Introducing PSUnit

An xUnit TDD Framework for PeopleSoft

Test Driven Development (TDD) is a software design technique that uses incremental, iterative unit tests to *drive* software design. TDD stipulates writing unit tests based on functional requirements before coding units. As unit tests grow in number, organizing, running, and maintaining tests becomes difficult. Writing tests, without testing standards, further exacerbates this test maintenance issue.

Testing frameworks help solve these maintenance problems by providing automation, organization, and testing standards. Many programming languages have testing frameworks that facilitate TDD, the xUnit testing frameworks being the most popular. Wikipedia maintains a list of Unit Testing Frameworks at http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Theoretically, we can drive PeopleCode development without a testing framework, but to follow TDD techniques, a framework certainly helps. With this document, I am pleased to introduce a unit testing framework for PeopleCode: PSUnit. PSUnit comes without warranty or support of any kind. PSUnit is nothing more than a collection of managed PeopleTools objects (pages, components, PeopleCode, etc) that facilitate testing and the development of test suites.

Before we dive into a test-driven example, let's quickly summarize the basics of TDD…

TDD steps:

1. Write a requirements based test
2. Run the test and watch it fail
3. Write the shortest, simplest solution to pass the test
4. Run the test and watch it pass
5. Refactor the test

Why test first?

- Ensure your designed units are testable
- Ensure you understand the requirements (see tire swing diagram)

If you are interested in learning more about TDD concepts, then take a look at Wikipedia's Test-driven development page. It explains the basics of TDD and serves as a portal to many other TDD resources.

The example that follows requires PSUnit. To install PSUnit, download PSUnit, extract the zip file contents, import the project, and build the project's records. The version downloadable from the PeopleTools blog was built using PeopleTools 8.49, but should run on newer versions of PeopleTools. The project includes the role TTS_UNIT_TEST. Once you add this role to your user profile, you will gain access to the online run component for PSUnit.

The best way to describe PSUnit is to test drive it with a TDD example. In this example, we will create a new tool to help debug applications. Here is the scenario:

> You, the developer, receive a notification from a user that page X of component COMP_X is calculating the wrong values. The user informs you that the calculation and error occur when he/she clicks save. From this information, you speculate that the calculation happens in the SavePreChange or SavePostChange event of the component or some record used in the component. Unfortunately, you are not familiar with this page or component.
>
> With a PeopleCode trace, you are able to identify six potential events. You notice that these events call FUNCLIB's and App Classes, creating a horrendously deep call stack. From what you have in front of you, it is obvious that a quick review of this 3,000+ line trace file won't provide an easy solution.

At this point you have several options:

- Continue to treat COMP_X as a black box, investigating it from the outside.
- Dig into the code and speculate as to its purpose.
- Configure app server debugging and step through the deep call stack.
- Start adding `MessageBox` statements to the delivered code so you can interrogate the state of the application as it runs.

We will choose the final option, the `MessageBox` option. Yes, this will require us to modify delivered code, but this modification should have no impact on the behavior of the code. The modification doesn't concern me as much as forgetting to delete one of those `MessageBox` statements after I find and fix the problem. And then, once I find the correct combination of `MessageBox` statements to show the problematic data or logic, I hate to delete them, knowing I may have to visit this code at a future date (sooner then I want, but far enough in the future that I've already forgotten how I solved the problem). Wouldn't it be nice if, once you found the appropriate combination of logging statements, you could just leave them in the code? Let's use TDD to see if we can develop this configurable debugging tool.

What I want is a logging facility that allows me to write statements like:

```
&logger.debug("Program made it to line 30");
&logger.debug("Number of rows in rowset: " | &rs);
```

And, I want to leave those statements in my code, turning them on and off as needed. I don't want to waste the effort I invested in writing targeted debugging statements, assuming I will need them at another time.

At this point, we know *what* we want to accomplish, but not *how*. TDD suggests that if we start with what we know, then what we don't know will become evident.

```
method testLogger()
```

```
   Local Logger &logger = create Logger();
   &logger.debug("Program made it to line 30");
   &logger.debug("Number of rows in rowset: " | &rs);
end-method;
```

From this test case, we see that we need a Logger object that implements a debug method.

With this information, let's create a stub for the Logger App Class.

```
class Logger
   method debug(&message as string);
end-class;

method debug
end-method;
```

At this point, our test case appears to test two things:

- Object creation/construction
- debug method execution

Now we need a way to determine if the test passes. We can skip the construction test because our method test will fail if construction fails.

