Software Testing, Test Case Design, and Debugging

CMPT 145

Learning Objectives

After studying this chapter, a student should be able to:

- Provide examples of software errors resulting in observable faults.
- Define the terms error, fault, specification, and oracle.
- Compare black box testing and white box testing.
- Explain the differences between unit testing, integration testing, and system testing.
- Identify unit test case equivalence classes for a given function.
- Explain why the creation of correct program components is important in the production of high-quality software.
- Apply a variety of strategies to the testing and debugging of simple programs.
- Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers.
- Construct and debug programs using the standard libraries available with a chosen programming language.
- Explain the importance of test coverage.

Software errors

- Easy to make.
- Will reduce the value of the software.
- Can be harmful to people.
- Can cause damage to equipment/environment.
- Take up a lot of programming time.

Disaster: Therac-25

- Medical linear accelerator for cancer treatment (radiation therapy)
- Introduced in 1983; novel use of software control.
- Some patients received massive radiation overdoses, leading to disability and 3 deaths.
- A combination of factors:
  - Software errors
  - Inadequate safety engineering
  - Poor design of user input module
  - Inadequate reporting of incidents

Disasters: NASA Mars Climate Orbiter

- Orbiter's software used Metric units (grams, meters, Celcius, etc)
- Mission control software used Imperial units (lbs, feet, Fahrenheit, etc)
- No one tested whether unit-conversion was being done
- $125M Orbiter crashed into Mars.

Disasters: ESA Cluster (Ariane 5 rocket)

- Safety mechanisms coded in control software for testing
- Safety mechanisms turned off for actual launch
- $370M of high tech destroyed 39 seconds after launch.

Disasters: Scripps Research Retractions

- Data analysis software to deduce protein crystal structure
- Five high ranking publications from the results (Science, JMB, PNAS)
- Researchers discover one function flipped numeric values from positive to negative
- Five high ranking publications retracted (Science, JMB, PNAS)

Why we test software

- Software errors are inevitable.
- Testing is not an after-thought, it is a professional responsibility.
- Testing and debugging has to be part of your development plan.

Software errors will happen

- We're not perfect.
- The sooner we find them and fix them the better.
- Time spent testing will reduce time spent debugging.

Terminology

- Fault: Your program exhibits behaviour that is not intended.
- Error: The cause of the fault.
- Specification: A written document stating the correct behaviour.
- Oracle: A method that tells us if an answer is correct or not.

Testing is the active process of fault discovery.

Where do software faults come from?

- There are two ultimate sources of software errors:
  1. Not writing the code you intended to write.
     - Typographical errors
     - Minor errors in logic.
  2. Not knowing what code to write.
     - Design was bad.
     - No design.
     - No plan.
- Debugging can only help with #1.

Scientific attitude for testing

- Hypothesis: Your program is correct.
- Experiment: Designed to show Hypothesis is false.
  - Try to demonstrate that your own program has errors.
  - Write test cases that try to find faults.
- Conclusion: Confidence in Hypothesis increases with every Experiment.

Testing Review: Strategies

- Black-box testing:
  - Design tests based on the interface of a function (inputs and outputs).
  - (black-box means: don't look at the code inside the function)
- White-box testing:
  - Design tests based on the actual instructions inside the function.
  - (white-box means: look at the code inside the function)
- These are not entirely distinct from each other.

Test coverage

- Measures how much of the code has been tested.
- Measured in terms of:
  - Lines of code
  - Number of branches
  - Number of functions
  - Number of modules
- More test coverage is better.

Testing levels

- Unit testing
  - Testing done on code units: e.g., function, method.
- Integration testing
  - Testing done on functions or modules working together.
- System testing
  - Testing done on completed applications.

Degrees of Testing

Testing can be considered at various levels of diligence:

1. No testing
2. Trivial, simple examples only
3. Reasonable, expected examples
4. Difficult examples
5. Unreasonable examples, illegal inputs
6. All possible inputs

Questions about testing

Answered from student perspective

- How should we test?
  white-box, black-box
- What should we test?
  unit, integration, system
- When should we test?
  immediately, or sooner
- How much time should we spend testing?
  equal to time spent coding
  It's not wasted time.

Test case design

- You can't test exhaustively!
- Each test case should be an example drawn from an equivalence class.
- To prevent bias, draw two test cases for every equivalence class.
- Draw test cases from the boundaries of equivalence classes.

