}

// role to class assignments
public interface Test provides role Command;
```

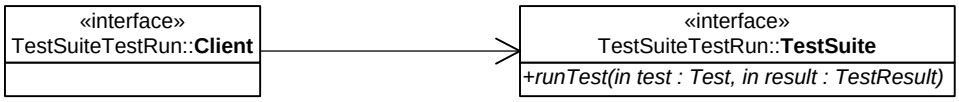
3.7 TestCaseTestRun

The TestCaseTestRun collaboration makes it easy for a Client to run a TestCase through convenience methods provided by a TestCase's TestCase role.


```
public collaboration TestCaseTestRun {

    free role Client {
        // no methods
    }

    role TestCase {
        public TestResult run();
        protected TestResult createResult();
    }

}

// role to class assignments
public class TestCase provides role TestCase;
```

3.8 TestSuiteTestRun

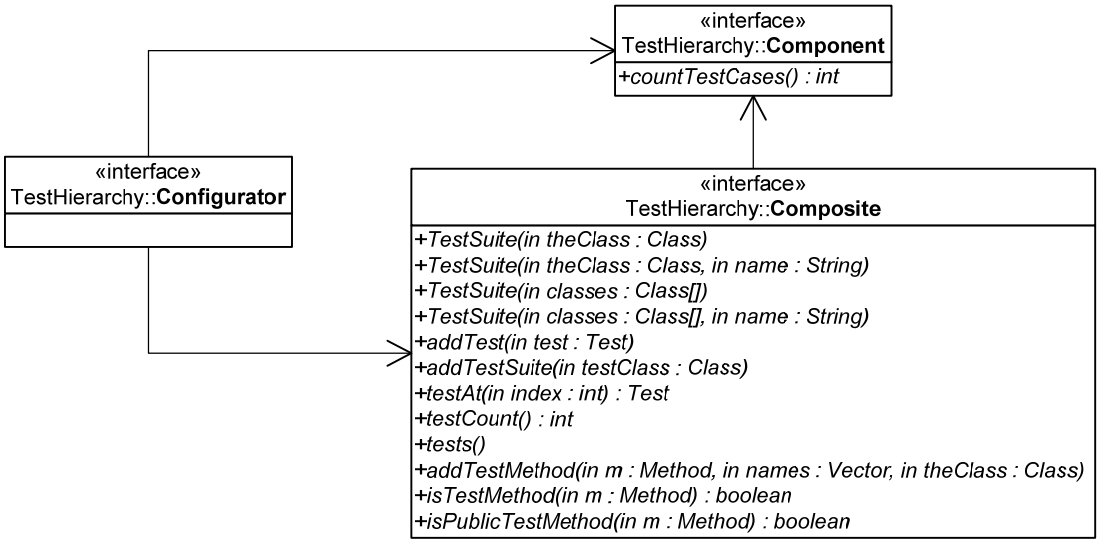
The TestSuiteTestRun collaboration lets a Client run a provided test with a given TestResult. This is a single method provided by the TestSuite role. The whole collaboration seems superfluous as there is exactly one use within the framework and none within the provided clients.



```
public collaboration TestSuiteTestRun {

    free role Client {
        // no methods
    }

    role TestSuite {
        public void runTest(Test test, TestResult result);
    }

