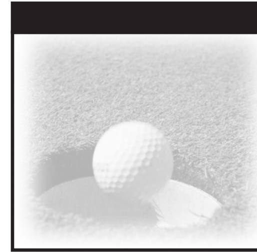
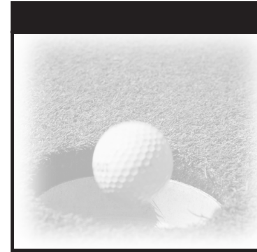


Jakarta Pitfalls

Time-Saving Solutions for Struts, Ant, JUnit, and Cactus

Bill Dudney
Jonathan Lehr

Wiley Publishing, Inc.



Jakarta Pitfalls

Time-Saving Solutions for Struts, Ant, JUnit, and Cactus

Bill Dudney
Jonathan Lehr

Wiley Publishing, Inc.

Executive Publisher: Robert Ipsen
Vice President and Publisher: Joe Wikert
Executive Editor: Robert Elliott
Assistant Development Editor: Eileen Bien Calabro
Editorial Manager: Kathryn A. Malm
Senior Production Editor: Angela Smith
Text Design & Composition: Wiley Composition Services

This book is printed on acid-free paper. ♾

Copyright © 2003 by Bill Dudney and Jonathan Lehr. All rights reserved.

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: permcoordinator@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-44915-6

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

For Sarah
—BD

For my wife, Kathryn
—JL

Contents

Acknowledgments **ix**

About the Authors **xi**

Introduction **xiii**

Chapter 1 Testing: Cactus and JUnit **1**

Pitfall 1.1: No Assert 4

Example 6

Solving Pitfall 1.1: Assert the Intent 9

Step-by-Step 9

Example 10

Example 2: Cactus 15

Pitfall 1.2: Unreasonable Assert 20

Example 21

Solving Pitfall 1.2: Assert the Intent 23

Pitfall 1.3: Console-Based Testing 24

Example 25

Solving Pitfall 1.3: System.out Becomes Assert 28

Step-by-Step 29

Example 29

Example 2: Cactus 31

Pitfall 1.4 Unfocused Test Method 34

Example 35

Solving Pitfall 1.4: Keep It Simple 38

Step-by-Step 38

Example 39

Pitfall 1.5: Failure to Isolate Each Test 45
Example 46
Solving Pitfall 1.5: Use setUp and tearDown and Introduce Test Decorators 50
Step-by-Step 51
Example 52
Example 2: Introduce Test Decorators 55
Pitfall 1.6: Failure to Isolate Subject 58
Example 59
Solving Pitfall 1.6: Introduce Mock Objects 62
Step-by-Step 62
Example 62

Chapter 2 Struts ActionForms 67
Pitfall 2.1: Copy/Paste Formatting 70
Example 72
Solving Pitfall 2.1: Consolidate and Generalize Formatting Code 74
Step-by-Step 75
Example 76
Pitfall 2.2: Copy/Paste Conversion 96
Example 98
Solving Pitfall 2.2: Consolidate and Generalize Bean Population Code 101
Step-by-Step 101
Example 102
Pitfall 2.3: Copy/Paste Validation 119
Example 121
Solving Pitfall 2.3: Consolidate and Generalize Validation Code 123
Step-by-Step 126
Example 127

Chapter 3 Struts Actions 147
Pitfall 3.1: Business-Tier Code in Actions 149
Example 151
Solving Pitfall 3.1: Move Business-Tier Code to BusinessDelegate 157
Step-by-Step 158
Example 159
Pitfall 3.2: Copy/Paste Code in Actions 172
Example 172
Solving Pitfall 3.2: Move Common Code to Base Class 173
Step-by-Step 174
Example 174

Pitfall 3.3: Accessing ActionForms in the Session 182
Example 183
Solving Pitfall 3.3: Add ActionForm Locator Method to Base Class 187
Step-by-Step 188
Example 188
Pitfall 3.4: Overloaded ActionMappings 192
Example 192
Solving Pitfall 3.4: Create Separate ActionMappings for Navigation and Form Submission 195
Step-by-Step 195
Example 195

