

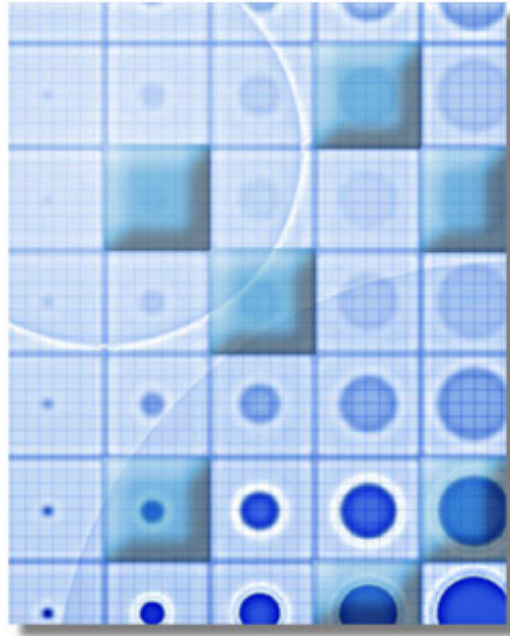
## JUnit Test Patterns in Rational XDE

by [Frank Hagenson](#)

Independent Consultant  
Northern Ireland

*Unit testing is one of the most important techniques available in the development of quality software. When you start to write unit tests, however, it's not long before you realize just how much effort is involved. Fortunately, a good unit-testing framework can really help, and one such framework is JUnit, which was originally developed by Kent Beck as part of his eXtreme Programming methodology.*

*Even with a good framework, however, you must still write a lot of repetitive and mundane code just to support your test harness. In this article, I show how I used Rational XDE to create a design pattern that automates the creation of test cases and suites, using the JUnit framework.*



*The goals of this project were:*

- 1. Automate the generation of all housekeeping code. I wanted my pattern to generate all required structural and behavioral code that doesn't need human input (i.e., doesn't require thinking).*
- 2. To make the generated code compile and run even when the pattern is applied multiple times.*

*This was the first iteration of my pattern, so to keep the scope manageable I restricted the pattern to working with one class at a time. I also did not do anything fancy, like trying to keep the test cases in sync with the application code. If a class name is changed, the test code will break, and it's a manual job to fix it.*

## JUnit

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

JUnit is a test framework. It provides a set of classes and tools that can be used to create and run tests for Java applications. The core of the JUnit framework is the *TestCase* class, which provides basic functionality to create and run tests. The normal usage scenario is to create a subclass of *TestCase*, and then add methods to it that test something.

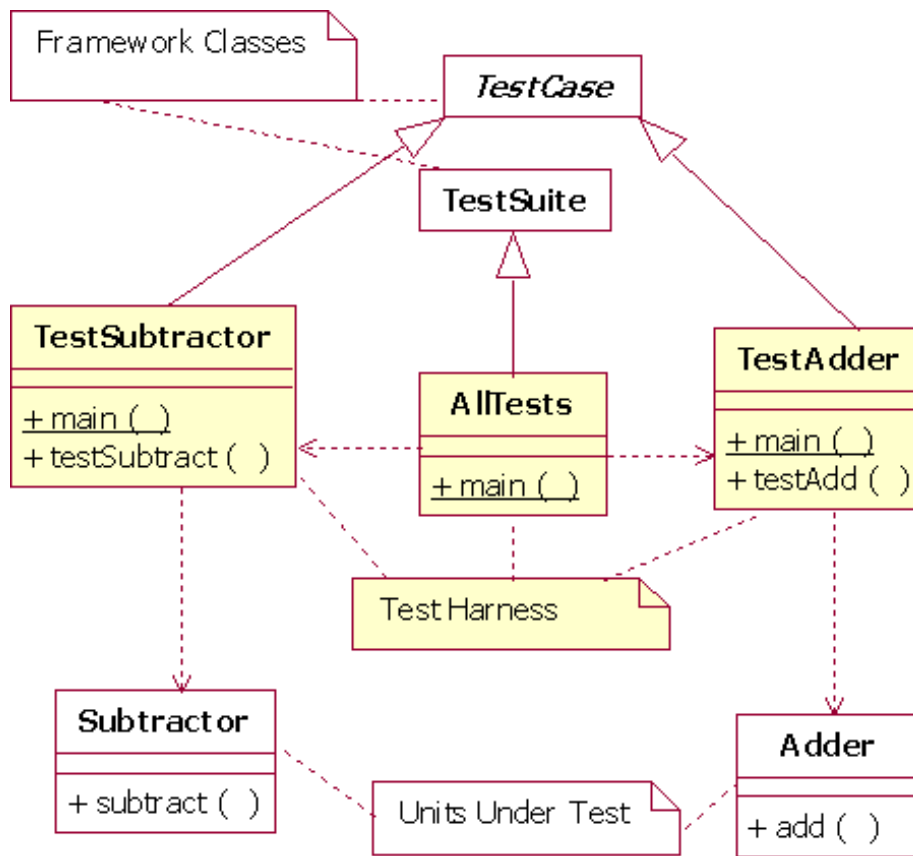
In order to run the tests, the new *TestCase* is added to a *TestSuite*, and the method `TestSuite.run` is called. The *TestSuite* then uses reflection to run the test methods of the *TestCase*. Any method in the *TestCase* that begins with "test" is treated as a test case and executed. So you use Java code to test Java code.

## The Test Harness

The test harness is all the code that is there simply to do testing. It adds nothing to the functionality of the released application. Any piece of code that I want to test is called the "Unit Under Test," or UUT for short. With Java, the UUT is generally a class, because in Java the class is the smallest unit of stand-alone code.

The test harness code can easily take up as much code as the application under test itself, so it makes good sense to structure the harness in a way that makes it easy to manage and modify. The best way I have found to do this is to create a subclass of *TestCase* for each UUT, and a subclass of *TestSuite* that allows you to run all test cases in one go. Each *TestCase* will invoke methods on the UUT and check that the values returned are as expected.

For example, if I had two classes, *Adder* and *Subtractor*, I would create two test cases called *TestAdder* and *TestSubtractor*, respectively, plus a test suite called something like *AllTests*. The UML representation of the test harness structure would look something like Figure 1.



