Unit testing: Test early, test often

Article · January 2003

CITATIONS
74

READS
11,793

1 author:

Michael Olan
Stockton University
16 PUBLICATIONS **109** CITATIONS

SEE PROFILE

UNIT TESTING: TEST EARLY, TEST OFTEN*

Michael Olan
Computer Science and Information Systems
Richard Stockton College
Pomona, NJ 08240
609-652-4587
olanm@stockton.edu

ABSTRACT

Testing is a critical part of good software development, but often gets only minimal coverage in introductory programming courses. Unit testing and selected aspects of test-driven development can be used to improve learning and encourage emphasis on quality and correctness. Tools like JUnit significantly simplify the generation of test cases. An additional benefit for instructors is that these tools can also be used to automate project grading.

1. INTRODUCTION

Validation is a process designed to increase confidence that a program functions as intended. Generally, validation involves testing. The desired outcome of testing is a guarantee that a program satisfies its specifications, but as noted by Edsger Dijkstra, testing can prove the presence of errors but not their absence. Nevertheless, careful testing greatly increases the confidence that a program works as expected.

Programmers should never assume that their code is correct. Yet beginning programmers, who often measure success as a program that compiles without syntax errors typically make such assumptions. By gently introducing testing early, students can gain confidence in and develop a sense of responsibility for the correctness of their work.

Software testing has come a long way from the use of debugging print statements cluttering a program's source code and contributing to "scroll blindness". Sorting through voluminous

---

output to interpret test results is a tedious and highly subjective process. Yet another major disadvantage of this practice is a lack of automation and flexibility that are needed to run tests early and often during the development process.

2. UNIT TESTING

The growth of object oriented programming has influenced the way programmers approach software testing. Being predominantly bottom-up, it is natural that object oriented programming favors a similar testing methodology that focuses on classes.

A unit test exercises a "unit" of code in isolation and compares actual with expected results. In Java, the unit is usually a class. Unit tests invoke one or more methods from a class to produce observable results that are verified automatically.

Proponents of Extreme Programming (XP) recommend test-driven development as a desirable and effective way to develop software. This technique involves first designing test cases, and then developing class behavior that will satisfy the tests. Class implementation is done incrementally, with tests performed whenever a change is made [2].

3. UNIT TESTING IN INTRODUCTORY COURSES

The choice of when (or if) to introduce sophisticated tools may vary, but planting the seeds of unit testing is entirely appropriate in CS1. For example, the instructor can provide students with a simple class complete with specifications, and have them experiment with the methods by writing a test driver. The goals of this exercise include learning to use prewritten software components, reading specifications, and testing to determine whether the components work as expected. Such experiments can also be used to illustrate the limitations of software components, and the importance of finding and documenting these limits. Experimenting out of context may also reduce the tendency of beginning students to program only for a specific application, rather than designing good reusable components.

Performing both positive and negative tests should be encouraged. Positive tests exercise the expected functionality of the test object. Negative testing uses cases that are expected to fail, where the code should handle the error in a robust way [1]. Students can experiment to determine which argument values work and which do not. Such an approach encourages examination of class and methods specifications. This analysis may then be used to determine restrictions on parameters, and formulate preconditions, postconditions, and invariants.

The BlueJ IDE provides a simple means to interactively test classes and methods without the need for writing a test driver. Using the object bench in BlueJ, an object can be created, and its methods executed interactively. Inspectors allow viewing the object's state, thus giving immediate feedback on the effects of executing the method [1].

The interactive testing provided in BlueJ has limitations. It is not repeatable or automated, so tests that uncover bugs must be repeated manually after corrections have been made. This is particularly troublesome when a significant amount of manually supplied data is needed, as

when filling a large array. These situations call for regression testing, which involves repeating tests to ensure that modifications have corrected the problem and not have not introduced new errors [7]. To be practical, such testing must be automated.

Minimizing the amount of human involvement required in testing is desirable. Automated regression testing is most effective when tests are self-checking, and results reported to the programmer indicating success or failure of each test. The development of regression tests can be tedious, and potentially error prone. Indeed, the amount of code needed for tests may exceed the amount of code being tested. However, tools such as JUnit can greatly reduce the amount of work needed to create test cases.

