JUnit IN ACTION

Vincent Massol

with Ted Husted

MANNING

JUnit in Action

JUnit in Action

VINCENT MASSOL
with TED HUSTED

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue   Fax: (203) 661-9018
Greenwich, CT 06830   email: orders@manning.com

Manning Publications Co.   Copyeditor: Tiffany Taylor
209 Bruce Park Avenue   Typesetter: Tony Roberts
Greenwich, CT 06830   Cover designer: Leslie Haimes

ISBN 1-930110-99-5

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – VHG – 06 05 04 03

contents

preface

To date tests are still the best solution mankind has found to deliver working software. This book is the sum of four years of research and practice in the testing field. The practice comes from my IT consulting background, first at Octo Technology and then at Pivolis; the research comes from my involvement with open source development at night and on weekends.

Since my early programming days in 1982, I've been interested in writing tools to help developers write better code and develop more quickly. This interest has led me into domains such as software mentoring and quality improvement. These days, I'm setting up continuous-build platforms and working on development best practices, both of which require strong suites of tests. The closer these tests are to the coding activity, the faster you get feedback on your code—hence my interest in unit testing, which is so close to coding that it's now as much a part of development as the code that's being written.

This background led to my involvement in open source projects related to software quality:

- Cactus for unit-testing J2EE components (http://jakarta.apache.org/cactus/)
- Mock objects for unit-testing any code (http://www.mockobjects.com/)
- Gump for continuous builds (http://jakarta.apache.org/gump/)
- Maven for builds and continuous builds (http://maven.apache.org/)

- The Pattern Testing proof of concept for using Aspect-Oriented Programming (AOP) to check architecture and design rules (http://patterntesting.sf.net/).[1]

JUnit in Action is the logical conclusion to this involvement.

Nobody wants to write sloppy code. We all want to write code that works—code that we can be proud of. But we're often distracted from our good intentions. How often have you heard this: "We wanted to write tests, but we were under pressure and didn't have enough time to do it"; or, "We started writing unit tests, but after two weeks our momentum dropped, and over time we stopped writing them."

This book will give you the tools and techniques you need to happily write quality code. It demonstrates in a hands-on fashion how to use the tools in an effective way, avoiding common pitfalls. It will empower you to write code that works. It will help you introduce unit testing in your day-to-day development activity and develop a rhythm for writing robust code.

Most of all, this book will show you how to control the entropy of your software instead of being controlled by it. I'm reminded of some verses from the Latin writer Lucretius, who, in 94–55 BC wrote in his *On the Nature of Things* (I'll spare you the original Latin text):

> Lovely it is, when the winds are churning up the waves on the great sea, to gaze out from the land on the great efforts of someone else; not because it's an enjoyable pleasure that somebody is in difficulties, but because it's lovely to realize what troubles you are yourself spared.

This is exactly the feeling you'll experience when you know you're armed with a good suite of tests. You'll see others struggling, and you'll be thankful that you have tests to prevent anyone (including yourself) from wreaking havoc in your application.

Vincent Massol
Richeville (close to Paris), France

[1] As much as I wanted to, I haven't included a chapter on unit-testing code using an AOP framework. The existing AOP frameworks are still young, and writing unit tests with them leads to verbose code. My prediction is that specialized AOP/unit-testing frameworks will appear in the very near future, and I'll certainly cover them in a second edition. See the following entry in my blog about unit-testing an EJB with JUnit and AspectJ: http://blogs.codehaus.org/people/vmassol/archives/000138.html.

acknowledgments

This book was one year in the making. I am eternally grateful to my wife, Marie-Albane, and my kids, Pierre-Olivier and Jean. During that year, they accepted that I spent at least half of my free time writing the book instead of being with them. I had to promise that I won't write another book… for a while….

Thank you to Ted Husted, who stepped up to the plate and helped make the first part of the book more readable by improving on my English, reshuffling the chapters, and adding some parsley here and there where I had made shortcuts.

JUnit in Action would not exist without Kent Beck and Erich Gamma, the authors of JUnit. I thank them for their inspiration; and more specially I thank Erich, who agreed to read the manuscript while under pressure to deliver Eclipse 2.1 and who came up with a nice quote for the book.

Again, the book would not be what it is without Tim Mackinnon and Steve Freeman, the original creators of the mock objects unit-testing strategy, which is the subject of a big part of this book. I thank them for introducing me to mock objects while drinking beer (it was cranberry juice for me!) at the London Extreme Tuesday Club.

The quality of this book would not be the same without the reviewers. Many thanks to Mats Henricson, Bob McWhirter, Erik Hatcher, William Brogden, Brendan Humphreys, Robin Goldsmith, Scott Stirling, Shane Mingins, and Dorothy Graham. I'd like to express special thanks to Ilja Preuß, Kim Topley, Roger D. Cornejo, and J. B. Rainsberger, who gave extremely thorough review comments and provided excellent suggestions.

With this first book, I have discovered the world of publishing. I have been extremely impressed by Manning's professionalism and obsession with perfection. Whenever I thought the book was done and I could relax, it had to go through another phase of verifications of some sort! Many thanks to publisher Marjan Bace for his continuing trust even though I kept pushing the delivery date. Developmental editor Marilyn Smith was an example of responsiveness, giving me back corrected chapters and suggestions just a few hours after I submitted chapters. Copy editor Tiffany Taylor fixed an incredible number of errors (I could almost not recognize my chapters after Tiffany stormed through them). Tony Roberts had the hard task of typesetting the book and supporting my numerous requests; thanks, Tony. Technical proofreader Robert McGovern did an excellent job of catching all my technical mistakes.

Last but not least, a big thank-you to Francois Hisquin, CEO of Octo Technology and Pivolis, the two companies I have been working for while writing *JUnit in Action*, for letting me write some parts of the book during the day!

about this book

JUnit in Action is an example-driven, how-to book on unit-testing Java applications, including J2EE applications, using the JUnit framework and its extensions. This book is intended for readers who are software architects, developers, members of testing teams, development managers, extreme programmers, or anyone practicing any agile methodology.

JUnit in Action is about solving tough real-world problems such as unit-testing legacy applications, writing real tests for real objects, employing test metrics, automating tests, testing in isolation, and more.

Special features

Several special features appear throughout the book.

Best practices

The JUnit community has already adopted several best practices. When these are introduced in the book, a callout box summarizes the best practice.