Chapter 4 Struts TagLibs and JSPs 197
Pitfall 4.1: Hard-Coded Strings in JSPs 199
Example 200
Solving Pitfall 4.1: Move Common Strings to Resource Bundles 201
Step-by-Step 201
Example 201
Pitfall 4.2 Hard-Coded Keys in JSPs 203
Example 203
Solving Pitfall 4.2:Replace Hard-Coded Keys with Constants 204
Step-by-Step 205
Example 205
Pitfall 4.3: Not Using Struts Tags for Error Messaging 209
Example 210
Solving Pitfall 4.3: Replace Custom Messaging with Struts Messaging 210
Step-by-Step 211
Example 211
Pitfall 4.4: Calculating Derived Values in JSPs 214
Example 214
Solving Pitfall 4.4: Move Calculations to Value Object 215
Step-by-Step 216
Example 216
Pitfall 4.5: Performing Business Logic in JSPs 218
Example 219
Solving Pitfall 4.5: Move Business Logic to a Helper Class 220
Step-by-Step 221
Example 221
Pitfall 4.6: Hard-Coded Options in HTML Select Lists 224
Example 224
Solving Pitfall 4.6: Move Options Values to a Helper Class 225
Step-by-Step 225
Example 226

Pitfall 4.7: Not Checking for Duplicate Form Submissions 229
Example 229
Solving Pitfall 4.7: Add Tokens to Generated JSP 230
Step-by-Step 230
Example 230

Chapter 5 Ant 233
Pitfall 5.1: Copy-and-Paste Reuse 235
Example 235
Solving Pitfall 5.1: Introduce Antcall 237
Step-by-Step 237
Example 238
Pitfall 5.2: No Distinction between Different Types of Builds 243
Example 244
Solving Pitfall 5.2: Introduce Properties File 245
Step-by-Step 245
Example 246
Pitfall 5.3: Building Subprojects 249
Example 249
Solving Pitfall 5.3: Centralize the Build 251
Step-by-Step 252
Example 253
Pitfall 5.4: No Logging from Custom Tasks 257
Example 257
Solving Pitfall 5.4: Add Logging 259
Step-by-Step 259
Example 260

Appendix A Pitfall Catalog 263

Appendix B References 273

Appendix C What's on the Web Site 277

Index 279

Acknowledgments

I would like to thank first and foremost Christ, for all He has done in my life to teach me to be more than I was and to inspire me to be more than I am. I would also like to thank my wonderful wife Sarah, without her support and love I'd be lost. And I'd also like to thank my great kids that keep life interesting. Andrew, Isaac, Anna and Sophia you are the definition of joy. I'd also like to thank my mom for always making me look the word up in the dictionary even though I complained enough to deserve to be sent to my room. I'd also like to thank Jon Crater and Bill Willis for all their great feedback on the content of this book. It's a better book because of them. I would also like to thank my co-workers Chris Noe and Sridhar Valavala for teaching me so much, and for listening to my endless Monty Python quotes. My hovercraft is indeed full of eels. And finally I'd like to thank Eileen Bien Calabro for all her hard work on turning my gibberish into English and helping me to deliver a better book. I hope you learn as much from reading this book as I did in writing it.

—Bill Dudney

Writing this book has truly been an adventure for me, and I am grateful to my co-author, Bill Dudney, for inviting me to participate. As is the case I suppose with most technical books, this one is the work of many hands, and I am indebted to Bill Willis, Jon Crater, and Eileen Calabro for their invaluable assistance.

For the past year and a half, I have had the very good fortune of working with a wonderful team of developers who have shared many insights that helped deepen my understanding of the Struts framework and the possibilities of web applications in general. In particular, I owe much to Carl

Lindberg and Harshal Chaudhari, as well as Shailesh Patel, Shoekai Yeh, Jason Jobe, Michael Cymerman, Nikolai Teleguine, Diana Schmidt, and Sergey Muzyka. I would also like to thank Chris Cordrey of Gale Force Software for his help in assembling the team, with a special note of thanks to Bob Leonard.

Above all, I am grateful to my beloved wife, Kathryn, for her patience, support, and sacrifice while I juggled the full-time responsibilities of leading a framework development team with the demands of co-authoring this book.

—Jonathan Lehr

About the Authors

Bill Dudney is a Java Architect with Object Systems Group. He has been building J2EE applications and software for 5 years and has been doing distributed computing for almost 14 years. Bill has been using Jakarta tools since there was a Tomcat and has been a major advocate of using open source tools and building unit tests on all his projects. After struggling for years to keep *make* off his resume he discovered Ant and was glad to be known as the 'build guy' again. He is the co-author of both *J2EE AntiPatterns* and *Mastering JavaServer Faces* (Wiley).

