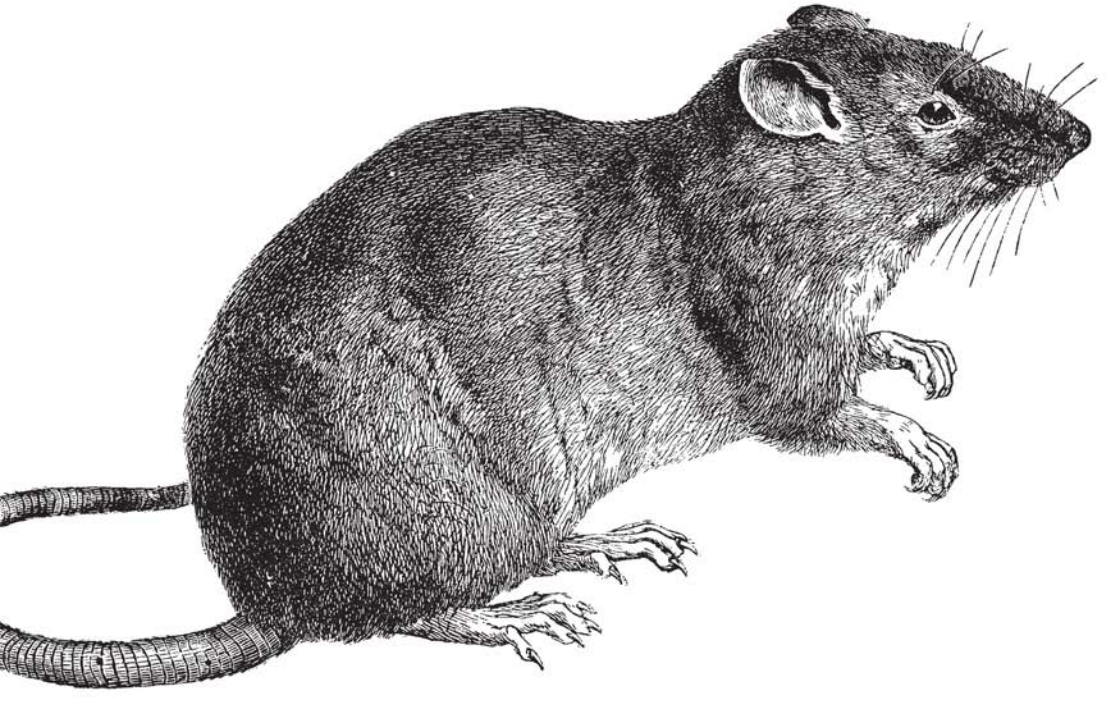
A Language-Independent Overview

Unit Test Frameworks



O'REILLY®

Paul Hamill

O'REILLY®

Unit Test Frameworks

Most people who write software have at least some experience with unit testing—even if they don't call it that. If you have ever written a few lines of throwaway code just to try something out, you've built a unit test. On the other end of the software spectrum, many large-scale applications have huge batteries of test cases that are repeatedly run and added to throughout the development process.

What are unit test frameworks and how are they used? Simply stated, they are software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the tests and report their results. They are not solely tools for testing; they can also be used as development tools on a par with preprocessors and debuggers. Unit test frameworks can contribute to almost every stage of software development and are key tools for doing Agile Development and building bug-free code.

Unit Test Frameworks covers the usage, philosophy, and architecture of unit test frameworks. The tutorials and code examples explain:

- Building a basic unit test framework from scratch
- Working with the xUnit family of test frameworks, including JUnit, CppUnit, NUnit, XMLUnit, and PyUnit
- Constructing both simple and complex unit tests

Tutorials and example code are platform-independent and compatible with Windows, Mac OS X, Unix, and Linux. The companion CD includes complete versions of JUnit, CppUnit, NUnit, and XMLUnit, as well as the complete set of code examples.