Design patterns in action

The JUnit framework puts several well-known design patterns to work. When we first discuss a component that makes good use of a design pattern, a callout box defines the pattern and points out its use in the JUnit framework.

Software directory

Throughout the book, we cover how to use extensions and tools with JUnit. For your convenience, references to all of these software packages have been collected in a directory in the references section at the end of this book. A bibliography of other books we mention is also provided in the references section.

Roadmap

The book is divided into three parts. Part 1 is "JUnit distilled." Here, we introduce you to unit testing in general and JUnit in particular. Part 2, "Testing strategies," investigates different ways of testing the complex objects found in professional applications. Part 3, "Testing components," explores strategies for testing common subsystems like servlets, filters, JavaServer Pages, databases, and even EJBs.

Part 1: JUnit distilled

Chapter 1 walks through creating a test for a simple object. We introduce the benefits, philosophy, and technology of unit testing along the way. As the tests grow more sophisticated, we present JUnit as the solution for creating better tests.

Chapter 2 delves deeper into the JUnit classes, life cycle, and architecture. We take a closer look at the core classes and the overall JUnit life cycle. To put everything into context, we look at several example tests, like those you would write for your own classes.

Chapter 3 presents a sophisticated test case to show how JUnit works with larger components. The subject of the case study is a component found in many applications: a controller. We introduce the case-study code, identify what code to test, and then show how to test it. Once we know that the code works as expected, we create tests for exceptional conditions, to be sure the code behaves well even when things go wrong.

Chapter 4 looks at the various types of software tests, the role they play in an application's life cycle, how to design for testability, and how to practice test-first development.

Chapter 5 explores the various ways you can integrate JUnit into your development environment. We look at automating JUnit with Ant, Maven, and Eclipse.

Part 2: Testing strategies

Chapter 6 describes how to perform unit tests using stubs. It introduces a sample application that connects to a web server and demonstrates how to unit-test the method calling the remote URL using a stub technique.

Chapter 7 demonstrates a technique called mock objects that lets you unit test code in isolation from the surrounding domain objects. This chapter carries on with the sample application (opening an HTTP connection to a web server); it shows how to write unit tests for the application and highlights the differences between stubs and mock objects.

Chapter 8 demonstrates another technique which is useful for unit-testing J2EE components: in-container testing. This chapter covers how to use Cactus to run unit tests from within the container. In addition, we explain the pros and cons of using an in-container approach versus a mock-objects approach, and when to use each.

Part 3: Testing components

Chapter 9 shows how to unit-test servlets and filters using both the mock-objects approach and the in-container approach. It highlights how they complement each other and gives strategies on when to use them.

Chapter 10 carries us into the world of unit-testing JSPs and taglibs. It shows how to use the mock-objects and in-container strategies.

Chapter 11 touches on a difficult but crucial subject: unit-testing applications that call databases using JDBC. It also demonstrates how to unit-test database code in isolation from the database.

Chapter 12 investigates how to unit-test all kind of EJBs using mock objects, pure JUnit test cases, and Cactus.

Code

The source code for the examples in this book has been donated to the Apache Software Foundation. It is available on SourceForge (http://sourceforge.net/projects/junitbook/). A link to the source code is also provided from the book's web page at http://www.manning.com/massol. Check appendix A for details on how the source code is organized and for software version requirements.

The Java code listings that we present have the Java keywords shown in bold to make the code more readable. In addition, when we highlight changes in a new listing, the changes are shown in bold font to draw attention to them. In that case, the Java keywords are displayed in standard, non-bold code font. Often, numbers and annotations appear in the code. These numbers refer to the discussion of that portion of the code directly following the listing.

In the text, a monotype font is used to denote code (JSP, Java, and HTML) as well as Java methods, JSP tag names, and most other source code identifiers:

- A reference to a method in the text may not include the signature because there may be more than one form of the method call.

- A reference to an XML element or JSP tag in the text usually does not include the braces or the attributes.

References

Bibliographic references are indicated in footnotes or in the body of text. Full publication details and/or URLs are provided in the references section at the end of this book. Web site URLs are given in the text of the book and cross-referenced in the index.

Author online

Purchase of *JUnit in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to http://www.manning.com/massol. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

about the authors

Vincent Massol is the creator of the Jakarta Cactus framework. He is also an active member of the Maven, Gump, and MockObjects development teams. After having spent four years as a technical architect on several major projects (mostly J2EE), Vincent is now the co-founder and CTO of Pivolis, a company specialized in applying agile methodologies to offshore software development. A consultant and lecturer during the day and open source developer at night, Vincent currently lives in Paris, France. He can be contacted through his blog at http://blogs.codehaus.org/people/vmassol/.

Ted Husted is an active member of the Struts development team, manager of the JGuru Struts Forum, and the lead author of *Struts in Action*.[2] As a consultant, lecturer, and trainer, Ted has worked with Java development teams throughout the United States. Ted's latest development project used test-driven development throughout and is available as open source (http://sourceforge.net/projects/wqdata/). Ted lives in Fairport, NY, with his wife, two children, four computers, and an aging cat.

[2] Ted Husted, Cedric Dumoulin, George Franciscus, and David Winterfeldt, *Struts in Action* (Greenwich, CT: Manning, 2002).

about the title

Manning's *in Action* books combine an overview with how-to examples to encourage learning *and* remembering. Cognitive science tells us that we remember best through discovery and exploration. At Manning, we think of exploration as "playing." Every time computer scientists build a new application, we believe they play with new concepts and new techniques—to see if they can make the next program better than the one before. An essential element of an *in Action* book is that it is example-driven. *In Action* books encourage the reader to play with new code and explore new ideas. At Manning, we are convinced that permanent learning comes through exploring, playing, and most importantly, *sharing* what we have discovered with others. People learn best *in action.*

There is another, more mundane, reason for the title of this book: Our readers are busy. They use books to do a job or solve a problem. They need books that allow them to jump in and jump out easily—books that will help them *in action.* The books in this series are designed for these "impatient" readers. You can start reading an *in Action* book at any point, to learn just what you need just when you need it.

about the cover illustration

The figure on the cover of *JUnit in Action* is a "Burco de Alpeo," taken from a Spanish compendium of regional dress customs first published in Madrid in 1799. The book's title page states:

> *Coleccion general de los Trages que usan actualmente todas las Nacionas del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en special para los que tienen la del viajero universal*

which we translate, as literally as possible, thus:

> *General collection of costumes currently used in the nations of the known world, designed and printed with great exactitude by R.M.V.A.R. This work is very useful especially for those who hold themselves to be universal travelers*

Although nothing is known of the designers, engravers, and workers who colored this illustration by hand, the "exactitude" of their execution is evident in this drawing, which is just one of many in this colorful collection. Their diversity speaks vividly of the uniqueness and individuality of the world's towns and regions just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The collection brings to life a sense of isolation and distance of that period‹and of every other historic period except our own hyperkinetic present. Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

At the time of publication, we were unable to decipher the meaning of the caption "Burco de Alpeo" but will keep you posted on our progress on the *JUnit in Action* web page. The first reader to come up with the correct translation will be thanked with a free copy of another Manning book of his or her choice. Please make postings to the Author Online forum at www.manning.com/massol.

