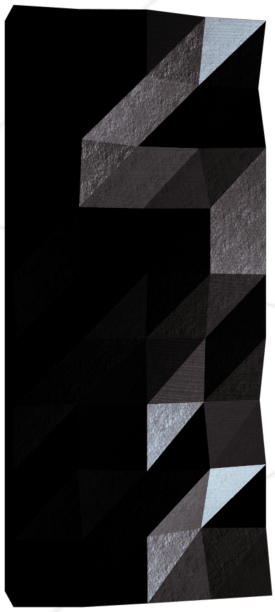


Create your own unit tests

Hack High School

Summary: Just because you've counted all the trees doesn't mean you've seen the forest.

Contents

Chapter I

Foreword

Consider Mr. T. He's been pitying fools since the early 80's, and the world is still awash in foolishness.

It's too bad, because the message is an important one. The general adoption of unit testing is one of the most fundamental advances in software development in the last 5 to 7 years.

How do you solve a software problem? How do they teach you to handle it in school? What's the first thing you do? You think about how to solve it. You ask, "What code will I write to generate a solution?" But that's backward. The first thing you should be doing – In fact, this is what they say in school, too, though in my experience it's paid more lip-service than actual service – The first thing you ask is not "What code will I write?" The first thing you ask is "How will I know that I've solved the problem?" We're taught to assume we already know how to tell whether our solution works. It's a non-question. Like indecency, we'll know it when we see it. We believe we don't actually need to think, before we write our code, about what it needs to do. This belief is so deeply ingrained, it's difficult for most of us to change.

King presents a list of 12 specific ways adopting a test-first mentality has helped him write better code:

- Unit tests prove that your code actually works
- You get a low-level regression-test suite
- You can improve the design without breaking it
- It's more fun to code with them than without
- They demonstrate concrete progress
- Unit tests are a form of sample code
- It forces you to plan before you code
- It reduces the cost of bugs
- It's even better than code inspections
- It virtually eliminates coder's block
- Unit tests make better designs
- It's faster than writing code without tests
- However, I think the test-first dogmatists tend to be a little too religious for their own good.

Asking developers to fundamentally change the way they approach writing software overnight is asking a lot. Particularly if those developers have yet to write their first unit test. I don't think any software development shop is ready for test-first development until they've adopted unit testing as a standard methodology on every software project they undertake. Excessive religious fervor could sour them on the entire concept of unit testing.

And that's a shame, because any tests are better than zero tests. And isn't unit testing just a barely more formal way of doing the ad-hoc testing we've been doing all along? I think Fowler said it best:

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. I encourage developers to see the value of unit testing; I urge them to get into the habit of writing structured tests alongside their code. That small change in mindset could eventually lead to bigger shifts like test-first development– but you have to crawl before you can sprint.

Chapter II

Introduction

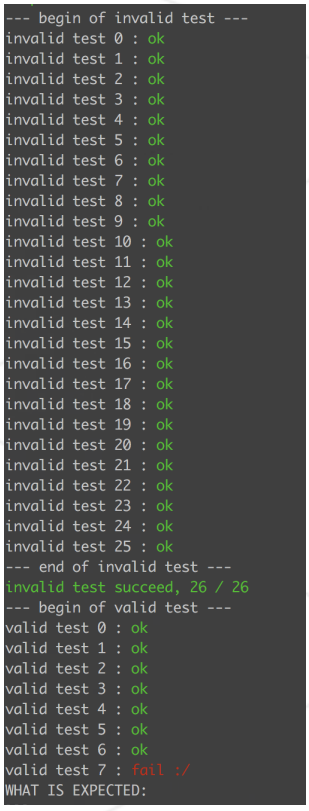
The purpose of this pdf is to show the mentors how to write unit tests that will allow students to test and grade their projects. For simplicity, we recommend that you use Bash to write the tests.

A test, for example, can be a basic comparison between the expected output of a working program and the student's program output. However, some of you probably have a specific format for your exercises, such as having a program the exits with a different status or programs that write to files. That's fine as long as you create the appropriate test suite that they can use to correct each other.

Chapter III

General instructions

- Make sure you create instructions on how to run your script
- Use colors if you can. Obviously green for pass and red for fail
- Depending on the type of project you are creating, you might or might not need to do every part of this
- Make the output easy to read like the picture below



Chapter IV

Mandatory part

IV.1 Output vs. Expected Output

In your project folder, you should have a subject directory where you will compile your .tex into a .pdf and another directory for your unit tests.

Inside your unit directory, create a bash script.

- Include your shebang on the top
- Echo a usage format
- Prepare at least 7 test cases for your program
- Optional: If the outputs were different, print out the expected output and student output

Here's an example of an extremely hard C program:

IV.1.1 C code

```c
#include    <stdio.h>

int main( void ) {

  puts( "hello world" );
  return (0);
}
```

Don't worry about understanding this code!

Here's an example script for that extremely hard program:

IV.1.2 Example

```
#!/bin/bash

RED=1
GREEN=2

log() {
  tput setaf $1
  printf "$2\n"
}

PROGRAM=$1

if [ -x "$PROGRAM" ]; then
  log $GREEN "FILE OK"
else
  log $RED "usage: bash $0 [PROGRAM]"
  exit 1
fi

testcase() {
  diff <(./$PROGRAM) <(echo $1) > /dev/null
  if [ $? -eq 0 ]; then
          log $GREEN "OK"
  else
          log $RED "KO"
  fi
}

testcase "hello world"      # OK
testcase "goodbye world"    # KO
```

This example shell script is simple. Consider making tests based on exit status of programs, tests that reads from standard input, assertions, timeouts, etc.

For information, go to these sites: http://eradman.com/posts/ut-shell-scripts.html || http://tldp.org/LDP/abs/html/

Chapter V

Turn-in and peer-evaluation

Turn this in to Kai or Ru so we can upload it onto the project evaluation page.