THE SIMPLEST UNIT TEST TOOL THAT COULD POSSIBLY WORK

Charles D. Allison
Utah Valley State College
Orem, UT 84663
801-863-6389
allisoch@uvsc.edu

ABSTRACT

A minimal tool for unit testing is presented that can be used to advantage by novices for simple testing as well as by more experienced students and programmers for more rigorous unit test strategies. The goal is to get students accustomed to thinking seriously about correctness as soon as possible in the curriculum.

INTRODUCTION

Testing is an important part of developing software solutions that doesn't always get proper emphasis in undergraduate CS curricula. This attitude carries over into industry where being a tester is perceived as a position inferior to that of developer. Nonetheless, the cost of software bugs is staggering, and software quality assurance (SQA) is a key, bona fide profession employed by most businesses that make a living by creating software. The National Institute of Standards and Technology estimated in 2002 that the total annual cost of software bugs in the U.S. alone was $60 billion.[1] The average-per-bug cost for many large companies is $7,000 for bugs caught during system test and double that for bugs found in the field, while the cost for bugs found by developers is a fraction of that.[2]

What if there were no bugs in the first place? This may not be feasible, since humans develop software, but developers can take steps to minimize the number of bugs they pass on to quality assurance personnel. Instead of thinking through a careful design and test plan, developers and students alike often just "throw code together." Programming luminary Edsger Dijkstra suggested that

> "Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with." [3]

Striving for quality on both ends of the software development process is crucial to producing affordable and reliable software. There are many factors that affect software quality, not the least of which is the effort developers put into testing their software before releasing it for independent verification.

BARRIERS TO TESTING

Beginning CS students struggle enough learning the rudiments of problem-solving with computers, such as control structures, file I/O, and basic algorithm construction. To get them to simultaneously acquire a testing discipline is unreasonable and counterproductive. They could benefit, however, from a painless technique for basic testing that integrates nicely with their elementary software development efforts. Such a practice needs to be trivial to employ or it will just get in the way. Thus we seek the simplest possible approach to validating the small units of code that students write. They should not have to understand object-oriented technology or even have to worry about the details using an integrated development environment or other GUI applications. They should be able to write simple test code and have it just work.

UNIT TESTING

A unit test is what developers create and execute to verify that a piece of code works as expected. In fact, Test-Driven Development, a development (not testing!) discipline popular in today's agile methodologies, encourages developers to write test code before they implement the code under test itself. This lets programmers thoroughly scrutinize requirements before getting lost in the internals of physical design and implementation, while at the same time enabling an incremental approach to software development based on refactoring and regression tests. [4]

A unit test is a program that exercises the code under test as a client of that code. It should employ a variety of input sets that determine how the code reacts under a full spectrum of runtime scenarios (for example, boundary conditions). Unit test should also be *automated*, meaning that the test program itself should be able to determine if the code under test functions correctly without the human reader having to visually analyze detailed output data. In other words, a good unit test will automatically answer whether the expected output was obtained, and if not, what and where the specific failures were. This suggests that tests should be an organized collection of Boolean expressions. Many unit test tools in popular use support this approach, but are beyond the ability of beginning students.

Suppose students need to create a function with the following C-like interface that finds the smallest value in a sequence of integers:

```
// Returns the zero-based index of the minimal element
int min_element(int a[], int n);
```

A typical CS1 student would code the function and pick some random array to pass to it, after which he or she would visually inspect output to see if the smallest value was obtained. That would likely be the end of the development cycle for that function.

Now assume the availability of a single auxiliary function, `test()`, which records the truth value of its Boolean argument. Before even coding `min_element()`, a student could write a simple test driver to examine several execution scenarios. What if the array is empty? (Let's have it return -1.) What if there are multiple identical elements representing the minimum? (We'll return the index of the first one.) Here's a reasonable first attempt at a simple test driver in C/C++:

```
// A unit test for min_element
```

```
#include “test.h”       // The test machinery

int main() {
    int a[] = {7,2,8,1,4};
    int b[] = {2,2,2};
    int c[] = {1,2,8,7,4};
    int d[] = {7,2,8,4,1};
    test(min_element(0,0) == -1);
    test(min_element(a,3) == 1);
    test(min_element(a,5) == 3);
    test(min_element(b,3) == 0);
    test(min_element(c,5) == 0);
    test(min_element(d,5) == 4);
    report();
}
```

Figure 1 – A Test Driver for the `min_element` function

By forming tests as a series of Boolean expressions, students learn early to think carefully about how their code should behave before they write it, and also become accustomed to the idea of having the test program itself do explicit verification.

If everything works as expected, `report()` will print something like the following to standard output:

```
Test Report:

      Number of Passes = 6
      Number of Failures = 0
```

Figure 1 contains a sample implementation of `min_element()`.

```
int min_element(int a[], int n) {
    int pos = -1;
    if (n > 0) {
        ++pos;
        for (int i = 1; i < n; ++i)
            if (a[i] < a[pos])      //*
                pos = i;
    }
    return pos;
}
```

Figure 1 – A Sample Implementation of `min_element`

The author of this code was distracted by finals and other concerns, and consequently wasn’t thinking clearly and initially rendered the line marked with an asterisk above as

```
if (a[i] < a[i+1])
```

Here’s the result of that test run:

```
FAILURE: min_element(a,5) == 3 in file C:\min_element.cpp on line 25
```

```
FAILURE: min_element(b,3) == 0 in file C:\min_element.cpp on line 26
FAILURE: min_element(c,5) == 0 in file C:\min_element.cpp on line 27

Test Report:

      Number of Passes = 3
      Number of Failures = 3
```

The bugs were quickly found and fixed, but the important thing to notice is that attention was immediately directed to the calls that failed.