Part 1

JUnit distilled

In part 1, you'll become test-infected! Through a simple example, chapter 1 will teach you what the JUnit framework is and what problems it solves. Chapter 2 will take you on a discovery tour of the core JUnit classes and how to best use them. In chapter 3, you'll practice your new JUnit knowledge on a real-world example. You'll also learn how to set up a JUnit project and how to execute the unit tests. Chapter 4 steps back and explains why unit test are important and how they fit in the global testing ecosystem. It also presents the Test-Driven Development methodology and provides guidance on measuring your test coverage. Chapter 5 demonstrates how to automate unit testing using three popular tools: Eclipse, Ant, and Maven.

At the end of part 1, you'll have a good general knowledge of JUnit, how to write unit tests, and how to run them easily. You'll be ready to start learning about the different strategies required to unit-test full-fledged applications: stubs, mock objects, and in-container testing.

1 JUnit jumpstart

This chapter covers

- Writing simple tests by hand
- Installing JUnit and running tests
- Writing better tests with JUnit

Never in the field of software development was so much owed by so many to so few lines of code.
—Martin Fowler

All code is tested.

During development, the first thing we do is run our own programmer's "acceptance test." We code, compile, and run. And when we run, we test. The "test" may just be clicking a button to see if it brings up the expected menu. But, still, every day, we code, we compile, we run…and *we test.*

When we test, we often find issues—especially on the first run. So we code, compile, run, and test again.

Most of us will quickly develop a pattern for our informal tests: We add a record, view a record, edit a record, and delete a record. Running a little test suite like this by hand is easy enough to do; so we do it. *Over and over again.*

Some programmers like doing this type of repetitive testing. It can be a pleasant break from deep thought and hard coding. And when our little click-through tests finally succeed, there's a real feeling of accomplishment: *Eureka! I found it!*

Other programmers dislike this type of repetitive work. Rather than run the test by hand, they prefer to create a small program that runs the test automatically. Play-testing code is one thing; running automated tests is another.

If you are a "play-test" developer, this book is meant for you. We will show you how creating automated tests can be easy, effective, and even fun!

If you are already "test-infected," this book is also meant for you! We cover the basics in part 1, and then move on to the tough, real-life problems in parts 2 and 3.

1.1 Proving it works

Some developers feel that automated tests are an essential part of the development process: A component cannot be *proven* to work until it passes a comprehensive series of tests. In fact, two developers felt that this type of "unit testing" was so important that it deserved its own framework. In 1997, Erich Gamma and Kent Beck created a simple but effective unit testing *framework* for Java, called JUnit. The work followed the design of an earlier framework Kent Beck created for Smalltalk, called SUnit.

DEFINITION *framework*—A framework is a semi-complete application.[1] A framework provides a reusable, common structure that can be shared between applications. Developers incorporate the framework into their own application and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure, rather than a simple set of utility classes.

If you recognize those names, it's for good reason. Erich Gamma is well known as one of the "Gang of Four" who gave us the now classic *Design Patterns* book.[2] Kent Beck is equally well known for his groundbreaking work in the software discipline known as Extreme Programming (http://www.extremeprogramming.org).

JUnit (junit.org) is open source software, released under IBM's Common Public License Version 1.0 and hosted on SourceForge. The Common Public License is business-friendly: People can distribute JUnit with commercial products without a lot of red tape or restrictions.

JUnit quickly became the de facto standard framework for developing unit tests in Java. In fact, the underlying testing model, known as xUnit, is on its way to becoming the standard framework for *any* language. There are xUnit frameworks available for ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk, and Visual Basic—just to name a few!

Of course, the JUnit team did not invent software testing or even the unit test. Originally, the term *unit test* described a test that examined the behavior of a single *unit of work*.

Over time, usage of the term *unit test* broadened. For example, IEEE has defined unit testing as "Testing of individual hardware or software units *or groups of related units*" (emphasis added).[3]

In this book, we use the term *unit test* in the narrower sense of a test that examines a single unit in isolation from other units. We focus on the type of small, incremental test that programmers apply to their own code. Sometimes these are called *programmer tests* to differentiate them from quality assurance tests or customer tests (http://c2.com/cgi/wiki?ProgrammerTest).

---

[1] Ralph Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* 1.5 (June/July 1988): 22–35; http://www.laputan.org/drc/drc.html.

[2] Erich Gamma et al., *Design Patterns* (Reading, MA: Addison-Wesley, 1995).

[3] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries* (New York: IEEE, 1990).

Here's a generic description of a typical unit test from our perspective: "Confirm that the method accepts the expected range of input, and that the method returns the expected value for each test input."

This description asks us to test the behavior of a method through its interface. If we give it value *x*, will it return value *y*? If we give it value *z* instead, will it throw the proper exception?

DEFINITION *unit test*—A unit test examines the behavior of a distinct *unit of work*. Within a Java application, the "distinct unit of work" is often (but not always) a single method. By contrast, *integration tests* and *acceptance tests* examine how various components interact. A *unit of work* is a task that is not directly dependent on the completion of any other task.

Unit tests often focus on testing whether a method is following the terms of its *API contract*. Like a written contract by people who agree to exchange certain goods or services under specific conditions, an API contract is viewed as a formal agreement made by the interface of a method. A method requires its callers to provide specific objects or values and will, in exchange, return certain objects or values. If the contract cannot be fulfilled, then the method throws an exception to signify that the contract cannot be upheld. If a method does not perform as expected, then we say that the method has broken its contract.

DEFINITION *API contract*—A view of an Application Programming Interface (API) as a formal agreement between the caller and the callee. Often the unit tests help define the API contract by demonstrating the expected behavior. The notion of an API contract stems from the practice of *Design by Contract*, popularized by the Eiffel programming language (http://archive.eiffel.com/doc/manuals/technology/contract).