It is also possible to test using a debugger. Setting breakpoints, watches, etc. is an effective way to trace execution of a program to determine the cause of a bug. But an interactive debugger is not suitable for regression testing. Debugging is done in retrospect to isolate an error once its existence has been detected. Unit testing is better suited to the early detection and isolation of bugs, and can significantly reduce the need for debugging.

Tests in the BlueJ object bench or debugger use do not compose well. Basically, they are limited to executing one method call or expression at a time. They also require human judgment to analyze the results.

4. TEST-DRIVEN DEVELOPMENT (TDD)

Testing software from the beginning and throughout the entire development cycle is an essential software engineering practice. The Extreme Programming (XP) method takes this a step further and recommends an evolutionary approach to design that follows a test-code cycle of continually switching between coding and testing. Test-driven development involves formulating the tests before writing implementation code [2]. The TDD cycle is:

- Add a simple test
- Run all tests (some or all will fail)
- Make a small change
- Run the tests and succeed
- Refactor to improve design

Refactoring involves restructuring existing classes and methods, for example dividing a class or method into parts. Restructuring source code is done to improve its design and quality. However, any change will potentially introduce errors and thus tests must be repeated often.

Specifications play an essential role in all aspects of software design, including unit testing. In TDD, the roles of specifications and tests are intertwined. Unit testing combines aspects of traditional black box and glass box testing. The specifications supply the guidelines for use of a method (preconditions) and its expected results (postconditions), a black box view. However, in TDD where implementation code is developed along with the tests, the programmer also has a glass box view.

Test first gives a better idea of what features need to be in a class, and how they should work. For example, the nature of the JUnit testing framework exposes the need for getter and setter methods almost immediately. Using this technique is more likely to result in reusable code.

In general, tools do not automate the generation of test data. Constructing appropriate test inputs is a difficult and nonalgorithmic process. Rules concerning the use of typical, boundary, and error cases still apply. The tester must also decide in advance what outputs are appropriate for any set of inputs. What can be automated is the invocation of test methods with predefined inputs and checking the acceptability of the results.

5. UNIT TESTING IN JAVA WITH JUNIT

This section will illustrate the technique of unit testing using the Java programming language. The example presented will test a simple Java class to represent bank accounts, with methods for creating accounts, making deposits and withdrawals, and querying the account balance. Unit tests will demonstrate the JUnit testing tool.

JUnit [6] is a framework for unit testing in Java. With JUnit, a test case is usually implemented as a subclass of `junit.framework.TestCase`. Predicate functions, including a variety of `assertEquals` methods for various argument types, `assertTrue`, `assertNull`, and their complements provide the basic testing functionality. These methods compare actual with expected results.

Objects subject to testing are referred to as the *test fixture*. A `setUp()` method is used to initialize the test fixture, and a `tearDown()` method releases the fixture. A test case can contain any number of test methods. A test method typically tests one method from the class under consideration.

The execution of a test method in JUnit follows the model:

```
setUp();
testXXX();
tearDown();
```

JUnit invokes `setUp` before each test method is executed, and invokes `tearDown` after each test method completes. If a test fails, execution of the current test method is aborted.

A JUnit test case repeats the above model, using reflection to find all test methods. Test methods must be public, void, take no arguments, and have a name that starts with "test" (so that reflection can determine which methods to call).

The following example illustrates a JUnit test case for a simple class representing bank accounts having a name and an integer balance.

```
import junit.framework.TestCase;
public class AccountTest extends TestCase {
   public AccountTest(String arg0) {
        super(arg0);
   }

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

