Software Testing Foundations

Owolabi Legunsen

CS 5154

Fall 2021

An earlier statement from this course

Testing is usually the last line of defense against bugs

But what exactly is a “bug”?

What does “bug” refer to in this program?

```
public static int numZero (int[] x) {
int count = 0;
for (int i = 1; i < x.length; i++) {
  if (x[i] == 0) count++;
}
return count;
}
```

What is a “bug” in this program?

Should start searching at 0, not 1

i is 1, not 0, on the first iteration

```
public static int numZero (int[] x) {
int count = 0;
for (int i = 1; i < x.length; i++) {
  if (x[i] == 0) count++;
}
return count;
}
```

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

count is 0, instead of 1, at the return statement

Building shared terminology in CS 5154

- **Fault** : static defect in the code
- **Error :** incorrect internal state caused by a fault
- **Failure** : incorrect observed behavior w.r.t. the expected behavior

Why is this shared terminology important?

- Show off the knowledge you gained in this class ☺
- Precise understanding will help us be on the same page in this class
- Many concepts that we will learn build on these terminologies
- They are software testing industry standard terminologies

Example: identify the fault, error, failure

Example: identify the fault, error, failure

The faults that caused major failures

Failure	Year	Impact	Root Cause
NASA's Mars lander	1999	$125,000,000 satellite lost	Failure to convert units from Pounds to Newtons
THERAC-25 radiation machine	1985 - 1987	6 patients died	Several: see link
Ariane 5 explosion	1996	$7,500,000,000 lost	Exception-handling fault (64-bit to 16-bit conversion)
Northeast blackout	2003	50 million people lost power in US and Canada, $6,000,000,000 lost	Buffer overflow in monitoring system

In software testing, we write tests to find faults before those faults find the users

So, what is a test?

```
public static int numZero (int[] x) {
int count = 0;
for (int i = 1; i < x.length; i++) {
  if (x[i] == 0) count++;
}
return count;
}
```

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Components of a test

- **Test Case Values**: input data needed to execute the code being tested
- **Expected Results:** output that is produced if the code is correct
- **Test Oracle:** decides if the observed output match expected output

Discuss: why does “well-tested” software fail?

Why does “well-tested” software fail?

- Are the tests effective for finding errors?
- Can testing guarantee the absence of failures?
- Has the testing been done with the right goals?
- Is the software really “well tested”?

A test is effective if it…

1. **Reaches** program location(s) that contain a fault

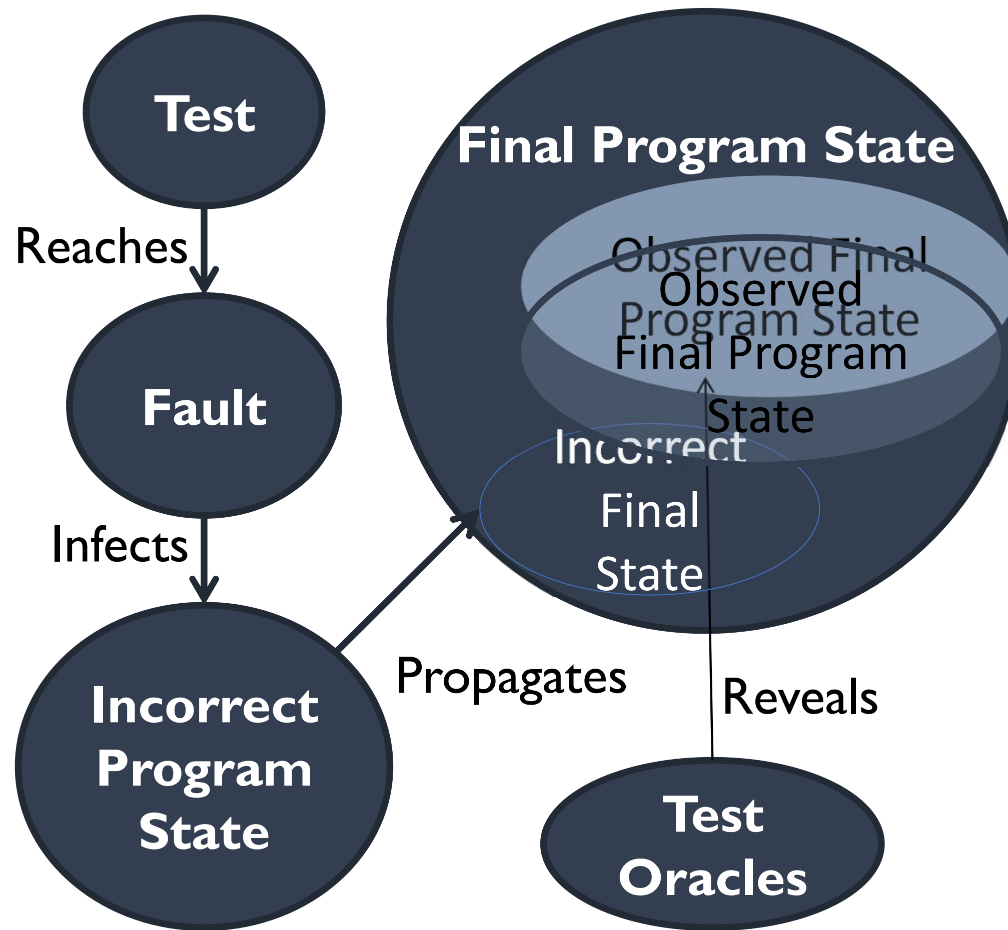
2. **Infects** the program state after executing a faulty location