In this chapter, we'll walk through creating a unit test for a simple class from scratch. We'll start by writing some tests manually, so you can see how we *used* to do things. Then, we will roll out JUnit to show you how the right tools can make life much simpler.

1.2 Starting from scratch

Let's say you have just written the `Calculator` class shown in listing 1.1.

Listing 1.1 The Calculator class

```
public class Calculator
{
  public double add(double number1, double number2)
  {
    return number1 + number2;
  }
}
```

Although the documentation is not shown, the intended purpose of the `Calculator`'s `add(double, double)` method is to take two `double`s and return the sum as a `double`. The compiler can tell you that it compiles, but you should also make sure it works at runtime. A ycore tenet of unit testing is: "Any program feature without an automated test simply doesn't exist."[4] The `add` method represents a core feature of the calculator. You have some code that allegedly implements the feature. What's missing is an automated test that proves your implementation works.

But isn't the add method "too simple to possibly break"?

The current implementation of the `add` method is too simple to break. If `add` were a minor utility method, then you might not test it directly. In that case, if `add` did fail, then tests of the methods that used `add` would fail. The `add` method would be tested indirectly, but tested nonetheless.

In the context of the calculator program, `add` is not just a method, it's a *program feature*. In order to have confidence in the program, most developers would expect there to be an automated test for the add feature, no matter how simple the implementation appears.

In some cases, you can prove program features through automatic functional tests or automatic acceptance tests. For more about software tests in general, see chapter 4.

Yet testing anything at this point seems problematic. You don't even have a user interface with which to enter a pair of `double`s. You could write a small command-line program that waited for you to type in two `double` values and then displayed the result. Of course, then you would also be testing your own ability to type a number and add the result ourselves. This is much more than you want to do. You just want to know if this "unit of work" will actually add two `double`s and return the correct sum. You don't necessarily want to test whether programmers can type numbers!

[4] Kent Beck, *Extreme Programming Explained: Embrace Change* (Reading, MA: Addison-Wesley, 1999).

Meanwhile, if you are going to go to the effort of testing your work, you should also try to preserve that effort. It's good to know that the `add(double,double)` method worked when you wrote it. But what you really want to know is whether the method works when you ship the rest of the application.

As shown in figure 1.1, if we put these two requirements together, we come up with the idea of writing a simple test program for the method. The test program could pass known values to the method and see if the result matches our expectations. You could also run the program again later to be sure the method continues to work as the application grows.

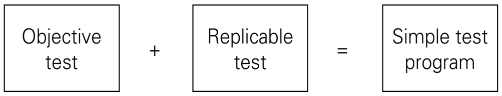


Figure 1.1 Justifying JUnit: Putting the two testing requirements together gives an idea for a simple test program.

So what's the simplest possible test program you could write? How about the simple `TestCalculator` program shown in listing 1.2?

Listing 1.2 A simple TestCalculator program

```
public class TestCalculator
{
  public static void main(String[] args)
  {
    Calculator calculator = new Calculator();
    double result = calculator.add(10,50);
    if (result != 60)
    {
      System.out.println("Bad result: " + result);
    }
  }
}
```

The first `TestCalculator` is simple indeed! It creates an instance of `Calculator`, passes it two numbers, and checks the result. If the result does not meet your expectations, you print a message on standard output.

If you compile and run this program now, the test will quietly pass, and all will seem well. But what happens if you change the code so that it fails? You will have to carefully watch the screen for the error message. You may not have to supply the input, but you are still testing your own ability to monitor the program's output. You want to test the code, not yourself!

The conventional way to handle error conditions in Java is to throw an exception. Since failing the test is an error condition, let's try throwing an exception instead.

Meanwhile, you may also want to run tests for other `Calculator` methods that you haven't written yet, like `subtract` or `multiply`. Moving to a more modular design would make it easier to trap and handle exceptions and make it easier to extend the test program later. Listing 1.3 shows a slightly better `TestCalculator` program.

Listing 1.3 A (slightly) better TestCalculator program

```
public class TestCalculator
{
  private int nbErrors = 0;

  public void testAdd()
  {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    if (result != 60)
    {
      throw new RuntimeException("Bad result: " + result);
    }
  }                                                                  ❶

  public static void main(String[] args)
  {
    TestCalculator test = new TestCalculator();
    try
    {
      test.testAdd();
    }
    catch (Throwable e)
    {
      test.nbErrors++;
      e.printStackTrace();
    }

    if (test.nbErrors > 0)
    {
      throw new RuntimeException("There were " + test.nbErrors
        + " error(s)");
    }
  }                                                                  ❷
}
```

Working from listing 1.3, at ❶ you move the test into its own method. It's now easier to focus on what the test does. You can also add more methods with more unit tests later, without making the main block harder to maintain. At ❷, you change the `main` block to print a stack trace when an error occurs and then, if there are any errors, to throw a summary exception at the end.

1.3 Understanding unit testing frameworks

There are several best practices that unit testing frameworks should follow. These seemingly minor improvements in the `TestCalculator` program highlight three rules that (in our experience) all unit testing frameworks should observe:

- Each unit test must run independently of all other unit tests.
- Errors must be detected and reported test by test.
- It must be easy to define which unit tests will run.

The "slightly better" test program comes close to following these rules but still falls short. For example, in order for each unit test to be truly independent, each should run in a different classloader instance.

Adding a class is also only slightly better. You can now add new unit tests by adding a new method and then adding a corresponding `try/catch` block to `main`.

A definite step up, but still short of what you would want in a *real* unit test suite. The most obvious problem is that large `try/catch` blocks are known to be maintenance nightmares. You could easily leave a unit test out and never know it wasn't running!

It would be nice if you could just add new test methods and be done with it. But how would the program know which methods to run?

Well, you could have a simple registration procedure. A registration method would at least inventory which tests are running.

Another approach would be to use Java's *reflection* and *introspection* capabilities. A program could look at itself and decide to run whatever methods are named in a certain way—like those that begin with the letters *test*, for example.

Making it easy to add tests (the third rule in our earlier list) sounds like another good rule for a unit testing framework.

The support code to realize this rule (via registration or introspection) would not be trivial, but it would be worthwhile. There would be a lot of work up front, but that effort would pay off each time you added a new test.

Happily, the JUnit team has saved you the trouble. The JUnit framework already supports registering or introspecting methods. It also supports using a different *classloader* instance for each test, and reports all errors on a case-by-case basis.

