

JUnit in Action

by Vincent Massol with Ted Husted

Detailed Table Of Contents

Chapter 1: JUnit jumpstart

- 1.1 Proving it works
- 1.2 Starting from scratch
- 1.3 Understanding unit testing frameworks
- 1.4 Setting up JUnit
- 1.5 Testing with JUnit
- 1.6 Summary

Chapter 2: Exploring JUnit

- 2.1 Exploring core JUnit
- 2.2 Launching tests with test runners
 - 2.2.1 Selecting a test runner
 - 2.2.2 Defining your own test runner
- 2.3 Composing tests with TestSuite
 - 2.3.1 Running the automatic suite
 - 2.3.2 Rolling your own test suite
- 2.4 Collecting parameters with TestResult
- 2.5 Observing results with TestListener
- 2.6 Working with TestCase
 - 2.6.1 Managing resources with a fixture
 - 2.6.2 Creating unit test methods
- 2.7 Stepping through TestCalculator
 - 2.7.1 Creating a test suite
 - 2.7.2 Creating a test result
 - 2.7.3 Executing the test methods
 - 2.7.4 Reviewing the full JUnit life cycle
- 2.8 Summary

Chapter 3 Sampling JUnit

- 3.1 Introducing the controller component
 - 3.1.1 Designing the interfaces
 - 3.1.2 Implementing the base classes
- 3.2 Let's test it!
 - 3.2.1 Testing the DefaultController
 - 3.2.2 Adding a handler
 - 3.2.3 Processing a request
 - 3.2.4 Improving testProcessRequest
- 3.3 Testing exception-handling

- 3.3.1 Simulating exceptional conditions
- 3.3.2 Testing for exceptions
- 3.4 Setting up a project for testing
- 3.5 Summary

Chapter 4 Examining software tests

- 4.1 Why we need unit tests
 - 4.1.1 The paradigm shifts
 - 4.1.2 To err is human ...
 - 4.1.3 To interact divine
- 4.2 Kinds of tests
 - 4.2.1 The four flavors of software tests
 - 4.2.2 The three flavors of unit tests
- 4.3 Determining how good tests are
 - 4.3.1 Measuring test coverage
 - 4.3.2 Generating test coverage reports
 - 4.3.3 Testing interactions
- 4.4 Test-Driven Development
 - 4.4.1 Tweaking the cycle
 - 4.4.2 The TDD two-step
 - 4.4.3 Having fun with it
- 4.5 Summary

Chapter 5 Automating JUnit

- 5.1 A day in the life
- 5.2 Running tests from Ant
 - 5.2.1 Ant, indispensable Ant
 - 5.2.2 Ant targets, projects, properties, and tasks
 - 5.2.3 The Javac task
 - 5.2.4 The JUnit Task
 - 5.2.5 Putting Ant to the task
 - 5.2.6 Pretty printing with JUnitReport
 - 5.2.7 Improving the buildfile
- 5.3 Running tests from Maven
 - 5.3.1 Maven the goalseeker
 - 5.3.2 Installing Maven
 - 5.3.3 Configuring Maven for a project
 - 5.3.4 Executing JUnit tests with Maven
 - 5.3.5 Handling dependent JARs with Maven
- 5.4 Running tests from Eclipse
 - 5.4.1 Creating an Eclipse project
 - 5.4.2 Running JUnit tests in Eclipse
- 5.5 Summary

Chapter 6 Coarse-grained testing with stubs

- 6.1 Introducing Stubs
- 6.2 Practicing on an HTTP Connection sample
 - 6.2.1 Choosing a stubbing solution
 - 6.2.2 Using Jetty as an embedded server
- 6.3 Stubbing the web server's resources
 - 6.3.1 Setting up the first stub test
 - 6.3.2 Testing for failure conditions
 - 6.3.3 Reviewing the first stub test
- 6.4 Stubbing the connection
 - 6.4.1 Producing a custom URL protocol handler
 - 6.4.2 Creating a JDK HttpURLConnection stub
 - 6.4.3 Running the test
- 6.5 Summary

Chapter 7 Testing in isolation with mock objects

- 7.1 Introducing mock objects
- 7.2 Mock tasting—a simple example
- 7.3 Using mock objects as a refactoring technique
 - 7.3.1 Easy refactoring
 - 7.3.2 Allowing more flexible code
- 7.4 Practicing on a HTTP connection sample
 - 7.4.1 Defining the mock object
 - 7.4.2 Testing a sample method
 - 7.4.3 Try #1: easy method refactoring technique
 - 7.4.4 Try #2: refactoring using a class factory
- 7.5 Using mocks as Trojan horses
- 7.6 Deciding when to use mock objects
- 7.7 Summary