Jonathan Lehr is an independent consultant in the Washington, D.C., area with over twenty years experience in software development and developer training. He is the author of over a dozen courses on Object-Oriented Programming and other development topics, and for the past eight years has designed and architected e-commerce applications in Objective C and Java for Fortune 100 financial and telecommunications companies. He currently leads a user-interface framework team that provides reusable Struts-based components and infrastructure enhancements for use by development teams at a major financial institution. He is also the co-author of *Mastering JavaServer Faces* (Wiley).

Introduction

What Is a Pitfall?

A pitfall is a common, overlooked, unsound way of developing and designing software. The consequences of pitfalls vary: Some are as mild as slightly decreased performance, but some have more severe consequences, like slipping schedules, difficult maintenance, and lack of changeability. Pitfall-strewn code can also be a major problem for new developers. The time it takes for a new developer to become effective is directly related to the cohesiveness of the code. Cohesive code is easy to follow and understand because it flows logically. Code filled with pitfalls is hard to follow because it does not flow logically.

The need to avoid pitfalls is paramount. Over time, any application that is being used will have bugs that need to be fixed and new features that need to be added, and thus it will require maintenance. The fewer pitfalls that are in the design and code, the easier this maintenance will be to perform. To illustrate this principal, imagine a shopping cart application. The application has browse, add products, checkout, and ship functionality. Over time, customers might request more functionality, such as the ability to search the product catalog. If some of the code to implement the search functionality is in the StrutsAction classes and some of the code is in a business tier object such as a JavaBean or an EJB, adding the new search functionality will be more difficult than if the original code is contained solely in the business tier.

Oddly, pitfalls rarely keep systems from working. An application can be riddled with pitfalls and still not have a problem functioning early on, but

the consequences of the pitfalls will eventually surface. For example, a project can be proceeding according to plan for months. Then, the second iteration begins, and mass chaos ensues. Many projects have failed during later iterations because the early code was too hard to maintain.

Also, unlike other development problems, the consequences of pitfalls don't make themselves obvious. If we do something like cast an object to the wrong type, the Java runtime is kind enough to inform us of that fact with an exception the first time the code is executed. But if our code is stuck in a pitfall, there is no runtime to tell us; we simply have to wait for the consequences to manifest themselves. This is why it is important to study the pitfalls that others have fallen into and to recognize them before we fall into them ourselves.

As a result of the delayed nature of the consequences, it is sometimes hard to justify fixing code stuck in a pitfall. After all, management is rarely keen on rewriting the whole system just to remove a couple of pitfalls (and rightly so because there would almost certainly be other pitfalls introduced as a result). So how do you get buy-in to fix the code? The most important thing to keep in mind when trying to justify fixing code is the longer-term payoff of code that is easier to understand and maintain. For example, copying and pasting the formatting code in one of your Struts forms (Pitfall 4.1) is particularly bad for the long-term maintainability of the form.

It is also important to remind management of the long-term consequences of pitfalls if you hope to ask for the time and resources to address them. The most important aspect to communicate to management is that fixing pitfalls does not have to mean rewriting. Instead, inform management that the code is changed in a disciplined way to achieve better design and implementation without starting over. Also emphasize that when fixing pitfalls, the internal structure of the code might change a lot, but the interface changes only slightly.

Of course, studying pitfalls before any of these issues occurs will save both you and management time, energy, and money down the road. But remember that studying pitfalls is not enough. You also need to find a way to work out of them when necessary or, better yet, to avoid them altogether. The good news is that every pitfall has at least one solution, and all of the Jakarta pitfalls discussed in this book come with both solutions and tips for avoidance.

Pitfalls in Jakarta

Jakarta is part of the Apache open source project, and its emphasis is on server-side Java solutions. There are many great projects hosted by the

Jakarta folks, but we focused on these three subprojects because they are widely used: Ant, Cactus, and Struts.

Ant

Ant has almost entirely replaced *make* as the build tool for Java developers. Ant allows developers to declare how their applications should be built and packaged. The declaration is written in a straightforward XML-based configuration file that is used to direct Ant from step to step. Ant is also customizable to allow developers to build their own tasks and use them in their configuration files.

Pitfalls in Ant arise typically from a lack of experience. Another area that gives rise to pitfalls is the perceived similarity between make files and build files. Many developers making the switch from make to Ant end up writing make files instead of build files. Chapter 5, “Ant,” deals with these issues.