**Figure 1: UML Class Diagram of Test Harness**

To make it simple to test a single class, I normally add a main method to the *TestCase* that simply runs the tests in the class and outputs the results to standard out. In this way I can just compile and run the test case to check whether the code I've just developed works or not. The JUnit framework provides everything I need to do this, so the main method for a class like *TestAdder* would look something like this:

```

public static void main(String[] args) {
    junit.textui.TestRunner.run(AdderTest.class);
}

```

I also put a main method on the *TestSuite* class, so that I can easily run all my tests from the command line. I generally make running this *TestSuite* a part of the build process, so that no application is said to build correctly until it has compiled **and** all its test cases have executed successfully.

The main method of the test suite would look something like this:

```

public static void main(String[] args) {
    AllTests suite = new AllTests();
    junit.textui.TestRunner.run(suite);
}

```

The *TestSuite* constructor also needs to do a bit of work. It must add all known *TestCases* to the suite, so that the main method can execute them.

So my test suite constructor would look something like this:

```
public AllTests() {
    this.addTest(new TestSuite(TestAdder.class));
    this.addTest(new TestSuite(TestSubtractor.class));
}
```

Then I can finally start adding test methods to the TestCases to exercise the behavior of the classes being tested. Supposing my *Adder* class had a method named *add* that returned the sum of two numbers passed to it; a test for this method might look something like this:

```
public void testAdd()
{
    Adder uut = new Adder();
    assertTrue("1+1!=2", uut.add(1,1) == 2);
}
```

This method instantiates a new *Adder* and checks that the result of the call to *add* equals two by calling the JUnit framework method *assertTrue*. If the second parameter passed to *add* does not evaluate to true, JUnit throws an exception and records this as a test failure.

Only the code in the *testAdd* method is really specific to testing the *Adder* class. So even with a framework like JUnit, there is a lot of housekeeping code that needs to go in place before I can actually write the tests. Another painful thing is that every time I create a new test case, I have to go and add it to my TestSuite. These are the kinds of tasks I'd like to automate.

In summary, the tasks to be automated are:

1. Write the structural code for a new TestCase.
2. Write the TestSuite code if we don't have one yet.
3. Add the code to add an instance of the new TestCase to the TestSuite at run time.
4. Write a main method for the new TestCase that runs the TestCase and outputs the results to standard out.
5. Write a main method for the TestSuite that runs all TestCases and outputs the results to standard out.

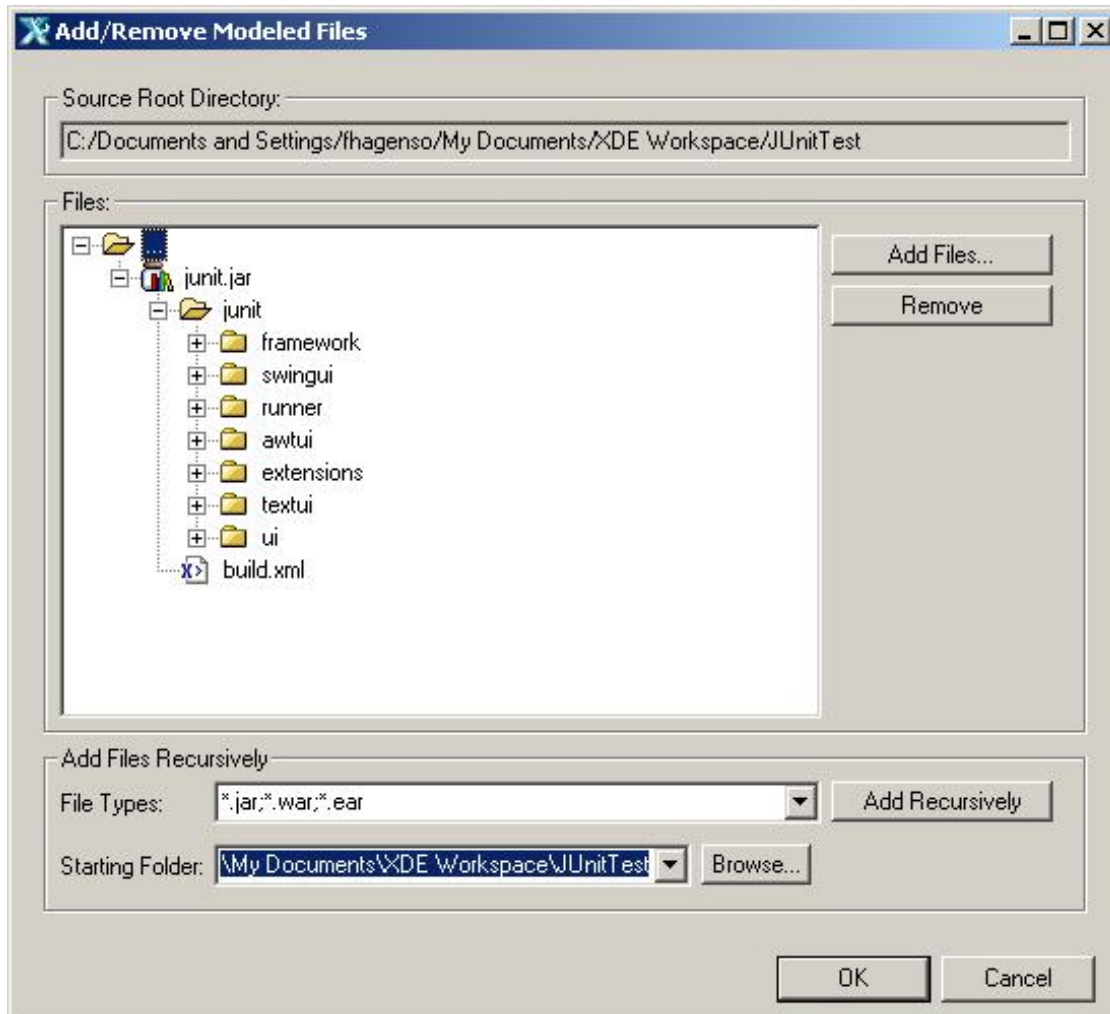
## The Structural Pattern

I decided to put the pattern and the JUnit framework classes together in a model, which can be reused with any application I'm developing. That way, I can just open the model and apply the pattern as required. All I have to remember to do is add a reference from my application model to my JUnit pattern model, so that the code will generate and compile correctly.