Our original intent was to display a message using the `MessageBox` function. Looking at the debug method, we could satisfy this requirement with one line:

```
method debug
   /+ &message as String +/
   MessageBox(0, "", 0, 0, &message);
end-debug;
```

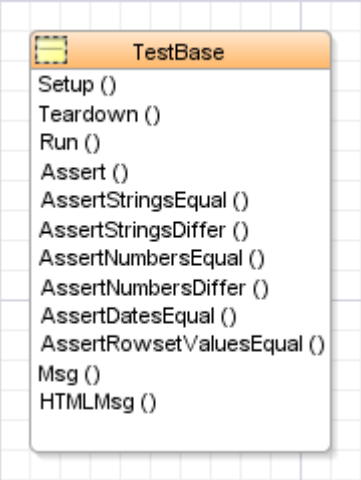
Following the guidelines of TDD, however, this approach is not testable. Even if it were testable, the behavior of this debug method would require us to test the `MessageBox` function, a delivered function that is outside the scope of our project.

When we implement the on/off switch for our logging facility, we will want a way to test the logic of the `debug` method without testing the `MessageBox` statement. As an alternative, let's delegate the debug printing logic to another class. This level of abstraction will allow us to extend our logging facility to log to files, databases, sockets, etc. Most importantly, this level of abstraction will allow us to test the `debug` method without testing the `MessageBox` statement.

With this new design in mind, let's set aside our original `testLogger` test case to develop a log target we can use to test the `debug` method. As always, we will start with step 1: develop a test. This time, however, we will develop our test case using the PSUnit test framework.

We define PSUnit test cases as Application Classes that extend the base class `TTS_UNITTEST:TestBase` (see Figure 1). `TestBase` provides the standard xUnit `Setup`, `Teardown`, and assertion methods as well as a single `Run` method. Setup code goes in the

Figure 1
TestBase Class Diagram



`Setup()` method, cleanup code goes in the `Teardown()` method, and test code goes in the `Run()` method. Most of the assertion methods are self explanatory. The `Assert` method requires a `boolean` test parameter and a `string` message parameter. If the `boolean` parameter is false, then the `Assert` method throws an exception, giving the message parameter as the Exception's text. The rest of the assertion methods are similar except these other methods compare the first two parameters for equality.

One way to verify the Logger's debug logic is to buffer printed log statements in a string and then compare the buffer contents to the strings we printed. Let's write a test for this logging target:

```
method Run
   Local StringBufferTarget &target =
         create StringBufferTarget();
   &target.print("Hello World");
   %This.AssertStringsEqual(&target.getBuffer(), "Hello World",
         "method failed");
end-method;
```

This test creates an instance of the `StringBufferTarget` class, calls the print method, and then compares the contents of the class's string buffer to the printed data.

Listing 1 contains the full `TestStringBufferTarget` class test case. Add this code to a new package named ADS_LOG_TEST.

Listing 1 TestStringBufferTarget

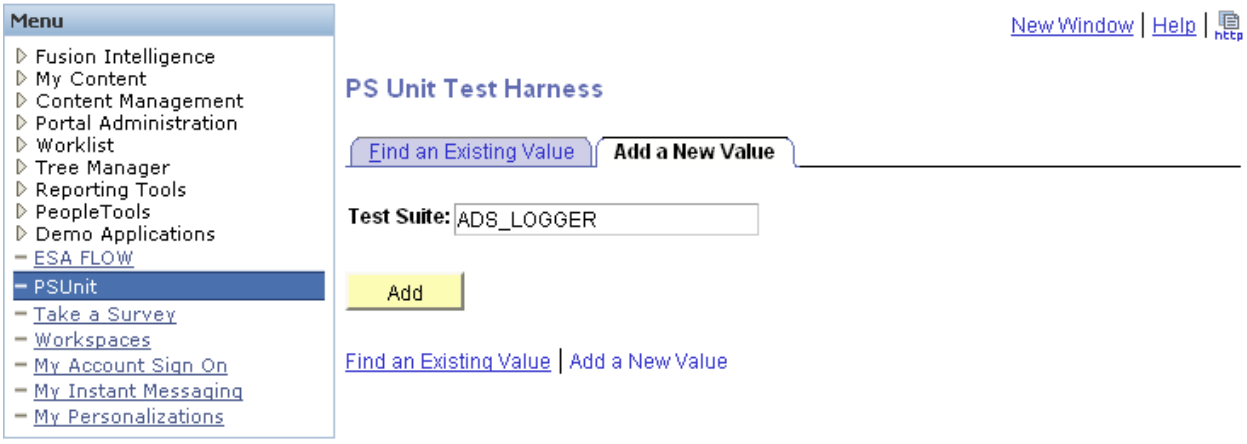
```
import TTS_UNITTEST:TestBase;
import ADS_LOGGER:StringBufferTarget;

class TestStringBufferTarget extends TTS_UNITTEST:TestBase
   method TestStringBufferTarget();
   method Run();
end-class;

method TestStringBufferTarget
   %Super = create TTS_UNITTEST:TestBase("TestStringBufferTarget");
end-method;

method Run
   /+ Extends/implements TTS_UNITTEST:TestBase.Run +/
   Local ADS_LOGGER:StringBufferTarget &target =
         create ADS_LOGGER:StringBufferTarget();
   &target.print("Hello World");
   %This.AssertStringsEqual(&target.getBuffer(), "Hello World",
         "method failed");
end-method;
```