LEARNING WITHOUT A CURVE

The appeal of this approach is that even the simplest code can be tested in this manner; no classes need to be defined, no inheriting from a testing framework, no configuring a GUI, just a simple call to functions defined in a header file. Students don't even need to know how to *write* a function—they just need to know how to *call* one. The Java version (shown later on) requires only that the file `Test.java` (or its byte code in `Test.class`) be in the default class path.

Figure 2 contains the test management implementation from the file `test.h`:

```
// test.h: Simple but effective automated test scaffolding
// Author: Chuck Allison, 2003-2007
// Permission to use granted if accompanied by source citation
// (Note: the reason for the trailing underscore in fail_() is to avoid a
// name clash, since the identifier "fail" is defined in the <iostream>
// header.
#ifndef TEST_H
#define TEST_H
#include <iostream>

namespace {
    size_t nPass = 0;
    size_t nFail = 0;
    void do_test(const char* condText, bool cond, const char* fileName,
                 long lineNumber) {
        if (!cond) {
            std::cout << "FAILURE: " << condText << " in file " << fileName
                      << " on line " << lineNumber << std::endl;
            ++nFail;
        }
        else
            ++nPass;
    }
    void do_fail(const char* text, const char* fileName, long lineNumber) {
        std::cout << "FAILURE: " << text << " in file " << fileName
                  << " on line " << lineNumber << std::endl;
        ++nFail;
    }
    void succeed() {
       ++nPass;
    }
    void report() {
        std::cout << "\nTest Report:\n\n";
        std::cout << "\tNumber of Passes = " << nPass << std::endl;
        std::cout << "\tNumber of Failures = " << nFail << std::endl;
    }
```

```
}
#define test(cond) do_test(#cond, cond, __FILE__, __LINE__)
#define fail_(text) do_fail(text, __FILE__, __LINE__)
#endif
```

Figure 2 – The Code in `test.h`

To make a C version, all that is needed is to remove the `namespace` syntax, make all definitions `static`, and replace the use of `cout` with calls to `printf()`. Students need know nothing whatsoever about the implementation details—they just invoke the `test()` function

Figure 3 illustrates a Java version.

```
// Test.java: Simple automated test scaffolding
// Author: Chuck Allison, 2007
// Permission to use granted if accompanied by source citation

public class Test {
    static int nPass = 0;
    static int nFail = 0;

    public static final void test(boolean condition) {
        if (!condition)
            fail();
        else
            succeed();
    }
    public static void fail() {
        // The failure is 2 levels back from here in the call chain
        fail(3);
    }
    public static void fail(int level) {
        try {
             throw new RuntimeException();
        }
        catch (RuntimeException x) {
             // Extract file/line number where test was called
             StringBuffer message =
                 new StringBuffer(x.getStackTrace()[level].toString());
             message.delete(0, message.indexOf("(")+1);
             message.deleteCharAt(message.length()-1);
             System.out.println("failure in line " + message);
        }
        ++nFail;
    }
    public static void succeed() {
        ++nPass;
    }
    public static int report() {
        System.out.println("Test Report:");
        System.out.println("\tPassed: " + nPass);
        System.out.println("\tFailed: " + nFail);
        return nFail;
    }
}
```

Figure 3 – A Java Version

When the no-arg version of `Test.fail()` executes, it immediately calls `Test.fail(3)`, which throws and catches a `RuntimeException`. This artificial "error" enables obtaining a stack trace from the runtime system. The code in the catch clause above extracts a sanitized string three levels back in the call stack, since execution proceeded as

```
<test driver> => Test.test() => fail() => fail(3),
```

a net distance of three stack frames.

The `fail(string)` and `succeed()` functions are available to verify proper exception handling, once students get to that point in the curriculum. As an example, suppose a function `f()` takes an integer parameter that represents a zero-based index. The following code tests whether an exception is thrown on a bad index:

```
try {
    MyClass.f(-1);
    Test.fail("Invalid index not caught");
}
catch (Exception x) {
    Test.succeed();
}
```

This approach can also be used to test functions that don't return a value by wrapping calls to such functions in another function that inspects the side effects of the function being tested and then returns a Boolean value to indicate success or failure.

This tool has used in test drivers for assignments given to C++ and Java students. The instructor provides tests for assignments at first, but students eventually write their own unit tests. The experience opens their eyes to the importance of thorough, automated testing.

SUMMARY

The examples in this article are by no means representative of the test strategies employed by quality assurance personnel using formal test methods; such discipline is not within the reach of beginning students. Students can and should, however, give due diligence to correctness by carefully considering program requirements. By designing simple, automated test drivers hand-in-hand with solving the given problem, they acquire skill in a disciplined approach to developing software solutions that work. The author has used this test mechanism for six years and has found that students readily grasp its use and are usually amazed at how many bugs they find in their code. Its simplicity facilitates early adoption and encourages good design and iterative development habits early in the CS curriculum.

REFERENCES

1. National Institute of Standards and Technology, The Economic Impacts of Inadequate Infrastructure for Software Testing, Planning Report 02-3, May 2002. See http://www.nist.gov/director/prog-ofc/report02-3.pdf.

2. Viega, J. and McManus, J., The Importance of Software Testing, Cutter Consortium, 11 anuary 2000. See http://www.cutter.com/research/2000/crb000111.html.
3. Dijkstra, E., The Humble Programmer, Turing Award Lecture, http://www.acm.org/awards/article/a1972-dijkstra.pdf, retrieved June 28, 2007.
4. Beck, K., *Test Driven Development By Example*, Addison-Wesley, Reading, MA, 2003.