First, I must reverse engineer the JUnit framework, to get access to the classes it contains for my pattern. To do this, I create a new Java Modeling Project named *JUnit*, and copy the JUnit framework archive, *junit.jar*, into the root folder of the project. In the model navigator, I right click on the model and select from the menu:

### More Java Actions | Add / Remove Modeled Files

In the **Add/Remove Modeled Files** dialog shown in Figure 2, I set the File Types combo to `*.jar` and click the **Add Recursively** button. XDE pulls *junit.jar* into the project. When I click the OK button, XDE reverse engineers all of the classes in the *.jar* file.



**Figure 2: Reverse Engineering JUnit**

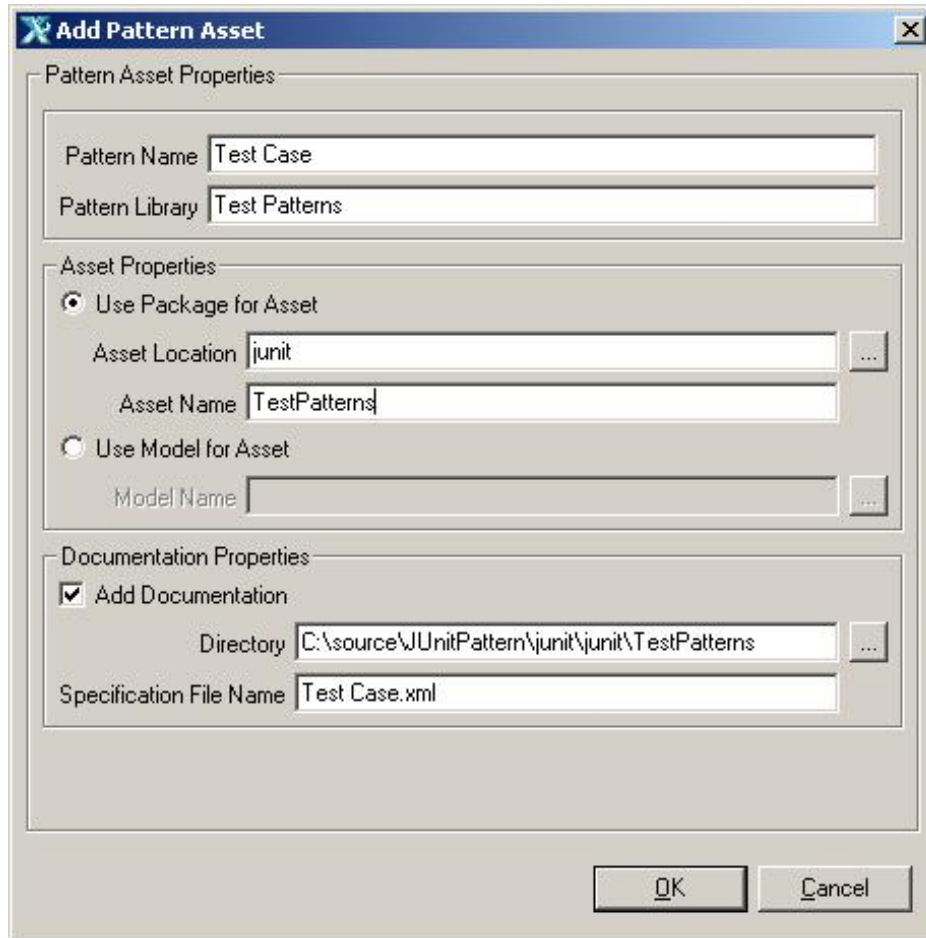
Abstracting out a general design pattern from the main usage scenario above, I come up with two parameters for our pattern. The first is the class I want to test, and the second is the suite, which may or may not exist.

In Rational XDE, creating a new design pattern is easy. I just right click on the JUnit model in the Model Explorer and select

### Add UML | Pattern Asset

from the menu. XDE presents me with a dialog box to specify how I want the

pattern asset created, as in Figure 3. Rational XDE will create a new package with the stereotype <<Asset>>. Inside that will be a new Parametized Collaboration.



**Figure 3: Creating a Pattern Asset**

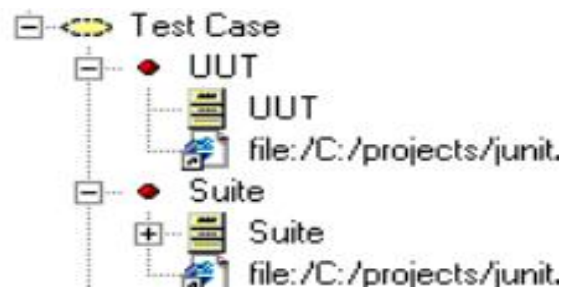
The Parametized Collaboration is a UML element I can use to specify any collaboration of UML elements that will modify or use a set of input parameters in some way. In this case, the input parameters are a class to be tested, and a test suite. The pattern will use these input parameters to generate the structure of a test harness. I add these parameters to the Collaboration by right clicking on it in the Model Explorer and selecting

### **Add UML | Template Parameter**

from the menu. To specify that the input parameters are classes, I then select each one, right click on it, and select