Before saving this test code, we need to create a stub for the `StringBufferTarget` class. Create a new Application Package named ADS_LOGGER and a new class named StringBufferTarget:

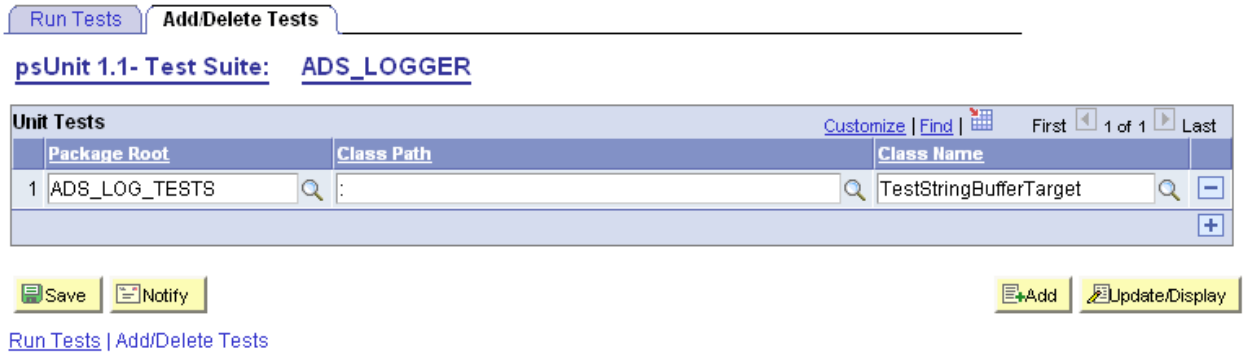
```
class StringBufferTarget
   method print(&message as string);
   method getBuffer() returns string;
end-class;

method print
end-method;

method getBuffer
   return "";
end-method;
```

Our code is ready for step 2: Run the test and watch it fail. The test prints “Hello World” to the log target, but, as you can see from the code above, our log target’s getBuffer() method returns an empty string. When this code runs, the assertion will fail because “Hello World” <> “”. In step 2, we run tests we know will fail to prove they fail. If we write a test that passes before we implement behavior, then we know the test is flawed. This first run of our test case provides minimal validation of the test.

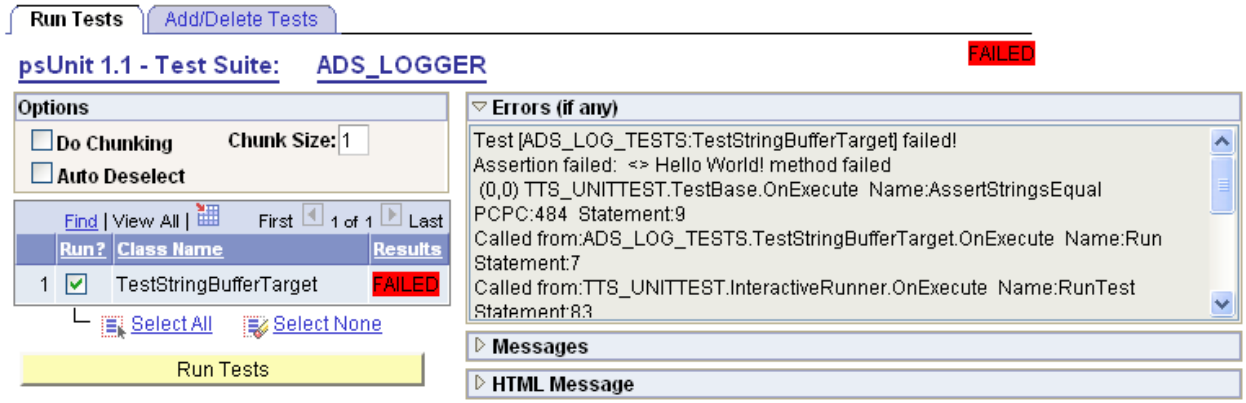
To run this test using the PSUnit test framework, launch your browser, connect to your PeopleSoft instance, navigate to PSUnit, and add a new value.



