AUnit Cookbook

AUnit - version 1.04
Document revision level $Revision: 1.2 $
Date: 22 May 2006

AdaCore

http://www.adacore.com

Table of Contents

1 Introduction

This is a short guide for using the AUnit test framework. AUnit is an adaptation of the Java JUnit (Kent Beck, Erich Gamma) unit test framework for Ada code. This document is adapted from the JUnit Cookbook document contained in the JUnit release package.

2 Simple Test Case

How do you write testing code?

The simplest way is as an expression in a debugger. You can change debug expressions without recompiling, and you can wait to decide what to write until you have seen the running objects. You can also write test expressions as statements which print to the standard output stream. Both styles of tests are limited because they require human judgment to analyze their results. Also, they don't compose nicely- you can only execute one debug expression at a time and a program with too many print statements causes the dreaded "Scroll Blindness".

AUnit tests do not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do:

1. Declare a package for a test case - a set of logically related test routines. A template for such a package is in `/AUnit/Template/pr_xxxx_xxx.ad*`. GPS provides an **Edit -> Unit Testing** menu to generate template code.
2. Derive from `AUnit.Test_Cases.Test_Case` in the new package.
3. The new derived type must provide implementations of `Register_Tests` and `Name`.
4. Write each test routine (see below) and register it with a line in routine *Register_Tests*, of the form:

   ```
   Register_Routine (T, Test_Name'Access, "Description of test routine");
   ```

5. When you want to check a value, use:

   ```
   AUnit.Assertions.Assert (Boolean_Expression, String_Description);
   ```

6. Create a suite function to gather together test cases and sub-suites. Alternatively, you can call the `Run` routine of a single test case directly, and then call `Test_Results.Test_Reporter.Report` on its `Result` parameter. This eliminates step 7.
7. At any level at which you wish to run tests, create a harness instantiating `Aunit.Test_Runner` with a suite function collecting together test cases and sub-suites to execute.
8. AUnit includes a GNAT project file that should be included into your application project to access the framework.&nbsp; For other compilation systems, be sure to include the subdirectories of *aunit* in your list of source directories.
9. Build the harness routine using gnatmake. The GNAT project file *aunit-1.04/aunit_tests.gpr* contains all the necessary links and switches for building test cases. When testing a new compiler, as opposed to incremental unit tests, the GNAT "-f" switch should be set for gnatmake. One can then use GNAT to build and run the tests.

For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, the test routine would look like:

```
procedure Test_Simple_Add
   (T : Aunit.Test_Cases.Test_Case'Class) is
    X, Y: Some_Currency;
begin
   X := 12; Y := 14;
   Assert (X + Y = 26, "Addition is incorrect");
end Test_Simple_Add;
```

The package spec (taken almost directly from '`pr_xxxx_xxx.ads`') looks as follows. The only modification was to remove support for a test fixture (next section), and to provide a name for the unit. Changes to "boilerplate code" are in bold:

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with AUnit.Test_Cases; use AUnit.Test_Cases;

package PR_xxxx_xxx is

   type Test_Case is new AUnit.Test_Cases.Test_Case with null record;

   procedure Register_Tests (T: in out Test_Case);
   -- Register routines to be run

   function Name (T: Test_Case) return String_Access;
   -- Provide name identifying the test case:

end PR_xxxx_xxx;
```

The package body, constructed by modifying 'pr_xxxx_xxx.adb' is:

```
with AUnit.Test_Cases.Registration; use AUnit.Test_Cases.Registration;
with AUnit.Assertions; use AUnit.Assertions;

-- Template for test case body.

package body PR_xxxx_xxx is

   -- Simple test routine

   procedure Test_Simple_Add
      (T    : Aunit.Test_Cases.Test_Case'Class) is
       X, Y : Some_Currency;
   begin
      X := 12; Y := 14;
      Assert (X + Y = 26, "Addition is incorrect");
   end;

   -- Register test routines to call

   procedure Register_Tests (T: access Test_Case) is
   begin
      -- Repeat for each test routine:
      Register_Routine (T, Test_Simple_Add'Access, "Test Addition");
   end Register_Tests;

   -- Identifier of test case.  Just change the string
   -- result of the function.

   function Name (T: Test_Case) return String_Access is
   begin
      return new String'("Money Tests");
   end Name;

end PR_xxxx_xxx;
```

The corresponding harness code, adapted from 'aunit-1.04/template/harness.adb' is:

```
with AUnit.Test_Cases; use AUnit.Test_Cases;
with AUnit.Test_Results.Test_Reporter; use AUnit.Test_Results.Test_Reporter;
--  Test case to run:
with PR_XXXX_XXX;
procedure Harness is
   Test : PR_XXXX_XXX.Test_Case;
   Result : AUnit.Test_Results.Result;
begin
   Run (Test, Result);
   Report (Result);
end Harness;
```

3 Fixture

Tests need to run against the background of a known set of objects. This set of objects is called a test fixture. When you are writing tests you will often find that you spend more time writing the code to set up the fixture than you do in actually testing values.

To some extent, you can make writing the fixture code easier by paying careful attention to the constructors you write. However, a much bigger savings comes from sharing fixture code. Often, you will be able to use the same fixture for several different tests. Each case will send slightly different messages or parameters to the fixture and will check for different results.

When you have a common fixture, here is what you do:

1. Create a package as in the previous section, starting from the templates '`pr_xxxx_xxx.ad*`'.
2. Add fields for elements of the fixture into the package body.
3. Override `Set_Up_Case` to initialize the fixture for all test routines.
4. Override `Set_Up` to initialize the variables before the execution of each routine.
5. Override `Tear_Down` to release any resources you allocated in `Set_Up` - to be executed after each test routine.
6. Override `Tear_Down_Case` to release any permanent resources you allocated in `Set_Up_Case` - executed after all test routines.

For example, to write several test cases that want to work with different combinations of 12 Euros, 14 Euros, and 26 US Dollars, first create a fixture. The package spec is now:

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with AUnit.Test_Cases; use AUnit.Test_Cases;

package PR_xxxx_xxx is
   type Test_Case is new AUnit.Test_Cases.Test_Case with null record;

   procedure Register_Tests (T: in out Test_Case);
   -- Register routines to be run

   function Name (T: Test_Case) return String_Access;
   -- Provide name identifying the test case

   Procedure Set_Up (T: in out Test_Case);
   -- Preparation performed before each routine

end PR_xxxx_xxx;
```

The body becomes:

```
with AUnit.Test_Cases.Registration; use AUnit.Test_Cases.Registration;
with AUnit.Assertions; use AUnit.Assertions;
with Currencies; use Currencies;

package body PR_xxxx_xxx is

   -- Fixture elements

   EU_12, EU_14 : Euro;
   US_26        : US_Dollar;

   -- Preparation performed before each routine

   Procedure Set_Up (T: in out Test_Case) is
   begin
      EU_12 := 12; EU_14 := 14;
      US_26 := 26;
   end Set_Up;

   -- Simple test routine

   procedure Test_Simple_Add
     (T : Aunit.Test_Cases.Test_Case'Class) is
   begin
       Assert
         (EU_12 + EU_14 /= US_26,
          "US and EU currencies not differentiated");
   end Test_Simple_Add;

   -- Register test routines to call

   procedure Register_Tests (T: in out Test_Case) is
   begin
      -- Repeat for each test routine:
      Register_Routine (T, Test_Simple_Add'Access, "Test Addition");
   end Register_Tests;

   -- Identifier of test case.  Just change the string
   -- result of the function.

   function Name (T: Test_Case) return String_Access is
   begin
      return new String'("Money Tests");
   end Name;

end PR_xxxx_xxx;
```

Once you have the fixture in place, you can write as many test routines as you like. Calls to `Set_Up` and `Tear_Down` bracket the invocation of each test routine.

Note that as of AUnit 1.01 a parameter of type `AUnit.Test_Cases.Test_Case'Class` has been added to test routines. This parameter allows access to the current `Test_Case` instance, so that a test routine can access per-instance (rather than package body global) data. This can be useful when part of the data to be used depends on a particular test case instance, while another part is global data of the test fixture.

Once you have several test cases, organize them into a Suite.

4 Suite

How do you run several test cases at once?

As soon as you have two tests, you'll want to run them together. You could run the tests one at a time yourself, but you would quickly grow tired of that. Instead, AUnit provides an object, `Test_Suite` which runs any number of test cases together.

For test routines that use the same fixture (i.e. those declared in the same package), the `Register_Routine` procedure is used to collect them into the single test case.