Paul Hamill has more than 10 years of experience developing code using C/C++, Java, and other languages. He has a BS in mechanical engineering from Cornell University and an MS in electrical engineering from the University of Colorado, and is the published coauthor of several academic papers on advanced CAD software. His recent experience includes work on a number of small entrepreneurial software development teams relying on eXtreme Programming (XP) and unit testing methodologies.

www.oreilly.com

US $29.95 **CAN $43.95**

ISBN: 978-0-596-00689-1

9 780596 006891 52995

Unit Test Frameworks

Other Resources from O'Reilly

Related titles

JUnit Pocket Guide
NUnit Pocket Reference
Extreme Programming Pocket Guide
Java Extreme Programming Cookbook

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

Unit Test Frameworks

Paul Hamill

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Unit Test Frameworks
by Paul Hamill

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Hendrickson

Production Editor: Mary Brady

Cover Designer: Ellie Volckhausen

Interior Designer: Melanie Wang

Printing History:

November 2004: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00689-6
[M]

Table of Contents

Preface

This book presents a comprehensive review of the xUnit family of unit test frameworks, including their usage, architecture, and theory. We begin by building a simple unit test framework from the ground up. The xUnit architecture is presented, using the JUnit framework as the reference implementation of xUnit. We progressively build an example application to demonstrate common practices and patterns of unit test development. Several popular versions of xUnit, including JUnit, CppUnit, NUnit, PyUnit, and XMLUnit, are covered in detail. Detailed class references are provided for JUnit and CppUnit as appendixes.

As a software development methodology, unit testing incorporates many rules and guidelines. However, writing unit tests is an art, not a science. Once you are familiar with the unit test driven approach to development, rigidly following its rules is optional. The true value of unit testing is in the focus on low-level software quality it gives developers, rather than as a formal process.

Audience

This book is intended for software developers, technical managers, and quality assurance staff who are learning about unit testing and agile development. Agile development is the wave of the future in software engineering, and many technical organizations are adopting it. Using unit test frameworks to enable test driven development is a key to becoming agile.

Contents of This Book

Here is a summary of the topics covered in each chapter and appendix:

Chapter 1, *Unit Test Frameworks: An Overview*
: An overview that explains what unit test frameworks are and how they are used.

Chapter 2, *Getting Started: Tutorial*
: A tutorial that creates a simple Java test framework. This provides the fundamentals of how unit test frameworks work. Appendix A contains the C++ version of this simple framework tutorial.

Chapter 3, *The xUnit Family of Unit Test Frameworks*
: A review of xUnit, using JUnit as a reference implementation to demonstrate basic xUnit architecture and usage.

Chapter 4, *Writing Unit Tests*
: An overview of writing unit tests. This offers a more detailed discussion of different types of unit tests and patterns of unit test development.

Chapter 5, *Unit Testing GUI Applications*
: A discussion of unit testing of GUI applications. This chapter explains how to build and test GUI objects following the smart object model.

Chapter 6, *JUnit*
: A description of the details of the usage and architecture of JUnit for Java.

Chapter 7, *CppUnit*
: A description of the details of the usage and architecture of CppUnit for C++.

Chapter 8, *NUnit*
: A description of the details of the usage and architecture for NUnit for .NET.

Chapter 9, *PyUnit*
: A description of the details of the usage and architecture of PyUnit for Python.

Chapter 10, *XMLUnit*
: A description of the details of the usage and architecture of XMLUnit for XML.

Chapter 11, *Resources*
: A list of additional resources for unit test frameworks and related topics.

Appendix A, *Simple C++ Unit Test Framework*
: The C++ version of the simple unit test framework from Chapter 2.

Appendix B, *JUnit Class Reference*
: A detailed class reference for JUnit's key package junit.framework.

Appendix C, *CppUnit Class Reference*
: A detailed class reference for CppUnit.

Glossary
: A list of definitions for important technical terms used in this book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text
: Indicates regular text and descriptions.

`Constant width`
: Indicates commands, methods, attributes, data types, class names, or the output from commands. It also shows the actual source code.