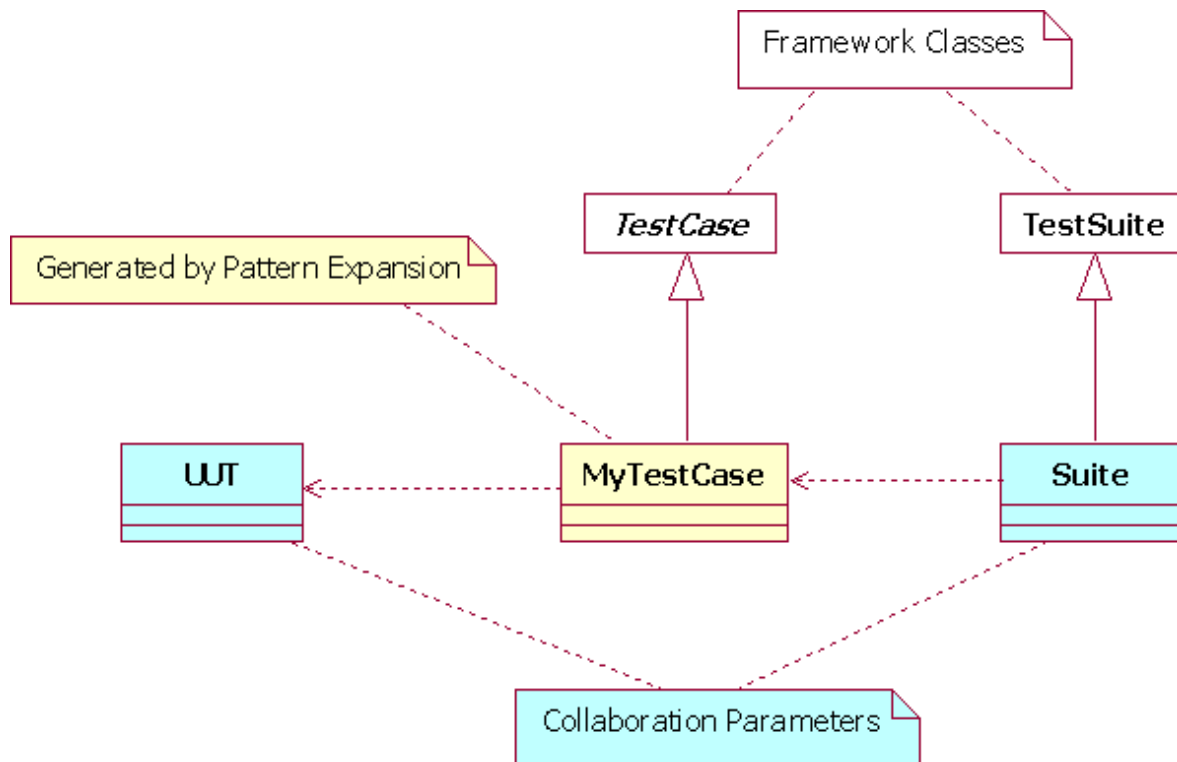
### **Add UML | Type | Class**

from the menu. This gives me a structure, as shown in Figure 4.



**Figure 4: Parametized Collaboration Structure in the Model Explorer**

Creating the structural aspect of the design pattern is straightforward. I simply create a new class diagram in the <<Asset>> package and put some classes on it with relationships between them. In this case, I need to add the *TestCase* and *TestSuite* classes from the JUnit framework, the classes that are parameters to the collaboration, any classes that will be generated as part of the pattern expansion, and then the required relationships between them. So my initial design looks like the class diagram in Figure 5. Now, when I apply the pattern, XDE prompts me for the class to be tested and the *TestSuite*, and then adds any other classes in the diagram to my model, including any relationships I have specified between these classes.



**Figure 5: Initial Structural Aspect of Pattern**

But here's where I have a problem. I've specified that the pattern should create a subclass of *TestCase* called *MyTestCase*, to be the test case for the UUT. However, that name is not very descriptive, and if I apply the pattern twice, I'll get two classes called *MyTestCase*. What I really want is to pre-pend "Test" to the name of the class I'm trying to test, so that my test case for *Adder* would be *TestAdder*, for example.

Luckily, XDE provides this capability through the use of *scriptlets*. A scriptlet is a little piece of JavaScript that will be executed when the pattern is

applied. To tell the pattern engine to execute the scriptlet, I just need to enclose it in `<%ý%>` markers. To tell the pattern engine to use the output of the execution as a replacement string, I just include a `=` (equals symbol) as the first character in the execution. Notice that this is very similar to the use of JavaScript in an ASP or JSP page.

I make this change and apply the pattern to a class. The pattern expansion creates a new test case and test suite class, and also a class diagram showing the expansion in the target package, as expected. Unfortunately, it also creates a copy of the framework classes in the destination package, but what I really want is for the expanded classes to reference the framework classes in the JUnit model.

To rectify this situation, I need to modify the merge behavior of the pattern via the Pattern Explorer. When I select the class diagram in the Pattern Explorer, I can view its Pattern Properties, one of which is the merge behavior. The different ways a UML element in a pattern can be merged into a model during pattern expansion are described in Table 1.

**Table 1: Merge Behaviors for Pattern Elements**

Value	Meaning
Merge	Create a new element if it doesn't exist; update if it's already there.
Preserve	Create a new element if it doesn't exist; don't touch if it's already there.
Replace	Always create a new element; overwrite if it's already there.
No Copy	Never create a new element; never update if it's already there.

By changing the merge behavior of the diagram from "Replace" (the default) to "No Copy," the diagram is not created in the model at expansion time; and, as a side effect, the framework classes that are present on the diagram are not re-created in the model when the pattern is expanded.

## Adding Behavior

Okay. Now I have a pattern that will create the structural aspect of a test-harness architecture. But architecture is more than just structure. What about behavior?

The actual tests that I will write to test my code will be different for each class, so they are not good candidates to automate as part of the pattern. But there are a lot of things I need to do for my test cases and suites that stay the same each time. For example, I always add a main method to each test case and the suite, and these main methods do pretty much the same things every time, the only difference being the class they're operating on.