When the psUnit 1.1 test suite opens in your browser, switch to the Add/Delete Tests tab. Before executing this test suite, we need to add the new `TestStringBufferTarget` test class.



Switch back to the Run Tests tab, click Select All, and then click Run Tests. You should see a red square indicating the test failed.



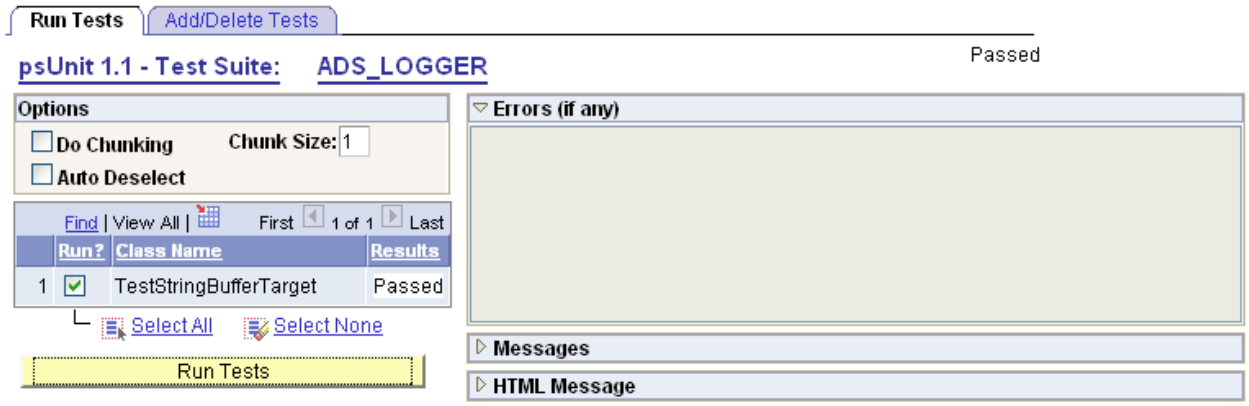
Moving on to step 3, our goal is to write as little code as possible to make our test pass. It doesn't matter how we make this test pass. Like step 2, step 3 is more concerned with validating test logic than testing implementation logic. We will clean up the code in step 5.

Considering the simplicity of the `StringBufferTarget`, I am sure you already know the code you would write to implement the `print` and `getBuffer` methods. At this stage, resist the temptation to write the obvious implementation. Remember, we are trying to validate the test, not the implementation. My first cut at implementing this class follows:

```
class StringBufferTarget
   method print(&message As string);
   method getBuffer() Returns string;
end-class;

method print
   /+ &message as String +/
end-method;

method getBuffer
   /+ Returns String +/
   Return "Hello World";
end-method;
```

Notice that I hard coded the result. Again, this is to validate the test case, not the implementation. We will refactor this code in step 5.

Step 4: Run the test again and watch as the indicator flips from red to green.



Well, actually, as you see from the screen print above, psUnit 1.1 result ('Passed') doesn't flip to green upon success. It flips to an anti-climatic white. As an experienced PeopleTools programmer, I am sure you know how to fix this (if not, check the code that flips it red).

On to step 5… it is time to refactor this code to eliminate redundancy, remove magic numbers, etc. For example, the string "Hello World" exists in our test code and our implementation. We can eliminate this redundancy by adding a private instance variable to store strings written to the `print` method and then display those strings with the `getBuffer()` method.

```
class StringBufferTarget
   method print(&message As string);
   method getBuffer() Returns string;

private
   instance string &buffer_;
end-class;

method print
   /+ &message as String +/
   &buffer_ = &buffer_ | &message;
end-method;

method getBuffer
   /+ Returns String +/
   Return &buffer_;
end-method;
```