}

// role to class assignments
public class TestSuite provides role TestSuite;
```

3.9 TestHierarchy (Composite Pattern)

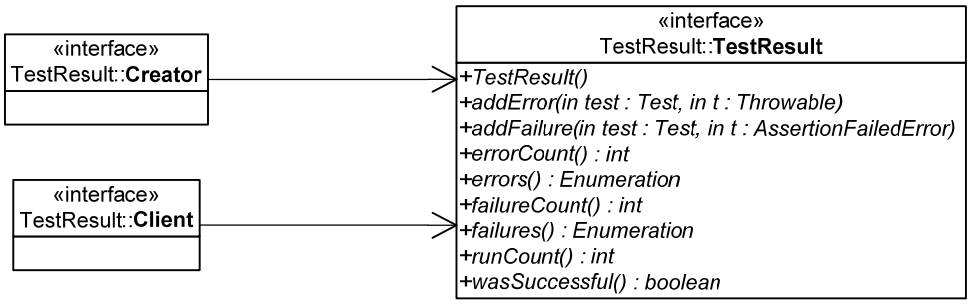
An application of the Composite pattern [6, 7], the TestHierarchy collaboration lets a Configurator object create a tree of Test objects. Every object in the tree is a Component and provides the countTestCases method; non-leaf objects are Composites and they provide methods for managing their children. In particular, they provide convenience methods for building child objects from TestCase classes, following the convention that any method in such a class that starts with “test” is a test case.



```
public collaboration TestHierarchy {

    free role Configurator {
        // no methods
    }

    role Component {
        public int countTestCases();
    }

    role Composite {
        public TestSuite(final Class theClass);
        public TestSuite(Class theClass, String name);
        public TestSuite (Public class[] public classes);
        public TestSuite(Public class[] public classes, String name);
        public void addTest(Test test);
        public void addTestSuite(Public class testPublic class);
        public Test testAt(int index);
        public int testCount();
        public Enumeration tests();
        private void addTestMethod(Method m, Vector names, Class theClass);
        private boolean isPublicTestMethod(Method m);
        private boolean isTestMethod(Method m);
    }

}

// role to class assignments
public interface Test provides role Component;
public class TestCase implements interface Test;
public class TestSuite implements interface Test provides role Composite;
```

3.10 TestResult (Collecting Parameter Pattern)

The TestResult collaboration lets a Client provide to and retrieve information from a TestResult object. Errors and Failures are collected, and statistics about them are provided. The TestResult collaboration is an instance of the Collecting Parameter Pattern [8]. The TestResult object is passed through the test case hierarchy while running the tests. It is important to note that the Creator and the Client are typically different objects, as the Creator role is played by an object outside the test case hierarchy. In a canonical implementation of the pattern, the Client role would be played by an object different from the one playing the TestResult role, but in JUnit the TestResult object itself is one of the objects that takes on the Client role, starting a test case and collecting its result. (Other objects that take on the Client role are objects from outside junit.framework as Client is a free role.)



```
public collaboration TestResult {

    free role Creator {
        // no methods
    }

    free role Client {
        // no methods
    }

    role TestResult {
        public TestResult()
        public synchronized void addError(Test test, Throwable t);
        public synchronized void addFailure(Test test, AssertionFailedError t);
        public synchronized int errorCount();
        public synchronized Enumeration errors();
        public synchronized int failureCount();
        public synchronized Enumeration failures();
        public synchronized int runCount();
        public synchronized boolean wasSuccessful();
    }

}

// role to class assignments
public class TestResult provides role TestResult;
```

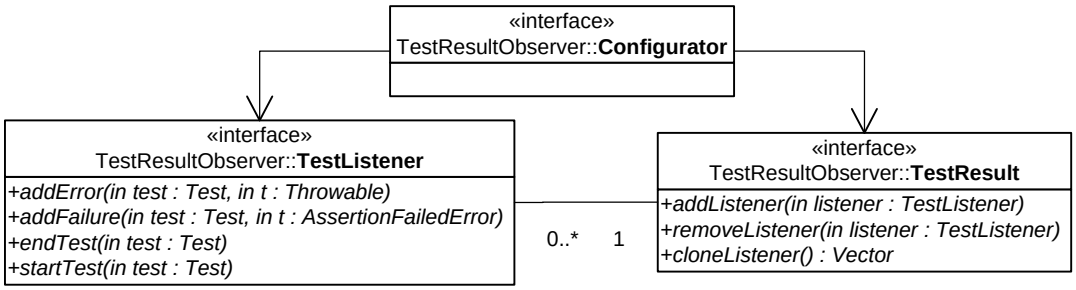
3.11 TestResultController

The TestResultController collaboration lets a Client object stop the execution of a current test run by calling stop on it. The Client role is a free role played by test runners and other clients from outside junit.framework. This necessitates the use of threading or other means of concurrency to separate clients from the actual test runs.