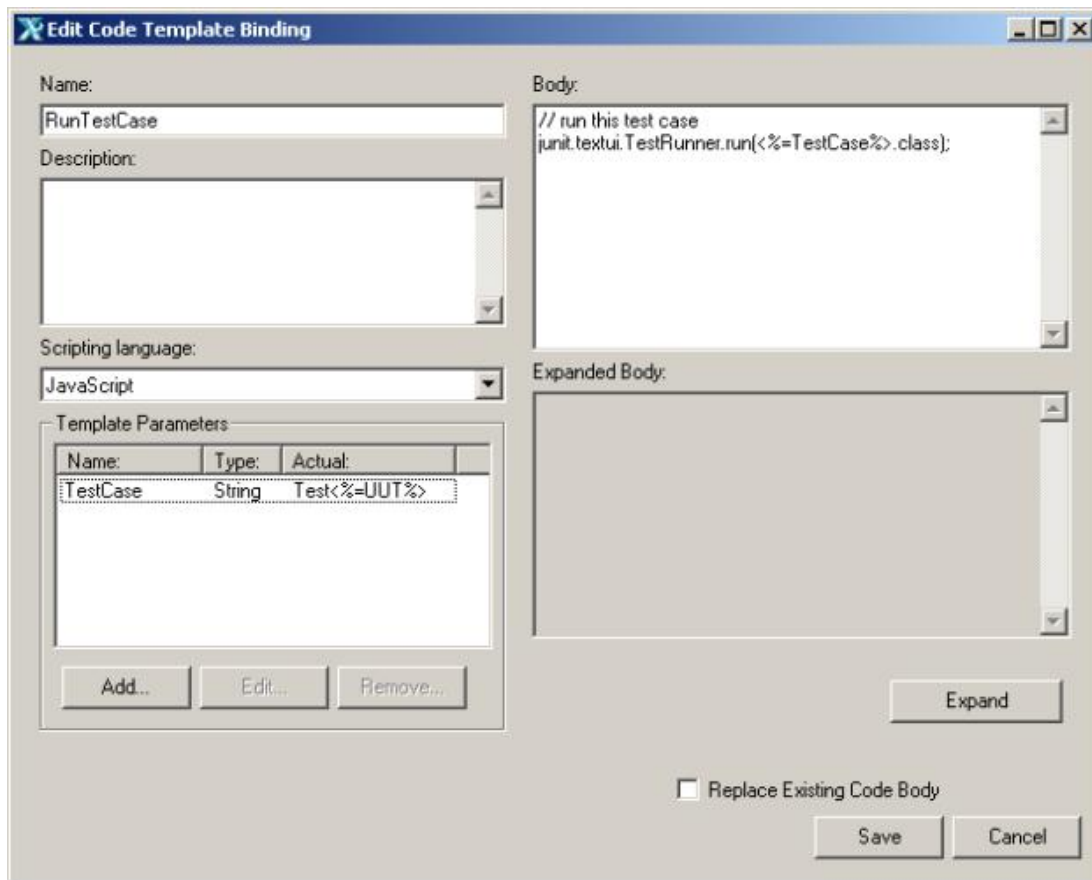
XDE provides me with a way to automate this kind of behavioral code

through the use of code templates. With a code template, I can bind a piece of Java code to an operation on a class. When I apply the pattern, the Java code is squirted into the implementation body for that class. What's even better is that I can use scriptlets in these code templates, and I can specify parameters to pass to the scriptlet when it is expanded.

For example, for my TestCase, I add a main method, and then right click on that method in the model explorer and select

### Code Templates | Bind

from the menu to bind a new code template to the method. The code template editor appears, and I can specify any existing model elements to use as parameters to the code template expansion. In this case, I want the code template to run the TestCase, so I specify the TestCase as a parameter to the code template. This can be seen in the Code Template Editor in Figure 6. I specify that a parameter named "TestCase" will be passed to the code template expansion, and that its value is expanded from the scriptlet `Test<%=UUT%>`, which is the name of the new TestCase the pattern creates.



**Figure 6: The Code Template Editor**

Inside the code template body, I use scriptlets to access these parameters. So the code template I bind to the main method of my TestCase looks like this:

```
junit.textui.TestRunner.run(<%=TestCase%>.class);
```

When the pattern is applied to the `Adder` class, this will expand to:

```
junit.textui.TestRunner.run(TestAdder.class);
```

which is exactly what I want.

When I bind a code template to an operation, XDE gives me the option of specifying if it is a one-time expansion or not. If I select one-time expansion, then the code is generated, and I can modify it at will. If I don't select one-time expansion, the code will be generated with a couple of marker comments around it. Then, any code inside those marker comments is overwritten by XDE whenever the code and model are synchronized.

In my pattern, I've made use of both ways of binding templates. For my main method, I'm not anticipating any customizations to the code, so I haven't made one-time expansions. Any changes made inadvertently will be cleaned up next time I synchronize.

I've also automated the addition of the `TestCases` to the `TestSuite`. First, I define a method called `addAllTests` in the `TestSuite` and bind a code template to the constructor of the `TestSuite` that calls this method. Then, I bind the following piece of code to the `addAllTests` method:

```
this.addTest(new TestSuite(<%=TestCase%>.class));
```

As you can see in Figure 6, I've set the parameter `TestCase` to be `Test<%=UUT%>` so it expands to the name of the `TestCase` class that's just been generated, and binds the resulting piece of code to the `addAllTests` method. I'm anticipating that this method might be customized, so I've bound the code template as a one-time expansion.

Just to provide the mandatory scope creep, I'll also automate the instantiation of the UUT. In my usage scenario above, I explicitly create a new `Adder` just before I test it. With my pattern, I add a method to do that for me; JUnit allows you to provide a method on your test case named `setUp`, which it will execute before it calls each `testy` method. I've added the method and bound the following code template to it:

```
mUut = new <%=UUT%>();
```

This expands to calling the default constructor for the UUT and storing the object in a data member named `mUut`.

JUnit also allows you to provide a `tearDown` method, which it calls immediately after calling each `testy` method. To this method, I've bound the following code template:

```
mUut = null;
```

To support these two method bodies, I change the dependency between the test case and the UUT into a private aggregation, with the role name `mUut`. XDE generates a field reference for this. I can use the generated `setUp` and `tearDown` methods as starting points or leave them as is.

The last thing I'll add is a default test operation named *testDoSomething*, and bind a code template to it that simply throws a failure back to the JUnit framework when executed. This is useful, because it doesn't mislead me into thinking I've got a test for something when I haven't. When I implement the test, I can replace the code that throws the failure with a valid test.

