AUnit Cookbook

AUnit - version 2011
Configuration level $Revision: 171929 $
Date: 19 April 2011

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) and C++ CppUnit (M. Feathers, J. Lacoste, E. Sommerlade, B. Lepilleur, B. Bakker, S. Robbins) unit test frameworks for Ada code.

1.1 What's new in AUnit 3

AUnit 3 brings several enhancements over AUnit 2 and AUnit 1:

- Removal of the genericity of the AUnit framework, making the AUnit 3 API as close as possible to AUnit 1.
- Emulates dynamic memory management for limited run-time profiles.
- Provides a new XML reporter, and changes harness invocation to support easy switching among text, XML and customized reporters.
- Provides new tagged types Simple_Test_Case, Test_Fixture and Test_Caller that correspond to CppUnit's TestCase, TestFixture and TestCaller classes.
- Emulates exception propagation for restricted run-time profiles (e.g. ZFP), by using the gcc builtin setjmp/longjmp mechanism.
- Reports the source location of an error when possible.

1.2 Examples

With this version, we have provided new examples illustrating the enhanced features of the framework. These examples are in the AUnit installation directory: <aunit-root>/share/examples/aunit, and are also available in the source distribution aunit-2011-src/examples.

The following examples are provided:

- simple_test: shows use of AUnit.Simple_Test_Cases (see Section 3.1 [AUnit.Simple_Test_Cases], page 5).
- test_caller: shows use of AUnit.Test_Caller (see Section 3.3 [AUnit.Test_Caller], page 7).
- test_fixture: example of a test fixture (see Chapter 4 [Fixture], page 9).
- liskov: This suite tests conformance to the Liskov Substitution Principle of a pair of simple tagged types. (see Section 7.2 [OOP considerations], page 15)
- failures: example of handling and reporting failed tests (see Chapter 6 [Reporting], page 13).
- calculator: a full example of test suite organization.

1.3 Note about limited run-times

AUnit allows a great deal of flexibility as to the structure of test cases, suites and harnesses. The templates and examples given in this document illustrate how to use AUnit while staying within the constraints of the GNAT Pro restricted and Zero Foot Print (ZFP) run-time libraries. Therefore, they avoid the use of dynamic allocation and some other features that would be outside of the profiles corresponding to these libraries. Tests targeted to the full Ada run-time library need not comply with these constraints.

1.4 Thanks

This document is adapted from the JUnit and CppUnit Cookbooks documents contained in their respective release packages.

Special thanks to Francois Brun of Thales Avionics for his ideas about support for OOP testing.

2 Overview

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 that 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. Derive a test case type from `AUnit.Simple_Test_Cases.Test_Case`.

   Several test case types are available:

   - `AUnit.Simple_Test_Cases.Test_Case`: the base type for all test cases. Needs overriding of `Name` and `Run_Test`.
   - `AUnit.Test_Cases.Test_Case`: the traditional AUnit test case type, allowing multiple test routines registration, each being run and reported independently.
   - `AUnit.Test_Fixtures.Test_Fixture`: used together with `AUnit.Test_Caller`, this allows easy creation of test suites comprising several test cases that share the same fixture (see Chapter 4 [Fixture], page 9).

   See Chapter 3 [Test Cases], page 5, for simple examples of use of these types.

2. When you want to check a value[1], use one of the following `Assert`[2] methods:

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

   or:

   ```
   if not AUnit.Assertions.Assert (Boolean_Expression, String_Description) then
      return;
   end if;
   ```

   In case you need to test that a method raises an expected exception, there also exists a generic Assert procedure that you can use for this purpose:

---

[1] While JUnit and some other members of the xUnit family of unit test frameworks provide specialized forms of assertions (e.g. `assertEqual`), we took a design decision in AUnit not to provide such forms. Ada has a much more rich type system giving a plethora of possible scalar types, and leading to an explosion of possible special forms of assert routines. This is exacerbated by the lack of a single root type for most types, as is found in Java. With the introduction of AUnit 2 for use with restricted run-time profiles, where even 'Image is missing, providing a comprehensive set of special assert routines in the framework itself becomes even more unrealistic. Since AUnit is intended to be an extensible toolkit, users can certainly write their own custom collection of such assert routines to suit local usage.

[2] Note that in AUnit 3, and contrary to AUnit 2, the procedural form of `Assert` has the same behavior whatever the underlying Ada run-time library: a failed assertion will cause the execution of the calling test routine to be abandoned. The functional form of `Assert` always continues on a failed assertion, and provides you with a choice of behaviors.

```
declare
   procedure Test_Raising_Exception is
   begin
      call_to_the_tested_method (some_args);
   end Test_Raising_Exception;

   procedure Assert_Exception_Raised is new AUnit.Assertions.Assert_Exception
     (Test_Raising_Exception);
begin
   Assert_Exception_Raised (String_Description);
end;
```

This procedure can handle exceptions with all run-time profiles (including zfp).