```
public collaboration TestResultController {

    free role Client {
        // no methods
    }

    role TestResult {
        public synchronized boolean shouldStop();
        public synchronized void stop();
    }

}

// role to class assignments
public class TestResult provides role TestResult;
```

3.12 TestResultObserver (Observer Pattern)

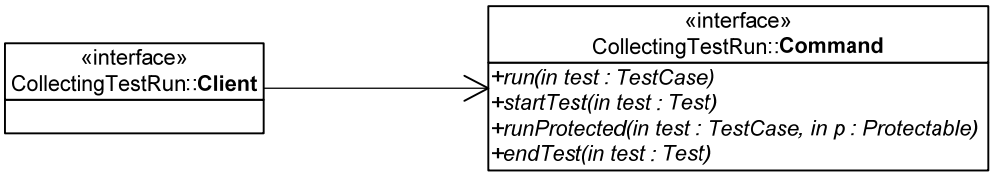
An instance of the Observer pattern [7], the TestResultObserver collaboration lets a Configurator register TestListeners at a TestResult object such that the TestResult can call back on the listeners to inform them about how the test run is progressing. The TestResult tells its listeners when a test case starts and when it ends as well as if an error or a failure occurred.




```
public collaboration TestResultObserver {

    free role Configurator {
        // no methods
    }

    free role TestListener {
        public void addError(Test test, Throwable t);
        public void addFailure(Test test, AssertionFailedError t);
        public void endTest(Test test);
        public void startTest(Test test);
    }

    role TestResult {
        public synchronized void addListener(TestListener listener);
        public synchronized void removeListener(TestListener listener);
        private synchronized Vector cloneListeners();
    }

}

// role to class assignments
public interface TestListener provides role TestListener;
public class TestResult provides role TestResult;
```

3.13 CollectingTestRun (Command Pattern)

In a second instantiation of the Command pattern [7], the Client, played by a TestCase object, delegates the test run handling to a Command, played by the TestResult object. Thus, the TestResult object shoulders yet another responsibility, here wrapping the actual test run with calls to the startTest and endTest methods, which track the test runs and notify test listeners.

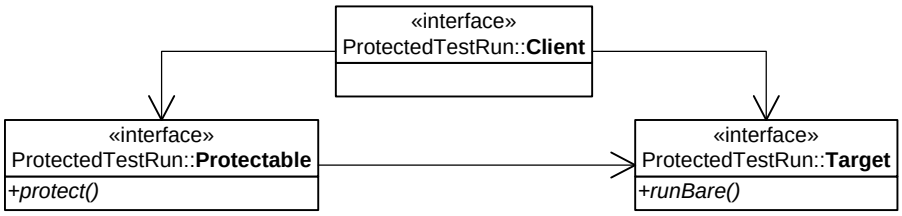


```
public collaboration CollectingTestRun {

    free role Client {
        // no methods
    }

    role Command {
        public void run(final TestCase test);
        public void startTest(Test test);
        public void runProtected(final Test test, Protectable p);
        public void endTest(Test test);
    }

