Junit

Junit is a **framework** for developing unit tests in Java. the **underlying testing model**, known as ***xUnit***, is on its way to becoming the standard framework for any language.

In this book, we use the term *unit test* in the narrower sense to mean **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 tests are called ***programmer tests***to differentiate them from **quality-assurance or customer tests**.

Here is **a generic description** of a typical unit test from the perspective of this book: “Confirms that the method **accepts the expected range of input** and that the method returns the expected value for each input.” This description asks us to **test the behavior of a method through its interface**.

DEFINITION *Unit test*—A test that examines the behavior of a distinct unit of work. A *unit of work* is a task that **is not directly dependent** on the completion of any other task. **Within a Java** application, **the distinct unit of work is often, but not always, a single method**.

In contrast, ***integration tests* and *acceptance tests*** examine **how various components interact**.

Unit tests often focus on testing whether a method is following the terms of its API contract. An *API contract* is a formal agreement **made by the signature of a method**. A method requires its callers to provide specific object references or taprimitive values and returns an object reference or primitive

value. If the method cannot fulfill the contract, **the test should throw an exception**, and we say that **the method has *broken* its contract**.

A core principle of unit testing is, “Any **program feature** without an automated test simply doesn’t exist”

# Testing From Scratch So You Know what Junit Brings to the Table

Take a calculator class as an example:

**public class** Calculator {

**public double** add(**double** number1, **double** number2) {

**return** number1 + number2;

}

}

Isn’t the add method too simple to break?

The current implementation of the add method is too simple to break with usual, everyday calculations. **If add were a minor utility method, you might not test it directly. In that case, if add did fail, 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**, but **also a *program feature***. 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 to be.** 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 5.

* You don’t want to create a user interface to enter the values and know whether this method works correctly.
* if you are going to go to the effort of testing your work, you should also **try to preserve that effort.**
* It is good to know that the add(double, double) method worked when you wrote it. What you really want to know, however, is whether the method works when you ship the rest of the application or whenever you make a subsequent modification.

If we put these requirements together, we come up with the idea of writing a simple test program for the add method.

So, what is the simplest possible test program you could write? What about this CalculatorTest program?

**public class** CalculatorTest {

**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);

}

}

}

* But what happens if you change the code so that it fails? You have to watch the

screen carefully 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 signal error conditions in Java is to throw an exception. Let’s throw an exception to indicate a test failure.

Meanwhile, you may also want to run tests for other Calculator methods that you have not written yet, such as subtract or multiply. Moving to a modular design will make catching and handling exceptions easier; it will also be easier to extend the test program later. The next listing shows a slightly better CalculatorTest program:

**public class** CalculatorTest {

**private int** nbErrors = 0;

**public void** testAdd() {

Calculator calculator = **new** Calculator();

**double** result = calculator.add(10, 50);

**if** (result != 60) {

**throw new** IllegalStateException("Bad result: " + result);

}

}

**public static void** main(String[] args) {

CalculatorTest test = **new** CalculatorTest();

**try** {

test.testAdd();

}

**catch** (Throwable e) {

test.nbErrors++;

e.printStackTrace();

}

**if** (test.nbErrors > 0) {

**throw new** IllegalStateException("There were " + test.nbErrors

+ " error(s)");

}

}

}

you move the test into its own testAdd method. Now it’s easier to focus on what the test does. You can also add more methods with more unit tests later without making the main method harder to maintain. You also change the main method to print a stack trace when an error occurs; then, if there are any errors, you end by throwing a summary exception.

you can see that even this small class and its **tests can benefit from the little bit of skeleton code** you created to run and manage test results. But as an application gets more complicated and the tests become more involved, continuing to build and **maintain a custom testing framework becomes a burden.**

## Understanding Unit Testing Frameworks

Unit testing has several best practices that frameworks should follow. The seemingly minor improvements in the CalculatorTest program in listing 1.3 highlight **three rules** that (in my experience) **all unit testing frameworks should follow:**

 **Each unit test should run independently of all other unit tests.**

 **The framework should detect and report errors test by test.**

 **It should 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 each unit test to be truly independent, for example, each should run in a different class instance.**

## Adding New Unit Tests

* You can add new unit tests by adding a new method and then adding a corresponding

try/catch block to main. This is a step up but still short of what you would want in a real unit test suite. Experience tells us that large try-catch blocks cause maintenance problems. You could easily leave out a unit test and never know it!

* 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 capabilities. A program could

look at itself and decide to run whatever methods follow a certain naming convention,

such as those that begin with test.

* The JUnit framework already supports discovering methods.
* It also supports using a different class instance and class loader instance for each test
* and reports all errors on a test-by-test basis

# Junit

## Introduction

* To use JUnit to write your application tests, you need to know about its dependencies.
* we’ll work with JUnit 5, the latest version of the framework when this document was written.
* Version 5 of the testing framework **is a modular one.**
* you can no longer simply add a jar file to your project compilation classpath and your execution classpath.
* In fact, starting with version 5, the architecture is **no longer monolithic** (as discussed in chapter 3).
* Also, with the introduction of annotations in Java 5, JUnit has also moved to using them. JUnit 5 is heavily based on annotations—a contrast with the idea of extending a base class for all testing classes and using naming conventions for all testing methods to match the text*XYZ* pattern, as done in previous versions.

What’s new in Junit 5 vs what we had in Junit 4 and why you should move towards it:

JUnit 5 represents the next generation of JUnit.

* You’ll use the programming capabilities introduced starting with Java 8 (Junit 4 came out in 2006 versus 2014, the release year of Java8)
* you’ll be able to build tests **modularly and hierarchically**
* and the tests will be easier to understand, maintain, and extend

Chapter 4 discusses the transition from JUnit 4 to JUnit 5 and shows that the projects you are working on may benefit from the great features of JUnit 5. As you’ll see, you can make this transition smoothly, in small steps.

## Setting up Junit

The dependencies that are always needed in the pom.xml file, are shown in the following listing. **In** **the beginning**, you need only **junit-jupiter-api** and **junitjupiter-engine**.

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.6.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.6.0</version>

<scope>test</scope>

</dependency>

To be able to run tests from the command prompt, make sure your pom.xml configuration file includes a JUnit provider dependency for the Maven Surefire plugin. Here’s what this dependency looks like:

<build>

<plugins>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.2</version>

</plugin>

</plugins>

</build>

Page11…