3. Create a suite function inside a package to gather together test cases and sub-suites[3].
4. At any level at which you wish to run tests, create a harness by instantiating procedure `AUnit.Run.Test_Runner` or function `AUnit.Run.Test_Runner_With_Status` with the top-level suite function to be executed. This instantiation provides a routine that executes all of the tests in the suite. We will call this user-instantiated routine **Run** in the text for backward compatibility to tests developed for AUnit 1. Note that only one instance of **Run** can execute at a time. This is a tradeoff made to reduce the stack requirement of the framework by allocating test result reporting data structures statically.

   It is possible to pass a filter to a `Test_Runner`, so that only a subset of the tests is run. In particular, this filter could be initialized from a command line parameter. See the package `AUnit.Test_Filters` for an example of such a filter. AUnit does not automatically initialize this filter from the command line both because it would not be supported with some of the limited runtimes (zero footprint for instance), and because you might want to pass the argument through different ways (as a parameter to switch, or a stand along command line argument for instance).
5. Build the code that calls the harness **Run** routine using gnatmake or gprbuild. The GNAT project file *aunit.gpr* contains all necessary switches, and should be imported into your root project file.

[3] If using the ZFP or the 'cert' run-time profiles, test cases and suites must be allocated using `AUnit.Memory.Utils.Gen_Alloc`, `AUnit.Test_Caller.Create`, `AUnit.Test_Suites.New_Suite`, or be statically allocated.

3 Test Case

In this chapter, we will introduce how to use the various forms of Test Cases. We will illustrate with a very simple test routine, that verifies that the sum of two Moneys with the same currency result in a value which is the sum of the values of the two Moneys:

```
declare
    X, Y: Some_Currency;
begin
   X := 12; Y := 14;
   Assert (X + Y = 26, "Addition is incorrect");
end;
```

The following sections will show how to use this test method using the different test case types available in AUnit.

3.1 AUnit.Simple_Test_Cases

`AUnit.Simple_Test_Cases.Test_Case` is the root type of all test cases. Although generally not meant to be used directly, it provides a simple and quick way to run a test.

This tagged type has several methods that need to be defined, or may be overridden.

- `function Name (T : Test_Case) return Message_String is abstract`:

  This function returns the Test name. You can easily translate regular strings to Message_String using `AUnit.Format`. For example:

  ```
  function Name (T : Money_Test) is
  begin
     return Format ("Money Tests");
  end Name;
  ```

- `procedure Run_Test (T : in out Test_Case) is abstract`:

  This procedure contains the test code. For example:

  ```
  procedure Run_Test (T : in out Money_Test) is
      X, Y: Some_Currency;
  begin
     X := 12; Y := 14;
     Assert (X + Y = 26, "Addition is incorrect");
  end Run_Test;
  ```

- `procedure Set_Up (T : in out Test_Case);` and `procedure Tear_Down (T : in out Test_Case);` (default implementations do nothing):

  These procedures are meant to respectively set up or tear down the environment before running the test case. See Chapter 4 [Fixture], page 9, for examples of how to use these methods.

You can find a compilable example of `AUnit.Simple_Test_Cases.Test_Case` usage in your AUnit installation directory: <aunit-root>/share/examples/aunit/simple_test/ or from the source distribution aunit-2011-src/examples/simple_test/

3.2 AUnit.Test_Cases

`AUnit.Test_Cases.Test_Case` is derived from `AUnit.Simple_Test_Cases.Test_Case` and defines its `Run_Test` procedure.

It allows a very flexible composition of Test routines inside a single test case, each being reported independently.

The following subprograms must be considered for inheritance, overriding or completion:

- `function Name (T : Test_Case) return Message_String is abstract;` Inherited. See Section 3.1 [AUnit.Simple_Test_Cases], page 5.
- `procedure Set_Up (T : in out Test_Case)` and `procedure Tear_Down (T : in out Test_Case)` Inherited. See Section 3.1 [AUnit.Simple_Test_Cases], page 5.
- `procedure Set_Up_Case (T : in out Test_Case)` and `procedure Tear_Down_Case (T : in out Test_Case)` Default implementation does nothing.

  The latter procedures provide an opportunity to Set Up and Tear Down the test case before and after all test routines have been executed. In contrast, the inherited `Set_Up` and `Tear_Down` are called before and after the execution of each individual test routine.
- `procedure Register_Tests (T : in out Test_Case) is abstract` This procedure must be overridden. It is responsible for registering all the test routines that will be run. You need to use either `Registration.Register_Routine` or the generic `Specific_Test_Case.Register_Wrapper` methods defined in `AUnit.Test_Cases` to register a routine. A test routine has the form:

  ```
  procedure Test_Routine (T : in out Test_Case'Class);
  ```

  or

  ```
  procedure Test_Wrapper (T : in out Specific_Test_Case'Class);
  ```

  The former procedure is used mainly for dispatching calls (see Section 7.2 [OOP considerations], page 15).