Italic
: Indicates new terms where they are defined, pathnames, file directories, filenames, and Internet names, such as email addresses, and URLs.

`Constant Width Bold`
: Indicates source code that is being emphasized for your attention.

Code in this book is formatted as shown here to distinguish it from the rest of the text. Code examples begin with the filename where the code resides.

```
MyClass.java
public class MyClass {

    myMethod( ) {
        int id = 3;
    }

}
```

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Unit Test Frameworks*, by Paul Hamill. Copyright 2005 O'Reilly Media, Inc., 0-596-00689-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/unitest/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

My sincere thanks go out to my reviewers: Ron Jeffries, James Newkirk, Philip Plumlee, J. B. Rainsberger, Simon Robbie, and Anthony Williams. Their shared experience and advice was incredibly useful and encouraging. This book could not have been completed without their help.

This book is built on the work of software pioneers. Kent Beck is the original author of the xUnit architecture in the form of SmalltalkUnit. Ward Cunningham, Kent Beck, and Ron Jeffries are the formulators of the Extreme Programming methodology, which led to many of the test driven development practices described in this book. Erich Gamma and Kent Beck ported SmalltalkUnit to Java to create JUnit, the most widely used and extended unit test framework. Many individual developers created and contributed to the different versions of xUnit, which are classic examples of open source software, built by the collective efforts of the software development community. The fingerprints of these talented engineers are all over the material covered by this book.

CHAPTER 1
Unit Test Frameworks: An Overview

Most people who write software have at least some experience with unit testing. If you have ever written a few lines of throwaway code just to try something out, you've built a unit test. On the other end of the software spectrum, many large-scale applications have huge batteries of test cases that are repeatedly run and added to throughout the development process. Unit tests are useful at all levels of programming.

What are unit test frameworks and how are they used? Simply stated, they are software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the tests and report their results. They are not solely tools for testing; they can also be used as development tools on a par with preprocessors and debuggers. Unit test frameworks can contribute to almost every stage of software development, including software architecture and design, code implementation and debugging, performance optimization, and quality assurance.

Unit tests usually are developed concurrently with production code, but are not built into the final software product. The relationship of unit tests to production code is shown in Figure 1-1.

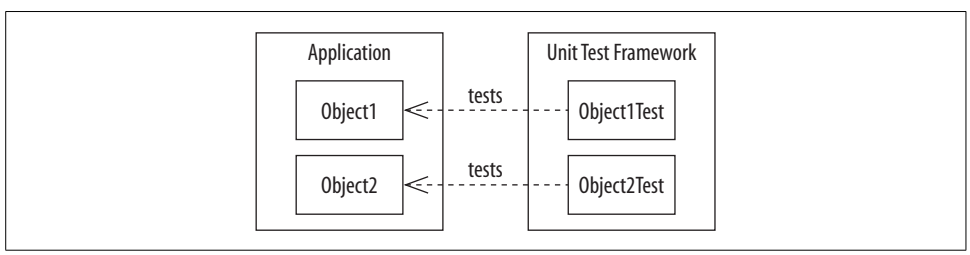


Figure 1-1. Production application and unit test framework

An application is built from software objects linked together. The unit tests use the application's objects, but exist inside the unit test framework. This approach has a

number of nice aspects. The production code is not cluttered up with built-in unit tests. The size of the compiled application tends to be kept smaller for the same reason. The tests can be run separately from the application, so the objects can be tested in isolation.

A single unit test should test a particular behavior within the production code. Its success or failure validates a single unit of code. Well-written tests set up an environment or scenario that is independent of any other conditions, then perform a distinct action and check a definite result. These tests should avoid dependencies on the results of other tests (called *test coupling*), and they should be short and simple. By starting with tests of the most basic functionality, then gradually building to tests of compound objects and behaviors, a unit test framework can be used to verify very complex architectures. Having such a test framework to build upon not only is much easier than developing standalone tests, but also produces more thorough, effective tests. A comprehensive suite of unit tests enables rapid application development, since the effects of every change can be immediately and thoroughly verified.

