AUnit Cookbook

Release 0.0

AdaCore

Feb 11, 2022

This page is intentionally left blank.

CONTENTS

This page is intentionally left blank.

Ada Unit Testing Framework

Version 0.0
Date: Feb 11, 2022

AdaCore

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*.

This page is intentionally left blank.

CHAPTER

ONE

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 Typographic conventions

For notational convenience, <*version*> will be used throughout this document to stand for the AUnit product version number. For example, aunit-<*version*>-src expands to aunit-0.0-src.

1.3 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-<version>-src/examples`.

The following examples are provided:

- simple_test: shows use of AUnit.Simple_Test_Cases (see *AUnit.Simple_Test_Cases*).
- test_caller: shows use of AUnit.Test_Caller (see *AUnit.Test_Caller*).
- test_fixture: example of a test fixture (see *Fixture*).

- liskov: This suite tests conformance to the Liskov Substitution Principle of a pair of simple tagged types. (see *OOP considerations*)
- failures: example of handling and reporting failed tests (see *Reporting*).
- calculator: a full example of test suite organization.

1.4 Note about limited run-time libraries

AUnit allows a great deal of flexibility for 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 Footprint (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.

1.5 Thanks

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

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

CHAPTER

TWO

OVERVIEW

How do you write testing code?

The simplest approach 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:

- 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 to be registered, where each one is 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 *Fixture*).

  See *Test Case* for simple examples of using these types.

- When you want to check a value [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 richer type system giving a large number 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. use one of the following Assert [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. methods:

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

  or:

[1]
[2]

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

If you need to test that a subprogram raises an expected exception, there is the procedure `Assert_Exception` that takes an access value designating the procedure to be tested as a parameter:

```
type Throwing_Exception_Proc is access procedure;

procedure Assert_Exception
(Proc    : Throwing_Exception_Proc;
Message : String;
Source  : String := GNAT.Source_Info.File;
Line    : Natural := GNAT.Source_Info.Line);
--  Test that Proc throws an exception and record "Message" if not.
```

Example:

```
-- Declared at library level:
procedure Test_Raising_Exception is
begin
  call_to_the_tested_method (some_args);
end Test_Raising_Exception;

-- In test routine:
procedure My_Routine (...) is
begin
  Assert_Exception (Test_Raising_Exception'Access, **String_Description**);
end My_Routine;
```

This procedure can handle exceptions with all run-time profiles (including zfp). If you are using a run-time library capable of propagating exceptions, you can use the following idiom instead:

```
procedure My_Routine (...) is
begin
  ...
  -- Call subprogram expected to raise an exception:
  Call_To_The_Tested_Method (some_args);
  Assert (False, 'exception not raised');
exception
  when desired_exception =>
    null;
end My_Routine;
```

An unexpected exception will be recorded as such by the framework. If you want your test routine to continue beyond verifying that an expected exception has been raised, you can nest the call and handler in a block.

- Create a suite function inside a package to gather together test cases and sub-suites. (If either the ZFP or the cert run-time profiles ia being used, test cases and suites must be allocated using `AUnit.Memory.Utils.Gen_Alloc`, `AUnit.Test_Caller.Create`, `AUnit.Test_Suites.New_Suite`, or else they must be statically allocated.)
- 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 with 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 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 run-time profiles (ZFP for instance), and because you might want to pass the argument in different ways (as a parameter to switch, or a stand-alone command line argument for instance).

It is also possible to control the contents of the output report by passing an object of type `AUnit_Options` to the *Test_Runner*. See package `AUnit.Options` for details.

- 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.

This page is intentionally left blank.

CHAPTER

THREE

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, which verifies that the sum of two Money values with the same currency unit is a value that is the sum of the two values:

```
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) return Message_String 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 *Fixture* 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-`*<version>*`-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 *AUnit.Simple_Test_Cases*.

- `procedure Set_Up (T : in out Test_Case);`
  `procedure Tear_Down (T : in out Test_Case);`

  Inherited. See *AUnit.Simple_Test_Cases*.

- `procedure Set_Up_Case (T : in out Test_Case);`
  `procedure Tear_Down_Case (T : in out Test_Case);`

  Default implementation does nothing.

  These last two 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` subprograms 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 *OOP considerations*).

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;
```

(continues on next page)

(continued from previous page)

```
  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 *Suite*) 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;
```

Here is 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-time libraries (in particular, those without dynamic allocation support). However, for non-native run-time libraries, you cannot extend `Test_Fixture` with a controlled component.

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

CHAPTER

FOUR

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:

- Create a *Test Case* package as in previous section.
- Declare variables or components for elements of the fixture either as part of the test case type or in the package body.
- According to the Test_Case type used, override its `Set_Up` and/or `Set_Up_Case` subprogram:
  - `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_Fixtures`: `Set_Up` is called before each test case created using `Aunit.Test_Caller`.
- 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:
```

(continues on next page)

(continued from previous page)

```
   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_Fixtures` in your AUnit installation directory: *<aunit-root>*`/share/examples/aunit/test_fixture/` or from the AUnit source distribution `aunit-`*<version>*`-src/examples/test_fixture/`.