Using this type to test our money addition, the package spec is:

```
with AUnit; use AUnit;
with AUnit.Test_Cases; use AUnit.Test_Cases;

package Money_Tests is

   type Money_Test is new Test_Cases.Test_Case with null record;

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

   function Name (T: Money_Test) return Message_String;
   -- Provide name identifying the test case

   -- Test Routines:
   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class);
end Money_Tests;
```

The package body is:

```
with AUnit.Assertions; use AUnit.Assertions;

package body Money_Tests is

   procedure Test_Simple_Add (T : in out 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;

   -- Register test routines to call
   procedure Register_Tests (T: in out Money_Test) is

      use AUnit.Test_Cases.Registration;

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

   -- Identifier of test case

   function Name (T: Money_Test) return Test_String is
   begin
      return Format ("Money Tests");
   end Name;

end Money_Tests;
```

3.3 AUnit.Test_Caller

`Test_Caller` is a generic package that is used with `AUnit.Test_Fixtures.Test_Fixture`. `Test_Fixture` is a very simple type that provides only the `Set_Up` and `Tear_Down` procedures. This type is meant to contain a set of user-defined test routines, all using the same Set up and Tear down mechanisms. Once those routines are defined, the Test_Caller package is used to incorporate them directly into a test suite.

With our money example, the Test_Fixture is:

```
with AUnit.Test_Fixtures;
package Money_Tests is
   type Money_Test is new AUnit.Test_Fixtures.Test_Fixture with null record;

   procedure Test_Simple_Add (T : in out Money_Test);

end Money_Tests;
```

The test suite (see Chapter 5 [Suite], page 11) calling the test cases created from this Test_Fixture is:

```
with AUnit.Test_Suites;

package Money_Suite is
   function Suite return AUnit.Test_Suites.Access_Test_Suite;
end Money_Suite;
```

With the corresponding body:

```
with AUnit.Test_Caller;
with Money_Tests;

package body Money_Suite is

   package Money_Caller is new AUnit.Test_Caller
     (Money_Tests.Money_Test);

   function Suite return Aunit.Test_Suites.Access_Test_Suite is
      Ret : AUnit.Test_Suites.Access_Test_Suite :=
              AUnit.Test_Suites.New_Suite;
   begin
      Ret.Add_Test
        (Money_Caller.Create
           ("Money Test : Test Addition",
            Money_Tests.Test_Simple_Add'Access));
      return Ret;
   end Suite;

end Money_Suite;
```

Note that `New_Suite` and `Create` are fully compatible with limited run-times (in particular, those without dynamic allocation support).

You can find a compilable example of `AUnit.Test_Caller` use in the AUnit installation directory: <aunit-root>/share/examples/aunit/test_caller/ or from the source distribution aunit-2011-src/examples/test_caller/

4 Fixture

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

You can make writing fixture code easier by sharing it. 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 Test Case package as in previous section.
2. Declare variables or components for elements of the fixture either as part of the test case type or in the package body.
3. According to the Test_Case type used, override its `Set_Up` and/or `Set_Up_Case` method:
   - `AUnit.Simple_Test_Cases`: `Set_Up` is called before `Run_Test`.
   - `AUnit.Test_Cases`: `Set_Up` is called before each test routine while `Set_Up_Case` is called once before the routines are run.
   - `AUnit.Test_Fixture`: `Set_Up` is called before each test case created using `Aunit.Test_Caller`.
4. You can also override `Tear_Down` and/or `Tear_Down_Case` that are executed after the test is run.

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 AUnit; use AUnit;

package Money_Tests is
   use Test_Results;

   type Money_Test is new Test_Cases.Test_Case with null record;

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

   function Name (T : Money_Test) return Test_String;
   -- Provide name identifying the test case

   procedure Set_Up (T : in out Money_Test);
   --  Set up performed before each test routine

   -- Test Routines:
   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class);
end Money_Tests;
```

The body becomes:

```
package body Money_Tests is

   use Assertions;

   -- Fixture elements

   EU_12, EU_14 : Euro;
  US_26       : US_Dollar;

   -- Preparation performed before each routine

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

   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class) is
       X, Y : Some_Currency;
   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 Money_Test) is

      use Test_Cases.Registration;

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

   -- Identifier of test case
   function Name (T: Money_Test) return Test_String is
   begin
      return Format ("Money Tests");
   end Name;

end Money_Tests;
```

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.

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

You can find a compilable example of fixture set up using `AUnit.Test_Fixture` in your AUnit installation directory: <aunit-root>/share/examples/aunit/test_fixture/ or from the AUnit source distribution aunit-2011-src/examples/test_fixture/.

5 Suite

5.1 Creating a Test 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`, that runs any number of test cases together.

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

```
with AUnit.Test_Suites;
package My_Suite is
   function Suite return AUnit.Test_Suites.Access_Test_Suite;
end My_Suite;
```

```
--  Import tests and sub-suites to run
with Test_Case_1, Test_Case_2;

package body My_Suite is
   use AUnit.Test_Suites;

   -- Statically allocate test suite:
   Result : aliased Test_Suite;

