EECS4313 week 1

Jerry Wu

2024-01-08

Contents

Lecture 1

Introduction

Testing knowing and unknowing

For knowing, we already have an idea of the features we need to implement.

1. Requirements

2. Deliverable

The opposite being testing stuff that we don't know we want to implement or the requirements don't cover(extrapolating new features). Oftentimes, the base requirements aren't enough to solve the problem that the software wants to solve.

Our goal is to find unknowing functional/unfunctional bugs and problems within code effectively.

Difference between testing goals

- An oracle is used for known testing, because we can get a definitive and objective answer as to whether something fulfills the requirements.

- Testing the **unknown** requires subjectivity and whether something is good or bad, which we will focus on in this course

Common causes of software failure

To put simply, a software failure is any kind of unpredictable/undesirable behavior. Some examples include:

- Crashing (the worst case scenario)
- Hanging
- Data corruption
- Incorrect functionality
- Performance degradation

Our goal is to completely prevent these issues and in the worst case scenario where they can't be prevented to recover or arrive at a termination point so as to create a functional system. Our goal is to be efficient as possible when testing as it is costly.

A few more possible causes

- Segfault (index out of bounds)
- Deadlock (in multithreaded applications with thread interaction over shared memory)
- Memory leaks
- Regression bugs (fixing one bug creating multiple new bugs)
- Incorrect implementation
  for example:

```
public int add(int x, int y)
{
    return x-y
}
```

Historical examples of bugs

- Ariane 5 rocket crash (64 bit float for horizontal velocity converted to 16 bit signed int. Number was > 32767, which is the largest 16 bit int, conversion failed)
- This ended up costing $8B

Testing challenges

In times past, we were able to conduct tests by creating a test suite with test cases. Nowadays when software is constantly evolving, we have to take into account these challenges in btoh a technical and moral context:

- Maintenance - evolving our testing as the application evolves
- Selection
- Minimization - how can we be efficient? How can we minimize the amount of harm done by bugs?
- Prioritization - which component of the application needs more rigorous testing?
- Augmentation - what happens when we make changes to our code?
- Evaluation
- Fault characterization - where is the problem located?
- Testing ML/DL applications - especially applicable in the year of 2024. A bug is something that **can cause significant harm**. So an image generator generating something innacurate wouldn't be considered a bug, but a self driving car misclassifying a pedestrian as something else leading to a crash would be a significant bug.

We can't be 100% accurate when conducting testing, so we should be as specific as possible when defining our testing criteria.

Lecture 2

Testing terminology

What is testing?

Put simply, testing is a rigorous **technical investigation** done to expose **quality related information** about the **product** under test. It is not done randomly or in an ad-hoc manner.

> *"Cyberpunk was terrible when it released because it wasn't tested" - H.V. Pham 2024*

Quality related problems

- Find important bugs - like with anything, different objectives require different testing strategies and will yield different results each time
- QOL issues
- Usability
- Interoperability with other products
- Help managers make ship/no ship decisions
- BLock premature product releases
- Minimize technical support costs
- Make sure technical specifications are followed and conforms to regulations
- Minimize safety related lawsuit risk, find safe scenarios for use of product

We don't need to pass all test cases, although we want to pass as many as possible before releasing. It's better to create something with less features and runs better than to create something with a robust library of features that is full of bugs.

Test case

- **Test case values/Input/Data**: The values that directly satisfy one test requirement
- **Expected output/results**: The result that is produced when executing the test if the program satisfies it's intended behavior

Combining both of these gives us a **test oracle**

Testing steps and activities

- **Test design**
  - Design test values to satisfy specific criteria or other engineering goal
  - Requires knowledge of discrete math, programming, and testing
- **Test automation**
  - Embed test values into executable scripts
  - Requires knowledge of scripting
- **Test execution**
  - Run tests on the software, record the results of said test
  - Requires electron brain
- **Test evaluation**
  - Evaluate results of testing, report to devs
  - Requires domain knowledge

Types of testing

- **Static testing**
  - Testing without executing the program
  - This method is very effective at finding **potential** faults, which are problems that could potentially cause other problems when new code is introduced
- **Dynamic testing**: Testing by executing the program with real inputs. Shows concrete issues with the program, but aren't necessarily a bug.

