

UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B
Fall 2002

P. N. Hilfinger

Testing with Jtest and Vjtest

1 Testing: The Essential Pariah

Testing has always been one of orphan children of software development. Programmers (especially students, it sometimes seems) are reluctant to spend significant time on an activity that doesn't *seem* to advance their products' development. But in fact, the time you don't spend writing tests generally turns into time you do spend debugging. Perhaps it's occurred to you—after typing the same long, failure-inducing input to your program for the umpteenth time—that you could have just written the input into a file somewhere once, and then re-used the file during debugging. Well, that file of input is a test. You're going to write it anyway, so why not do so in as useful and re-usable a fashion as possible?

Of course, essentially every software company releases programs that are erroneous to some degree and receives “bug reports” from its customers. In well-run companies, such reports themselves get turned into tests, called *regression tests*, that may be run in order to insure that future versions of the software don't re-introduce the problem. Indeed, an increasingly common procedure is to run a suite of regression tests automatically (each night, for example), recording the results and notifying appropriate individuals.

It isn't necessary to have a complete program to begin testing. A *unit test* is a test of a particular component or subsystem of some program or other system. Indeed, in many homework problems we don't ask you for main programs at all, but just some methods. You have to write extra stuff to do any testing.

To facilitate unit testing, or indeed testing in general, we have provided you with a testing framework generically called the *JTest Framework*. The design of the JTest framework borrows heavily from the [JUnit 3.2](#) framework developed by Kent Beck and Erich Gamma.

2 Using the JTest Framework

There are two user interfaces to the JTest framework for running tests: a text-based command-line program called simply `jtest`, which you will find used in Makefiles we supply; and a graphical user interface called `vjtest`. Once you have compiled your program files and your testing files (see §3, below), you can run either of these to test. Assuming that your testing class is named `MyTests` (presumably compiled from a file called `MyTests.java`), the command

```
jtest MyTests
```

to your shell (command-line processor) will run the tests and report problems, and

```
jtest -v MyTests
```

will do the same, but provide more detailed output.

For a graphical interface, simply type

```
vjtest MyTests
```

or simply

```
vjtest
```

(after which you can enter “**MyTests**” under “Name of test class”). You’ll get a window with buttons and text areas for running tests, looking at the results, and displaying the detailed results of individual tests (“Show”).

3 Writing Your Own Tests

The JTest framework uses classes from package `ucb.tools.testing`, which we have installed for this course on the instructional machines. Suppose that you have written an `Account` class and wish to test that three methods you’ve written—`deposit`, `withdraw`, and `balance`—are giving correct results. You might write a test class such as `AccountTests` in Figure 1.

First, the `AccountTests` class extends `Tester`, one of the classes in `ucb.tools.testing`, a package we supply. The `Tester` class provides various methods used for testing, and introduces `AccountTests` into the JTest framework. Each public, parameterless method in `AccountTests` whose name starts with “`test`” constitutes one test; we’ll call these “test methods,” appropriately enough. You may also have other methods used to help implement the test methods; they won’t confuse the testing framework if you don’t make them public.

Each test method performs whatever computations it needs to; their content is entirely up to you. The JTest framework provides a set of methods that you can use to communicate the results of a test. First, there are a number of assertion-checking procedures:

check(*condition*) tests that *condition* (a boolean expression) is true, and reports a test failure otherwise.

checkEquals (*test-value*, *ref-value*) tests whether *test-value* and *ref-value* (which can be any expressions) are “equal”. For numbers, “equal” means just `==`. If *ref-value* is a `String`, then *test-value* is first converted to a string using the `.toString()` method, and then the two values are compared using string comparison. For other reference (object) types, comparison uses the `.equals()` method.

checkEquals (*test-value*, *ref-value*, *modifiers*) tests that *test-value* and *ref-value* are equal, with details of the comparison controlled by *modifiers*. The *modifiers* are ignored unless *ref-value* is a `String`. Possible modifiers are defined as symbolic constants in `Tester`:

IGNORE_BLANKS throws away all blanks and tabs before comparison.

IGNORE_BLANK_LINES throws away all lines containing only whitespace before comparison.

TRIM_LINES throws away all whitespace at the beginning and end of lines before comparison.

IGNORE_TRAILING throws away all whitespace at the end of lines before comparison.

IGNORE_CASE treats upper- and lower-case versions of each letter as identical in comparisons.

IGNORE_EXTRA_BLANKS collapses adjacent runs of blank and tab characters into a single blank and also trims all leading and trailing blanks (as for **TRIM_LINES**) before comparisons.

You may combine several modifiers by using `+` or `|` (bitwise or).

checkEquals (*test-value*, *ref-value*, *relative-error*) tests whether *test-value* and *ref-value* (both can be any expression yielding a floating-point value) are equal to within a given relative error. That is, it tests to see if the difference between *test-value* and *ref-value* is greater than *relative-error* times *ref-value*. For example, a relative error of 0.01 means that the *test-value* may be off by no more than 1%.

Each of the methods above takes an optional string as a last argument, which is printed in the error message delivered if the check fails. This is useful to help you tell which of several checks is failing. You'll find examples in several of the calls on **check** and **checkEquals** in Figure 1.

Finally, there is also one convenience routine, illustrated in the **testOverdraw** method of **AccountTests**:

fail(*msg*) Causes immediate test failure, with *msg* as the error message.

4 Failures vs. Errors

In JTest terminology, there are both *failures* and *errors*. A “failure” refers specifically to a problem reported by **fail** or detected by one of the **check** methods. An “error” is any other kind of problem that halts the test. Mostly, these will be unhandled exceptions (as, for example, when the method you are testing dereferences a null pointer). The one other possibility is that your program exceeds the execution-time limit (which is imposed to prevent a runaway test from permanently holding up an entire test suite).