   --  Statically allocate test cases:
   Test_1 : aliased Test_Case_1.Test_Case;
   Test_2 : aliased Test_Case_2.Test_Case;

   function Suite return Access_Test_Suite is
   begin
      Add_Test (Result'Access, Test_Case_1'Access);
      Add_Test (Result'Access, Test_Case_2'Access);
      return Result'Access;
   end Suite;
end My_Suite;
```

Instead of statically allocating test cases and suites, you can also use `AUnit.Test_Suites.New_Suite` and/or `AUnit.Memory.Utils.Gen_Alloc`. These routines emulate dynamic memory management (see Chapter 8 [Using AUnit with Restricted Run-Time Libraries], page 23). Similarly, if you know that the tests will always be executed for a run-time profile that supports dynamic memory management, you can allocate these objects directly with the Ada `"new"` operator.

The harness is:

```
with My_Suite;
with AUnit.Run;
with AUnit.Reporter.Text;

procedure My_Tests is
  procedure Run is new AUnit.Run.Test_Runner (My_Suite.Suite);
  Reporter : AUnit.Reporter.Text.Text_Reporter;
begin
   Run (Reporter);
end My_Tests;
```

5.2 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 compose both suites and test cases, and provide a harness for any given suite:

```
-- Composition package:
with AUnit; use AUnit;
package Composite_Suite is
   function Suite return Test_Suites.Access_Test_Suite;
end Composite_Suite;

--  Import tests and suites to run
with This_Suite, That_Suite;
with AUnit.Tests;

package body Composite_Suite is
   use Test_Suites;

   --  Here we dynamically allocate the suite using the New_Suite function
   --  We use the 'Suite' functions provided in This_Suite and That_Suite
   --  We also use Ada 2005 distinguished receiver notation to call Add_Test

   function Suite return Access_Test_Suite is
      Result : Access_Test_Suite := AUnit.Test_Suites.New_Suite;
   begin
      Result.Add_Test (This_Suite.Suite);
      Result.Add_Test (That_Suite.Suite);
      return Result;
   end Suite;
end Composite_Suite;
```

The harness remains the same:

```
with Composite_Suite;
with AUnit.Run;

procedure My_Tests is
  procedure Run is new AUnit.Run.Test_Runner (Composite_Suite.Suite);
  Reporter : AUnit.Reporter.Text.Text_Reporter;
begin
   Run (Reporter);
end My_Tests;
```

As can be seen, this is a very flexible way of composing test cases into execution runs: any combination of test cases and sub-suites can be collected into a suite.

6 Reporting

Test results can be reported using several 'Reporters'. By default, two reporters are available in AUnit: `AUnit.Reporter.Text.Text_Reporter` and `AUnit.Reporter.XML.XML_Reporter`. The first one is a simple console reporting routine, while the second one outputs the result using an XML format. These are invoked when the `Run` routine of an instantiation of `AUnit.Run.Test_Runner` is called.

New reporters can be created using children of `AUnit.Reporter.Reporter`.

The Reporter is selected by specifying it when calling **Run**:

```
with A_Suite;
with AUnit.Run;
with AUnit.Reporter.Text;

procedure My_Tests is
  procedure Run is new AUnit.Run.Test_Runner (A_Suite.Suite);
  Reporter : AUnit.Reporter.Text.Text_Reporter;
begin
   Run (Reporter);
end My_Tests;
```

The final report is output once all tests have been run, so that they can be grouped depending on their status (passed or fail). If you need to output the tests as they are run, you should consider extending the `Test_Result` type and do some output every time a success or failure is registered.

6.1 Text output

Here is an example where the test harness runs 4 tests, one reporting an assertion failure, one reporting an unexpected error (exception):

```
--------------------

   Total Tests Run: 4

   Successful Tests: 2
      Test addition
      Test subtraction

   Failed Assertions: 1

      Test addition (failure expected)
            Test should fail this assertion, as 5+3 /= 9
            at math-test.adb:29


   Unexpected Errors: 1

      Test addition (error expected)
            CONSTRAINT_ERROR



Time: 2.902E-4 seconds
```

This reporter can optionally use colors (green to report success, red to report errors). Since not all consoles support it, this is off by default, but you can call `Set_Use_ANSI_Colors` to activate support for colors.

6.2 XML output

Following is the same harness run using XML output. The XML format used matches the one used by CppUnit.

Note that text set in the Assert methods, or as test case names should be compatible with utf-8 character encoding, or the XML will not be correctly formatted.