A single `Test_Case` and its collection of routines can be executed directly in a harness like so:

```
...
Test   : PR_XXXX_XXX.Test_Case;
Result : AUnit.Test_Results.Result;
...
Run (Test, Result);
Report (Result);
```

To create a suite of two test cases and run them together, execute:

```

with AUnit.Test_Suites; use AUnit.Test_Suites;
with AUnit.Test_Runner;

--  List of tests and suites to run

with Test_Case_1, Test_Case_2;

procedure Harness is

   function Suite return Access_Test_Suite is
      Result : Access_Test_Suite := new Test_Suite;
   begin
      --  You may add multiple tests or suites here:
      Add_Test (Result, new Test_Case_1.Test_Case);
      Add_Test (Result, new Test_Case_2.Test_Case);
      return Result;
   end Suite;

   procedure Run is new AUnit.Test_Runner (Suite);

begin
   Run;
end Harness;
```

5 Composition of Suites

Typically, one will want the flexibility to execute a complete set of tests, or some subset of them. In order to facilitate this, we can reorganize the harness so that the composition of test cases and suites is done in a separate library function, and each composition level can have its own harness:

```
-- Composition function:
with AUnit.Test_Suites; use Aunit.Test_Suites;

-- List of tests and suites to compose:
with Test_Case_1;
with Test_Case_2;

function This_Suite return Access_Test_Suite is
   Result : Access_Test_Suite := new Test_Suite;
begin
   Add_Test (Result, new Test_Case_1.Test_Case);
   Add_Test (Result, new Test_Case_2.Test_Case);
   return Result;
end Suite;

-- More general form of harness for a given level:
with AUnit.Test_Runner;
--  Composition function for this level:
with This_Suite;
procedure Harness is
   procedure Run is new AUnit.Test_Runner (This_Suite);
begin
   Run;
end Harness;
```

At a higher level, we may wish to combine two suites of units tests that are composed with functions `This_Suite` and `That_Suite`.

The corresponding composition function and harness would be:

```
-- Composition function:
with AUnit.Test_Suites; use Aunit.Test_Suites;

-- List of tests and suites to compose:
with Suite_1;
with Suite_2;

function Composition_Suite return Access_Test_Suite is
   Result : Access_Test_Suite := new Test_Suite;
begin
   Add_Test (Result, Suite_1);
   Add_Test (Result, Suite_2);
   return Result;
end Composition_Suite;

-- More general form of harness for a given level:
with AUnit.Test_Runner;
--  Composition function for this level:
with Composition_Suite;
procedure Harness is
   procedure Run is new AUnit.Test_Runner (Composition_Suite);
begin
   Run;
end Harness;
```

As can be seen, this is a very flexible way of composing test cases into execution runs.

Note that the `Aunit.Test_Runner.Run` routine has a defaulted parameter to control whether timing information is reported. Its speficiation is:

```
procedure Run (Timed : Boolean := True);
```

By default the execution time for a harness is reported. If you are running some number of harnesses from a scripting language, and comparing the result to an existing file, using `Timed -> False` ensures that the output will be identical across successful runs.

6 Reporting

Currently test results are reported using a simple console reporting routine:

```
Test_Results.Text_Reporter.Report (Result);
```

A sample run on a set of problem reports submitted to ACT prints the following to the console when executed:

```
[efalis@dogen AUnit]$ ./harness
   Total Tests Run:  10
   Failed Tests: 1
      PR 7503-008.Allocation_Test:: Bad discriminant check
   Unexpected Errors: 0
```

The switch "-v" may be used with any harness to cause the list of successful tests to be printed along with any failures or errors:

```
[efalis@dogen AUnit]$ ./harness -v
   Total Tests Run:  17
   Successful Tests: 17
      PR 7112-001: Record_Initialization
      PR 7210-005: Test_1
      PR 7210-005: Test_2
      PR 7210-005: Test_3
      PR 7210-005: Test_4
      PR 7210-005: Test_5
      PR 7210-005: Test_6
      PR 7210-005: Test_A
      PR 7210-005: Test_B
      PR 7503-008: Allocation_Test
      PR 7605-009: Modular_Bounds
      PR 8010-001b: Test calculation of constant with modular sub-expression
      PR 7522-012: Subtype not recognized in initialization
      PR 7617-011: Test renaming in instantiation I
      PR 7624-003: Use of multi-dimensional aggregate as generic actual parameter
      PR 7813-010: Test -gnatc for bogus semantic error
      PR 8010-009: Overload resolution with enumeration literals
   Failed Tests: 0
   Unexpected Errors: 0
Time:  0.001011000 seconds
```

7 GPS Support

Note that the GPS IDE has a menu **Edit -> Unit Testing** to generate the template code for test cases, test suites and harnesses.