Advanced Java Features for Using JUnit in CS1/CS2

Hans Dulimarta  Scott Grissom
{dulimarh,grissom}@gvsu.edu
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan 49401

Abstract—The Computer Science Curriculum 2008 [2] (CS2008) published jointly by ACM and IEEE includes “software testing” as one of the core topics in several areas of knowledge (Human-Computer Interaction, Social and Professional Issues, and Software Engineering). Out of 31 core hours of Software Engineering knowledge, CS2008 recommends a minimum of 3 hours dedicated to Software Verification and Validation. It is important that software testing is introduced early in the curriculum and we believe that CS1 is an appropriate course to do so. The first exposure to using testing tools in CS1 will prepare students to continue using automated testing tools when they work on larger projects in future classes.

In this paper, we describe a few techniques in designing JUnit test cases to address some challenges commonly encountered in CS1 programming assignments. Some of these techniques are also applicable to programming assignments in CS2.

Index Terms—Unit Testing, Introductory Course, Test Case Design

I. What is JUnit?

As a unit testing framework for Java programs, JUnit [6] provides a large set of assertion methods for comparing the actual value of variables obtained from the program under test with their expected value. In addition to this main feature for asserting various conditions, JUnit also provides powerful and extendable features that have attracted professional software developers as well as computer science educators to use JUnit in their software development.

Test cases developed as supplemental material to a programming assignment can benefit both the instructor and the students. The instructor will feel confident that the solution to the assignment is correct and the test cases can later be used to automate most of the grading[4]. Likewise, students will have the assurance that the work they submit satisfy the assignment requirements.

As one of the most popular choices for unit testing framework, JUnit has been adapted to support more advanced features for software testing. Szeder [5] implemented a unit testing framework for multi-threaded Java programs by integrating JUnit with Java Path Finder. All the possible scheduling of threads are first discovered using Java Path Finder and JUnit test methods are then applied to each thread schedule. Artho and Biere [1] developed JNuke that integrates automatic test coverage measurement based on the result produced by GNU C compiler utility gccov. This approach enables JNuke to detect untested or dead code.

The rest of this paper is organized as follows. The next section describes the common challenges faced in using JUnit for testing programming projects assigned to CS1/CS2 students. In Section III we briefly describe how our CS1 and CS2 courses are administered and how projects are assigned. In Section IV we present relevant assignments where the above challenges appeared. By presenting these assignments, we hope that readers will be able to adopt the techniques for their own courses. Section V presents the advanced Java features we used in conjunction with JUnit assertion methods to address the above challenges. In section VI, we describe some of the issues encountered by our students when they ran our tester code. These issues served as feedback for improvement in designing our test cases. And finally, we conclude with new insights we learned from using JUnit in our CS1 and CS2.

II. Challenges

A. Capturing Standard Output

When a class is designed to include the typical setter and getter methods, writing a JUnit test suite for this class is straight forward; the test cases will mainly consist of test methods that call one of the
"setter" methods followed by one or more assertions on the corresponding "getter(s)". However, during the first few weeks of a CS1 course it may be too early for students to understand the concept of returning a result from a function. In fact, first-time programmers may (mis)understand that "returning a result" means printing the result to standard output. Consequently, the first few programming assignments in the semester are typically designed to include methods that print the results via standard output. This approach allows students to immediately verify the results. However, designing automated testing for this kind of assignment is not straightforward.

B. Generating Random Input

Programming assignments that require conditional statements are usually designed to include methods that process varying input data. Based on the range of input values, these methods are supposed to produce different output. A common approach for handling this situation is to require the student program to read input from a file, but it may add extra complexity of using files and handling input/output exceptions. To prepare such an assignment, the instructor will have to prepare both the input data and the expected output for each corresponding input. An alternative technique is to read data from standard input. The JUnit tests for this kind of assignment must be designed to "inject" input data into the student's program's standard input. In addition to injecting input data, the instructor's tester code can also generate the expected result for the corresponding injected input.

C. Escaping Infinite Loops

Incorrect implementation of a programming assignment that requires loops may contain bugs that cause a method to run in an infinite loop. Consequently, a JUnit test method that attempts to verify the correctness of the method may get trapped inside the infinite loop. These loops must be detected and escaped.

D. Testing Extra Credit

Instructors occasionally offer extra credit or programming assignments are designed to allow students to implement additional features. Testing the additional features poses a design challenge for the instructor. If the test cases for the extra features are hardcoded into the test file, it may generate compile errors for students who decide not to implement the extra features. One solution is to provide a separate test file for testing only the extra features, but this may require extra setup on the students' part as well as maintenance overhead for the instructor. A better solution is to use Java reflection to detect at runtime whether a student's implementation includes the extra method(s).