}

// role to class assignments
public class TestCase provides role Client;
public class TestResult provides role Command;
```

3.14 ProtectedTestRun (Adapter pattern)

An instance of the (Object) Adapter pattern, the ProtectedTestRun collaboration continues the test run delegation chain of calls by injecting a Protectable object, an instance of an inner class, into the test run call sequence. Here, the Client, played by the TestResult object, creates the Protectable and configures it with a Target, the actual TestCase. The Protectable's run method is called by the Client and in turn forwards it to the Target object, while "protecting" the call with some exception handling. The adaptation being performed is the enhancement of the signature of the run method with the declaration of a possible Throwable (exception) being thrown.

Thus, reviewing the call sequence, first run is called on a TestCase, which then calls run on the TestResult argument, which in turn calls protect on an anonymous Protectable, which then calls runBare on the original TestCase object.

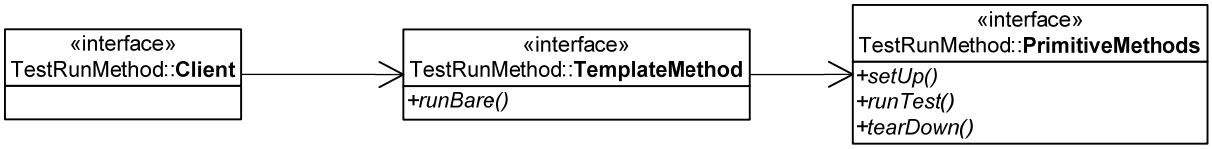



```
public collaboration ProtectedTestRun {

    free role Client {
        // no methods
    }

    role Protectable {
        public void protect() throws Throwable;
    }

    role Target {
        public void runBare() throws Throwable;
    }

}

// role to class assignments
public class TestResult provides role Client;
interface Protectable provides role Protectable;
public class TestCase provides role Target;
```

3.15 TestRunMethod (Template Method)

An instance of the Template Method pattern [6, 7], the TestRunMethod collaboration structures the actual test case execution into before and after method calls, to setUp and tearDown respectively, and a call to the final test case

code provided by the runTest method. Subclasses use setUp and tearDown to provide test case specific initialization and de-initialization.

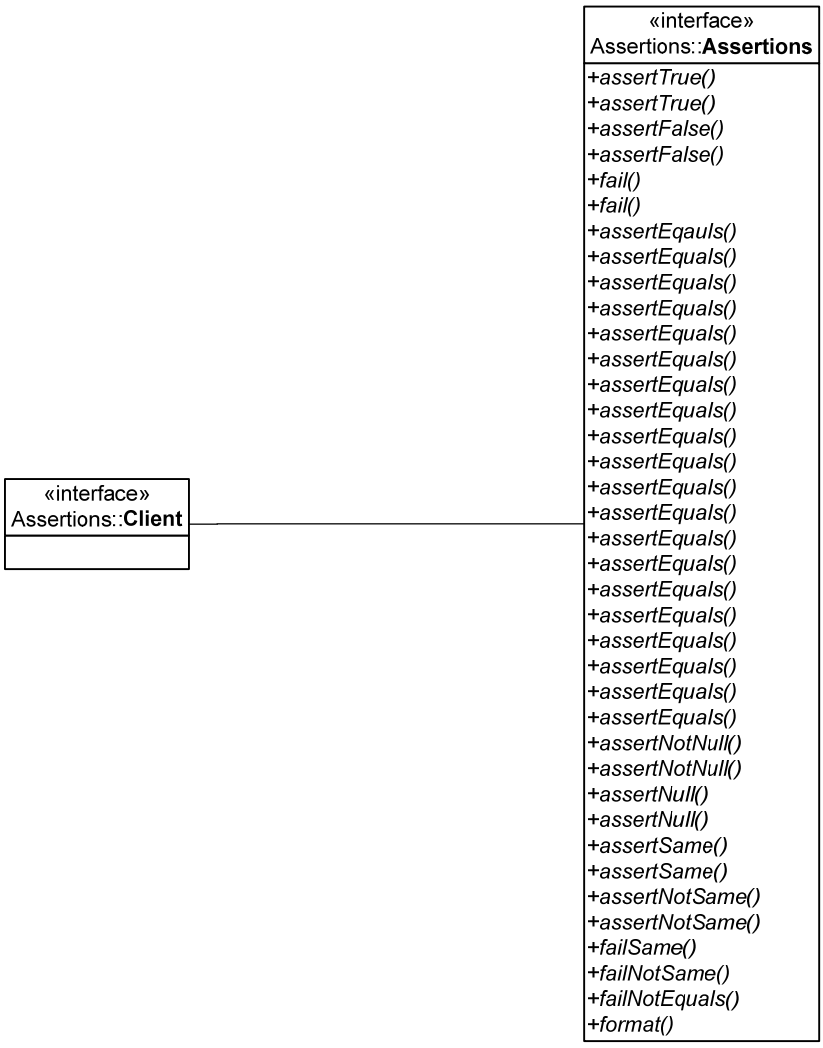