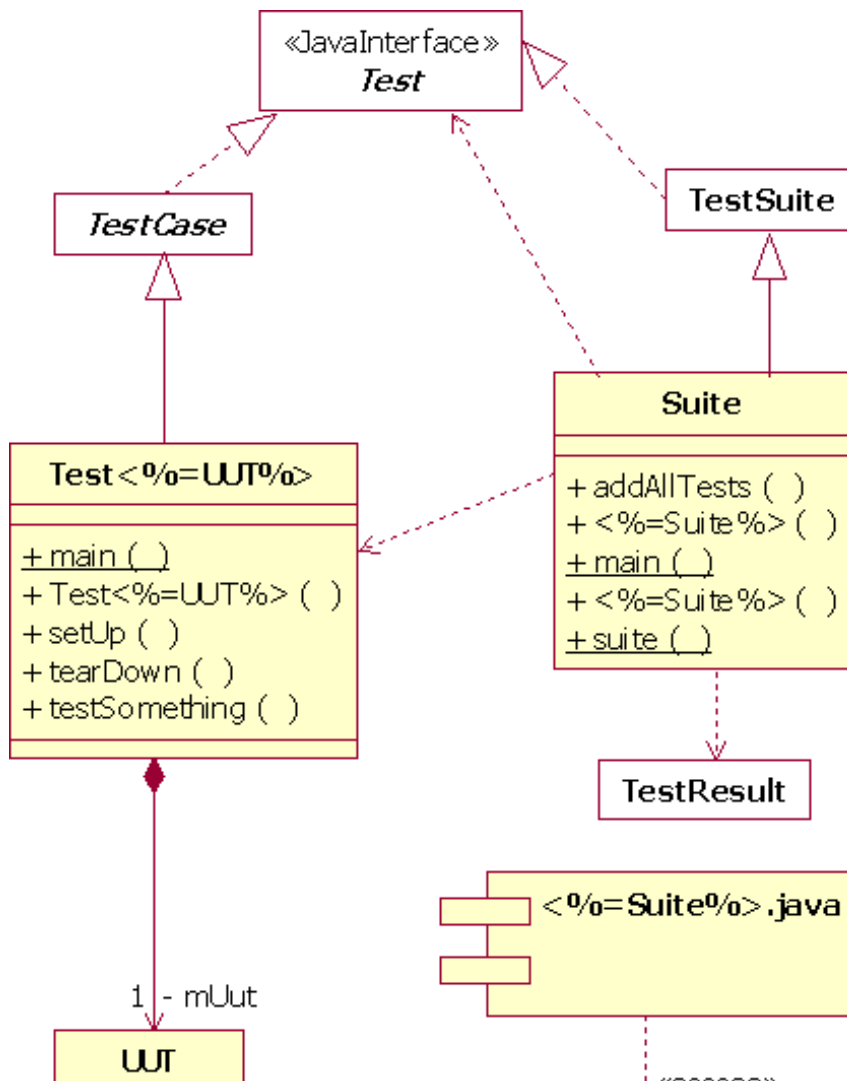
## Merging the Pattern

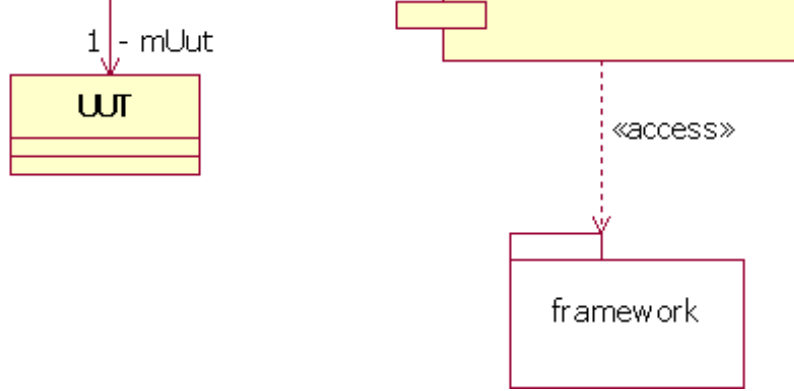
So now we have a pattern that creates both the structure and behavior of a test harness. We have satisfied the first goal of the project: to generate all housekeeping code automatically. Unfortunately, the second goal requires a bit more work.

If the pattern is expanded at this point, the resulting code won't compile. The generated TestSuite is missing `import` statements for the JUnit framework classes it requires; *Test* and *TestResult*. I could add dependencies to both of these classes, but I really want the TestSuite to have access to all classes in the `junit.framework` package, as other classes may be needed when I start coding inside the TestSuite. I really want the statement

```
import junit.framework.*
```

in the TestSuite class. To do this I create an `<<access>>` dependency between the Java component the test suite resides in and the `junit.framework` package. The resulting class diagram is shown in Figure 7.





**Figure 7: The Final Structural Pattern**

Now the generated code will compile and run, although the test case will fail with a "Not implemented" message. But as soon as the pattern is applied to a second class using the same TestSuite, the first TestCase is no longer executed. What's happened? The *addAllTests* method of the TestSuite has stopped working. It now only adds the last TestCase created to the TestSuite. The second application of the pattern has overwritten the existing method definition with its own definition. What I really want is for the body of the second code template to be appended, so that both TestCases are added. I do this, by delving into the merge options for the pattern again.