E. Setting Up Preconditions

It is quite common to find methods that require a precondition be true before we can verify that these methods satisfy the requirements. For instance, a method that draws cash from a bank account requires a non-zero balance. Setting up the precondition for such simple methods is straightforward, but other methods may require more involved setup. In a game of Chess, testing whether a King is "in-check" requires the chess board be set up by placing certain chess pieces at selected positions.

F. Discovering Class Information at Runtime

Java interfaces give programmers freedom to choose the name of the implementing class. For a programming assignment that requires students to implement instructor-provided interfaces, students may choose any name for their implementing class. As a consequence, instructor tester code cannot assume a particular class name will be used in students' implementation; the class name must be determined at runtime.

III. Course Administration

Our CS1 was taught in Java with BlueJ 3.x [3] as the required IDE. Five projects were assigned in the semester, three of them were released with JUnit test cases. The first and last projects were "do-it-yourself" projects where each student was given the flexibility to design the final content of the projects, with some minimal features provided by the instructor; no unit testing was used for these two projects. Our CS2 was also taught in Java but we let our students use an IDE of their preference. Unlike in CS1 where test cases were provided by the instructors, our CS2 students are required to write JUnit test cases.

A team of several instructors taught multiple sections, and one of the instructors was assigned as the course coordinator. For our CS1, all projects were designed by the course coordinator, but the corresponding test cases were developed by a different instructor on the team. For our CS2, each instructor designed one project which later shared by the team.
Having a different test developer helps to reduce ambiguous instructions in the project handouts. Each project release included an online handout of the project requirements and a set of JUnit test cases contained in a single .java file.

IV. Programming Assignments

A. CashRegister

The class CashRegister simulates a simple cash register object that maintains two instance variables: currentAmountDue and totalDailySales. In addition to the typical get methods, the class also provides the following methods:

- void scanPrice(double price): add the given price to the current amount due
- void completeSale(): calculate the tax and add it to the current amount due. This method is invoked after the last item is scanned
- void cancelSale(): cancel the current transaction
- makePayment(double pay): receive payment of the given amount from the customer
- clearAllSales(): clear the total daily sales. This method requires the user to confirm the operation by answering a "y" or "n" via the standard input.

In contrast to typical getter methods that return non-void, the above methods return a void and they must display (via System.out.println) textual output that contains either a monetary amount formatted with NumberFormat or an error/warning message.

For this project, the following challenges were addressed when writing the JUnit test cases:

- to capture these textual outputs and verify that they are correct and satisfy the requirements.
- to automatically generate a random "y" or "n" response when testing clearAllSales().

B. A Game of Pig

The class Pig is an implementation of a two-dice game in which two players take turns to earn at least 100 points. Rolling a pair of 1s (1,1) causes the current player to lose all his points, while rolling a 1 (1,n) causes the current player to lose only the points earned in the current round. In both cases, the current player loses his turn and the opponent continues the game. At any time, the current player may decide to hold and let the opponent continue. One of the players is human and the other is the computer.

In addition to the typical getter methods, the class must provide the following methods:

- void playerRolls() to simulate the current player rolling the two dice
- void playerHolds() to simulate the current player holding his current turn
- void computerTurn() to simulate the computer's entire turn, the computer continues to roll until 20 points are accumulated or turn is lost
- void playerTurn() to simulate the human player's entire turn
- void restart() reset all the instance variable to start a new game
- boolean playerWon() and boolean computerWon() to check whether the human/computer player has won the game

Unlike the CashRegister project, this project has a GUI component with most of the solution provided by the instructor. The students were required to adapt the provided GUI code to display appropriate states from the Pig game engine. In addition, this game requires a supporting class Die, which is also provided by the instructor.

The challenge in designing JUnit test cases for this project is to capture and verify textual output produced on the program's standard output.

C. The Game of Chess

The game of chess assigned to our CS2 students was designed to enforce the concept of behavior-based inheritance. For this assignment, we provide the following interfaces in a JAR file.

- IChessPiece an interface that defines the behavior of a chesspiece. The core method of this interface is boolean isValidMove (Move m, IChessBoard b);
- IChessBoard an interface that mainly serves as a wrapper of a two-dimensional structure of a chessboard.
- IChessModel an interface that defines the behavior of a typical chess game.