In the traditional jargon of testing, tests are categorized as *black box* or *white box*, depending on the amount of access to the internal workings of whatever is being tested. *Functional* and *structural* tests are related ideas. For example, a test that simply runs a program and checks its return code is a black box (functional) test, since nothing is known about how the program is written. Unit tests are usually white box (structural) tests, since the test framework is able to access the internal structure of the code being tested. Most object-oriented languages provide access protection, preventing outside classes from accessing protected or private code elements. Because of this, unit tests often are written to test only the public interfaces of the objects tested. This encourages the design of objects with discrete, testable interfaces and a minimum of complex hidden behavior. Thus, writing testable objects promotes good object-oriented development practices.

Another distinction is drawn between *programmer* and *acceptance tests*. Developers write programmer tests as they design and build code. These usually test low-level code elements, such as methods and interfaces. Acceptance tests may be specified or written by a nonprogrammer, such as a quality-assurance person or product manager. These generally are functional tests of high-level behavior, such as producing output or performing a user task. Unit tests may fall into either of these categories.

Test Driven Development

Unit test frameworks are a key element of Test Driven Development (TDD), also known as "test-first programming." TDD is one of the most significant and widely used practices in Extreme Programming (XP) and other Agile Development methodologies. Test frameworks achieve their maximum utility when used to enable TDD, although they still are useful when TDD is not followed. This book concentrates on

unit test frameworks as a family of tools, rather than specifically on TDD, but the two topics are closely related.

The key rule of TDD can be summarized as "test twice, code once," by analogy to the carpenter's rule of "measure twice, cut once." "Test twice, code once" refers to the three-step procedure involved in any code change:

1. Write a test of the new code and see it fail.
2. Write the new code, doing "the simplest thing that could possibly work."
3. See the test succeed, and refactor the code.

These three basic steps are the *TDD cycle*.

Step 1 is to write a test, run it, and verify the resulting failure. The failure is important because it validates that the test fails as expected. It is often tempting to skip running the test and seeing the failure. Don't.

In Step 2, code is written to make the test succeed. A wise guideline is doing "the simplest thing that could possibly work." This may be a completely trivial implementation, such as having the new code return a constant value or copying and pasting code from one place to another. It doesn't have to be pretty; it just has to pass the test. The temptation in this step is to do a little extra work and make some additional code change not directly related to passing the test. Again, don't do this.

In Step 3, the test succeeds, verifying both the new code and its test. At this point, the new code may be refactored. *Refactoring* is a software engineering concept defined as "behavior-preserving transformation." More formally, refactoring is the process of transforming code to improve its internal design without changing its external functionality. Within the TDD cycle, refactoring starts with the inelegant code that was written to pass the unit test and improves it by removing duplication or other ugliness. Since the unit test is in place, the details of how the code is implemented can be altered with confidence.

New code should only be written when a test fails. Code changes are only expected to occur when you are refactoring, adding new functionality, or debugging. Continuously repeating the TDD cycle is the most atomic level of the software development process. Software changes generally fall under two categories: adding new functionality or fixing bugs.

When adding new functionality, the first step is always to write a unit test that anticipates and uses the new code. After the unit test runs and fails, add the new code and re-test to verify success. The unit test has value aside from simply demonstrating that the new functionality works. Writing the test forces you to think in advance about the ideal design of the new code. Thus, in a sneaky and subtle way, TDD makes all new development part of a methodical, low-level software design process. Once the new unit test and functionality are in place, the unit test serves as the definitive, working example of how the new code is supposed to be used. For these reasons,

time spent writing unit tests is not solely testing effort. Investments in testing are equal investments in design.

When debugging, you should first write a unit test that fails because of the bug. This is a useful effort in itself, because it determines exactly how the bug occurs. Once the unit test is in place and failing, fix the bug and re-run the test to verify that the bug is closed. Aside from fixing the bug, this process has the additional benefit of creating a test that will catch it. If the bug is ever re-introduced, the test will fail and highlight the problem.