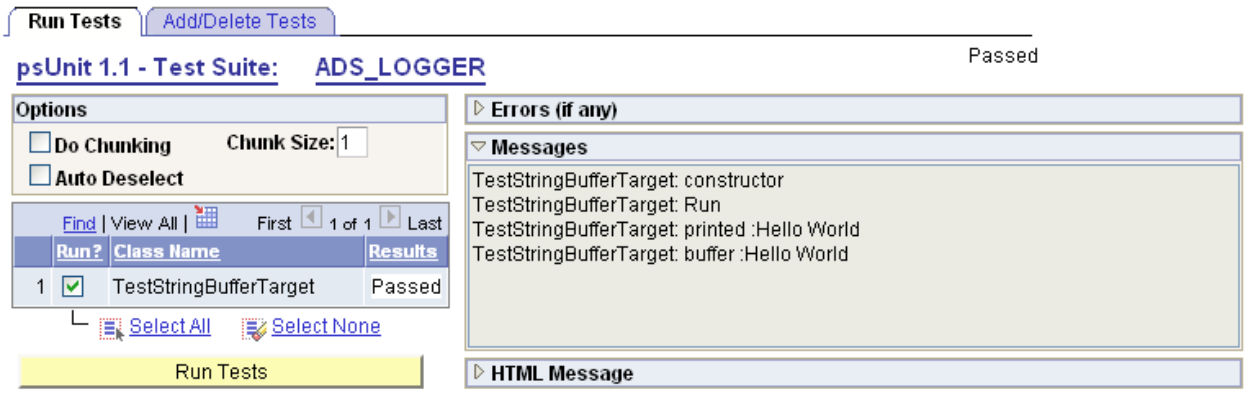
For the sake of brevity, I made 3 changes at once. Pure TDD suggests making one change at a time, testing between each change. If refactoring causes a test to fail, single step modifications are the easiest way to identify the failure's cause.

Testing our code one more time shows these changes were successful. Since we are running this code interactively, it would be nice to see some output describing the test's accomplishments. Let's use the `Msg` method of `TestBase` to print comments to the PSUnit test page. Listing 2 contains the code for this modified test case. Notice that I eliminated the "Hello World" redundancy as well.

Listing 2 TestStringBufferTarget, version 2

```
import TTS_UNITTEST:TestBase;
import ADS_LOGGER:StringBufferTarget;

class TestStringBufferTarget extends TTS_UNITTEST:TestBase
   method TestStringBufferTarget();
   method Run();
end-class;

method TestStringBufferTarget
   %Super = create TTS_UNITTEST:TestBase("TestStringBufferTarget");
   %This.Msg("TestStringBufferTarget: constructor");
end-method;

method Run
   /+ Extends/implements TTS_UNITTEST:TestBase.Run +/
   %This.Msg("TestStringBufferTarget: Run");

   Local ADS_LOGGER:StringBufferTarget &target =
         create ADS_LOGGER:StringBufferTarget();
   Local string &message = "Hello World";
   Local string &buffer;

   &target.print(&message);
   &buffer = &target.getBuffer();

   %This.Msg("TestStringBufferTarget: printed :" | &message);
   %This.Msg("TestStringBufferTarget: buffer :" | &buffer);

   %This.AssertStringsEqual(&buffer, &message, "method failed");
end-method;
```

Running this code again, we see that `%This.Msg(string)` prints text to the **Messages** group box.



Recapping the process thus far, we started with a logger. The logger's test identified the need for a target. Since we didn't have a target, we set the logger's test aside and used TDD steps 1 through 5 to design and implement a target. With this target complete, we can return to step 1: write a test for the logger. When we write the test this time, we will create it as a PSUnit test case in the ADS_LOG_TESTS Application Package.

Listing 3 TestLogger Test Case

```
import TTS_UNITTEST:TestBase;
import ADS_LOGGER:StringBufferTarget;
import ADS_LOGGER:Logger;

class TestLogger extends TTS_UNITTEST:TestBase
   method TestLogger();
   method Run();
end-class;

method TestLogger
   %Super = create TTS_UNITTEST:TestBase("TestLogger");
   %This.Msg("TestLogger: constructor");
end-method;

method Run
   /+ Extends/implements TTS_UNITTEST:TestBase.Run +/
   %This.Msg("TestLogger: Run");

   Local ADS_LOGGER:StringBufferTarget &target =
         create ADS_LOGGER:StringBufferTarget();
   Local ADS_LOGGER:Logger &logger =
         create ADS_LOGGER:Logger();

   Local string &message = "Hello World";
   Local string &buffer;

   &logger.setTarget(&target);

   &logger.debug(&message);
   &buffer = &target.getBuffer();

   %This.Msg("TestLogger.Run: printed: " | &message);
```

```
   %This.Msg("TestLogger.Run: buffer: " | &buffer);

   %This.AssertStringsEqual(&buffer, &message, "method failed");
end-method;
```