You may set the maximum execution time for a test to *S* seconds by including the call

```
setTimeLimit (S);
```

in the appropriate test method. Otherwise you get a default limit.

```
import ucb.tools.testing.*; // Location of JTest classes

public class AccountTests extends Tester {

    /** Test initial balance. */
    public void testBalance () {
        check ((new Account (100)).balance () == 100,
              "Initial balance for 100");
        check ((new Account (0)).balance () == 0,
              "Initial balance for 0");
    }

    /** Test deposit. */
    public void testDeposit () {
        Account a = new Account (0);
        a.deposit (3);
        check (a.balance () == 3, "After deposit(3)");
        a.deposit (100);
        check (a.balance () == 103);
    }

    /** Test withdrawal */
    public void testWithdraw () {
        Account a = new Account (103);
        a.withdraw (50);
        checkEquals (a.balance (), 53, "After withdrawing 50");
        a.withdraw (53);
        checkEquals (a.balance (), 0);
    }

    /** Test overdraft */
    public void testOverdraw () {
        Account a = new Account (50);
        try {
            a.withdraw (51);
            fail ("Overdraft not caught");
        } catch (IllegalArgumentException e) {
        }
    }
}
```

Figure 1: Example of a test class for a simple bank-account class.

5 Output to files

While a test is running, any output to the standard output (`System.out` or `ucb.io.StdIO.stdout`) and to the standard error output (`System.err` or `ucb.io.StdIO.stderr`) is redirected to internal buffers, rather than being printed. The collected output is available (as Strings) to your test methods using the two functions

```
getOutContents ()    and    getErrContents ()
```

So you might test a main program that is supposed to print “Hello, world!” with something like

```
public void testHello () {
    // Call main program with empty argument list. */
    Hello.main (new String[] { });
    checkEquals (getOutContents (), "Hello, world!\n");
    checkEquals (getErrContents (), "");
}
```

The call `clearOutputFiles ()` throws away any accumulated output, allowing multiple calls to output-producing functions during a single test.

The JTest framework limits the amount you may output to each of these files. You may change the limit from the default (4096 characters) to B bytes with the call

```
setOutputLimit (B);
```

When testing the output of a program to a file, it is often useful to compare that output to the contents of another file. For this purpose, the call `getFileContents(filename)` returns the contents of the named file as a String. For example:

```
Prog.doOutput (...);
checkEquals (getOutContents (), getFileContents ("trueout.1"));
```

6 Input from a String

By default, `System.in` reads from the empty file on each test (so there is an immediate end-of-file when it is read from). For convenience, the JTest framework provides a way to provide a source of input for `System.in` from either a file or from a string:

setInput(S) Sets the contents of `System.in` to S , which must be a String.

setInputFile(N) Sets the `System.in` to read from the file named N , (where N is a String).

7 Grouping Tests

You will often develop your tests in clusters, one test class for each major subsystem of your program, perhaps, or one test class per class. In this case, it's nice to have a way to produce a combined test set consisting of the union of these tests, while still retaining the option of keeping the clusters separate. For this purpose, the JTest framework provides a simple way to produce a test set that is simply a combination of other test sets. For example, suppose that you have two test collections: `AccountTests` and `UtilityTests`. To create a combine test set called, let's say, `AllTests`, you can write:

```
import ucb.tools.testing.*;
public class AllTests extends TestGroup {

    public AllTests () {
        super (new Class[] { AccountTests.class, UtilityTests.class });
    }
}
```

This gives a test set in which the tests are slightly renamed to avoid ambiguity: the `Balance` test, for example, will be identified as `AccountTests.Balance` in output. Simply follow the pattern above, substituting your combined test-class name for `AllTests` and your list of test class names for `AccountTests` and `UtilityTests` (the list can be as long as you like).

8 Handling a Main Program

It is possible to call a main program directly from within a test method. For example, to test that

```
java pig all gaul is divided into three parts
```

prints

```
allay aulgay isay ividedday intoay eethray artspay}
```

you could write a test containing

```
 pig.main (new String[] { "all", "gaul", "is", "divided",
                        "into", "three", "parts" });
  assertEquals (getOutContents (),
                "allay aulgay isay ividedday intoay eethray artspay\n");
```

However, there is a small problem: main programs will often terminate by calling `System.exit`, which in turn will halt the entire program, including the test method, before the check occurs.

To address this problem, the testing framework provides a method for executing an arbitrary command line, including a call to the java interpreter. So, for example, you can write the test above as

```
int code = runProgram ("java pig all gaul is divided into three parts");
assertEquals (getOutContents (),
              "allay aulgay isay ividedday intoay eethray artspay\n");
check (code == 0, "java pig had an abnormal return code");
```

The `runProgram` command runs the given command almost as if you had typed it as a command line to the shell, returning the “exit code” (which by convention is 0 for normal termination and non-zero otherwise; in Java, `System.exit(N)` exits a program with exit code N). The standard output and standard error of the command are collected as usual, and returned by `getOutContents()` and `getErrContents()`.

I said “almost as if” in the last paragraph because the argument to `runProgram` is broken into arguments at whitespace boundaries only; normal shell quotation characters are treated as ordinary characters. Thus,

```
runProgram ("java foo 'a sentence'");
```

passes *two* strings to the main function of `foo`: `"'a"` and `"sentence'"` (whereas if that string were typed on the usual Unix command-line, the main program would receive the one argument string `"a sentence"`). When this is a problem, there is an alternative form of `runProgram` in which you list the arguments in an array:

```
runProgram (new String[] {"java", "foo", "a sentence"});
```