In the project, students are required to implement a base class (ChessPiece) that implements the IChessPiece interface. Other chesspiece classes(Pawn, Rook, Knight, Bishop, Queen, King) then inherit from ChessPiece. Each chess piece has a unique set of rules to move, but they all also share a common set of rules:

- A chess piece can only be moved to a different location (the source and destination must be two different locations on the board)
A chess piece can only be moved by the player who owns the piece.

A chess piece can capture only another piece of a different color.

The above common set of rules are verified by the base class ChessPiece, while rules specific to individual chess piece are verified by the corresponding classes.

For our CS2 projects, students must implement provided interfaces and develop their own test cases. Unlike CS1 projects where students must use specific class names in their projects, CS2 students were allowed to use any name for their Java classes as long as they implement the specified interfaces. The main challenges for writing JUnit test cases for such project is to use Java reflection to discover the actual class names at runtime.

V. Developing Test Cases

This section describes how we developed the project test cases.

A. Capturing Standard Output

To capture a program’s standard output, we use the System.setOut method for output redirection. Combining this method with an OutputStream subclass that overrides its write() method, we were able to log any System.out.println originating from a student’s program to a StringBuilder object created in the JUnit test fixture. This technique allows the test cases to verify the expected output by inspecting the StringBuilder object. The overall setup is shown in Figure 1. Once the output redirection setup in a JUnit test fixture, a test method to verify the System.out.println() output produced by one of the CashRegister methods is shown in Figure 2.

B. Generating Random Input

The CashRegister project requires our tester framework to supply a random "y" or "n" for testing its cancelAllSales() method. The technique we used to achieve this task is to combine input redirection with a PipedInputStream. The overall setup is shown in Figure 3.

With the above input redirection initialized in a JUnit test fixture, a test method that requires a controlled input to the standard input (cancelAllSales() in the following example) can now be written as follows:

```java
public class CashTest {
  private static StringBuilder textLog;
  private static PrintString originalStdout;

  private static OutputStream logger =
    new OutputStream() {
      @Override
      public void write (int b) {
        textLog.append ((char) b);
      }
    };

  @BeforeClass
  public static void setUpBeforeClass() {
    textLog = new StringBuilder();
    originalStdout = System.out;
    System.setOut (new PrintString (logger));
  }

  @AfterClass
  public static void tearDown() {
    System.setOut (originalStdout);
  }
}
```

Fig. 1. Redirecting Standard Output to an OutputStream

```java
@Test
public void testScanPrice() {
  CashRegister cash = new CashRegister (...);
  // reset the text logger
  textLog.setLength (0);
  cash.scanPrice (25.99);
  String tester = textLog.toString();
  assertTrue (tester.indexOf(EXPECTED) >= 0);
  assertEquals (EXPECTED_LENGTH, tester.length());
  // more Assert calls
}
```

Fig. 2. Using textLog in a test method

In Figure 4 the invocation of the method yesNo.generateResponse() seems to be out of order; it was called prior to calling clearAllSales(). The sequence is actually correct because the “n” input must be injected to the program standard input prior to the actual reading of the standard input by clearAllSales(). Swapping the order of the two calls will result in a deadlock.

C. Escaping Infinite Loops

Getting trapped in an infinite loop is a common mistake made by novice programmers. When a method may potentially get stuck in an infinite loop,
public class CashTest {
    private class YesNoResponder extends PipedOutputStream {
        public void generateResponse (boolean flag) {
            write (flag ? 'y' : 'n');
            // place a newline character to simulate
            // the <ENTER> key
            write ('\n');
        }
    }
    private static YesNoResponder yesNo;
    @BeforeClass
    public static void setUpBeforeClass() {
        // other code here
        yesNo = new YesNoResponder();
        PipedInputStream fakeKbd =
            new PipedInputStream();
        yesNo.connect (fakeKbd);
        System.setIn (fakeKbd);
        // other code here
    }
}

Fig. 3. Using PipedOutputStream for injecting input

private class TesteeThread implements Runnable {
    @Override
    public void run() {
        // invoke the testee method here
    }
}

private class TesteeThread implements Runnable {
    @Override
    public void run() {
        // invoke the testee method here
    }
}

@Test
public void testClearSales() {
    CashRegister cash = new CashRegister (...);
    textLog.setLength (0);
    // reset the text logger
    boolean resp = myRandom.nextBoolean();
    // place a random "y" or "n"
    yesNo.generateResponse (resp);
    cash.clearAllSales();
    // more Assert calls
}

Fig. 4. Generating a random response

the associated tester method should be designed to
provide a graceful way of escaping the infinite loop.
JUnit4 provides the @Timeout annotation for this
purpose.

