Using ABLUnit for Testing

ABLUnit helps you to orgainize and manage your test code, and automate testing in all parts of a distributed application. In this video, we focus on testing a Business Entity class that runs in an application server.

The first step is to create an ABLUnit project in Developer Studio. We create a new OpenEdge project.

We name the project and then we specify that the project type is ABLUnit.

We leave the defaults for the top-level folders.

Because we are testing server-side code, we must ensure that this project has the required folders in its PROPATH. We select the Common Infrastructure and Server folders.

We accept the defaults for the rest of the project.

When we click Finish, we are informed that a new OpenEdge ABLUnit perspective will be opened.

We check the **remember my decision** box and click **Yes** to open this new perspective.

The OpenEdge ABLUnit perspective looks very similar to the OpenEdge Editor perspective, with the exception of the ABLUnit view where you monitor test runs.

In your ABLUnit project, you create a set of test classes to test various parts of your application. You organize your test classes under the **tests** folder that is created for you by the wizard.

Here we add a sub-folder named **BusinessEntity** that will contain the test class we will use to test our Business Entity.

Our CustomerBE class contains methods for retrieving and updating data. We will create a client test class for testing this Business Entity's public methods, GetData, GetOrderData, and UpdateData.

In the BusinessEntity folder, we create a test class by selecting the New ABLUnit Test Class wizard.

We name the test class testCustomerBE.

We then select which stubs we want the wizard to create for us. The Before and After stubs are methods that are executed before and after every test method is called. We do not need this capability. But we want our test client to connect to the application server so it can call the Business Entity. We will use the Setup method to connect and the TearDown method to disconnect.

Next, we select the class that we will test, CustomerBE.

Finally, we select the methods that our test class will test. We select all of the public methods for this Business Entity class.

When the wizard finishes, our new test class opens.

This class contains method stubs for the setup and teardown methods as well as test methods for each of the public methods we specified.

Notice that the test methods are created with no input or output parameters and are preceded by the @Test annotation. You can create additional methods to ensure that all test cases are covered, but they cannot have input or output parameters and must be preceded by the @Test annotation.

To call a Business Entity from a client, the client must define and initialize data members for the server connection and for the service interface procedures. In addition, the client must define a dataset that matches the dataset used by the Business Entity.

The first method we implement is the setUp() method. Here, we connect to the application server. In this case, we provide a connection string to connect to the classic AppServer. Then we load the procedure file for the service interfaces that are used to access the business entity.

Next, we implement the tearDown() method which disconnects from the application server and deletes the server and procedure objects.

Now, we implement code to test the GetData() method. For this test, we first empty the static dataset to ensure it has no data in it before we attempt to retrieve data.

We add code to call the service interface procedure for the GetData() method. Because we do not specify a filter, the call will retrieve all of the customer data.

Then we add code to open an output file and write to it. Logging test data is useful for documenting test results.

And finally, we close the output file.

We also want to include the test case where we retrieve some of the data so we create a new test method called testGetSomeData.We add the code that will filter the data to retrieve some of the data.

We save our test class and we are now ready to test. Before we test the business entity, we must start the application server.

With our test class open, we click the run icon and select that we want to run it as a Progress ABLUnit Application.

Notice that the test run for each method is displayed in the ABLUnit Test view. If errors occur, then you must determine whether the error is in your test code, or in the business entity you are testing.

In this case, the test class could not open the output file because the testlog directory does not exist. This is an error in the test code.

We add the testlog directory and we rerun one of the test methods. The test is successful.

Finally we rerun all of the tests and we see that they all ran to completion.

We examine the log files to ensure that the test results were as expected.
You have now seen how to set up your Developer Studio environment for testing an application using the ABLUnit test framework.

To learn about developing ABL applications, take the course, Developing a Progress OpenEdge ABL Application.