Listing 3 contains the code for this rewritten test case. This code further clarifies the `Logger` interface by demonstrating the need to identify a logger's target. Even though I chose to implement this behavior using the `setLogger(&target)` mutator method, constructor injection would be equally satisfactory. Let's add this new method to the `Logger` stub and make `Logger` a member of the ADS_LOGGER Application Package.

```
import ADS_LOGGER:Target;

class Logger
   method debug(&message as string);
   method setTarget(&target as Target);
end-class;

method debug
end-method;

method setTarget
end-method;
```

The `setTarget` parameter type `Target` is a placeholder for the `StringBufferTarget` we previously created. We want to strongly type the target parameter while still allowing for flexible target implementations. We can provide this strongly typed flexibility by extracting the following interface from the `StringBufferTarget` class and adding it to our ADS_LOGGER Application Package:

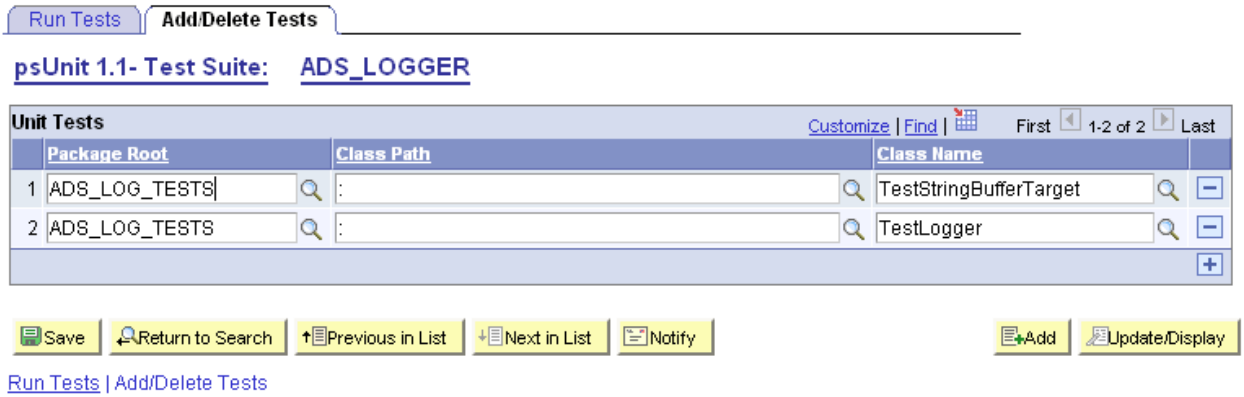
```
interface Target
   method print(&message as string);
end-interface;
```

For the `Logger` class to use the `StringBufferTarget` the `StringBufferTarget` must implement this new interface. To accomplish this, replace the first line of the `StringBufferTarget` class with the following two lines:

```
import ADS_LOGGER:Target;
class StringBufferTarget implements ADS_LOGGER:Target
```

Our Logger test is ready to run. Step 2: run the test and watch it fail.

In your web browser, add the ADS_LOG_TESTS:TestLogger test case to your PSUnit test suite and run the `TestLogger` test case.



Running this test should produce a red failure indicator. Listing 4 provides a solution for step 3: write the shortest, simplest solution to pass the test.

Listing 4 Logger Implementation

```
import ADS_LOGGER:Target;

class Logger
   method debug(&message As string);
   method setTarget(&target As ADS_LOGGER:Target);

private
   instance ADS_LOGGER:Target &target_;
end-class;

method debug
   &target_.print(&message);
end-method;

method setTarget
   &target_ = &target
end-method;
```

Step 4: Run the test again and watch it pass.

With steps 1 through 4 complete, we can refactor this `Logger`. Looking at this code, however, I don't see anything to refactor.

Our objective is to turn logging on and off without touching code. This implies some type of configuration. As TDD is an iterative process, this requirement takes us back to step 1: write a test. Rather than write a new test, giving us two tests that cover the same code, we can refactor the original test by adding coverage for this new requirement. Listing 5 contains this refactored `TestLogger` test case.