By following the TDD cycle, you can come as close as humanly possible to writing flawless code on the first try—in other words, "code once." The process gives you a clear indication that a piece of work is done. When a new unit test is written and then fails, the task is halfway completed. You cannot move on to something else until the test succeeds.

Unit Testing and Quality Assurance

Unit test frameworks are valuable when used for automated software testing as part of a quality assurance (QA) process. In many software development groups, the QA process starts when new code is submitted, built, and unit tested. Often, the unit tests include not only programmer tests, but also acceptance tests designed or written by the QA team. If all the unit tests succeed, the code is provisionally accepted and sent to a QA engineer for inspection and testing.

Running the full suite of unit tests as the first step in QA has many benefits. Most importantly, the tests ensure that the code is solid the moment it has left the developers' hands. No human intervention is required to run the tests and evaluate the results. Either they all succeed, or there is a failure. Such Boolean (true/false) results are ideal because an automated system can understand them. The success of the unit tests confirms that the developers' assumptions are valid, and that the low-level functionality is working correctly at a level of scrutiny that functional tests can never achieve. When numerous developers are making changes at once, the unit tests provide confidence that nobody's changes caused someone else's code to break. Furthermore, unit tests help to provide accountability. Knowing exactly which test fails usually makes it apparent whose change broke things. "Breaking the build" once meant submitting code that caused a compile to fail, but now often refers to causing a unit test failure as well. Many teams employ heinous punishments (such as making the responsible developer buy donuts or beer for coworkers) to remind everyone that breaking the build is a serious offense. The failure of a unit test clearly places a high priority on fixing the problem. If TDD is followed rigorously, the code should never be left in a state in which a unit test fails.

Unit testing doesn't replace all other types of testing. It is entirely possible to develop thoroughly unit-tested, completely bulletproof code that is lacking in usability and

performance. Stress testing, performance testing, and usability testing usually are separate considerations from unit testing. QA effort is still necessary to try out the completed application, decide whether it performs acceptably in real-world conditions, observe how things work outside of a controlled development environment, and otherwise apply human judgment. There are elements of software functionality for which it is difficult or impossible to write good unit tests. These include GUI "look and feel," responses to system events, interaction with distributed application components, and many other possibilities. Sometimes unit tests can be written to simulate these types of situations, but ultimately, there is no substitute for reality or for a user's objective feedback.

Although manual QA testing is still important, unit tests are a powerful tool for QA. Developers who use test-centric development report dramatic improvements in software quality, speed of development, and ability to make significant design changes on the fly. These speed and quality advantages rapidly become apparent from the QA perspective as well.

Homegrown Unit Testing

Writing simple tests comes naturally to most programmers. The classic beginner exercise of writing a three-line program that prints "Hello world!" is a basic unit test of the development language and environment. Find a software shop with no unit test framework in place (if such a prehistoric place could possibly exist), and you may see developers writing their own little "toy programs" or "test utilities" to try out new code. The sad thing about this approach is that the toy programs are thrown away once the developer is done with them. Later, when something breaks, someone has to laboriously debug the production code, without benefit of the developer's test.

Another common low-level testing technique is to build tests into the production code with ASSERT macros. In debug builds, the macro tests a condition and sends a message if it fails. In release builds, the macro is defined to be empty, so no test code is included. This allows a developer to sprinkle assertions throughout the code, reporting any condition that is worthy of someone's attention. Asserts can be a useful thing to have in your software toolbox, but far less so than true unit testing. For an assert to be evaluated, the production code must be run to the point where it is defined. It's not convenient for automated testing, since an automated system doesn't know how to cause a particular assert to fire. Failures don't leave the developer with a clear path to correct the problem. Fixing a failure is no guarantee that the same problem will not happen again under different circumstances. Reliance on testing with this type of assert is unlikely to produce high-quality software. It is a forerunner to formal unit testing, which uses test asserts contained within well-defined tests, rather than placed randomly in the production code.