Cactus

Cactus is a derivative of JUnit that provides server-side unit testing for J2EE components. With Cactus, unit tests can be written to perform on the server side fully integrated with the application server. This setup provides a great way to ensure that your components will perform as expected in an actual J2EE runtime environment. Because Cactus runs in the application server environment, the tests can use the actual server objects instead of having to try to build mock objects.

Although unit testing server-side objects with Cactus is far easier than it would be without Cactus, developers still make mistakes and end up with code that is hard to understand or maintain. The most common cause for these mistakes is lack of understanding of how to do unit testing in the first place. Chapter 1, “Testing: Cactus and JUnit,” explains in depth what goes wrong and how to fix it.

Struts

Struts is the Web-based UI framework that has become a de facto standard in the J2EE community. Struts provides an implementation of the Model-View-Controller (MVC) framework for building Web applications. The view is built from a large array of custom tags and JSPs. There are two controllers in Struts: a central servlet that listens to requests and delegates to application-specific controllers to perform the task specified in the request, and the Action that you write. The model is left to the developer to build. The application controller classes (that is, Actions) are responsible for converting the

model-level data into data that Struts is able to understand (that is, Action-Forms) as well as converting from the ActionForms back to the application-specific model.

Struts makes building Web-based applications easier than it has ever been before, but there is a common set of things that developers, especially new Struts developers, do wrong. Several pitfalls come from not having a good understanding of the architecture of Struts. Other pitfalls arise from not building the Struts components (Actions and ActionForms) in a way that will work well in a three-tier environment. Chapters 2, 3, and 4 capture these pitfalls as well as their solutions.

These three tools from Jakarta have proven to be a major force in the J2EE community. Many projects have been greatly enhanced by using Struts for their Web-based UI, Ant to build, and Cactus to test the project.

Why This Book?

The kinds of mistakes that are chronicled in this book are real-world experiences we have faced as developers working with these tools—the kinds of experiences that cause actual delays in schedule, or allow major bugs to get into the users' hands, or led to lots of rewrites in maintenance because the code was so hard to change or understand.

The Jakarta open source community has exploded over the last couple of years with insanely popular—and useful—projects, including the ones covered here: Ant, Cactus, and Struts. Given the relative newness of the technology, many developers are inexperienced with these tools and are getting trapped by the same pitfalls over and over. This book is an attempt to capture some of the most common pitfalls and the means to arrive at solutions. It is our hope that you will be saved the frustration of being trapped by the same pitfalls that have trapped us.

Even if you are an experienced developer, that doesn't mean that you can't get something out of this book. After all, just because everyone uses a technology does not always mean that they use it correctly. It takes time for common mindshare to develop around a concept and for common problems and solutions to become well known. For example, early adoptors of Ant, Cactus, and Struts suffered from poor documentation. Over time, the documentation has become very good for all three of these Jakarta projects, but some developers are still building bad code out of habit. What we provide here, therefore, is not an introductory Ants, Cactus, or Struts book; it's a way to improve code incrementally and to find out about the nooks and crannies that could make your code hard to maintain or perform poorly. In

the end, the pitfalls and solutions in this book will help you build better applications that are easier to maintain and that will perform better.

Organization of the Book

In each chapter, we first give a brief introduction to the chapter topic and offer lists of pitfalls and their related solutions. Sometimes, a single solution applies to one or more pitfalls in a chapter. When that happens, we cover the solution in detail under the first pitfall to which it applies and refer you to the original solution the next time it is applicable.

Pitfalls

Every pitfall in this book is numbered and named. We describe each pitfall in detail, and we explain how developers typically become trapped in it. We also provide information on how a developer can avoid being trapped and what the common symptoms and consequences of each pitfall are. For example, Pitfall 5.1: Business Tier Code in Chapter 5 documents the typical bad practice of putting code that belongs in the model into the StrutsAction classes and describes how to clean up the code so that it is better partitioned.

Where applicable, the pitfall descriptions also document the pitfall from different perspectives. Often a pitfall will manifest in different ways, depending on a number of factors. The pitfall descriptions in this book address each of these different manifestations in a way that will help developers identify the pitfall in their code or design.

To make the discussion more concrete, we also provide an example for each pitfall. Sometimes, the example is abbreviated in an effort to make it more clear. It is better to have an example that clearly illustrates the pitfall than to explain all the details.