Listing 5 TestLogger Test Case that Tests Configuration

```
import TTS_UNITTEST:TestBase;
```

```
import ADS_LOGGER:StringBufferTarget;
import ADS_LOGGER:Logger;

class TestLogger extends TTS_UNITTEST:TestBase
   method TestLogger();
   method Run();

private
   method Run_OnTest();
   method Run_OffTest();

   Constant &MESSAGE = "Hello World";
end-class;

method TestLogger
   %Super = create TTS_UNITTEST:TestBase("TestLogger");
   %This.Msg("TestLogger: constructor");
end-method;

method Run
   /+ Extends/implements TTS_UNITTEST:TestBase.Run +/
   %This.Msg("TestLogger: Run");

   %This.Msg("TestLogger: Turn on logging");
   SQLExec("UPDATE PS_ADS_LOG_CONFIG SET FLAG = 'Y'");
   %This.Run_OnTest();

   %This.Msg("TestLogger: Turn off logging");
   SQLExec("UPDATE PS_ADS_LOG_CONFIG SET FLAG = 'N'");
   %This.Run_OffTest();
end-method;


method Run_OnTest
   %This.Msg("TestLogger.Run_OnTest");

   Local ADS_LOGGER:StringBufferTarget &target = create
         ADS_LOGGER:StringBufferTarget();
   Local ADS_LOGGER:Logger &logger = create ADS_LOGGER:Logger();

   &logger.setTarget(&target);
   &logger.debug(&MESSAGE);

   %This.AssertStringsEqual(&target.getBuffer(), &MESSAGE, "method failed");
end-method;

method Run_OffTest
   %This.Msg("TestLogger.Run_OffTest");
   Local ADS_LOGGER:StringBufferTarget &target = create
         ADS_LOGGER:StringBufferTarget();
   Local ADS LOGGER:Logger &logger = create ADS LOGGER:Logger();


   &logger.setTarget(&target);
   &logger.debug(&MESSAGE);
```

```
   %This.AssertStringsEqual(&target.getBuffer(), "", "method failed");
end-method;
```

Our requirements call for two tests, an *on* test and an *off* test. To facilitate this requirement, I converted the `Run` method into a controller that updates the configuration and then calls the appropriate private test method. From the `Run` method, you can see that I chose to store the on/off configuration flag in a one row table named `ADS_LOG_CONFIG` with one field named `FLAG`.

The `Run_OnTest` method is the same test from listing 3. The `Run_OffTest` method is similar to `Run_OnTest` except that the `Run_OffTest` method verifies that the log buffer contains no data. When you save and run this test (step 2), the `Run_OnTest` method will pass and the `Run_OffTest` method will fail. We can determine which private method failed by reviewing the call stack in the Errors group box. The test failed because the Logger class hasn't implemented the on/off flag.

With this test validated, we can modify the Logger class (step 3) to implement this on/off behavior. Listing 6 contains the modified Logger code.

Listing 6 Logger class that implements the on/off behavior

```
import ADS_LOGGER:Target;

class Logger
   method Logger();
   method debug(&message As string);
   method setTarget(&target As ADS_LOGGER:Target);

private
   instance ADS_LOGGER:Target &target_;
   instance boolean &isOn;
end-class;

method Logger
   Local string &flag;
   SQLExec("SELECT FLAG FROM PS_ADS_LOG_CONFIG", &flag);

   If (&flag = "Y") Then
      &isOn = True;
   End-If;
end-method;

method debug
   /+ &message as String +/
   If (&isOn) Then
      &target_.print(&message);
   End-If;
end-method;

method setTarget
   /+ &target as ADS_LOGGER:Target +/
   &target_ = &target
end-method;
```

Run the test again to see success (steps 4 and 5).

With our framework exhibiting the desired behavior, we can implement a `MessageBox` target to use for debugging.

Listing 7 MesageBoxTarget class

```
import ADS_LOGGER:Target;

class MessageBoxTarget implements ADS_LOGGER:Target
   method print(&message As string);
end-class;

method print
   /+ &message as String +/
   /+ Extends/implements ADS_LOGGER:Target.print +/
   MessageBox(0, "", 0, 0, &message);
end-method;
```