Now that you have a better idea of why you need unit testing frameworks, let's set up JUnit and see it in action.

1.4 Setting up JUnit

JUnit comes in the form of a jar file (`junit.jar`). In order to use JUnit to write your application tests, you'll simply need to add the junit jar to your project's compilation classpath and to your execution classpath when you run the tests.

Let's now download the JUnit (JUnit 3.8.1 or newer[5]) distribution, which contains several test samples that you will run to get familiar with executing JUnit tests. Follow these steps:

1. Download the latest version of JUnit from junit.org, referred to in step 2 as `http://junit.zip`.
2. Unzip the `junit.zip` distribution file to a directory on your computer system (for example, `C:\` on Windows or `/opt/` on UNIX).
3. Underneath this directory, unzip will create a subdirectory for the JUnit distribution you downloaded (for example, `C:\junit3.8.1` on Windows or `/opt/junit.3.8.1` on UNIX).

You are now ready to run the tests provided with the JUnit distribution. JUnit comes complete with Java programs that you can use to view the result of a test. There is a *graphical*, Swing-based test runner (figure 1.2) as well as a *textual* test runner (figure 1.3) that can be used from the command line.

To run the graphical test runner, open a shell in `C:\junit3.8.1` on Windows or in `/opt/junit3.8.1` on UNIX, and type the appropriate command:

Windows:

```
java -cp junit.jar;. junit.swingui.TestRunner junit.samples.AllTests
```

UNIX:

```
java -cp junit.jar:. junit.swingui.TestRunner junit.samples.AllTests
```

To run the text test runner, open a shell in `C:\junit3.8.1` on Windows or in `/opt/junit3.8.1` on UNIX, and type the appropriate command:

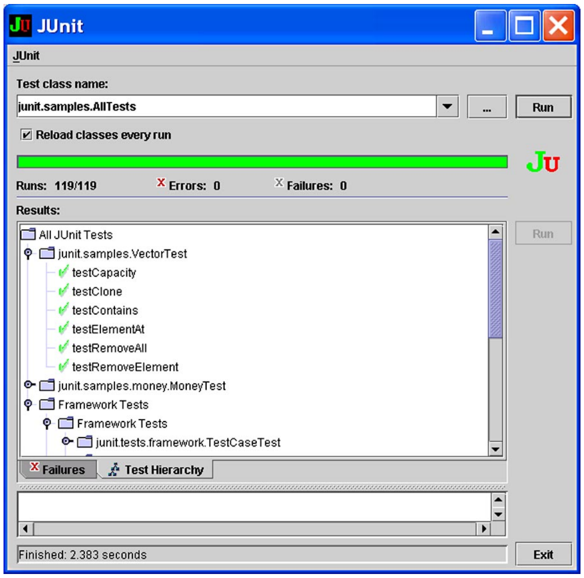
Windows:

```
java -cp junit.jar;. junit.textui.TestRunner junit.samples.AllTests
```

UNIX:

```
java -cp junit.jar:. junit.textui.TestRunner junit.samples.AllTests
```

[5] Earlier versions of JUnit will not work with all of our sample code.



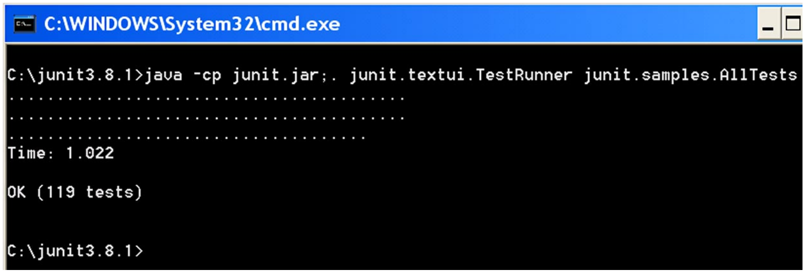
```
JUnit
Test class name:
junit.samples.AllTests
[✓] Reload classes every run
Runs: 119/119    Errors: 0    Failures: 0
Results:
All JUnit Tests
  junit.samples.VectorTest
    testCapacity
    testClone
    testContains
    testElementAt
    testRemoveAll
    testRemoveElement
  junit.samples.money.MoneyTest
  Framework Tests
    Framework Tests
      junit.tests.framework.TestCaseTest
Failures | Test Hierarchy
Finished: 2.383 seconds
```

Figure 1.2
Execution of the JUnit distribution sample tests using the graphical Swing test runner

Notice that for the text test runner, tests that pass are shown with a dot. Had there been errors, they would have been displayed with an *E* instead of a dot.

As you can see from the figures, the runners report equivalent results. The textual test runner is easier to run, especially in batch jobs, though the graphical test runner can provide more detail.

The graphical test runner also uses its own classloader instance (a reloading classloader). This makes it easier to use interactively, because you can reload classes (after changing them) and quickly run the test again without restarting the test runner.



```
C:\WINDOWS\System32\cmd.exe

C:\junit3.8.1>java -cp junit.jar;. junit.textui.TestRunner junit.samples.AllTests
.........................................
.........................................
.....................................
Time: 1.022

OK (119 tests)


C:\junit3.8.1>
```

Figure 1.3 Execution of the JUnit distribution sample tests using the text test runner

In chapter 5, “Automating JUnit,” we look at running tests using the Ant build tool and from within integrated development environments, like Eclipse.

1.5 *Testing with JUnit*

JUnit has many features that make tests easier to write and to run. You’ll see these features at work throughout this book:

- Alternate front-ends, or test runners, to display the result of your tests. Command-line, AWT, and Swing test runners are bundled in the JUnit distribution.
- Separate classloaders for each unit test to avoid side effects.
- Standard resource initialization and reclamation methods (`setUp` and `tear-Down`).
- A variety of assert methods to make it easy to check the results of your tests.
- Integration with popular tools like Ant and Maven, and popular IDEs like Eclipse, IntelliJ, and JBuilder.

Without further ado, let’s turn to listing 1.4 and see what the simple `Calculator` test looks like when written with JUnit.

Listing 1.4 The TestCalculator program written with JUnit

```
import junit.framework.TestCase;

public class TestCalculator extends TestCase         ❶
{
  public void testAdd()                              ❷
  {
    Calculator calculator = new Calculator();        ❸
    double result = calculator.add(10, 50);          ❹
    assertEquals(60, result, 0);                     ❺
  }
}
```

Pretty simple, isn’t it? Let’s break it down by the numbers.