Solutions

After each pitfall's example, we offer a solution, called Solving Pitfall X.X. Each solution contains general information, ways the solution can be applied to all the variations of the pitfall, step-by-step guidance, and a detailed example. The solutions essentially walk you through taking your pitfall-riddled code and converting to better, pitfall-free code.

Some solutions will affect the design of the application, but others will affect only the code. During the discussion of the solutions, however, we will focus mostly on the code because as the code is changed the design will be changed as well.

A Note about JUnit Testing

Testing often gets a bad reputation. It is often pushed to the end of a project, then dropped because of schedule issues. Then the project goes into the hands of testers without any developer-based testing, leading to a landslide of bugs. A unit test makes sure that small units of functionality on a particular class are working correctly, so this cycle doesn't repeat itself. With enough unit tests in place, the official testers on your project will be bored, and your application will sail through testing.

Why Unit Test?

First and foremost, it is necessary to have unit tests in place in order to refactor code—or, for our purposes, dig yourself out of pitfalls. Unit tests help you to make sure that what is documented in the API of your classes is actually what you have implemented—which is, of course, valuable if you want to change the implementation. When the change is complete (your pitfall is resolved), you can just rerun your unit tests. If the tests are complete, then you know the clients of your class will not be affected by your solution. Another benefit of unit tests is that as long as the tests are run before and after any change, problems will be found right away. Without unit tests, it can be quite a while before a bug related to the change surfaces, making the bug harder to track down.

Unit testing is also an efficient way to validate design and implementation assumptions. For example, with a unit test, you can validate your expectations about the way hashCode and equals work in a Hash Map or Set. If you have a set of unit tests that assert the contract as it is spelled out in the documentation, you can be fairly sure that when you put the object into the set it will act as expected. More tests to make sure that it is acting the correct way will expose missed requirements, assumptions, and bugs.

Unit testing is especially important in reusable components or frameworks. If you want your reusable code to be used by others, then you need to write tests for it. If that code is poorly tested, your teammates will lack confidence in your code; if it is well tested (and well documented), the code can be used with confidence. The tests not only will help to make sure that the code works but will give your users some valuable hints on how to use the framework. Further, code changes in one part of a system often show up as bugs in other parts of the system. Unit tests that are run often help prevent this from happening by isolating change and finding bugs right away.

Testing with JUnit

JUnit and its derivatives make unit testing easy. In fact, for some developers, testing with JUnit is addictive. It's a great feeling to have your code go into the hands of testers knowing that they won't find any major show-stopper bugs. And with any test, it's as simple as overriding setUp, tearDown, and the suite method; then you are ready to add test methods.

To prove how easy it is to perform a test with JUnit, here is an example test that examines the substring method on the String class.

```
public StringTest extends TestCase {
  private String subject = "Monty Python";

  public static void main(String args[]) {
    String classes[] = {StringTest.class.getName()};
    junit.swingui.TestRunner.main(classes);
  }

  public static TestSuite suite() {
    return new TestSuite(StringTest.class);
  }

  public void testSimpleSubstring() throws Exception {
    assertEquals("Python", subject.substring(6));
  }

  public void testBeginEndSubstring() throws Exception {
    assertEquals("Monty", subject.substring(0, 4));
  }

  public void setUp() throws Exception {
    // no need to Initialize anything because the subject
    // Is already Initialized as 'Monty Python'
  }
}
```

That is it. In just these few lines, we have all that we need to build and run a JUnit test. You can visit www.junit.org for more information and motivation.

Note to the Reader

This book assumes that you are a Java/J2EE developer familiar with the technologies discussed. This is not a book on how to build Ant files or

Struts applications. Instead, this book is about how not to build Cactus tests, how not to do Struts, and how not to use Ant.

We hope both inexperienced and experienced developers enjoy reading this book as much as we enjoyed writing it. With the experience captured here, we hope that you will be able to avoid the countless hours we spent frustrated, trying to work our way out of the pitfalls we had created.

Testing: Cactus and JUnit

With the advent of Extreme Programming (XP) and its emphasis on refactoring, unit testing has gained in popularity and exposure. In order to refactor anything, a good set of unit tests must be in place to make sure that current clients of the implementation will not be affected by the changes that are made. Many developers, as they embrace the XP approach, are suddenly "test infected" and writing all kinds of JUnit tests. Many developers who were doing unit testing with code in the main method of their Java classes are finding JUnit and Cactus to be a more thorough means to test their classes. This chapter is about what goes wrong when building a real-world test set for real-world applications with these tools.