Test Case Equivalence Classes

A set of test cases is called an equivalence class if:

- White-box: They all cause same path through the code (white-box).
- Black-box: The result on any test case in the set tells you nothing more than any other test case.

An equivalence class expresses a range of possible test cases:

- Numbers: ranges, sets, etc.
- Lists: different sizes, different values, etc.

Boundary test cases

- An equivalence class expresses a range of possible test cases
- Values on the boundary of the classes are called boundary cases
- Include test cases at every boundary, and on both sides.

Test scripts

- Unit testing: An if-statement checking a function's output against a known value.
- Silent unless an error is detected.
- Easy to do when output is simple, e.g., numbers, strings, lists.
- Harder to do with more sophisticated programs with more sophisticated data.

Test Harnesses

- Test Harness: A script or function that initializes data to be used in a test.
  - May have to simulate the effects of other functions to be tested
  - Possibly uses other functions that are being tested together.
- Silent unless an error is detected.
- Test harnesses may have to be tested too!

Where do software errors come from?

- There are two ultimate sources of software error:
  1. Not writing the code you intended to write.
     - Typographical errors
     - Minor errors in logic.
  2. Not knowing what code to write.
     - Design was bad.
     - No design.
     - No plan.
- Debugging can only help with #1.

Debugging attitude

- To find out what your program is doing wrong, you need to know what your program is doing.
- Do not assume that an error has a quick or easy fix.
- Errors are almost always where you are not looking.
- Challenge your own assumptions.
- Assume everything is broken.

Scientific Debugging

- The output of your test script is data.
- It tells you something went wrong. It does not tell you what the error is.
- Do not make a change to your code too soon

  1. Make a hypothesis about the error. Write it down.
  2. Create new test cases that should fail if your hypothesis is true.
  3. Run the new test cases to verify your hypothesis.
  4. Only change the code after you have gained confidence in your hypothesis.
  5. If your fix eliminates the errors, you were (probably) right.

Debugging techniques

- For faults discovered by unit testing:
    - Use the debugging tool
    - Careful reading of the function/unit you are testing
- For faults discovered by integration testing:
    - Print statements (Wolf-fencing) to narrow down the location of the error.
    - Set break-points to use the debugging tool on small regions of your code.
- For faults discovered by system testing:
    - Do not debug the application.
    - Identify the conditions that cause the fault.
    - Create an integration test with those conditions, and debug that!

Debugging: Wolf-fencing

- An error in module A might not be cause a fault until module B gets the incorrect data.
- You don't have the time to use the debugger across a whole application.
- Add output to your program, e.g.,
  - `print` statements
  - `assert` statements (Chapter 13).
  - (advanced) `logging output`
- Using a kind of binary search for the location of an error.
- Gather information about what happened.

Debugging non-Python languages

- Many languages (Java, C#, Ruby, etc) have run-time errors like Python.
    - You will at least know where the run-time error occurred.
- Some languages (C, C++) do not.
    - A C/C++ program will halt abnormally without any warning or hint about why.
- Use wolf-fencing to find out where the program halted.
- Almost all languages have interactive debuggers, and they all have the same features. Learn the one you need for your work.

Debugging: Code Walk Throughs

- Your errors may be invisible to you.
- So explain your code to someone else.
- Professional software engineers review each others' code regularly.
- Pair programming is a good practice, when permitted by courses.
- If no one is around, use a puppet. Silly, but it works.

The difference between you and a professional

- Professional developers are learning new technologies all the time.
  - They are not done learning.
- The difference between you, and accomplished software developers is
  - Not knowledge
  - But experience
- Be patient with yourself to gain that experience while you are learning new things.

Example 1

Consider the following interface documentation:

```
def insertion_sort(U):
    """
    Purpose:
        creates new sequence containing sorted data of U
    Pre-conditions:
        :param U: sequence to sort
    Post-Conditions:
        None
    Return:
        :return: new sequence where U is sorted
    """
```

- Design some black-box test cases.
- Identify some test case equivalence classes.

Example 2

Consider the following interface documentation:

```
def f(x):
    return x + 3
```

- Why is it impossible to test this function?

Example 3

Consider the functions in the file `sorting_algs_141.py`.

- Design some black-box test cases.
- Identify some test case equivalence classes.
- Use the debugger to step through some of the functions.