In listing 1.4 at ❶, you start by extending the test class from the standard JUnit `junit.framework.TestCase`. This base class includes the framework code that JUnit needs to automatically run the tests.

At ❷, you simply make sure that the method name follows the pattern `test`*`XXX`*`()`. Observing this naming convention makes it clear to the framework that the method is a unit test and that it can be run automatically. Following the `test`*`XXX`* naming convention is not strictly required, but it is strongly encouraged as a best practice.

At ❸, you start the test by creating an instance of the `Calculator` class (the "object under test"), and at ❹, as before, you execute the test by calling the method to test, passing it two known values.

At ❺, the JUnit framework begins to shine! To check the result of the test, you call an `assertEquals` method, which you inherited from the base `TestCase`. The Javadoc for the `assertEquals` method is:

```
/**
 * Asserts that two doubles are equal concerning a delta. If the
 * expected value is infinity then the delta value is ignored.
 */

static public void assertEquals(double expected, double actual,
    double delta)
```

In listing 1.4, you passed `assertEquals` these parameters:

- `expected` = 60
- `actual` = `result`
- `delta` = 0

Since you passed the calculator the values 10 and 50, you tell `assertEquals` to expect the sum to be 60. (You pass 0 as you are adding integer numbers, so there is no delta.) When you called the `calculator` object, you tucked the return value into a local `double` named `result`. So, you pass that variable to `assertEquals` to compare against the expected value of 60.

Which brings us to the mysterious `delta` parameter. Most often, the `delta` parameter can be zero, and you can safely ignore it. It comes into play with calculations that are not always precise, which includes many floating-point calculations. The `delta` provides a plus/minus factor. So if the `actual` is within the range (`expected-delta`) and (`expected+delta`), the test will still pass.

If you want to enter the test program from listing 1.4 into your text editor or IDE, you can try it using the graphical test runner. Let's assume you have entered the code from listings 1.1 and 1.4 in the `C:\junitbook\jumpstart` directory (`/opt/junitbook/jumpstart` on UNIX). Let's first compile the code by opening a shell prompt in that directory and typing the following (we'll assume you have the javac executable on your `PATH`):

JUnit Design Goals

The JUnit team has defined three discrete goals for the framework:

- The framework must help us write useful tests.
- The framework must help us create tests that retain their value over time.
- The framework must help us lower the cost of writing tests by reusing code.

In listing 1.4, we tried to show how easy it can be to write tests with JUnit. We'll return to the other goals in chapter 2.

Windows:

```
javac -cp ..\..\junit3.8.1\junit.jar *.java
```

UNIX:

```
javac -cp ../../junit3.8.1/junit.jar *.java
```

You are now ready to start the Swing test runner, by typing the following:

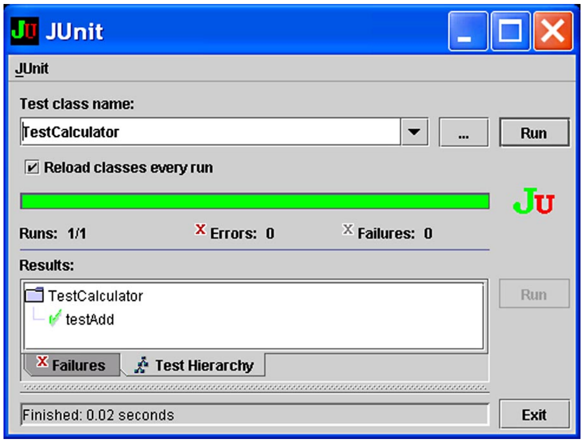
Windows:

```
java -cp .;..\..\junit3.8.1\junit.jar
   ➔ junit.swingui.TestRunner TestCalculator
```

UNIX:

```
java -cp .:../../junit3.8.1/junit.jar
   ➔ junit.swingui.TestRunner TestCalculator
```

The result of the test is shown in figure 1.4.



Test class name: TestCalculator

☑ Reload classes every run

Runs: 1/1   Errors: 0   Failures: 0

Results:

```
TestCalculator
  testAdd
```

Finished: 0.02 seconds

Figure 1.4
Execution of the first JUnit test `TestCalculator` using the Swing test runner

The remarkable thing about the JUnit `TestCalculator` class in listing 1.4 is that the code is every bit as easy to write as the first `TestCalculator` program in listing 1.2, but you can now run the test automatically through the JUnit framework.

NOTE If you are maintaining any tests written prior to JUnit version 3.8.1, you will need to add a constructor, like this:

```
public TestCalculator(String name) { super(name); }
```

It is no longer required with JUnit 3.8 and later.

1.6 Summary

Every developer performs some type of test to see if new code actually works. Developers who use automatic unit tests can repeat these tests on demand to ensure the code still works later.

Simple unit tests are not difficult to write by hand, but as tests become more complex, writing and maintaining tests can become more difficult. JUnit is a unit testing framework that makes it easier to create, run, and revise unit tests.

In this chapter, we scratched the surface of JUnit by installing the framework and stepping through a simple test. Of course, JUnit has much more to offer.

In chapter 2, we take a closer look at the JUnit framework classes and how they work together to make unit testing efficient and effective. (Not to mention just plain fun!)

2 Exploring JUnit

This chapter covers

- Using the core JUnit classes
- Understanding the JUnit life cycle

Mistakes are the portals of discovery.
—James Joyce

In chapter 1, we decided that we need an automatic testing program so that we can replicate our tests. As we add new classes, we often want to make changes to classes under test. Of course, experience has taught us that sometimes classes interact in unexpected ways. So, we'd really like to keep running all of our tests on all of our classes, whether they've been changed or not. But how can we run multiple test cases? And what do we use to run all these tests?

In this chapter, we will look at how JUnit provides the functionality to answer those questions. We will begin with an overview of the core JUnit classes `TestCase`, `TestSuite`, and `BaseTestRunner`. Then we'll take a closer look at test runners and `TestSuite`, before we revisit our old friend `TestCase`. Finally, we'll examine how the core classes work together.

2.1 Exploring core JUnit

The `TestCalculator` program from chapter 1, shown in listing 2.1, can run a single test case on demand. As you see, to create a single test case, you can extend the `TestCase` class.

Listing 2.1 The TestCalculator program written with JUnit

```
import junit.framework.TestCase;

public class TestCalculator extends TestCase
{
  public void testAdd()
  {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    assertEquals(60, result, 0);
  }
}
```

When you need to write more test cases, you create more `TestCase` objects. When you need to run several `TestCase` objects at once, you create another object called a `TestSuite`. To run a `TestSuite`, you use a `TestRunner`, as you did for a single `TestCase` object in the previous chapter. Figure 2.1 shows this trio in action.