Many pitfalls in unit tests come from the complexity of the components being tested or the complexity of the tests themselves. Also, the lack of assertions in the test code can cause problems. Without an assertion, a test just confirms that no exceptions were thrown. Although it is useful to know when exceptions are thrown, it is rarely enough. For example, not all unexpected state changes in a test subject will throw an exception. Developers shouldn't simply rely on printouts so that they can visually inspect the result of calling the tested code. While visual inspection is better than nothing, it's not nearly as useful as unit testing can be. This chapter shows several ways in which a lack of assertions shows up and provides strategies

MOCK OBJECT VERSUS "IN CONTAINER" TESTING

There are two ways to approach testing your server-side objects. They can be isolated from the containers in which they are intended to run and tested separately to ensure that the objects do what is expected of them. The other way is to build a framework that works with the container to allow your objects to be tested inside the container.

The first approach, called Mock Object testing (or the Mock Objects Pattern), is very effective at isolating the test subject. There is significant burden, though, in building and maintaining the Mock Objects that simulate the configuration and container objects for the test subject. They have to be built and maintained in order for the testing to be effective. Even though there is virtually no complexity to the actual Mock Objects, there is a lot of complexity in maintaining the large number of Mock Objects required to simulate the container.

Cactus takes the other approach and facilitates testing inside the container. Cactus gets between the test cases and the container and builds an environment for the test subject to be run in that uses the container-provided objects instead of Mock Objects. Both approaches are helpful in stamping out bugs.

to migrate existing tests (visual or not) to solid unit tests that assert the contract implied in the API being tested.

A quick word about the differences between JUnit and Cactus: JUnit tests run in the same JVM as the test subject whereas Cactus tests start in one JVM and are sent to the app server's JVM to be run. Cactus has a very clever means to do the sending to the remote machine. Just enough information is packaged so that the server side can find and execute the test. The package is sent via HTTP to one of the redirectors (ServletTestRedirector, FilterTestRedirector, or the JSPTestRedirector). The redirector then unpacks the info, finds the test class and method, and performs the test. Figure 1.1 represents this process.

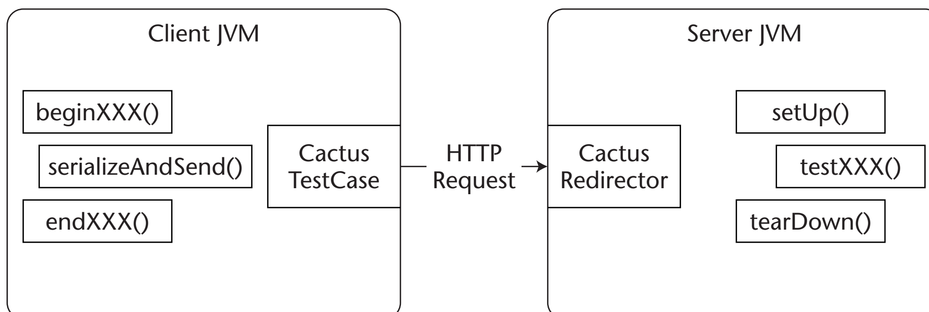


Figure 1.1 Cactus and test methods.

Misunderstanding the distributed nature of Cactus can lead to frustration if you are an old hand at building JUnit tests. Keep this in mind as you start to build Cactus tests.

Another thing to keep in mind as you build Cactus tests is that the redirector provides access to the container objects only on the server. Because the container objects are not available until the test is running on the server side, you cannot use container objects in the methods that are executed on the client side. For example, the config object in a ServletTest is not available in the beginXXX and endXXX methods, but you can use it in your setUp, textXXX, and tearDown methods.

JUnit has become the de facto standard unit testing framework. It is very simple to get started with it, and it has amazing flexibility. There are probably dozens, if not a couple of hundred, of extensions to JUnit available on the Web. This chapter focuses on JUnit (www.junit.org) and Cactus (www.jakarta.apache.org/cactus). Cactus allows "in container" testing of J2EE components. As stated earlier, this book assumes some experience with these tools.

Pitfall 1.1: No assert is the result of developers that do not realize that calling a method is not the same as testing it.