Chapter 8 In-container testing with Cactus

- 8.1 The problem with unit-testing components
- 8.2 Testing components using mock objects
 - 8.2.1 Testing the servlet sample using EasyMock
 - 8.2.2 Pros and cons of using mock objects for testing components
- 8.3 Integration unit tests
- 8.4 Introducing Cactus
- 8.5 Testing components using Cactus
 - 8.5.1 Running Cactus tests
 - 8.5.2 Executing tests using the Cactus/Jetty integration
 - 8.5.3 Drawbacks of in-container testing
- 8.6 How Cactus works
 - 8.6.1 Executing client-side and server-side steps
 - 8.6.2 Stepping through a test
 - 8.6.3 Gathering test results
- 8.7 Summary

Chapter 9 Unit-testing servlets and filters

- 9.1 Presenting the Administration application
- 9.2 Writing servlet tests with Cactus
 - 9.2.1 Designing the first test
 - 9.2.2 Using Maven to run Cactus tests
 - 9.2.3 Finishing the Cactus servlet tests
- 9.3 Testing servlets with mock objects
 - 9.3.1 Writing a test using DynaMocks and DynaBeans
 - 9.3.2 Finishing the DynaMock tests
- 9.4 Writing filter tests with Cactus
 - 9.4.1 Testing the filter with a SELECT query
 - 9.4.2 Testing the filter for other query types
 - 9.4.3 Running the Cactus filter tests with Maven
- 9.5 When to use Cactus, and when to use mock objects
- 9.6 Summary

Chapter 10 Unit-testing JSPs and taglibs

- 10.1 Revisiting the Administration application
- 10.2 JSP unit testing
- 10.3 Unit-testing a JSP in isolation with Cactus
 - 10.3.1 Executing a JSP with SQL results data
 - 10.3.2 Writing the Cactus test
 - 10.3.3 Executing Cactus JSP tests with Maven
- 10.4 Unit-testing taglibs with Cactus
 - 10.4.1 Defining a custom tag
 - 10.4.2 Testing the custom tag
 - 10.4.3 Unit-testing tags with a body
 - 10.4.4 Unit-testing collaboration tags
- 10.5 Unit-testing taglibs with mock objects
 - 10.5.1 Introducing MockMaker and installing its Eclipse plugin
 - 10.5.2 Using MockMaker to generate mocks from classes
- 10.6 When to use mock objects, and when to use Cactus
- 10.7 Summary

Chapter 11 Unit-testing database applications

- 11.1 Database unit testing
- 11.2 Testing business logic in isolation from the database
 - 11.2.1 Implementing a database access layer interface
 - 11.2.2 Setting up a mock database interface layer
 - 11.2.3 Mocking the database interface layer
- 11.3 Testing persistence code in isolation from the database
 - 11.3.1 Testing the execute() method
 - 11.3.2 Using expectations to verify state
- 11.4 Writing database integration unit tests
 - 11.4.1 Filling the requirements for database integration tests
 - 11.4.2 Presetting database data
- 11.5 Running the Cactus test using Ant
 - 11.5.1 Reviewing the project structure
 - 11.5.2 Introducing the Cactus/Ant integration module

- 11.5.3 Creating the Ant build file step by step
- 11.5.4 Executing the Cactus tests
- 11.6 Tuning for build performance
 - 11.6.1 Factor-out read-only data
 - 11.6.2 Grouping tests in functional test suites
 - 11.6.3 Using an in-memory database
- 11.7 Overall database unit-testing strategy
 - 11.7.1 Choosing an approach
 - 11.7.2 Applying continuous integration
- 11.8 Summary

Chapter 12 Unit-testing EJBs

- 12.1 Defining a sample EJB application
- 12.2 Using a façade strategy
- 12.3 Unit testing JNDI code using mock objects
- 12.4 Unit-testing session beans
 - 12.4.1 Using the Factory Method strategy
 - 12.4.2 Using the Factory Class strategy
 - 12.4.3 Using the Mock JNDI implementation strategy
- 12.5 Using mock objects to test message-driven beans
- 12.6 Using mock objects to test entity beans
- 12.7 Choosing the right mock object strategy
- 12.8 Using integration unit tests
- 12.9 Using JUnit and remote calls
 - 12.9.1 Requirements for using JUnit directly
 - 12.9.2 Packaging the Petstore application in an EAR
 - 12.9.3 Performing automatic deployment and execution of tests
 - 12.9.4 Writing a remote JUnit test for PetstoreEJB
 - 12.9.5 Fixing JNDI names
 - 12.9.6 Running the tests
- 12.10 Using Cactus
 - 12.10.1 Writing an EJB unit test with Cactus
 - 12.10.2 Project directory structure
 - 12.10.3 Packaging the Cactus tests
 - 12.10.4 Executing the Cactus tests
- 12.11 Summary