```
public collaboration TestRunMethod {

    free role Client {
        // no methods
    }

    role TemplateMethod {
        public void runBare() throws Throwable;
    }

    role PrimitiveMethods {
        protected void runTest() throws Throwable;
        protected void setUp() throws Exception;
        protected void tearDown() throws Exception;
    }

}

// role to class assignments
public class TestCase provides role TestCase;
```

The method runBare is shared with ProtectedTestRun::Target because in the call sequence they are joined at the hip.

3.16 Assertions

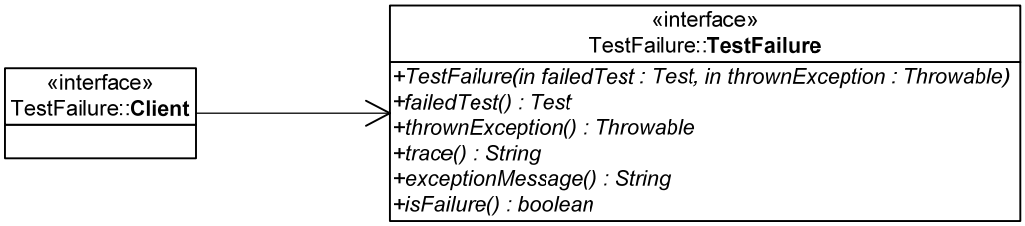
Test code typically performs some action and then tests whether the action led to the desired results. This checking for expected results is codified by calls to assertion methods, which cast the checking as a (possibly arbitrarily complex) expression that evaluates to either true or false. If the expression evaluates to false, the assertion fails, and hence the test fails. If the test fails, an AssertionFailedError will be thrown.


«interface»
Assertions::Assertions
+assertTrue()
+assertTrue()
+assertFalse()
+assertFalse()
+fail()
+fail()
+assertEqauls()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertEquals()
+assertNotNull()
+assertNotNull()
+assertNull()
+assertNull()
+assertSame()
+assertSame()
+assertNotSame()
+assertNotSame()
+failSame()
+failNotSame()
+failNotEquals()
+format()
«interface»
Assertions::Client

```
public collaboration Assertions {

    free role Client {
        // no methods
    }

    role Assertions {
        protected Assert();
        public static void assertTrue(String message, boolean condition);
        public static void assertTrue(boolean condition);
        public static void assertFalse(String message, boolean condition);
        public static void assertFalse(boolean condition);
        public static void fail(String message);
        public static void fail();
        public static void assertEquals(String message, Object expected, Object actual);
        public static void assertEquals(Object expected, Object actual);
        public static void assertEquals(String message, String expected, , String actual);
        public static void assertEquals(String expected, String actual);
        public static void assertEquals(String message, double ex, double actual, double delta);
        public static void assertEquals(double expected, double actual, double delta);
        public static void assertEquals(String message, float ex, float actual, float delta);
        public static void assertEquals(float expected, float actual, float delta);
        public static void assertEquals(String message, long expected, long actual);
        public static void assertEquals(long expected, long actual);
        public static void assertEquals(String message, boolean expected, boolean actual);
        public static void assertEquals(boolean expected, boolean actual);
        public static void assertEquals(String message, byte expected, byte actual);
        public static void assertEquals(byte expected, byte actual);
        public static void assertEquals(String message, char expected, char actual);
        public static void assertEquals(char expected, char actual);
        public static void assertEquals(String message, short expected, short actual);
        public static void assertEquals(short expected, short actual);
        public static void assertEquals(String message, int expected, int actual);
        public static void assertEquals(int expected, int actual);
        public static void assertNotNull(Object object);
        public static void assertNotNull(String message, Object object);
        public static void assertNull(Object object);
        public static void assertNull(String message, Object object);
        public static void assertSame(String message, Object expected, Object actual);
        public static void assertSame(Object expected, Object actual);
        public static void assertNotSame(String message, Object expected, Object actual);
        public static void assertNotSame(Object expected, Object actual);
        public static void failSame(String message);
        public static void failNotSame(String message, Object expected, Object actual);
        public static void failNotEquals(String message, Object expected, Object actual);
        static String format(String message, Object expected, Object actual);
    }