Pitfall 1.2: Unreasonable assert examines the tendency of developers new to unit testing to start asserting everything, even things that won't happen unless the JVM is not working properly.

Pitfall 1.3: Console-Based Testing addresses the problem of developers who get into the habit of using "System.out" to validate their applications. This method of testing is very haphazard and error-prone.

Pitfall 1.4: Unfocused Test Method is common to more experienced developers who get a little lazy about writing good tests and let the test become overly complex and hard to maintain.

Pitfall 1.5: Failure to Isolate Each Test is fixed using the setUp and tearDown methods defined in the JUnit framework. These methods allow each test to be run in an isolation that contains only the test subject and the required structure to support it.

Pitfall 1.6: Failure to Isolate Subject is related to the discussion of Mock Objects in the previous sidebar.

Pitfall 1.1: No Assert

This pitfall describes the tendency of developers new to unit testing to forget about asserts completely. New developers often assume that invoking the method is a sufficient test. They assume that if no exceptions are thrown when the method is called, then everything must be OK. Many bugs escape this kind of testing.

Here is some code for a simple addStrings method that returns the result of concatenating the value returned from the toString method of its two arguments. The current implementation should not be putting a space into the returned value, but it is. The initial test will not expose this bug because it is stuck in this pitfall; we will apply the solution to the test, though, and it will expose the bug.

```
public String appendTwoStrings(Object one, Object two) {
    StringBuffer buf = new StringBuffer(one.toString());
    buf.append(" ");
    buf.append(two.toString());
    return buf.toString();
}
```

Here is a sample test that simply invokes the method without really testing anything.

```
public void testAppendTwoStrings() throws Exception {
    myAppender.appendTwoStrings("one", "two");
}
```

This situation is typical of tests stuck in this pitfall. Even though it looks as if the appendTwoStrings method is tested, it is not. Users of the appendTwoStrings method have expectations of what the return value will be as a result of calling the method. And, in this case, the expectations will not be met. The API for a class is an implied contract for the users of the code. Whenever that contract is not met, the users of the code will see that failure as a bug. Unit tests should make sure that every unit of code performs as expected, that it fulfills the implied contract in the API. Unit tests that are stuck in this pitfall do not make sure that code is performing as expected, and they need to be fixed.

INTENT OF THE API

The "intent of the API" is what is documented or expected that the API will do with the inputs provided. It is also what the API will do to the internal state of the object on which the method is being called. For example, the *append* method on the StringBuffer class is documented to append the argument to its internal buffer such that the StringBuffer is longer by the length of the argument that is passed into the method. The internal state of the StringBuffer has changed, and nothing has happened to the argument. The test suite for StringBuffer should assert both "intents" of the StringBuffer API.

A test should make sure that the stated intentions of the API are met by asserting that what is expected to be true actually is. Another way to think of the intent of the API is that the API is like a contract between the clients that use the API and the provider of the API. The provider of the API is guaranteeing that the class will perform certain tasks, and the consumer is expecting those tasks to be performed. Formal ideas surrounding Design by Contract (DBC) are helpful in building tests for classes.

No assert is usually exposed at the point at which the application is passed over to the testing team, which tests the application by looking at the state changes that are occurring. As the testing team looks into the database to confirm that what was supposed to change did change, they will notice that the data is not changing as expected. As a result, many bug reports will be filed, and the bug fixes will more often than not be made in code that was tested with few asserts. Bug reports are no fun, especially when some effort was made to do unit testing. This situation will make it appear that unit testing added little value.

One of two things, laziness or lack of knowledge and experience, usually causes this pitfall. Everyone gets lazy from time to time. Developers are no exception, but it is important that we build good unit tests so that we can afford to be a little lazy. A good set of unit tests will expose bugs right when they are introduced, which is when they are easiest to fix. And because the bugs are easier to fix, we have less work.

Lack of knowledge and experience is fixed only through mentorship and experience. Over time, developers will begin to see how valuable unit tests are, especially if they have found, fixed, and prevented anyone else from seeing bugs in their code. You can encourage and teach good unit testing by doing periodic peer reviews with junior members. Peer reviews provide a great mechanism to mentor people, and if the senior people allow junior people to review their code, junior people will be able to see good examples on a regular basis.

