Software Testing IOOP Lecture 1

Justin Pearson

September 9, 2011

Four Questions

- Does my software work?
- Does my software meet its specification?
- I've changed something does it still work?
- How can I become a better programmer?

Testing

Software Failures

NASA's Mars lander, September 1999, crashed due to a units integration fault — cost over $50 million.

> *The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, Small Forces, used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). A file called Angular Momentum Desaturation (AMD) contained the output data from the SM_FORCES software. The data in the AMD file was required to be in metric units per existing software interface documentation, and the trajectory modelers assumed the data was provided in metric units per the requirements.*[1]

[1] `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`

Ariane 5 explosion

Flight 501, which took place on Tuesday, June 4, 1996, was the first, and unsuccessful, test flight of the European Space Agency's Ariane 5 expendable launch system. Due to an error in the software design (inadequate protection from integer overflow), the rocket veered off its flight path 37 seconds after launch and was destroyed by its automated self-destruct system when high aerodynamic forces caused the core of the vehicle to disintegrate. It is one of the most infamous computer bugs in history.

Exception Handling

```
try {
  .......
} catch (ArithmeticOverflow()) {
     ... Self Destruct ....
   }
```

In fact it was an integration problem. The software module was for Ariane 4 and people forgot that Ariane 5 had higher initial acceleration.

Therac-25

- Radiation therapy machine. At least 6 patients where given 100 times the intended dose of radiation.
- Causes are complex [2] but one cause identified:
  - **Inadequate Software Engineering Practices** ... including:

    The software should be subject to extensive testing and formal analysis at the module and software level; system testing alone is not adequate. Regression testing should be performed on all software changes.

[2] `http://sunnyday.mit.edu/papers/therac.pdf`

Intel's `fdiv` bug

Some pentiums returned

$$\frac{4195835}{3145727} = 1.333739068902037589$$

instead of

$$\frac{4195835}{3145727} = 1.333820449136241002$$

Intel's `fdiv` bug

With a goal to boost the execution of floating-point scalar code by 3 times and vector code by 5 times, compared to the 486DX chip, Intel decided to use the SRT algorithm that can generate two quotient bits per clock cycle, while the traditional 486 shift-and-subtract algorithm was generating only one quotient bit per cycle. This SRT algorithm uses a lookup table to calculate the intermidiate quotients necessary for floating-point division. Intel's lookup table consists of 1066 table entries, of which, due to a programming error, five were not downloaded into the programmable logic array (PLA). *When any of these five cells is accessed by the floating point unit (FPU), it (the FPU) fetches zero instead of +2, which was supposed to be contained in the "missing" cells. This throws off the calculation and results in a less precise number than the correct answer(Byte Magazine, March 1995).*

Intel's `fdiv` bug

- Simple programming error, not getting your loop termination condition correct.
- Later we'll see that this might of been avoided with testing.

Software Failures

- These are just some of more spectacular examples. There is lots and lot of bad software out there. Anything we can do to improve the quality of software is a good thing.
- Formal methods are hard to implement, but software testing with some discipline can become part of any programmers toolbox.

Testing

There are lots of different testing activities. We can't cover them all but they include:

- Unit Testing: Testing your functions/methods as you write your code.
- Regression testing: maintaining a possibly large set of test cases that have to passed when ever you make a new release.
- Integration testing: testing if your software modules fit together.

As we learn about software engineering you'll see how different testing can be in different phases of the software engineering process.

Can we test?

"Program testing can be used to show the presence of bugs, but never to show their absence!" Edsger Dijkstra.

This is true, but it is no reason to give up on testing. All software has bugs. Anything you do to reduce the number of bugs is a good thing.

Test Driven Development

Later on we will look at test driven development which is a programming discipline where you write the tests before you write the code.

What is a Test?

A test is simply some inputs and some expected outputs.

Types of Testing

1. Test Design
2. Test Automation
3. Test Execution
4. Test Evaluation

Very important test execution should be as automated as possible. It should be part of your `Makefile`. Some systems even automatically run tests when you check in code.

Test Design

- Writing good tests is hard.
- It requires knowledge of you problem.
- Knowledge of common errors.
- Often test designer is a separate position in a company.

Test Design

- Adversarial view of test design:

  How do I break software?

- Constructive view of test design:

  How do I design software tests that improve the software process?

Test Automation

- Designing tests is hard.
- If you don't make running the tests an automated process then people will never run them.
- There are many automated systems, but you can roll your own via scripting languages.
- The xUnit framework has support in most languages for the automated running of tests.
- It should be as simple as `make tests`.

Test Automation

- There are tools for automatically testing web systems.
- There are tools for testing GUIs.
  - If you design you software correctly you should decouple as much of the GUI behaviour from the rest of the program as you can. This will not only make your program easier to port to other GUIs it will make it easier to test.
- Don't forget to include test automation in your make files.
- Consider integrating automated testing into your version management system.

Test Execution

You need to think of test execution as separate activity. You have to remember to run the tests. In a large organization this might require some planning.

- Easy if testing is automated.
- Hard for some domains e.g. GUI.
- Can be hard in distributed or real time environments.

Test Evaluation

- My software does not pass some of the tests. Is this good or bad?
- My software passes all my tests. Can I go home now? Or do I have to design more tests?

Important Terminology

- **Validation:** The proess of evaluation software at the end of software development to ensure compliance with intended usage.
- **Verification:** The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

Important Terminology

- **Software Fault:** A static defect in the software
- **Software Error:** An incorrect internal state that is the manifestation of some fault
- **Software Failure:** External, incorrect behavior with respect to the requirements or other description of the expected behavior

Understanding the difference will help you fix faults. Write your code so it is testable.