These three classes are the backbone of the JUnit framework. Once you understand how `TestCase`, `TestSuite`, and `BaseTestRunner` work, you will be able to

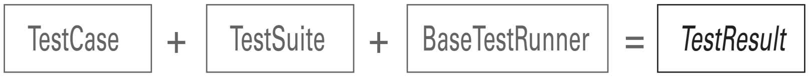


Figure 2.1 The members of the JUnit trio work together to render a test result.

DEFINITION *`TestCase` (or test case)*—A class that extends the JUnit `TestCase` class. It contains one or more tests represented by `testXXX` methods. A test case is used to group together tests that exercise common behaviors. In the remainder of this book, when we mention a *test,* we mean a `testXXX` method; and when we mention a *test case*, we mean a class that extends `TestCase`—that is, a set of tests.

`TestSuite` (or test suite)—A group of tests. A test suite is a convenient way to group together tests that are related. For example, if you don't define a test suite for a `TestCase`, JUnit automatically provides a test suite that includes all the tests found in the `TestCase` (more on that later).

`TestRunner` (or test runner)—A launcher of test suites. JUnit provides several test runners that you can use to execute your tests. There is no `TestRunner` interface, only a `BaseTestRunner` that all test runners extend. Thus when we write `TestRunner` we actually mean any test runner class that extends `BaseTestRunner`.

write whatever tests you need. On a daily basis, you only need to write test cases. The other classes work behind the scenes to bring your tests to life. These three classes work closely with four other classes to create the core JUnit framework. Table 2.1 summarizes the responsibilities of all seven core classes.

Table 2.1 The seven core JUnit classes and interfaces (interfaces are indicated by italics)

Class / *interface*	**Responsibilities**	**Introduced in...**
`Assert`	An assert method is silent when its proposition succeeds but throws an exception if the proposition fails.	Section 2.6.2
`TestResult`	A `TestResult` collects any errors or failures that occur during a test.	Section 2.4
`Test`	A *`Test`* can be run and passed a `TestResult`.	Section 2.3.2
`TestListener`	A *`TestListener`* is apprised of events that occur during a test, including when the test begins and ends, along with any errors or failures.	Section 2.5

continued on next page

Table 2.1 The seven core JUnit classes and interfaces (interfaces are indicated by italics) *(continued)*

Class / *interface*	Responsibilities	Introduced in…
`TestCase`	A `TestCase` defines an environment (or *fixture*) that can be used to run multiple tests.	Section 2.1
`TestSuite`	A `TestSuite` runs a collection of test cases, which may include other test suites. It is a composite of *`Test`*s.	Section 2.3
`BaseTestRunner`	A test runner is a user interface for launching tests. `BaseTestRunner` is the superclass for all test runners.	Section 2.2.2

Figure 2.2 shows the relationships among the seven core JUnit classes. You'll see how these core classes and interfaces work together in this chapter and throughout the book.

2.2 Launching tests with test runners

Writing tests can be fun, but what about the grunt work of running them? When you are first writing tests, you want them to run as quickly and easily as possible. You should be able to make testing part of the development cycle—*code : run : test : code* (or *test : code : run : test* if you are test-first inclined). There are IDEs and compilers for quickly building and running applications, but what can you use to run the tests?

2.2.1 Selecting a test runner

To make running tests as quick and easy as possible, JUnit provides a selection of test runners. The test runners are designed to execute your tests and provide you with statistics regarding the outcome. Because they are specifically designed for this purpose, the test runners can be very easy to use. Figure 2.3 shows the Swing test runner in action.

The progress indicator running across the screen is the famous JUnit green bar. *Keep the bar green to keep the code clean* is the JUnit motto.

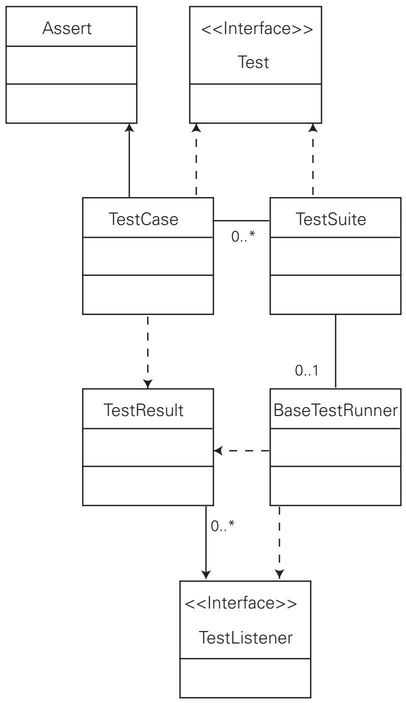


Figure 2.2 The core JUnit classes used to run any JUnit test program

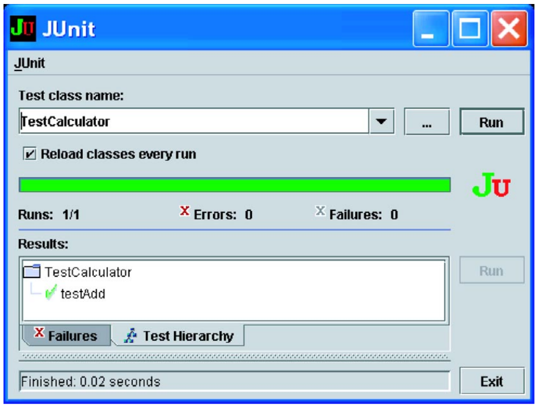


Figure 2.3 The graphical test runner in action

When tests fail, the bar shows up red instead. JUnit testers tend to refer to passing tests as *green-bar* and failing tests as *red-bar.*

The JUnit distribution includes three `TestRunner` classes: one for the text console, one for Swing, and even one for AWT (the latter being a legacy that few people still use).

2.2.2 Defining your own test runner

Unlike other elements of the JUnit framework, there is no `TestRunner` interface. Instead, the various test runners bundled with JUnit all extend `BaseTestRunner`. If you needed to write your own test runner for any reason, you could also extend this class yourself. For example, the Cactus framework that we'll discuss in later chapters extends `BaseTestRunner` to create a `ServletTestRunner` that can run JUnit tests from a browser.