Just as many developers take the initiative and write test programs to try out small pieces of code, it's common to find developers putting together basic, home-grown unit test frameworks that take care of their testing needs. As demonstrated in the Chapter 2, a test framework can be just a few lines of code to run unit tests and report the results. Even a very simple framework can be the foundation for thorough testing of complex applications.

CHAPTER 2
Getting Started: Tutorial

Software concepts are best explained by example. In this tutorial, you will set up a simple unit test framework and use it to help build a basic application. Following the primary rule of TDD, every change to the code is preceded by a unit test.

Why build our own test framework, instead of starting with one of the xUnits? The xUnit test frameworks are powerful tools. They not only support writing unit tests, running them, and reporting the results, but also include test classes, helper code, test runners, and utilities. Such features minimize the amount of code required to write a unit test and maximize your ability to test complex code. They include much more than the minimum needed to build unit tests.

The core functionality of running tests and reporting the results is fundamentally simple. Developers working in cross-platform environments, using older compilers or uncommon languages or needing closer control over how unit tests and their results are handled may not be able to use the xUnits or want to invest the time to set them up. The proliferation of very basic unit test frameworks available online demonstrates the popular belief that "simpler is better" when it comes to unit test frameworks. Most importantly, creating your own framework clearly demonstrates how unit tests work and how straightforward the unit test framework concept really is.

The example code is given in Java. Appendix A contains the C++ version. The code can be found on the CD accompanying this book in the directory */examples/chapter2*. Consider entering the code in this chapter by hand as if you were coding it from scratch. It's an illuminating exercise that will help you to understand how quick and easy it is to set up and start using a unit test framework.

This tutorial assumes that you have a Java runtime environment and compiler installed. Sun's *javac* compiler is recommended, as is the GNU *gcj* Java compiler. Versions of both compilers are readily available for most platforms.

The step-by-step procedures given here assume that you are compiling and running the code from the command line. If you are using a graphical Integrated Development Environment (IDE), the details of how you build and run the example code will differ.

Outline of an Application: the Virtual Library

This book presents an increasingly complex series of code examples to illustrate unit test framework usage. The examples fit into an overall system concept. Your mission, should you choose to accept it, is to build a system for managing a library full of books. Books will have the attributes you might expect, such as title and author. Users of the system will need to be able to perform a variety of library operations: adding new books, searching for books, checking them in and out of the library, and so forth.

Example 1: Create a Book

For the first example, we will create a representation of a book and its title. Since we'll do test-first development, we need to set up a unit test framework prior to writing any code for the book class. This test framework serves both as the foundation for the example's unit tests and also as an illustration of just how simple a functional test framework can be. Building it is Step 0.

The subsequent steps are the usual three steps in the TDD cycle. Step 1 is to write a unit test to verify that a book has been created. At first, the unit test will fail, because the functionality to create a book does not yet exist. Step 2 is to build the functionality to create a book. In Step 3, the test succeeds, proving that the functionality works and providing an example of how to use it.

Step 0: Set Up the Unit Test Framework

The unit test framework initially is built on a single class, `UnitTest`, shown in Figure 2-1.

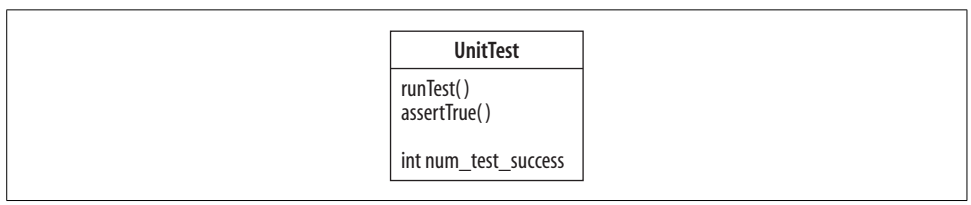


Figure 2-1. The class UnitTest

The source code for `UnitTest` is given in Example 2-1.

Example 2-1. The base class UnitTest

UnitTest.java

```
public abstract class UnitTest {

   protected static int num_test_success = 0;
```