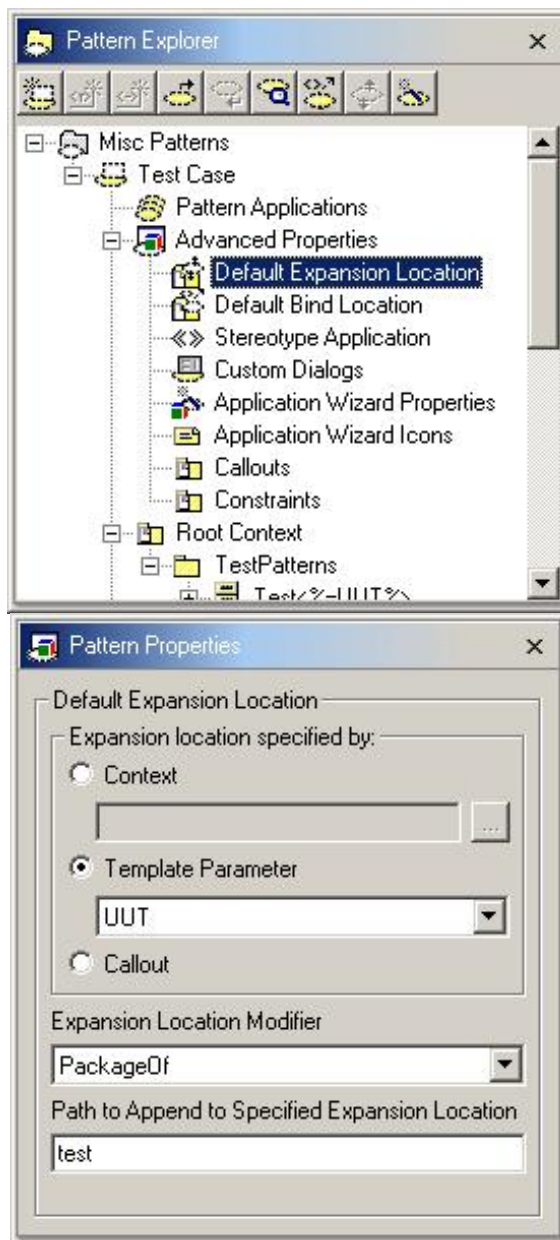
By default, the merge value for the *addAllTests* operation is set to "Replace," so if there is an existing element in the target model with the same name, it is overwritten by the element from the pattern application. This is why the body of *addAllTests* gets replaced when the pattern is applied to a suite the second time. By changing the merge value to "Merge," the code templates bound to *addAllTests* will actually be merged into one method body, so all test cases will be added when the method is called.

To help organize my model, I want to put test cases and suites into a package separate from the real application code. It is easiest to place them into a subpackage of the package where the UUT resides. I'll name this subpackage *test*, so I'll know what is in there.

To do this, I need to open the pattern in the Pattern Explorer again, and go to the

### **Advanced Properties | Default Expansion Location**

property. I change the property to expand to the package of the template parameter *UUT*, and append "test" to that, as shown in Figure 8.



**Figure 8: Setting the Default Expansion Location for the Pattern**

The last tweak I'll make is to give the user the option of either selecting an existing TestSuite class from the model or providing a name to generate a new one. To do this, I need to set the pattern properties for the Suite template parameter. So in the Pattern Explorer, I browse to the

**Suite | Advanced Properties | Value Sources**

property and change the value source. The possible values for this are summarized in Table 2. The value I want to set it to is "User or Generated," which gives me the option of selecting an existing model element or entering the name for a new TestSuite.

**Table 2: Value Sources for Pattern Template Parameters**

Value	Meaning
User	Parameter must be selected from model.
Generated	Parameter must be typed as a string.
Collection	Parameter must be a collection of an element selected from model.
User or Generated	Parameter may either be selected from the model or typed in.
User or Collection	Parameter may either be an element selected from model or a collection selected from model.
Generated or Collection	Parameter may either be typed in or a collection of an element selected from model.
Any	Parameter can be entered in any one of the three ways.

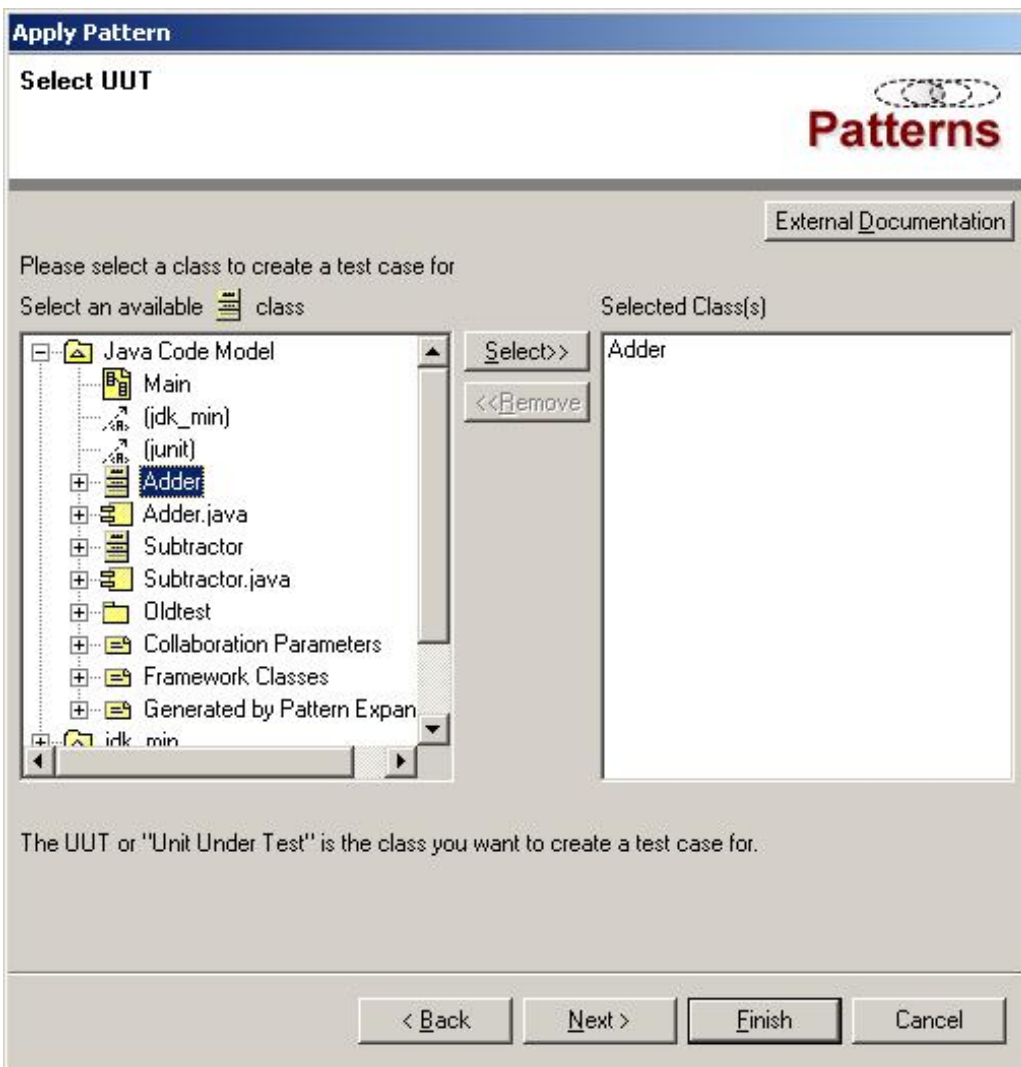
By default, XDE would still prompt me for the expansion location and default it to the value I have specified here. To make my life easier, I've also specified that the prompt for this value and the bindings location should be suppressed. To do this, I browse to the

### **Advanced Properties | Application Wizard Properties**

node in the Pattern explorer and check both the **Suppress Bind Location Dialog** and **Suppress Expand Location in Dialog** checkboxes.