}

// role to class assignments
public class Assert provides role Assertions;
```

3.17 TestFailure

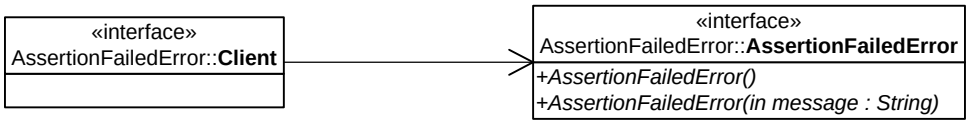
The TestFailure collaboration lets a Client create and update a TestFailure object. For each failed test, a TestFailure object is created that provides information about the failed test. The TestFailure object is created when the test fails and its data is read and analyzed by the client that started the test run.



```
public collaboration TestFailure {

    free role Client {
        // no methods
    }

    role TestFailure {
        public TestFailure(Test failedTest, Throwable thrownException);
        public Test failedTest();
        public Throwable thrownException();
        public String trace();
        public String exceptionMessage();
        public boolean isFailure();
    }

}

// role to public class assignment
public class TestFailure provides TestFailure;
```

3.18 AssertionFailedError

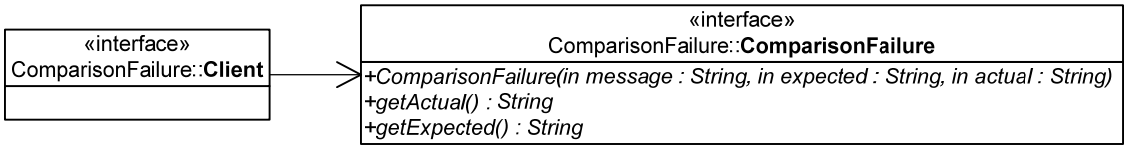
The AssertionFailedError collaboration is a simple client/service collaboration that lets a Client create and configure an AssertionFailedError exception. AssertionFailedErrors are exceptions that are created and thrown by the various assertion methods of Assertions::Assertions as listed above. To a Client they only provide a simple String message (as defined in the Throwable collaboration).




```
public collaboration AssertionFailedError {

    free role Client {
        // no methods
    }

    role AssertionFailedError {
        public AssertionFailedError();
        public AssertionFailedError(String message);
    }

}

// role to class assignments
public class AssertionFailedError provides role AssertionFailedError;
```

3.19 ComparisonFailure

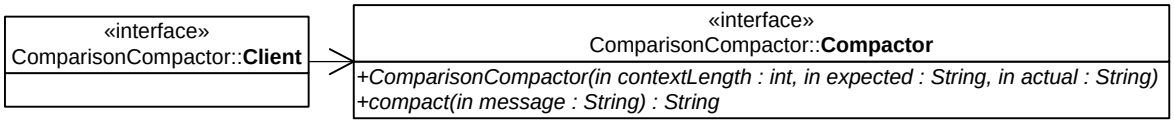
The ComparisonFailure collaboration is another simple client/service collaboration that lets a Client create and configure a ComparisonFailure exception. Over the simple Throwable protocol it adds methods to get an actual and an expected string value that represent the actual and expected values from some failed assertion.



