Unit Testing Notes

James Wu

Background: Testing in the Real World

Unit testing is one of the most important and essential skills of a software engineer. Good unit tests give a programmer the ability to pinpoint bugs and design flaws in their code, and can even act as a guide toward designing and implementing a complex project. In real-world software engineering situations, unit testing is not only helpful, but required for proper maintainance of large codebases. Testing, when done correctly, leads to well-written, bug-free programs.

Testing in 15-122

In 122, we'll be giving you plenty of opportunities to test your code. There will be many instances where the unit tests you write will directly be graded. Moreover, unit tests are highly encouraged in every single programming assignment, even if they are not required.

Moreover, it's important to note the passive difference good testing practices make. Writing tests with your code may seem like busywork, but debugging a failed test is almost always exponentially easier than debugging a failed autolab result, or an uncaught segmentation fault. Overall, writing tests is *always* a good idea.

What makes a test good?

There are a few very important traits that a good unit test should have:

Fidelity

A high *fidelity* test should fail *if and only if* there is a bug in your code. That is, the test should simulate a similar behavior to how your code is used in practice. This is important so that in writing these tests you become keenly aware of what behaviors your program should exhibit, and will not be led astray by false negatives.

Precision

A *precise* test should be able to pinpoint the exact method or line number that a test fails on. An example of a low-precision test is one that simply runs your entire program on a single input. If such a test fails, you will only know that something in your program is wrong, not where the error is located. Tests with high precision allow the computer scientist to pinpoint and debug problems quickly.

Simplicity

A simple test is one in which it's easy to understand what the test is actually testing. Complicated loops and control structures should be kept to a minimum, since they make the tests hard to understand. If your test is complicated enough to require testing, it's probably too complex.

How to write and organize tests

Here's a simple way to keep your unit tests organized and efficient. Say you have a file you want to test called `foo.c0`. First, make a file named `foo-test.c0`, which will contain all unit tests for functions in `foo.c0`. In 15-122, our assignments usually include empty files.

Next, organize your tests by writing at least one test function for each function in your original file. Let's say that your `foo.c0` has three functions. Then you can organize your `foo-test.c0` file like this:

```c
void test_f1() {
  //tests go here
  assert(f1(5) == 2);
}
void test_f2() {} //do the same for the other two.. omitted here
void test_f3() {}

int main() {
  test_f1(); //Run tests in order
  test_f2();
  test_f3();
  printf("All tests pass!\n");
  return 0;
}
```

This way, if an assertion fails, you know exactly which function has the problem. If a function is complicated, you may want to split it into more parts. Don't be afraid to name your tests long names as long as they describe exactly what you're testing: a test named `test_function()` is pretty undescriptive when it fails, but a test named `test_readArray_when_array_empty()` is much more helpful when it fails.

A lighthearted example

Unit testing is incredibly important to maintaining working code! Take this program, for example:

```c
//Read an array and return a copy of it to the user.
int[] readArray(int[] A, int length)
//@requires length > 0;
{
  int[] new = alloc_array(int, length);
  for (int i = 0; i < length; i++) {
    new[i] = A[i];
  }
  return new;
}
```

Spot the bug in the code? In this simple example, it may be easy to see that nothing guarantees the length of the array is the same as the length passed in. In fact, had the code been this simple, then OpenSSL would probably have caught this before it became known as the **Heartbleed bug**.

Obviously, the heartbleed bug was a lot more complicated in terms of actual code, but the same error occurred: nowhere was it guaranteed that the length of a object in memory, which is specified by the user, was the same as the actual length of the object.

In C0, this would simply go over the edge of the array and crash. Unfortunately, in C, reading past an array simply just reads into the next portion of memory, allowing for dangerous exposure of information in memory to a malicious hacker. A two line unit test, or even a contract, would have saved the code from ever seeing the light of day.

Some sample tests

Let's consider some sample tests for the code up there. A good set of tests for a function should go through all code paths in the function, and handle all reasonable inputs and edge cases for the

program (an edge case is often an empty array, a boundary integer, etc).

Here are a few tests that all expose potential problems with `readArray`:

```
//reveals precondition too limiting
void test_readArray_zero_array() {
  int[] empty = alloc_array(int, 0);
  //@assert \length(readArray(empty, 0)) == 0;
}

//reveals array out of bounds error
void test_readArray_length_too_large() {
  int[] testing = alloc_array(int, 5);
  testing[0] == 5;
  assert(readArray(testing, 6)[0] == 5);
}

//reveals information loss
void test_readArray_length_too_small() {
  int[] testing = alloc_array(int, 4);
  testing[3] = 7;
  //@assert \length(readArray(testing, 2)) == 4;
  assert(readArray(testing, 2)[3] == 7);
}
```

In particular, note that we use contract assertions only when we need to call the *\length* function, which can only be used in a C0 contract. Also, make sure to check all reasonable edge cases: it's logical that `readArray` should be able to handle an empty array, and we should expect it to return an empty array. The empty array test above reveals a bug in the contracts of the above function.

Additional Resources

Here are some more readings on testing if you want more information (click on the hyperlink to read them):

- Unit Testing as Hypothesis Testing by Jonathan Clark
- Coding Horror blog post on unit testing

Ultimately, it's up to you to write good, high fidelity, precise, and simple unit tests. You never know when someday, the internet's security may depend upon it.