```
        junit.swingui.TestRunner.run(AccountTest.class);
    }
    Account a;   // test fixture
    public void setUp() {
       a = new Account("D. Morebucks", 500);
    }
    public void testDeposit() {
       a.deposit(200);
       assertEquals("deposit error: ", 700, a.getBalance());
    }
    public void testWithdraw() {
       a.withdraw(200);
       assertEquals("withdraw error: ", 300, a.getBalance());
    }
}
```

The Account object a is the test fixture in this example. The setUp method initializes the fixture before each test method is run. Each test method exercises a single method from the Account class, and use assertEquals to analyze the results. A tearDown method is not needed in this example since Java automatically garbage collects unused objects. Typically, tearDown is used to release resources, such as closing a file or database connection.

JUnit has both text and GUI interfaces. This example uses the Swing GUI interface to JUnit, with the results shown in Figure 1. The JUnit window shows the number of test methods

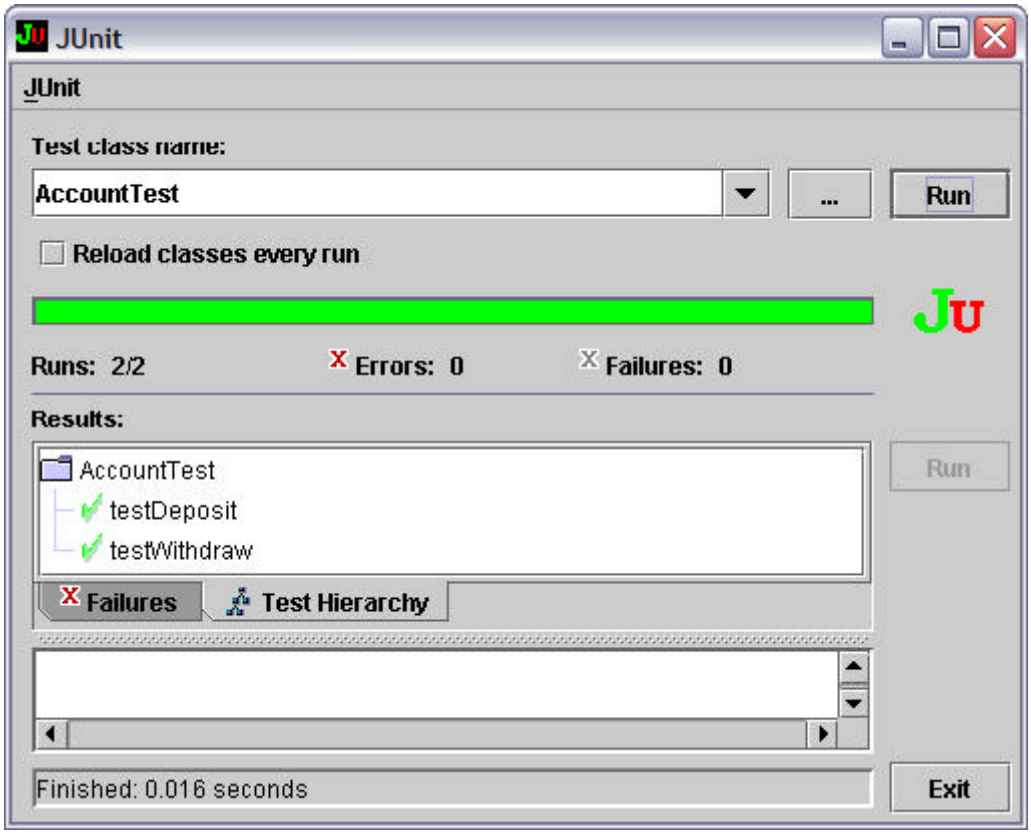

Figure 1

run, number of errors, number of failures, and a "Green Bar" for a fully successful test. The **Test Hierarchy** tab shows which methods succeeded and which failed. The **Failures** tab would display details of any failed test methods.

Suppose that the `Account` class uses the following naïve implementation of the withdraw method.

```
// Subtracts amt from balance of this Account
public boolean withdraw(double amt) {
  balance = balance - amt;
}
```

If transactions that cause a negative balance are to be rejected, the following test would expose a bug.