CHAPTER

FIVE

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 *Using AUnit with Restricted Run-Time Libraries*). 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` operation.

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
```

(continues on next page)

(continued from previous page)

```
   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.

This page is intentionally left blank.

CHAPTER

SIX

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)
```

(continues on next page)

(continued from previous page)

```
            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* subprograms 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>
```

(continues on next page)

(continued from previous page)

```
  </FailedTests>
</TestRun>
```

This page is intentionally left blank.

CHAPTER

SEVEN

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, for example:

- 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.

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-<version>-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;
```

(continues on next page)

(continued from previous page)

```
   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 from 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 type `Test_Fixture`, 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
```

(continues on next page)

(continued from previous page)

```
   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
```

(continues on next page)

(continued from previous page)

```
      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:
```

(continues on next page)

(continued from previous page)

```
   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
```

(continues on next page)

(continued from previous page)

```
   -- 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 automatically define the `Tests` child package that can be directly instantiated as follows:

```
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.

CHAPTER

EIGHT

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's 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 some restricted profiles, dynamic memory management is not provided as part of the run-time, and should not be used unless you have provided implementations as described in the GNAT User's Guide Supplement for GNAT Pro Safety-Critical and GNAT Pro High-Security.

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, an array of 100KB is allocated. The size can be changed by modifying the value in the file `aunit-<version>-src/aunit/framework/staticmemory/aunit-memory.adb` before building AUnit.

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

Additional restrictions relevant to the default ZFP profile include:

- 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. In order to be compatible with this restriction, the user-defined last chance handler routine can be defined as a "weak" symbol; this way, it will still be linked into the standalone executable, but will be replaced by the AUnit implementation when linked with the harness. The pragma `Weak_External` can be used for that; e.g.:

```
pragma Weak_External (Last_Chance_Handler);
```

- 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.
- 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.
- The timed parameter of the Harness `Run` routine has no effect when used with the ZFP profile, and on profiles not supporting `Ada.Calendar`.

This page is intentionally left blank.

CHAPTER

NINE

INSTALLATION AND USE

AUnit 3 contains support for restricted runtimes such as the zero-foot-print (ZFP) and certified (cert) profiles. It can now be installed simultaneously for several targets and runtimes.

9.1 Note on gprbuild

In order to compile, install and use AUnit, you need *gprbuild* and *gprinstall* version 2.2.0 or above.

9.2 Support for other platforms/run-times

AUnit should be built and installed separately for each target and runtime it is meant to be used with. The necessary customizations are performed at AUnit build time, so once the framework is installed, it is always referenced simply by adding the line

```
with "aunit";
```

to your project.

9.3 Installing AUnit

Normally AUnit comes preinstalled and ready-to-use for all runtimes in your GNAT distribution. The following instructions are for rebuilding it from sources for a custom configuration that the user may have.

- Extract the archive:

```
$ gunzip -dc aunit-<version>-src.tgz | tar xf -
```

- To build AUnit for a full Ada run-time:

```
$ cd aunit-<version>-src
$ make
```

- To build AUnit for a ZFP run-time targeting powerpc-elf platform:

```
$ cd aunit-<version>-src
$ make TARGET=powerpc-elf RTS=zfp
```

- To build AUnit for a reconfigurable runtime zfp-leon3 targeting leon3-elf platform:

```
$ cd aunit-<version>-src
$ make TARGET=leon3-elf RTS=zfp RTS_CONF="--RTS=zfp-leon3"
```

Once the above build procedure has been performed for the desired platform, you can install AUnit:

```
$ make install INSTALL=<install-root>
```

We recommend that you install AUnit into the standard location used by *gprbuild* to find the libraries for a given configuration. For example for the case above (runtime *zfp-leon3* targeting *leon3-elf*), the default location is *<gnat-root>*/leon3-elf/zfp-leon3. If the runtime is located in a custom directory and specified by the full path, using this exact path also as *<install_root>* is a sensible choice.

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, set to gprbuild's base installation directory as found in the PATH.
  - 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 both the run-time used to compile AUnit and the value given to the AUnit project as RUNTIME scenario variable.
  - RTS_CONF: defines the *gprbuild* Runtime config flag. The value is set to --RTS=$(RTS) by default. Can be used when compiling AUnit for a configurable run-time.

- 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

9.4 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

CHAPTER

TEN

GPS SUPPORT

The GPS IDE relies on the *gnattest* tool that creates unit-test skeletons as well as a test driver infrastructure (harness). A harness can be generated for a project hierarchy, a single project or a package. The generation process can be launched from the *Tools* -> *GNATtest* menu or from a contextual menu.

This page is intentionally left blank.

APPENDIX

A

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Copyright 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.