```
<?xml version='1.0' encoding='utf-8' ?>
<TestRun elapsed='1.107E-4'>
  <Statistics>
    <Tests>4</Tests>
    <FailuresTotal>2</FailuresTotal>
    <Failures>1</Failures>
    <Errors>1</Errors>
  </Statistics>
  <SuccessfulTests>
    <Test>
      <Name>Test addition</Name>
    </Test>
    <Test>
      <Name>Test subtraction</Name>
    </Test>
  </SuccessfulTests>
  <FailedTests>
    <Test>
      <Name>Test addition (failure expected)</Name>
      <FailureType>Assertion</FailureType>
      <Message>Test should fail this assertion, as 5+3 /= 9</Message>
      <Location>
        <File>math-test.adb</File>
        <Line>29</Line>
      </Location>
    </Test>
    <Test>
      <Name>Test addition (error expected)</Name>
      <FailureType>Error</FailureType>
      <Message>CONSTRAINT_ERROR</Message>
    </Test>
  </FailedTests>
</TestRun>
```

7 Test Organization

7.1 General considerations

This section will discuss an approach to organizing an AUnit test harness, considering some possibilities offered by Ada language features.

The general idea behind this approach to test organization is that making the test case a child of the unit under test gives some useful facilities. The test case gains visibility to the private part of the unit under test. This offers a more "white box" approach to examining the state of the unit under test than would, for instance, accessor functions defined in a separate fixture that is a child of the unit under test. Making the test case a child of the unit under test also provides a way to make the test case share certain characteristics of the unit under test. For instance, if the unit under test is generic, then any child package (here the test case) must be also generic: any instantiation of the parent package will require an instantiation of the test case in order to accomplish its aims.

Another useful concept is matching the test case type to that of the unit under test:

- When testing a generic package, the test package should also be generic.
- When testing a tagged type, then test routines should be dispatching, and the test case type for a derived tagged type should be a derivation of the test case type for the parent.
- etc.

Maintaining such similarity of properties between the test case and unit under test can facilitate the testing of units derived in various ways.

The following sections will concentrate on applying these concepts to the testing of tagged type hierarchies and to the testing of generic units.

A full example of this kind of test organization is available in the AUnit installation directory: <AUnit-root>/share/examples/aunit/calculator, or from the AUnit source distribution aunit-2011-src/examples/calculator.

7.2 OOP considerations

When testing a hierarchy of tagged types, one will often want to run tests for parent types against their derivations without rewriting those tests.

We will illustrate some of the possible solutions available in AUnit, using the following simple example that we want to test:

First we consider a `Root` package defining the `Parent` tagged type, with two procedures P1 and P2.

```
package Root is
   type Parent is tagged private;

   procedure P1 (P : in out Parent);
   procedure P2 (P : in out Parent);
private
   type Parent is tagged record
      Some_Value : Some_Type;
   end record;
end Root;
```

We will also consider a derivation of type `Parent`:

```
with Root;
package Branch is
   type Child is new Root.Parent with private;

   procedure P2 (C : in out Child);
   procedure P3 (C : in out Child);
private
   type Child is new Root.Parent with null record;
end Branch;
```

Note that `Child` retains the parent implementation of `P1`, overrides `P2` and adds `P3`. Its test will override `Test_P2` when we override `P2` (not necessary, but certainly possible).

7.2.1 Using AUnit.Test_Fixtures

Using `Test_Fixture` type, we first test `Parent` using the following test case:

```
with AUnit; use AUnit;
with AUnit.Test_Fixtures; use AUnit.Test_Fixtures;

--  We make this package a child package of Parent so that it can have
--  visibility to its private part
package Root.Tests is

   type Parent_Access is access all Root.Parent'Class;

   --  Reference an object of type Parent'Class in the test object, so
   --  that test procedures can have access to it.
   type Parent_Test is new Test_Fixture with record
      Fixture : Parent_Access;
   end record;

   --  This will initialize P.
   procedure Set_Up (P : in out Parent_Test);

   --  Test routines. If derived types are declared in child packages,
   --  these can be in the private part.
   procedure Test_P1 (P : in out Parent_Test);
   procedure Test_P2 (P : in out Parent_Test);

end Root.Tests;

package body Root.Tests is

   Fixture : aliased Parent;

   --  We set Fixture in Parent_Test to an object of type Parent.
   procedure Set_Up (P : in out Parent_Test) is
   begin
      P.Fixture := Parent_Access (Fixture'Access);
   end Set_Up;

   --  Test routines: References to the Parent object are made via
   --  P.Fixture.all, and are thus dispatching.
   procedure Test_P1 (P : in out Parent_Test) is ...;
   procedure Test_P2 (P : in out Parent_Test) is ...;

end Root.Tests;
```

The associated test suite will be:

```
with AUnit.Test_Caller;
with Root.Tests;

package body Root_Suite is
   package Caller is new AUnit.Test_Caller with (Root.Tests.Parent_Test);

   function Suite return AUnit.Test_Suites.Access_Test_Suite is
      Ret : Access_Test_Suite := AUnit.Test_Suites.New_Suite;
   begin
      AUnit.Test_Suites.Add_Test
        (Ret, Caller.Create ("Test Parent : P1", Root.Tests.Test_P1'Access));
      AUnit.Test_Suites.Add_Test
        (Ret, Caller.Create ("Test Parent : P2", Root.Tests.Test_P2'Access));
      return Ret;
   end Suite;
end Root_Suite;
```

Now we define the test suite for the `Child` type. To do this, we inherit a test fixture from `Parent_Test`, overriding the `Set_Up` procedure to initialize `Fixture` with a `Child` object. We also override `Test_P2` to adapt it to the new implementation. We define a new `Test_P3` to test `P3`. And we inherit `Test_P1`, since `P1` is unchanged.