```
public void testWithdraw2() {
   a.withdraw(800);
   assertEquals("withdraw error: ", 500, a.getBalance());
}
```

The results shown in Figure 2 indicate that a failure occurred by displaying a "Red Bar". Again, the number of test methods run, and the number of errors and failures are shown. The **Failures** panel identifies the method that failed, and displays the message string argument of

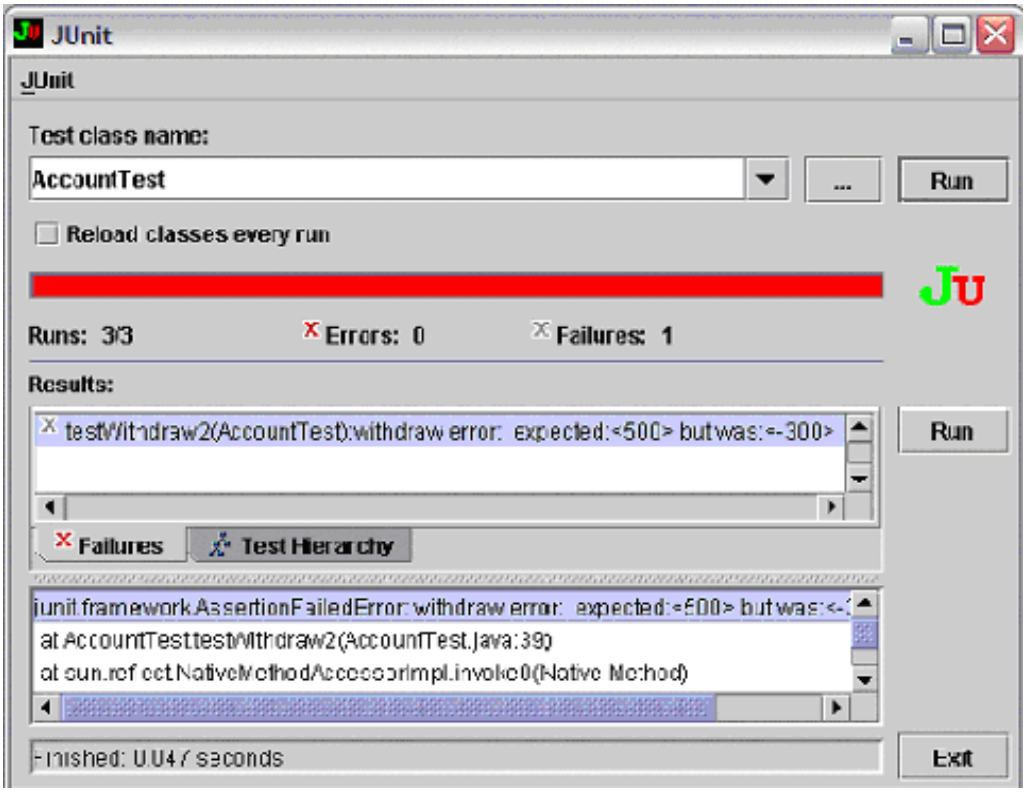

Figure 2

`assertEquals`, the expected result, and the observed result.

The **Run** button to the right of the **Results** panel is used to selectively run any single test method in the hierarchy. The bottom panel gives a stack trace of the failed test methods.

JUnit distinguishes errors from failures. Failures are anticipated by the assert methods. Errors are unanticipated problems resulting from uncaught exceptions propagated from a test method.

Now that a problem in withdraw has been detected, it needs to be fixed. We can improve the method by throwing an exception.

```
// Subtracts amt from balance of this Account
// Throws InsufficientFundsException if balance < amt
public void withdraw(double amt) {
     if (balance < amt)
        throw new InsufficientFundsException();
     else
        balance = balance - amt;
}
```

The `testWithdraw2` method now succeeds, but an additional test is needed to verify that the exception is thrown when appropriate.

```
public void testWithdraw3 () {
   try {
       a.withdraw(501);
      fail( "Expected InsufficientFundsException");
     } catch (InsufficientFundsException e) {
           // exception test succeeded
     }
}
```

If the expected exception is not thrown, the `fail` method causes the test method to fail immediately. Since the `InsufficientFundsException` would be thrown and caught, the test in this example succeeds.

As a project grows and the number of test cases increases, *test suites* become useful for organizing test cases. A test suite is a collection of related tests. There are several ways to build a JUnit test suite. One is to add each test method individually by name, as in the following example.

```
public static Test suite() {
   TestSuite suite = new TestSuite("Test everything");
   suite.addTest(new AccountTest("testDeposit"));
   suite.addTest(new AccountTest("testWithdraw1"));
    suite.addTest(new AccountTest("testWithdraw2"));
    suite.addTest(new OtherAcctTests("testWithdraw3"));
    suite.addTest(new OtherAcctTests("testWithdraw4"));
    return suite;
}
```

A simpler approach is to add all test methods from a test case as in this example.

