

JUnit - Extensions

Following are the JUnit extensions –

- Cactus
- JWebUnit
- XMLUnit
- MockObject

Cactus

Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters). The intent of Cactus is to lower the cost of writing tests for server-side code. It uses JUnit and extends it. Cactus implements an in-container strategy that executes the tests inside a container.

Cactus ecosystem is made of several components –

Cactus Framework is the heart of Cactus. It is the engine that provides the API to write Cactus tests.

Cactus Integration Modules are front-ends and frameworks that provide easy ways of using the Cactus Framework (Ant scripts, Eclipse plugin, and Maven plugin).

The following code demonstrates how Cactus can be used.

```
import org.apache.cactus.*;
import junit.framework.*;

public class TestSampleServlet extends ServletTestCase {
    @Test
    public void testServlet() {
        // Initialize class to test
        SampleServlet servlet = new SampleServlet();

        // Set a variable in session as the doSomething()
        // method that we are testing
        session.setAttribute("name", "value");
    }
}
```

```
// Call the method to test, passing an  
// HttpServletRequest object (for example)  
String result = servlet.doSomething(request);  
  
// Perform verification that test was successful  
assertEquals("something", result);  
assertEquals("otherValue", session.getAttribute("otherName"));  
}  
}
```

JWebUnit

JWebUnit is a Java-based testing framework for web applications. It wraps existing testing frameworks such as HtmlUnit and Selenium with a unified, simple testing interface to test the correctness of your web applications.

JWebUnit provides a high-level Java API for navigating a web application combined with a set of assertions to verify the application's correctness. This includes navigation via links, form entry and submission, validation of table contents, and other typical business web application features.

The simple navigation methods and ready-to-use assertions allow for more rapid test creation than using only JUnit or HtmlUnit. And if you want to switch from HtmlUnit to other plugins such as Selenium (available soon), there is no need to rewrite your tests.

Here is a sample code.

```
import junit.framework.TestCase;  
import net.sourceforge.jwebunit.WebTester;  
  
public class ExampleWebTestCase extends TestCase {  
    private WebTester tester;  
  
    public ExampleWebTestCase(String name) {  
        super(name);  
        tester = new WebTester();  
    }  
  
    //set base url  
    public void setUp() throws Exception {  
        getTestContext().setBaseUrl("http://myserver:8080/myapp");  
    }  
  
    // test base info
```

```

@Test
public void testInfoPage() {
    beginAt("/info.html");
}
}

```

Explore our **latest online courses** and learn new skills at your own pace. Enroll and become a certified expert to boost your career.

XMLUnit

XMLUnit provides a single JUnit extension class, XMLTestCase, and a set of supporting classes that allow assertions to be made about –

- The differences between two pieces of XML (via Diff and DetailedDiff classes).

- The validity of a piece of XML (via Validator class).

- The outcome of transforming a piece of XML using XSLT (via Transform class).

- The evaluation of an XPath expression on a piece of XML (via classes implementing the XpathEngine interface).

- Individual nodes in a piece of XML that are exposed by DOM Traversal (via NodeTest class).

Let us assume we have two pieces of XML that we wish to compare and assert that they are equal. We could write a simple test class like this –

```

import org.custommonkey.xmlunit.XMLTestCase;

public class MyXMLTestCase extends XMLTestCase {

    // this test method compare two pieces of the XML
    @Test
    public void testForXMLEquality() throws Exception {
        String myControlXML = "<msg><uuid>0x00435A8C</uuid></msg>";
        String myTestXML = "<msg><localId>2376</localId></msg>";
        assertEquals("Comparing test xml to control xml", myControlXML, myTestXML)
    }
}

```

MockObject

In a unit test, mock objects can simulate the behavior of complex, real (non-mock)

objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

The common coding style for testing with mock objects is to –

- Create instances of mock objects.

- Set state and expectations in the mock objects.

- Invoke domain code with mock objects as parameters.

- Verify consistency in the mock objects.

Given below is an example of MockObject using Jmock.

```
import org.jmock.Mockery;
import org.jmock.Expectations;

class PubTest extends TestCase {
    Mockery context = new Mockery();
    public void testSubReceivesMessage() {
        // set up
        final Sub sub = context.mock(Sub.class);

        Pub pub = new Pub();
        pub.add(sub);

        final String message = "message";

        // expectations
        context.checking(new Expectations() {
            oneOf (sub).receive(message);
        });

        // execute
        pub.publish(message);

        // verify
        context.assertIsSatisfied();
    }
}
```