TESTING FIRST

Many people in the JUnit community suggest that tests be written before the code that they are intended to test. The tests become almost a coded set of requirements for the test subject. This is a great habit to get into. The next time you are transitioning from design to development, try writing a few tests for the new code before implementing. When it comes time to use the code, you will thank yourself. The great benefit of testing first is that it forces the developer to focus on providing good APIs to future clients. The tests will expose nuances of what was expected to be true at design time versus what is really true on the ground in the code. And besides, if you write the tests first you will have a concrete gauge of when the class is done (that is, when it passes all the tests, it is done).

To stay out of this pitfall, you have to assert the intent stated in the API being tested. The intent of the API is what is expected to happen when the API is called. The intent is usually captured in the form of JavaDoc and the exception list that a method throws (sometimes referred to as the contract for the class). Unit tests should make sure that each piece of the API is doing what it should. As an example, if a method claims to throw an IllegalArgumentException when a null is passed, at least one test should assert that that exception is thrown when a null is passed.

Example

This example of No assert relies on a test for the contrived class called StringPair. Instances of StringPair will be used as keys in a map. An object that will be used as a key in a map must implement two methods: "equals" and hashCode. The two methods must be consistent, which means that if true is returned from the two objects involved in an equals comparison (that is, the receiver of the method call and the argument to the method), then hashCode must return the same value for both objects.

The StringPair class has two string properties, *right* and *left*. These two values are used in the equals and hashCode methods. To further complicate the subject, let's say that the StringPair class is used in a performance-sensitive environment and that it caches the hashCode so that it does not have to be recomputed each time. The hashCode should be reset to –1 when either the right or left value changes. This resetting behavior is crucial to the functioning of the StringPair class as a signal that the hashCode should be recomputed. The unit test here makes sure that the important methods on the StringPair class are called.

A good test for the StringPair class would assert that every intent described earlier is true (that hashCode and equals are consistent). The

JUnit test, however, is not good, as the test in Listing 1.1 does not assert anything in particular; in other words, this test case is trapped in this pitfall.

```
public class StringPairTest extends TestCase {
    private StringPair one = new StringPair("One", "Two");
    private StringPair oneA = new StringPair("One", "Two");
    private StringPair two = new StringPair("Three", "Four");
    private StringPair twoA = new StringPair("Three", "Four");

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

    public StringPairTest(String name) {
        super(name);
    }

    /**
     *  Test equals.
     */
    public void testEquals() throws Exception {
        one.equals(oneA);
        oneA.equals(one);
    }

    /**
     *  Test not equals.
     */
    public void testNotEquals() throws Exception {
        one.equals(two);
        two.equals(one);
    }

    /**
     *  Test hashCode.
     */
    public void testHashCode() throws Exception {
        one.hashCode();
        two.hashCode();
    }

    /**
     *  Test setting the values.
     */
    public void testSetValues() throws Exception {
        one.setRight("ROne");
        one.setLeft("LOne");
```

Listing 1.1 StringPairTest. *(continues)*

```
        two.setRight("RTwo");
        two.setLeft("LTwo");
    }

    /**
     *  Should throw an exception.
     */
    public void testNullPointerProtection() throws Exception {
        // since this will throw an exception the test will fail
        StringPair busted = new StringPair(null, "Four");
    }
}
```

Listing 1.1 *(continued)*

There are no asserts in the code for StringPairTest, so it is actually not testing very much. For example, take a look at the testSetValues method. All that happens is that the right and left property set methods are called. No check is made to make sure that the expected state changes happened on the StringPair instances. All this test is making sure of is that if valid strings are passed into the set methods (that is, not null) no exceptions are thrown. A lot of code is written in this way and then called test code. The StringPairTest test case is a classic example of this pitfall.

In this example, because the StringPair class is so simple, it might seem like overkill to put tests in place to make sure that equals and hashCode are performing as they should. Others, however, will be using this class and will expect it to function as advertised in its API. Which kind of class would you rather depend on in your code, one that is well tested (even when the code seems simple) or code that is not tested? A well-tested StringPair class can be used confidently. A poorly tested StringPair class that is tested only in integrated tests with the larger process will likely lead to much harder-to-find bugs. If StringPair is tested only through the Big Process test cases, then bugs in StringPair will be much harder to find because it cannot be stated with certainty that the bug is not in StringPair. The test needs to assert that the intent of the class as laid out in its API is actually being met, meaning that the hashCode is being reset when a value changes. If tests are in place that assert the intent of the StringPair API, then when bugs arise in the Big Process, they can be attributed confidently to something in the Big Process code.