```
public static Test suite() {
   TestSuite suite = new TestSuite("Test everything");
   suite.addTestSuite(AccountTest.class);
    suite.addTestSuite(OtherAcctTests.class);
    return suite;
}
```

This will include all methods from the indicated classes with names that begin with "test".

Since JUnit test cases are Java classes, javadoc comments can be inserted and documentation pages easily generated. Thus, documentation of the test plan is a simple and integral part of the overall test-development process.

JUnit is relatively easy to utilize manually or with an IDE. Both BlueJ and DrJava, simple IDEs for educational use, now have support for JUnit built in.

At a more sophisticated level, the open source eclipse IDE integrates JUnit in a particularly flexible way. Eclipse includes a wizard for automatically generating test method stubs. IDE's like eclipse also support hot code updates, which will allow JUnit tests to be run without recompiling the test suite. Eclipse can also be configured to launch a debugger when a failure occurs.

6. EXTENSIONS AND ADVANCED FEATURES

Testing programs with graphical user interfaces can be very tedious and challenging. Users can click buttons randomly, and supply input in random order. A number of extensions of JUnit have been developed for testing GUI programs by recording and replaying keystrokes, mouse clicks, etc. (Abbot, Marathon).

Other extensions have been developed for server-side code testing (Cactus), J2EE web applications (JUnitEE), and more.

Unit tests should test a single piece of functionality, but most nontrivial code is difficult to test in isolation. Code that interfaces with remote servers, accesses databases, etc. can be particularly challenging to test. A technique called Mock Objects replaces domain code with dummy implementations that emulate the real functionality of complex component interaction Several JUnit extensions are available for mock objects (EasyMock, Mock Maker).

Information about extensions is available at the JUnit web site [6].

Unit testing tools were first developed for Smalltalk by Kent Beck who also was involved in developing JUnit. A growing family of similar tools for other languages includes Ada, C, C++, Curl, Delphi, Eiffel, JavaScript, Perl, Python, and the Microsoft .NET languages [10]. The tutorial paper "JUnit Cookbook" [3] has been adapted for a number of these languages as well.

7. UNIT TESTING FOR AUTOMATED PROJECT GRADING

With tools such as JUnit, at least partial automation of project grading is relatively easy. For example, with the test harness in place, student submissions can be dragged into an eclipse project, and the test run. Dragging a Java source file into eclipse automatically compiles it. Dynamic hot code updates allow the JUnit test framework to rerun a test using the new code.

On a Unix/Linux system, it would be straightforward to write a script to automate this process even further by cycling through submissions, and directing output results to files.

The Ant build tools for Java can further automate the testing process [5], with the results gathered in a formatted report and emailed to a recipient. Although additional grader comments should be provided, a significant savings in the grading process is possible.

Another possibility is setting up a submission mechanism where students can run their code through a prepared test driver that would provide feedback about failed tests. This mechanism would also reinforce the importance of precise specifications and standards. If a student does not write code conforming to the specification of a class, in terms of method names and signatures, the test cannot be run. It could also reduce the number of unacceptable submissions since errors that students would likely miss could be exposed and corrections made before a final submission is due. Although this might encourage programming for the test driver, it is consistent with recommendations for TDD made by Kent Beck in [2].

8. CONCLUSIONS

Building reliable software requires a testing framework. Adding unit testing to the programmer's toolkit can improve design and significantly reduce the time spent hunting for mysterious bugs. Tools like JUnit can be effective for encouraging students to thoroughly test the software they write. Such tools enhance the learning process, and are not simply bells and whistles added to an IDE. With an early introduction to unit testing, and support from these tools, students too may become "test infected" [4].

9. REFERENCES

[1] Barnes, David J and Kölling, Michael *Objects First With Java: A Practical Introduction Using BlueJ*, Prentice Hall, 2003.

[2] Beck, Kent *Test-Driven Development By Example*, Addison-Wesley, 2003.

[3] Beck, Kent and Erich Gamma "JUnit Cookbook", http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[4] Beck, Kent and Erich Gamma "Test Infected: Programmers Love Writing Tests", *Java Report*, July 1998.

[5] Hightower, Richard and Nicholas Lesiecki *Java Tools for eXtreme Programming*, Wiley, 2002.

[6] JUnit website, http://www.junit.org.

[7] Liskov, Barbara and John Guttag *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, 2001.

[8] Savarese, Dan "Take the Work Out of Unit Testing", *JavaPro*, October 2001.

[9] Succi, Giancarlo and Michele Marchesi *Extreme Programming Examined*, Addison-Wesley, 2001.

[10] XProgramming website, http://www.xprogramming.com/software.htm.