Software Testing Lecture 1

Justin Pearson

2021

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?

The Answer

Testing

Motivation for Software Testing

Before we talk about what software testing is, I would like to give you some examples of some spectacular software failures.

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 implemented for Ariane 4 and the programers forgot that the Ariane 5 model had a higher initial acceleration and a different mass.

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 the loop termination condition correct.
- Later we'll see that this might have been avoided with testing.

Software Failures

- These are just some of the most spectacular examples. There is a 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 programmer's toolbox.

The reality of Software Development

- All code has problems.
- Anything that we can do to improve the quality of our code is important.

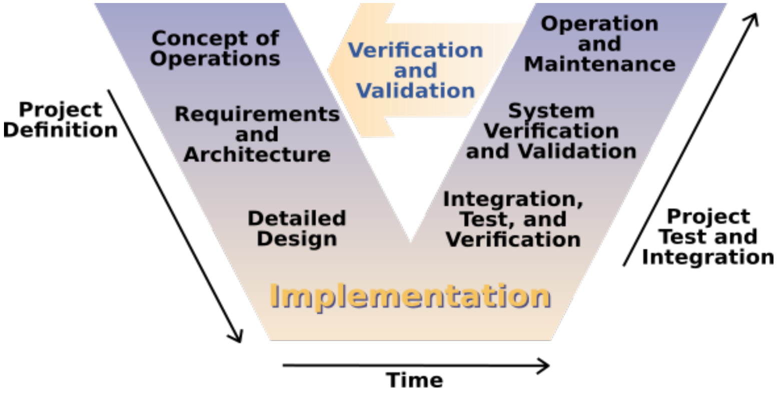
Software Engineering and Testing

Software Engineering tells us how to develop software. There are many models and processes including

- The Waterfall Model
- Agile Development
- Scrum
- Extreme Programming

We will not cover software engineering methodology, but it is important to think about how software testing fits in with your development process.

The V model — Software Engineering


Concept of Operations
Requirements and Architecture
Detailed Design
Implementation
Integration, Test, and Verification
System Verification and Validation
Operation and Maintenance
Verification and Validation
Project Definition
Project Test and Integration
Time

Even if you don’t develop software this way, it is a useful way of thinking about software development.

Testing

There are lots of different testing activities with names inspired by the V model. We cannot 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.

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.

What is a Test?

This is quite a complex question and depends on what you are developing.

- How do I test a GUI?
- How do I test a real-time system?
- How do I load-test a web-server?
- How do I test a database system?

How do I write code that can be tested?

What is a Test?

In this course we will look testing functions or methods.

- A test is simply some inputs and some expected outputs.

This simple description hides a lot of complexity, though.

- How do I know what my code is supposed to do, so that I can work out what the expected outputs are?
- What exactly are the inputs and outputs of my system?

Aspects of Testing

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

It is very important that 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, and
- Knowledge of common errors.
- Often, a test designer is a separate position in a company.
- Test design helps the tester understand the system.

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?
- Often you design tests to uncover common programming errors for example off by one errors.

Test Automation

- Designing good tests is hard.
- If you don't make the execution of 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 your 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, but also it will make it easier to test.
- Don't forget to include test automation in your compilation process.
- 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.
- Very 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 and Concepts

- **Validation:** The process 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 and Concepts

- **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 behaviour.

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

Pop Quiz

How many times does this loop execute? errors

```
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.

Fault/Error/Failure

- Fault/Error/Failure is an important tool for thinking about how to test something (not just software).
- I am trying to correct faults that cause errors that cause failures.
- How do I design test cases that give failures that are caused by errors that are due to faults in the code.

The RIP Model

Reachability, Infection and Propagation.

- Reachability: The test causes the faulty statement to be reached.
- Infection: The test case causes the faulty statement to result in an incorrect state.
- Propagation: The incorrect state propagates to incorrect output.

State

What is the state of a program?

- For a deterministic single threaded program it is the values of all variables, the program counter (where you are in the program), and possibly the contents of files on external storage.
- For multi-threaded, concurrent programs it is much more complicated, and we will only deal with deterministic single threaded programs.
- It can get even more complicated: for example, what about the state of the cache to take into account timing effects.

Analogy with disease

- The symptom is only an indication of what is wrong with you.
- Test cases are a diagnostic tool. We can only see the symptoms and not inside the software.

The RIP Model — Example

```
1    int count_spaces(char* str) {
2      int length, i, count;
3      count = 0;
4      length = strlen(str);
5      for(i=1; i<length; i++) {
6        if(str[i] == '␣') { count++; }
7      }
8      return(count);
9    }
```

We need to *reach* the fault one line 5, that is easy in this case, because this will always happen regardless of the input string.

The RIP Model — 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);
}
```

How do we *infect* the program? How do we get in to an incorrect state?

In this case, the final value of `count` should not actually be the length of the string. So the input string should be a string that starts with a space character.

The RIP Model — 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);
}
```

Propagation again is easy, because it always returns the value of count.

The RIP Model

In the example reachability and propagation were easy. This is not always the case.
All you can do if give inputs to a function and look at outputs.
Testing is different from debugging, you do not have the ability to observe and change the program state.
This means that you have to think about what input values will get you to the fault in the program.
Another time when you need to think about reachability is when you correct some failure (fix a bug), you want a test case that reproduces the failure that you fixed. So studying the changes you made and trying to come up with test cases involves thinking about the RIP model.

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. This is a bit theoretical.
- The book *Test-Driven Development by Example* by Kent Beck. Who is one of the fathers of unit testing, agile programming and extreme programming.
- A classic "The ART of software testing" Glenford Meyers. Online at the university library.

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

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