2.3 Composing tests with TestSuite

Simple things should be simple … and complex things should be possible. Suppose you compile the simple calculator test program from listing 2.1 and hand it to a graphical test runner, like this:

```
>java junit.swingui.TestRunner TestCalculator
```

It should run just fine, assuming you have the correct classpath. (See figure 2.3 for the Swing test runner in action.) Altogether, this seems simple—at least as far as running a single test case is concerned.

But what happens when you want to run multiple test cases? Or just some of your test cases? How can you group test cases?

Between the TestCase and the TestRunner, it would seem that you need some type of container that can collect several tests together and run them as a set. But, by making it easier to run multiple cases, you don't want to make it harder to run a single test case.

JUnit's answer to this puzzle is the TestSuite. The TestSuite is designed to run one or more test cases. The test runner launches the TestSuite; which test cases to run is up to the TestSuite.

2.3.1 Running the automatic suite

You might wonder how you managed to run the example at the end of chapter 1, when you didn't define a TestSuite. To keep simple things simple, the test runner automatically creates a TestSuite if you don't provide one of your own. (*Sweet!*)

The default TestSuite scans your test class for any methods that start with the characters *test*. Internally, the default TestSuite creates an instance of your TestCase for each test*XXX* method. The name of the method being invoked is passed as the TestCase constructor, so that each instance has a unique identity.

For the TestCalculator in listing 2.1, the default TestSuite could be represented in code like this:

```
public static Test suite()
{
    return new TestSuite(TestCalculator.class);
}
```

And this is again equivalent to the following:

```
public static Test suite()
{
    TestSuite suite = new TestSuite();
    suite.addTest(new TestCalculator("testAdd"));
    return suite;
}
```

NOTE To use this form, the hypothetical TestCalculator class would need to define the appropriate constructor, like this:

```
public TestCalculator(String name) { super(name); }
```

JUnit 3.8 made this constructor optional, so it is not part of the source code for the original TestCalculator class. Most developers now rely on the automatic TestSuite and rarely create manual suites, so they can omit this constructor.

If you added another test, like `testSubtract`, the default `TestSuite` would automatically include it too, saving you the trouble of maintaining yet another block of fluff:

```
public static Test suite()
{
    TestSuite suite = new TestSuite();
    suite.addTest(new TestCalculator("testAdd"));
    suite.addTest(new TestCalculator("testSubstract"));
    return suite;
}
```

This is trivial code and would be easy to copy, paste, and edit—but why bother with such drudgery when JUnit can do it for you? Most important, the automatic test suite ensures that you don't forget to add some test to the test suite.

2.3.2 Rolling your own test suite

The default `TestSuite` goes a long way toward keeping the simple things simple. But what happens when the default suite doesn't meet your needs? You may want to combine suites from several different packages as part of a master suite. If you're working on a new feature, then as you make changes, you may want to run a small set of relevant tests.

There are many circumstances in which you may want to run multiple suites or selected tests within a suite. Even the JUnit framework has a special case: In order to test the automatic suite feature, the framework needs to build its own suite for comparison!

If you check the Javadoc for `TestSuite` and `TestCase`, you'll notice that they both implement the `Test` interface, shown in listing 2.2.

Listing 2.2 The Test interface

```
package junit.framework;

public interface Test {
    public abstract int countTestCases();
    public abstract void run(TestResult result);
}
```

If you are an over-achiever and also look up the Javadoc for `TestSuite`, you'll probably notice that the `addTest` signature doesn't specify a `TestCase` type—any old `Test` will do.

The ability to add both test suites and test cases to a suite makes it simple to create specialty suites as well as an aggregate `TestAll` class for your application.

JUnit design goals

The simple but effective combination of a TestRunner with a TestSuite makes it easy to run all your tests every day. At the same time, you can select a subset of tests that relate to the current development effort. This speaks to JUnit's second design goal:

The framework must create tests that retain their value over time.

When you continue to run your tests, you minimize your investment in testing and maximize your return on that investment.

Typically, the TestAll class is just a static suite method that registers whatever Test objects (TestCase objects or TestSuite objects) your application should be running on a regular basis. Listing 2.3 shows a typical TestAll class.

Listing 2.3 A typical TestAll class

```
import junit.framework.Test;
import junit.framework.TestSuite;
import junitbook.sampling.TestDefaultController;

public class TestAll
{
    public static Test suite()     ❶
    {
        TestSuite suite = new TestSuite("All tests from part 1");     ❷
        suite.addTestSuite(TestCalculator.class);                ❸
        suite.addTestSuite(TestDefaultController.class);
           // if TestDefaultController had a suite method
           // (or alternate suite methods) you could also use
           // suite.addTestSuite(TestDefaultController.suite());
        return suite;
    }
}
```

❶ Create a suite method to call all your other tests or suites.

❷ Give the TestSuite a legend to help identify it later.

❸ You call addTestSuite to add whatever TestCase objects or TestSuite objects you want to run together. It works for both types because the addTestSuite method accepts a Test object as a parameter, and TestCase and TestSuite both implement the Test interface.

In chapter 5, we look at several techniques for automating tasks like this one, so that you do not have to create and maintain a TestAll class. Of course, you still may want to create specialty suites for discrete subsets of your tests.

Design patterns in action: Composite and Command

Composite pattern. "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."[1] JUnit's use of the Test interface to run a single test or suites of suites of suites of tests is an example of the Composite pattern. When you add an object to a TestSuite, you are really adding a *Test*, not simply a Test*Case*. Because both TestSuite and TestCase implement Test, you can add either to a suite. If the Test is a TestCase, the single test is run. If the Test is a TestSuite, then a group of tests is run (which could include other TestSuites).

Command pattern. "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."[2] The use of the Test interface to provide a common run method is an example of the Command pattern.

2.4 Collecting parameters with TestResult

Newton taught us that for every action there is an equal and opposite reaction. Likewise, for every TestSuite, there is a TestResult.

A TestResult collects the results of executing a TestCase. If all your tests always succeeded, what would be the point of running them? So, TestResult stores the details of all your tests, pass or fail.

The TestCalculator program (listing 2.1) includes a line that says

```
assertEquals(60, result, 0);
```

If the result did not equal 60, JUnit would create a TestFailure object to be stored in the TestResult.

The TestRunner uses the TestResult to report the outcome of your tests. If there are no TestFailures in the TestResult collection, then the code is clean, and the bar turns green. If there are failures, the TestRunner reports the failure

[1] Erich Gamma et al., *Design Patterns* (Reading, MA: Addison-Wesley, 1995).

[2] Ibid.