Pop Quiz

How many times does this loop execute?

```
for(i=10; i<5; i++) {
    do_stuff(i);
}
```

Fault/Error/Failure Example

```
int count_spaces(char* str) {
 int length, i,count;
 count = 0;
 length = strlen(str);
 for(i=1; i<length; i++)  {
   if(str[i] == ' ') { count++; }
 }
 return(count);
}
```

- Software Fault: `i=1` should be `i=0`.
- Software Error: some point in the program where you incorrectly count the number of spaces.
- Failure inputs and outputs that make the fault happen. For example `count_spaces("H H H");` would not cause the failure while `count_spaces(" H");` does.

Suggested Reading

- One of the testing gods is James Bach see his website[3]
- The book *Introduction to Software Testing*[4] by Ammann and Offutt.
- The book *Test-Driven Development by Example* by Kent Beck. Who is one of the fathers of unit testing, agile programming and extreme programming.

[3] `http://www.satisfice.com/`

[4] `http://cs.gmu.edu/~offutt/softwaretest/`

Unit Testing

- xUnit testing is a framework where individual functions and methods are tested.
- It is not particularly well suited to integration testing or regression testing.
- The best way to write testable code is to write the tests as you develop the code.
- Writing the test cases after the fact takes more time and effort than writing the test during the code. It is like good documentation you'll always find something else to do if you leave until after you've written the code.

The heart of Unit Testing

The unit test framework is quite powerful. But the heart are two functions:

- **assertTrue**
- **assertFalse**

assertTrue

Suppose we want to test our string length function `int istrlen(char*)` The the following things should be true:

- The length of `"Hello"` is 5.
- The length of `""` is 0.
- The length of `"My kingdom for a horse."` is 23.

`assertTrue`

Then we would assert that the following things are true:

- `assertTrue` (The length of `"Hello"` is 5.)
- `assertTrue`(The length of `""` is 0.)
- `assertTrue` (The length of `"My kingdom for a horse."` is 23.)

`assertTrue`

- Key idea in xUnit:
- `assertTrue`(executable code)
  - Runs the executable code which should evaluate to true.
- `assertFalse`(executable code)
  - Runs the executable code which should evaluate to false.

- Different xUnit frameworks run the tests in different ways.
- CUnit has a notion of test suites and registries.
- See the code in the repository for examples.
- Key to success in understanding complex APIs. Take example code and modify it to do what you want.

CUnit

```
void testSIMPLE(void)
{
  char *str1 = istring_mk("Hello");
  char *str2 = istring_mk("");
  char *str3 = istring_mk("My kingdom for a horse."
   CU_ASSERT(istrlen(str1) == 5);
   CU_ASSERT(istrlen(str2) == 0);
   CU_ASSERT(istrlen(str3) == 23);
  istring_rm(str1);
  istring_rm(str2);
  istring_rm(str3);
}
```

Important Unit Testing Concepts

- Setup. You might need to initialize some data structures.
- The tests. Well you need to do tests.
- Teardown. Always clean up after you.

Important idea.

- Each test should be able to be run independently of the other tests. You don't know what order the tests will be run. Or even if all tests will be run. The programmer might just rerun the test that caused problems.

CUnit — Setup

```
char *str1 = istring_mk("Hello");
char *str2 = istring_mk("");
char *str3 = istring_mk("My kingdom for a horse."
```

CUnit — The tests

```
CU_ASSERT(istrlen(str1) == 5);
CU_ASSERT(istrlen(str2) == 0);
CU_ASSERT(istrlen(str3) == 23);
```

CUnit — Teardown

```
istring_rm(str1);
istring_rm(str2);
istring_rm(str3);
```

Teardown is very important in languages like C with no automatic garbage collection.

More Testing Concepts

- Sometimes you don't have all the functionality implemented. Write dummy functions *stubs* that simply return null values rather doing any real work. Means that you can get your code going.
- *Mock* or *Fake* objects (people make a distinction but don't worry) implement enough of an object to get the test going.

In fact in test driven development you write the test implement the mocks first and as you introduce more tests you add code to make the tests pass.

Test functions

- It is a matter of judgment and taste how many tests you put in each function.
- You don't want individual tests to take too much time to run. This will discourage the programmer from running individual test often.
- Whenever you compile your code you should run the tests.
- Often IDEs implement red and green bars for tests. Green means the test has passed and red means the test has failed.
- Green is good.

How to use Unit Tests

- When you are writing functions use test cases to see if the behaviour is as expected.
- Use tests as another form of documentation. Helps other programmers understand your API.
- When you find a bug write a test case and then correct the bug. Leave the test case there just in case fixing another bug reintroduces a bug.

Some things to test for

- Extreme values. Empty strings, large values.
- Loops executing zero, once, many times, and test the loop termination condition.

```
for(int i=0; i<M; i++) {
    do_something(i);
}
```

- Find a test case that makes `M` be 0, 1 and some larger number.
- Often `M` will not be an input parameter. You might have to work out how the input parameters affect `M`

```
void whatever(char* str) {
    M = strlen(str);
    ....
}
```

- So you have to have a string of length 0,1 and some bigger number.

Some things to test for

- Code coverage. This is a complex area, but you should not really have untested code.

```
if (X==Y) { do_something(); }
else {do_something_else();}
```

- Find a test case where `X` equals `Y` and a test case where they are different.

Have a reason

- You can't test code just by letting a monkey type random things on the keyboard (although it sometimes helps).
- Try to have a reason for every test.
- Document reasons.
- When you test suite gets too large you have to work out which tests to delete.