```
public collaboration ComparisonFailure {

    free role Client {
        // no methods
    }

    role ComparisonFailure {
        public ComparisonFailure(String message, String expected, String actual);
        public String getActual();
        public String getExpected();
    }

}

// role to class assignments
public class ComparisonFailure provides role ComparisonFailure;
```

3.20 ComparisonCompactor (Strategy Pattern)

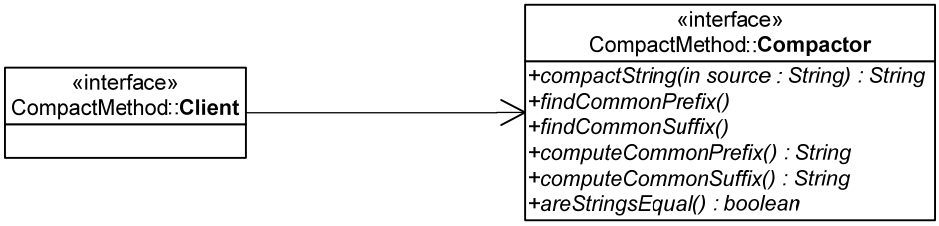
An application of the Strategy pattern, the ComparisonCompactor collaboration lets a client use a Compactor object to cut out repeating contents from an actual and an expected string value. This is used to shorten the textual display of a failed test so that a user can more quickly grasp the actual and expected values whose difference made the test fail.




```
public collaboration ComparisonCompactor {

    free role Client {
        // no methods
    }

    role Compactor {
        public ComparisonCompactor(int contextLength, String expected, String actual);
        public String compact(String message);
    }
}

// role to class assignments
public class ComparisonCompactor provides role Compactor;
```

3.21 CompactMethod (Composed Method Pattern)

Following the Composed Method pattern [8], the CompactMethod collaboration lets a Client, the compact method of ComparisonCompactor, realize its implementation by composinge a set of further methods, as provided by the Compactor role. It is basically a code factoring that eases comprehension and implementation.



```
public collaboration CompactMethod {

    role client {
        // no methods
    }

    role Compactor {
        private String compactString(String source);
        private void findCommonPrefix();
        private void findCommonSuffix();
        private String computeCommonPrefix();
        private String computeCommonSuffix();
        private boolean areStringsEqual();
    }
}

// role to class assignments
public class ComparisonCompactor provides role Client, Compactor;
```

4 Statistics

Finally, some data gathered from this simple but exemplary framework. Printable and Throwable are ignored.

4.1 Basic data

The overall number of methods:	114
The overall number of roles:	43
The overall number of collaborations:	19
The overall number of interfaces and classes:	11

4.2 Interpreted data

Average number of roles per collaboration:	2.3
The overall number of design pattern instances:	9
Percentage of collaborations that are patterns:	47%

4.3 Method sharing

There is surprisingly little overlap between different roles. The TestSuite::TestSuite and TestHierarchy::Composite roles share methods, and the ProtectedTestRun::Target and TestRunMethod::TestCase roles share methods, but that's about it.

5 Interpretation

See forthcoming publications.

References

[1] Trygve Reenskaug et al. *Working With Objects: The OORAM Software Engineering Method.* Prentice Hall, 1995.

[2] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[3] Dirk Riehle. *Framework Design: A Role Modeling Approach.* Dissertation No. 13509, ETH Zurich, 2000. Available from www.riehle.org/computer-science/research/dissertation.

[4] UML 2.0. *Unified Modeling Language.* OMG, 2007. Available from www.uml.org.

[5] Dirk Riehle and Thomas Gross. "Framework Design". In *Proceedings of the 1998 Conference on Object Oriented Programming Systems, Languages, and Applications* (OOPSLA '98). ACM Press, 1998. Available from www.riehle.org/computer-science/research.

[6] Kent Beck and Erich Gamma. "JUnit: A Cook's Tour". Available from www.junit.org.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[8] Kent Beck. *Smalltalk Best Practice Patterns.* Addison-Wesley, 1996.