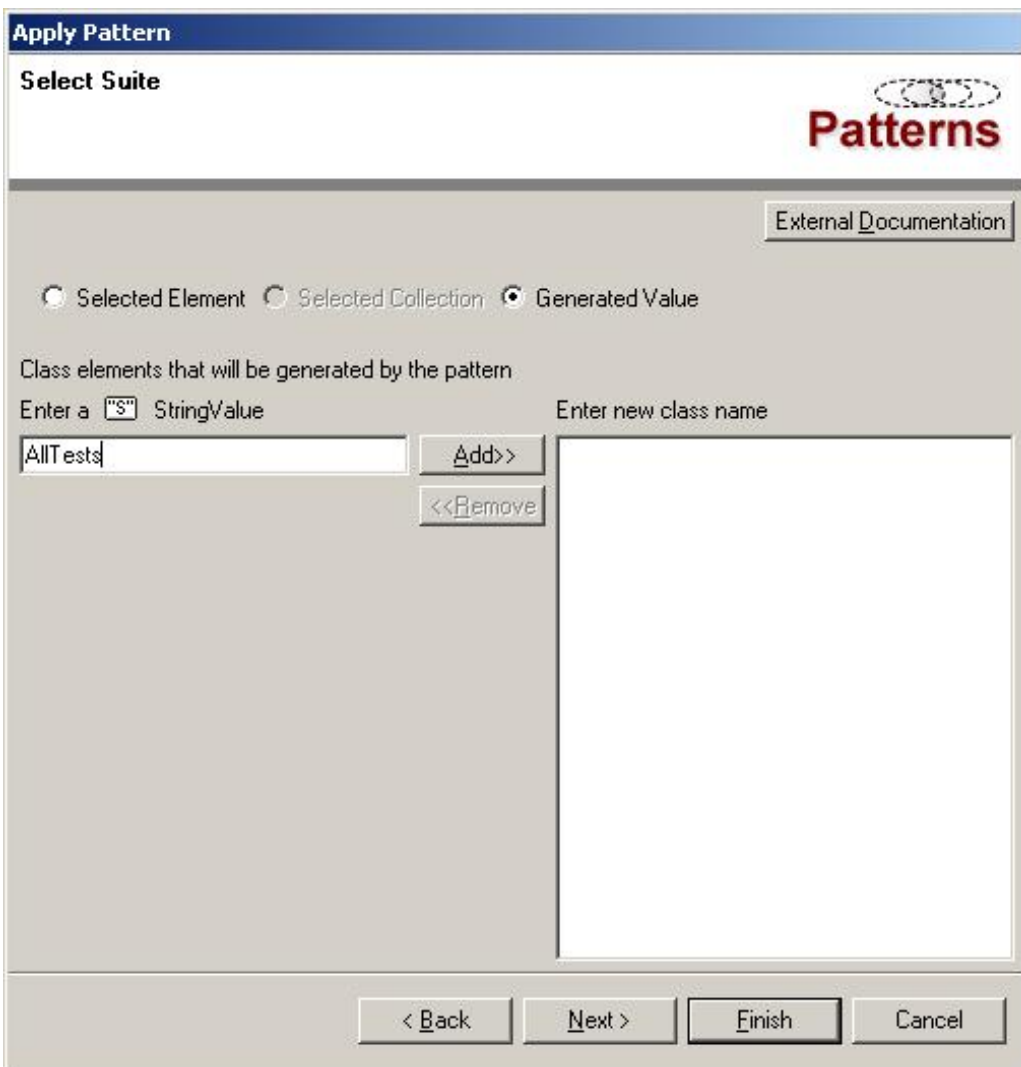
## **The Final Product**

Finally, the acid test. The first time I apply the pattern, I am presented with the dialog shown in Figure 9.



**Figure 9: Selecting a Unit Under Test (UUT)**

I select the class I want to test as the UUT, and click next. But when prompted for the suite, there is no existing class in my model that is a TestSuite, so I select the generated value option, enter the name of my new TestSuite, and click the **Add** button, as shown in Figure 10.



**Figure 10: Specifying a New Test Suite**

My pattern generates a TestCase called *TestAdder*, and a TestSuite called *AllTests*. When I compile and run *AllTests*, I get one failure because the test *testDoSomething* has not being implemented yet. So far so good.

When I generate a TestCase for *Subtractor*, I select the TestSuite I've just created. When I compile and run the TestSuite *AllTests*, two tests are executed. Success! Now that XDE has done the donkeywork, I can go and add some real test methods to my two new TestCases.

## Next Steps

My pattern deals with only one simple but common use of the JUnit framework. There are other common uses that could be implemented fairly easily -- for example, creating a TestCase as a nested class of the UUT. Also, there are many places this particular pattern could go. Selecting a package and generating TestCases for all the classes inside is one suggested enhancement. Generating a default test method for each public method of the UUT is another.

All of these options are outside the scope of my project, however, so they will have to wait for another article. Meanwhile, my project did create a small but

useful pattern and afforded a good dig around the XDE pattern engine in the process. Keep in mind that the pattern engine is extremely powerful, and I've really only scratched the surface of its potential. I hope this article will serve as an introduction to what is possible and set your creative juices flowing.

## References

To get more information about JUnit and download the latest version of the framework, visit <http://www.junit.org>.

For information on eXtreme programming, visit <http://www.xprogramming.org> or <http://www.xp.co.nz>.

## Acknowledgments

My deepest thanks to Yves Holvoet for his help in producing both this article and the XDE model it is based on.



---

***For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!***