An alternate solution is to run the testee method
in a thread separate from the tester thread. After
spawning a new thread, the tester/main thread can
use a timer to set a time limit for the testee thread.
The overall setup is shown in Figure 5

D. Exploiting Inheritance

The test cases for the Game of Chess are organized
using the same structure as the inheritance hierarchy
of the classes under test. We first created a JUnit
test ChessPieceTest for testing ChessPiece. The
rest of JUnit test files inherit from ChessPieceTest.
Moreover, to handle differences between the black
and white pawns “forward” directions, we created an
intermediate abstract parent PawnTest that inherits
from ChessPieceTest. Then, we created Black-
PawnTest and WhitePawnTest that overrides the
"forward()" method.

E. Include Required Constructors

As Java interfaces are introduced in a program-
ing assignment, the instructor test cases must be
designed to discover the implementing classes at
runtime. Since the signature of a constructor cannot
be enforced via an interface, the assignment should
specify the expected signature of the constructor(s) of
the implementing classes. Without this requirement it
would be impossible for the instructor tester code to
instantiate Java objects needed for testing. Some of
the constructors can also be used to facilitate testing
those methods that require a particular precondition.

VI. Issues in Designing Test Cases

A. Avoid Silent Failures

For each assert method, JUnit provides at least the
following two variants:
assertTrue (expectedValue, actualValue);
assertTrue ("hint / message", expectedValue, actualValue);

Using the first variant does not help students diagnose why their code failed. JUnit simply reports that the test failed. The second variant can include useful hints, provided in the string message, to help students fix their code. For example, consider a method designed to calculate the average of numbers within an array. The following messages provide a range of feedback.

- **Poor**: no message
- **Good**: calcAverage() does not work correctly
- **Better**: calcAverage() result is close but not correct. Are you using a double for the average?
- **Best**: calcAverage() result is not correct. Did you consider the possibility of zero elements?

B. Avoid False Positives

We found cases where an incomplete student implementation produced positive test results. One cause of these false positives was due to Java default behavior of automatically initializing variables to zero,false, or null. In other situations, there was an insufficient number of test cases to cover all possible conditions. For example, we neglected to test for a divide by zero exception in the calcAverage() method. Student solutions passed the test but did not check for zero elements. The error was identified when reading student solutions but was not identified as intended with the automated JUnit tests. Test cases must be robust and anticipate a range of input permutations and error conditions. Otherwise, student may develop a false-sense of confidence in the instructor-provided test cases.

C. Provide JUnit Documentation

JUnit testing can be used throughout the first programming course. Students can start by using JUnit tests as a black box and eventually progress from reading the instructor-provided test cases to writing their own by the end of the course. An effective way for students to learn is to read the instructor-provided cases if they are well commented. The more advanced students are naturally curious and want to know what the test cases do.

D. Avoid Shadowing of Protected Variables

We also found cases where students’ test cases may show positive results while instructor’s provided test cases did not due to shadowing of a protected variable. In one of the assignments where students were required to use inheritance, a student incorrectly redeclared an instance variable in a child class which has been declared in the parent class, thus hiding the visibility of the parent’s variable. This type of bug can be difficult to locate (even for an experienced programmer), but very easy to fix. If type of bugs causes a potential source of confusion to the students, the instructor’s test cases should use Java reflection to identify such bugs.

VII. Conclusion

Novice programmers can use and benefit from JUnit tests in an introductory programming course. Robust tests can be created using advanced Java features such as reflection to detect at runtime if a method is present, capturing standard output using OutputStream, generating random input from a PipedInputStream, and using the @Test annotation for detecting infinite loops.

There are several best practices that help make the process more effective from the student’s perspective. First, it is essential to write clear warnings and error messages that provide helpful clues. Second, warn students not to get too stressed about a single failed test. It could be a minor error and they should use their time implementing additional methods before returning to identify the error. Otherwise, students spend too much time on trivial problems and fail to make sufficient progress on the entire assignment. Third, JUnit java code should have internal comments written for a novice programmer. Students can read the code to learn how to adapt and write their own tests. And finally, tests must be accurate and comprehensive. Students must be able to trust and rely upon the instructor-provided test cases.

Acknowledgment

We would like to recognize our colleague, Zack Kurmas for the time and energy he spent on designing the Game of Chess assignment. We believe the assignment is very well-designed and has helped our students to gain comprehensive understanding of inheritance, interfaces, and polymorphism.

References