Black box testing

Deriving tests from external descriptions of software including specifications, requirements, and design.

- Testing as **attacker/hacker**
- We don't know the source code itself, but we have a set of inputs with a set of expected outputs.
- This is effective when we write the test cases first based on the design of our software model before we write any of the executable code itself.
- Works better for general implementations

White box testing

Opoosite of black box testing. Test cases are derived directly from the functions in our source code. This allows us to minimize the number of test cases we need to write to cover the entire scope of the code/code snippet.

- Testing as **developer**
- This includes branches, individual conditions, and statements.
- Works better for specific implementations of requirements.

Moral of the story, just use both in your test suite!

Top down vs bottom up testing

- **Top down**: Test the whole system together first, and go down through specific functions in depth first or breadth first manner. Similar to black box testing.
- **Bottom up** is the exact opposite, where the details are tested first until we arrive at the whole system at the last step. Similar to white box testing.

Testing at different levels

- **Level 1**: Unit testing - Test each individual method
- **Level 2**: Module testing - Test each class, file, module, or component individually
- **Level 3**: Integration testing - Test how modules interact with each other
- **Level 4**: System testing - Test the overall functionality of the system
- **Level 5**: Acceptance testing - Is the software acceptable to be shipped to end users?

Fault, error, and failure

- **Fault** - essentially a bug, which is a static defect in software caused by incorrect lines of code. No code execution necessary. (example: mutation in DNA)
- **Error** - An incorrect and **unobserved internal state**. Only occurs when code is executed, an internal issue. (example: having 2 hearts instead of 1)
- **Failure** - An **observed external incorrect behavior** with respect to the expected behavior. Only occurs when program is executed. (example: getting a disease like cancer)
- Faults can lead to error, or both error and failure, or none at all.

Each of these are considered as a type of fault, which is undesirable behavior that occurs within the code. External issues always occur due to internal error, but internal errors don't necessarily have external consequences.

States

A state contains:

- Values of parameters and variables
- Program counter (PC), which is also a variable signifying which line of code is being executed
- If there is something wrong with the program counter, it is considered as an **error**. $\exists$ a set of states which is considered "correct", and everything outside of that set is either a fault, error, or failure.
- It is important to observe as many failures as possible in order to expose faults in the code as possible.

To construct state diagram

1. Identify the fault, fix the fault
2. Identify a test case that does not **execute** the fault
3. Identify a test case that **does** execute the fault, but doesn't result in an error
4. Identify a test case that does result in an error, but not failure (do not forget about the program counter)
5. For the given test case, determine the first error state. Be sure to describe the complete state.

RIP model

These are conditions that must be present for a failure to be observed

- **Reachability** - the location(s) in the program that contain the fault must be reached
- **Infection** - After executing the locaiton, the state of the program must be incorrect
- **Propagation** - The infected state must propagate to cause some output of the program to be incorrect

How do we avoid bugs?

Ways to deal with faults

"Let's just make the bug a feature" - H.V. Pham 2024

It's impossible to avoid bugs, but we can mitigate them as well as possible by creating better design and architecture for our software models. More solutions include:

- **Fault avoidance**
  - Better design
  - Better PL
- **Fault detection**
  - Testing
  - Debugging
- **Fault tolerance**
  - Avoid redundancy
  - Isolation of faults (so as to avoid it from propagating)
- **Patching** (fixing the fault)

Testing vs debugging

- **Testing** - evaluating software by observing its execution. Often, only specific inputs will trigger the fault into creating a failure
- **Debugging** - The process of finding a fault given a failure. It is difficult to know what the fault is

"Debugging is an art"-H.V. Pham 2024