```
with Root.Tests; use Root.Tests;
with AUnit; use AUnit;
with AUnit.Test_Fixtures; use AUnit.Test_Fixtures;

package Branch.Tests is

   type Child_Test is new Parent_Test with null record;

   procedure Set_Up (C : in out Child_Test);

   --  Test routines:
   --  Test_P2 is overridden
   procedure Test_P2 (C : in out Child_Test);
   --  Test_P3 is new
   procedure Test_P3 (C : in out Child_Test);

end Branch.Tests;

package body Branch.Tests is
   use Assertions;

   Fixture : Child;
   --  This could also be a field of Child_Test

   procedure Set_Up (C : in out Child_Test) is
   begin
      --  The Fixture for this test will now be a Child
      C.Fixture := Parent_Access (Fixture'Access);
   end Set_Up;

   --  Test routines:
   procedure Test_P2 (C : in out Child_Test) is ...;
   procedure Test_P3 (C : in out Child_Test) is ...;

end Branch.Tests;
```

The suite for Branch.Tests will now be:

```
with AUnit.Test_Caller;
with Branch.Tests;

package body Branch_Suite is
   package Caller is new AUnit.Test_Caller with (Branch.Tests.Parent_Test);

   --  In this suite, we use Ada 2005 distinguished receiver notation to
   --  simplify the code.

   function Suite return Access_Test_Suite is
      Ret : Access_Test_Suite := AUnit.Test_Suites.New_Suite;
   begin
      --  We use the inherited Test_P1. Note that it is
      --  Branch.Tests.Set_Up that will be called, and so Test_P1 will be run
      --  against an object of type Child
      Ret.Add_Test
        (Caller.Create ("Test Child : P1", Branch.Tests.Test_P1'Access));
      --  We use the overridden Test_P2
      Ret.Add_Test
        (Caller.Create ("Test Child : P2", Branch.Tests.Test_P2'Access));
      --  We use the new Test_P2
      Ret.Add_Test
        (Caller.Create ("Test Child : P3", Branch.Tests.Test_P3'Access));
      return Ret;
   end Suite;
end Branch_Suite;
```

7.2.2 Using AUnit.Test_Cases

Using an `AUnit.Test_Cases.Test_Case` derived type, we obtain the following code for testing `Parent`:

```
with AUnit; use AUnit;
with AUnit.Test_Cases;
package Root.Tests is

   type Parent_Access is access all Root.Parent'Class;

   type Parent_Test is new AUnit.Test_Cases.Test_Case with record
      Fixture : Parent_Access;
   end record;

   function Name (P : Parent_Test) return Message_String;
   procedure Register_Tests (P : in out Parent_Test);

   procedure Set_Up_Case (P : in out Parent_Test);

   --  Test routines. If derived types are declared in child packages,
   --  these can be in the private part.
   procedure Test_P1 (P : in out Parent_Test);
   procedure Test_P2 (P : in out Parent_Test);

end Root.Tests;
```

The body of the test case will follow the usual pattern, declaring one or more objects of type **Parent**, and executing statements in the test routines against them. However, in order to support dispatching to overriding routines of derived test cases, we need to introduce class-wide wrapper routines for each primitive test routine of the parent type that we anticipate may be overridden. Instead of registering the parent's overridable primitive operations directly using `Register_Routine`, we register the wrapper using `Register_Wrapper`. This latter routine is exported by instantiating **AUnit**`.Test_Cases.Specific_Test_Case_Registration` with the actual parameter being the parent test case type.

```
with AUnit.Assertions; use AUnit.Assertions
package body Root.Tests is

   --  Declare class-wide wrapper routines for any test routines that will be
   --  overridden:
   procedure Test_P1_Wrapper (P : in out Parent_Test'Class);
   procedure Test_P2_Wrapper (P : in out Parent_Test'Class);

   function Name (P : Parent_Test) return Message_String is ...;

   --  Set the fixture in P
   Fixture : aliased Parent;
   procedure Set_Up_Case (P : in out Parent_Test) is
   begin
      P.Fixture := Parent_Access (Fixture'Access);
   end Set_Up_Case;

   --  Register Wrappers:
   procedure Register_Tests (P : in out Parent_Test) is

      package Register_Specific is
        new Test_Cases.Specific_Test_Case_Registration (Parent_Test);

      use Register_Specific;

   begin
      Register_Wrapper (P, Test_P1_Wrapper'Access, "Test P1");
      Register_Wrapper (P, Test_P2_Wrapper'Access, "Test P2");
   end Register_Tests;

   --  Test routines:
   procedure Test_P1 (P : in out Parent_Test) is ...;
   procedure Test_P2 (C : in out Parent_Test) is ...;

   --  Wrapper routines. These dispatch to the corresponding primitive
   --  test routines of the specific types.
   procedure Test_P1_Wrapper (P : in out Parent_Test'Class) is
   begin
      Test_P1 (P);
   end Test_P1_Wrapper;

   procedure Test_P2_Wrapper (P : in out Parent_Test'Class) is
   begin
      Test_P2 (P);
   end Test_P2_Wrapper;

end Root.Tests;
```

The code for testing the `Child` type will now be:

```
with Parent_Tests; use Parent_Tests;
with AUnit; use AUnit;
package Branch.Tests is

   type Child_Test is new Parent_Test with private;

   function Name (C : Child_Test) return Message_String;
   procedure Register_Tests (C : in out Child_Test);

   --  Override Set_Up_Case so that the fixture changes.
   procedure Set_Up_Case (C : in out Child_Test);

   --  Test routines:
   procedure Test_P2 (C : in out Child_Test);
   procedure Test_P3 (C : in out Child_Test);

private
   type Child_Test is new Parent_Test with null record;
end Branch.Tests;

with AUnit.Assertions; use AUnit.Assertions;
package body Branch.Tests is

   --  Declare wrapper for Test_P3:
   procedure Test_P3_Wrapper (C : in out Child_Test'Class);

   function Name (C : Child_Test) return Test_String is ...;

   procedure Register_Tests (C : in out Child_Test) is

      package Register_Specific is
         new Test_Cases.Specific_Test_Case_Registration (Child_Test);
      use Register_Specific;

   begin
      -- Register parent tests for P1 and P2:
      Parent_Tests.Register_Tests (Parent_Test (C));

      -- Repeat for each new test routine (Test_P3 in this case):
      Register_Wrapper (C, Test_P3_Wrapper'Access, "Test P3");
   end Register_Tests;

   --  Set the fixture in P
   Fixture : aliased Child;
   procedure Set_Up_Case (C : in out Child_Test) is
   begin
      C.Fixture := Parent_Access (Fixture'Access);
   end Set_Up_Case;

   --  Test routines:
   procedure Test_P2 (C : in out Child_Test) is ...;
   procedure Test_P3 (C : in out Child_Test) is ...;

   --  Wrapper for new routine:
   procedure Test_P3_Wrapper (C : in out Child_Test'Class) is
   begin
      Test_P3 (C);
   end Test_P3_Wrapper;

end Branch.Tests;
```

Note that inherited and overridden tests do not need to be explicitly re-registered in derived test cases - one just calls the parent version of `Register_Tests`. If the application tagged type hierarchy is organized into parent and child units, one could also organize the test cases into a hierarchy that reflects that of the units under test.

7.3 Testing generic units

When testing generic units, one would like to apply the same generic tests to all instantiations in an application. A simple approach is to make the test case a child package of the unit under test (which then must also be generic).

For instance, suppose the generic unit under test is a package (it could be a subprogram, and the same principle would apply):

```
generic
   -- Formal parameter list
package Template is
   -- Declarations
end Template;
```

The corresponding test case would be:

```
with AUnit; use AUnit;
with AUnit.Test_Fixtures;
generic
package Template.Gen_Tests is

   type Template_Test is new AUnit.Test_Fixtures.Test_Fixture with ...;

   --  Declare test routines

end Template.Gen_Tests;
```

The body will follow the usual patterns with the fixture based on the parent package `Template`. Note that due to an Ada AI, accesses to test routines, along with the test routine specifications, must be defined in the package specification rather than in its body.

Instances of `Template` will define automatically the Tests child package that can be directly instantiated as follow:

```
with Template.Gen_Test;
with Instance_Of_Template;
package Instance_Of_Template.Tests is new Instance_Of_Template.Gen_Test;
```

The instantiated test case objects are added to a suite in the usual manner.

8 Using AUnit with Restricted Run-Time Libraries

AUnit 3 - like AUnit 2 - is designed so that it can be used in environments with restricted Ada run-time libraries, such as ZFP and the cert run-time profile on Wind River Systems' VxWorks 653. The patterns given in this document for writing tests, suites and harnesses are not the only patterns that can be used with AUnit, but they are compatible with the restricted run-time libraries provided with GNAT Pro.

In general, dynamic allocation and deallocation must be used carefully in test code. For the cert profile on VxWorks 653, all dynamic allocation must be done prior to setting the application partition into "normal" mode. Deallocation is prohibited in this profile. For the default ZFP profile, dynamic memory management is not provided as part of the run-time, as it is not available on a bare board environment, and should not be used unless you have provided implementations as described in the GNAT Pro High Integrity User Guide.

Starting with AUnit 3, a simple memory management mechanism has been included in the framework, using a kind of storage pool. This memory management mechanism uses a static array allocated at startup, and simulates dynamic allocation afterwards by allocating parts of this array upon request. Deallocation is not permitted.

By default, the allocated array is a 100 KB array. This value can be changed by modifying its size in the file: '`aunit-2011-src/aunit/framework/staticmemory/aunit-memory.adb`'

To allocate a new object, you use `AUnit.Memory.Utils.Gen_Alloc`.

Additional restrictions relevant to the default ZFP profile include:

1. Normally the ZFP profile requires a user-defined `__gnat_last_chance_handler` routine to handle raised exceptions. However, AUnit now provides a mechanism to simulate exception propagation using gcc builtin setjmp/longjmp mechanism. This mechanism defines the `__gnat_last_chance_handler` routine, so it should not be redefined elsewhere.
2. AUnit requires `GNAT.IO` provided in '`g-io.ad?`' in the full or cert profile run-time library sources (or as implemented by the user). Since this is a run-time library unit it must be compiled with the gnatmake "-a" switch.
3. The AUnit framework has been modified so that no call to the secondary stack is performed, nor any call to memcpy or memset. However, if the unit under test, or the tests themselves require use of those routines, then the application or test framework must define those symbols and provide the requisite implementations.
4. The timed parameter of the Harness `Run` routine has no effect when used with the ZFP profile, and on profiles not supporting Ada.Calendar.

9 Installation and Use

AUnit 3 contains support for limited run-times such as zero-foot-print (ZFP) and certified run-time (cert). It can now be installed simultaneously for several targets and run-times.

9.1 Note on gprbuild

In order to compile and install AUnit, you need gprbuild. On cross-platforms gprbuild version 1.1 or later is required, while on native platforms any version can be used.

9.2 aunit project file and support for other platforms/run-times

AUnit can be installed once for several targets and run-times. To support this, the AUnit project includes scenario variables to select those targets and/or run-times. The scenario variables used are:

- **PLATFORM**: the supported values are "native" (default), "powerpc-elf", "vxworksae", "erc32-elf" and "leon-elf".
- **RUNTIME**: the supported values are "full" (default), "cert", "zfp", "hi" and "ravenscar".

9.3 Installing AUnit on UNIX systems

- Extract the archive:
  ```
  $ gunzip -dc aunit-2011-src.tgz | tar xf -
  ```
- To build AUnit for a full Ada run-time:
  ```
  $ cd aunit-2011-src
  $ make
  ```
  To build AUnit for a zfp run-time targeting powerpc-elf platform[1]:
  ```
  $ cd aunit-2011-src
  $ make TARGET=powerpc-elf RTS=zfp
  ```
  Once the above build procedure has been performed for all desired platforms, you can install AUnit:
  ```
  $ make install INSTALL=<gnat-root>
  ```
  Where <gnat-root> is for example /opt/gnat/6.0.1. If INSTALL is not specified, then AUnit will use the root directory where gprbuild is installed.
- Specific installation:

  The AUnit makefile supports some specific options, activated using environment variables. The following options are defined:
  - INSTALL: defines the AUnit base installation directory, should always be set.
  - TARGET: defines the gnat tools prefix to use. For example, to compile AUnit for powerpc VxWorks, TARGET should be set to powerpc-wrs-vxworks. If not set, the native compiler will be used.
  - RTS: defines the run-time used to compile AUnit.
- To test AUnit:

  The AUnit test suite is in the test subdirectory of the source package.
  ```
  $ cd test
  $ make
  ```
  The test suite's makefile supports the following variables: * RTS * TARGET

[1] Note that cross platforms require gprbuild version 1.1 or later

9.4 Installing AUnit on Windows systems

On Windows, an InstallShield wizard is available to easily install AUnit. If you are installing from the source package, you will first need a tool to uncompress the tar.gz file, then you can manually launch the 'doinstall.bat' script included in the source package.

Both methods need user responses during the installation:

- Selection of targets (optional): the InstallShield will try to detect all GNAT compilers available on your system. If only one compiler is found, then this compiler is automatically selected, and the script starts with the next step. If compilers for several targets are found, then the script will display a list of available targets. You can then select one or several targets, pressing enter to continue.

```
*** Select the targets you want AUnit to be compiled for.
*(1) pentium-mingw32msv (full)
 (2) pentium-mingw32msv (zfp)
 (3) powerpc-elf (zfp)
Enter a target number, or <enter> to continue the setup:
```

- At this stage, the AUnit library starts compiling. Once the compilation is done, the message

```
*** AUNIT COMPILED SUCCESSFULLY ***
```

  should be displayed.
- Selection of the installation directory: by default, AUnit is installed in the same root directory as gprbuild. Enter another directory at this stage if you want to install AUnit in another directory.

```
Please enter AUnit base installation directory
** Press enter for installing in 'C:\GNATPRO\6.1.1':
```

After entering the information above, the InstallShield will install AUnit in the selected installation directory.

Note that in contrast to the AUnit 2 installation procedures, support for exception handling, memory allocation and Ada.Calendar is automatically selected depending on the run-time used.

9.5 Installed files

The AUnit library is installed in the specified directory (<aunit-root> identifies the root installation directory as specified during the installation procedures above):

- the aunit.gpr project is installed in <aunit-root>/lib/gnat
- the AUnit source files are installed in <aunit-root>/include/aunit
- the AUnit library files are installed in <aunit-root>/lib/aunit
- the AUnit documentation is installed in <aunit-root>/share/doc/aunit
- the AUnit examples are installed in <aunit-root>/share/examples/aunit

10 GPS Support

The GPS IDE has a menu **Edit -> Unit Testing** to generate template code for test cases, test suites and harnesses. The current templates are for AUnit 1.x, and AUnit 2.x. The next version of GPS will also include support for AUnit 3. In the meantime, the AUnit 1.x templates require only very minor modification for use with AUnit 3.