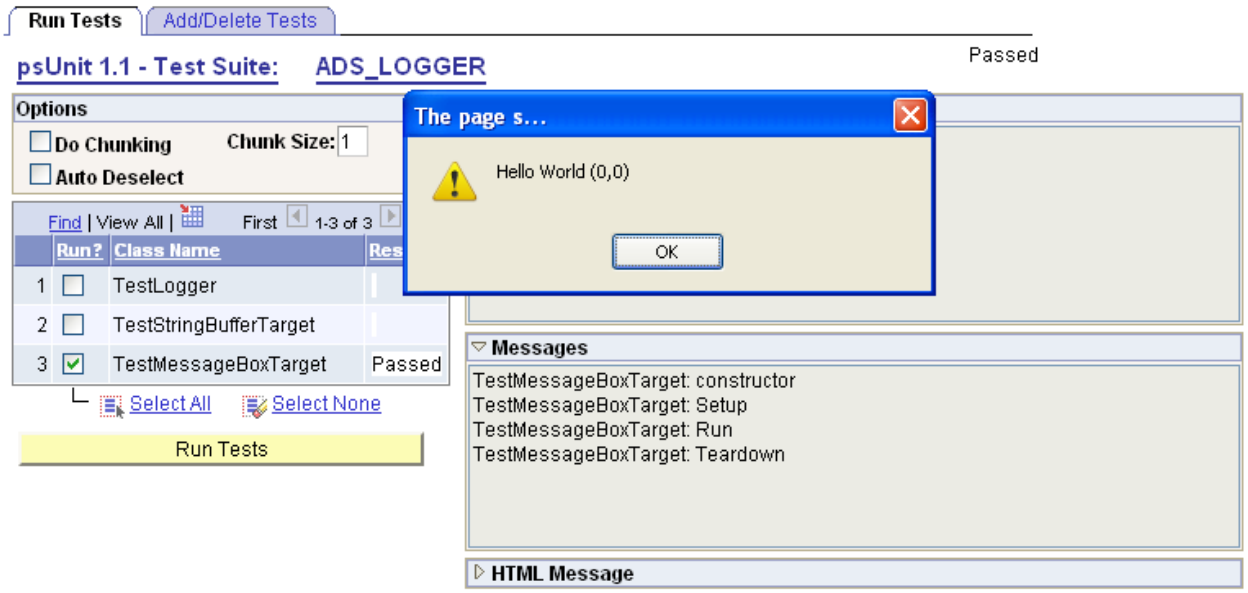
We can visually inspect the behavior of this new target with another test case. Unfortunately, we can't automate this test with Asserts (MessageBox/JavaScript alert), but we can manually verify the `MessageBox` popup. Listing 8 contains the code for this test.

Listing 8 MesageBoxTarget test case

```
import TTS_UNITTEST:TestBase;
import ADS_LOGGER:MessageBoxTarget;
import ADS_LOGGER:Logger;

class TestMessageBoxTarget extends TTS_UNITTEST:TestBase
   method TestMessageBoxTarget();
   method Setup();
   method Run();
   method Teardown();

private
   instance string &flag_;
end-class;

method TestMessageBoxTarget
   %Super = create TTS_UNITTEST:TestBase("TestMessageBoxTarget");
   %This.Msg("TestMessageBoxTarget: constructor");
end-method;

method Setup
   /+ Extends/implements TTS_UNITTEST:TestBase.Setup +/
   Local string &flag;
   %This.Msg("TestMessageBoxTarget: Setup");
   SQLExec("SELECT FLAG FROM PS_ADS_LOG_CONFIG", &flag);
   &flag_ = &flag;
```

```
end-method;


method Run
   /+ Extends/implements TTS_UNITTEST:TestBase.Run +/
   %This.Msg("TestMessageBoxTarget: Run");

   SQLExec("UPDATE PS_ADS_LOG_CONFIG SET FLAG = 'Y'");
   Local ADS_LOGGER:Logger &logger = create ADS_LOGGER:Logger();

   &logger.setTarget(create ADS_LOGGER:MessageBoxTarget());

   &logger.debug("Hello World");
end-method;


method Teardown
   /+ Extends/implements TTS_UNITTEST:TestBase.Teardown +/
   Local string &flag = &flag_;
   %This.Msg("TestMessageBoxTarget: Teardown");

   SQLExec("UPDATE PS_ADS_LOG_CONFIG SET FLAG = :1", &flag);
end-method;
```

Listing 8 introduces `Setup` and `Teardown`. The `TestMessageBoxTarget` class uses the `Setup` method to enable the framework's database log flag and then uses `Teardown` to reset the flag. For each test case in a test suite, the interactive test runner runs `Setup`, `Run`, and then `Teardown`. Use `Setup` to put data, files, configurations, etc in the state expected by the test and use `Teardown` to reset changes made by `Setup`.



The test suite presented here is fairly small. As I expand this test suite by adding tests for new log framework features, the time required to run tests may cause the on-line user interface to timeout. PSUnit provides test chunking to work around this issue. The screenshot above shows an options group box with a chunking checkbox and a chunk size text box. If you enable chunking and set the size to 3, then PSUnit will run three tests, return, and then run the next three.

The online PSUnit automated test suite runs as FieldChange PeopleCode and, therefore, is subject to all the FieldChange PeopleCode rules (CommitWork, think-time actions, etc).

PSUnit is a test framework starting point. I encourage you to enhance it as needed. For example, PSUnit does not provide an offline test runner. Running tests offline through an AppEngine program would eliminate FieldChange limitations and allow developers to schedule long running tests to run after hours.