3. **Propagates** the infected state into incorrect output

4. **Reveals** part of the incorrect output to the test oracle

The RIPR model of test effectiveness

- **R**eachability
- **I**nfection
- **P**ropagation
- **R**evealability



We will use the RIPR model to learn how to write effective tests

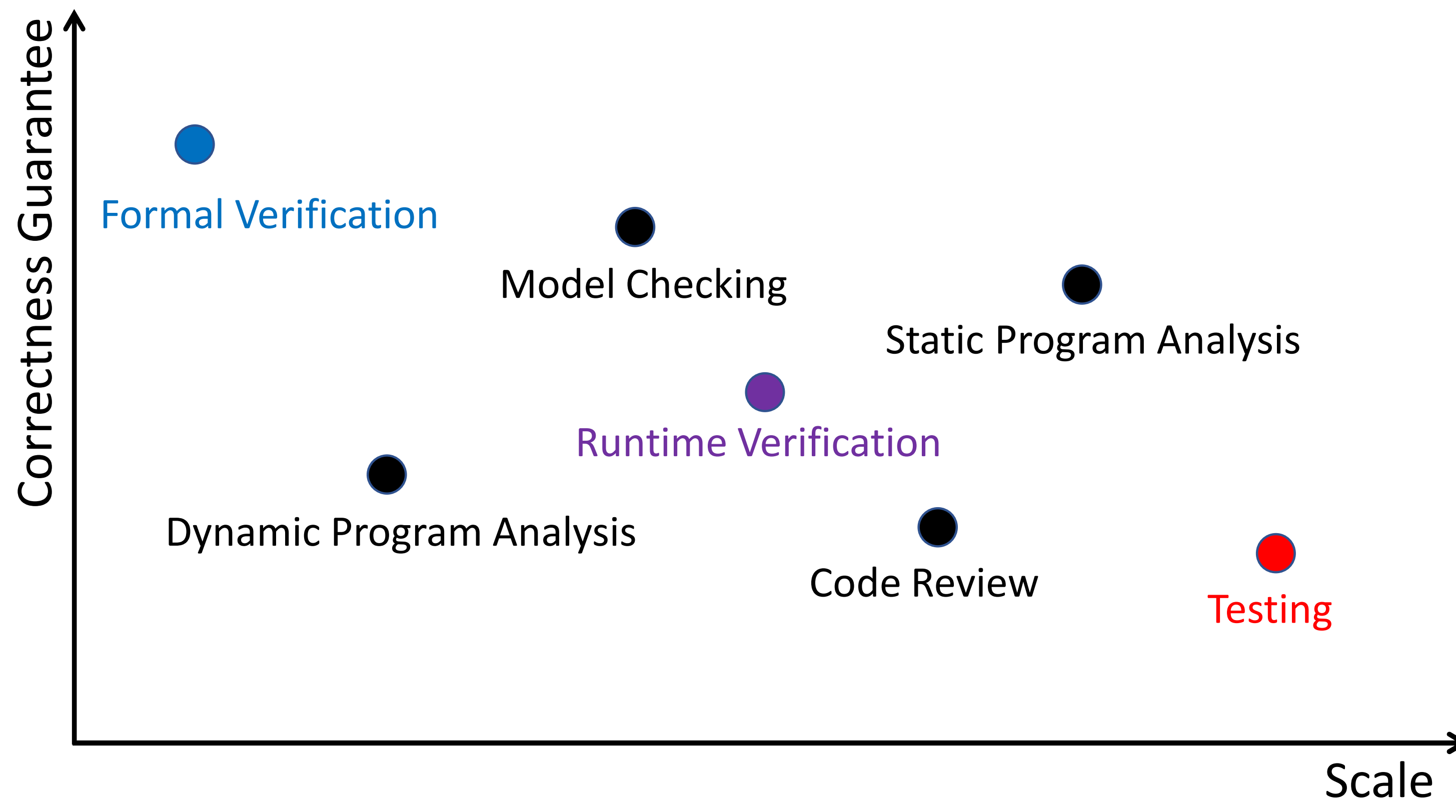
A fundamental limitation of software testing

- Claim: testing can only show the presence of failures, not their absence

- Is this claim true?

- Lesson: testing is one of many tools for improving software quality

Other software quality assurance techniques



Testing goals at different levels of maturity

- Level 0: testing == debugging
- Level 1: testing is done to show program correctness
- Level 2: testing is done to show that software does not work
- Level 3: testing is done to reduce the risk of using software
- Level 4: testing is a mental discipline that helps build high-quality software

Level 0 thinking

- Purpose: show that program runs on few arbitrary/provided inputs
- Debug the program if it does not work on said inputs
- Problem: no distinction between incorrect programmer behavior and programmer mistakes

Level 1 thinking

- Purpose: use tests to show that a program is correct
- Problems:
  - If there are no failures, is software good or tests are not effective?
  - Hard to know when to stop testing (testing cannot prove programs correct)

Level 2 thinking

- Purpose: use tests to show that a program is incorrect
- Problems:
  - Can lead to adversarial relationship among developers ☹
  - What if the tests do not fail?

Level 3 thinking

- Purpose: team-based approach to reducing risk of software failures
- Problems:
  - Testing is the only way to improve software quality
  - Focus is on software, and not on the developers that write the software

Level 4 thinking

- Purpose: testing as a mental discipline that improves software quality
- Effects:
  - Improve the ability of developers to write high-quality software
  - Invest in continued quality measurement and improvement
  - Make testers part of project leadership

Poll: what level of testing maturity are you at?

- Level 0: testing == debugging
- Level 1: testing is done to show program correctness
- Level 2: testing is done to show that software does not work
- Level 3: testing is done to reduce the risk of using software
- Level 4: testing is a mental discipline that helps build high-quality software

Some goals of CS 5154

- Moving you (and your organization) to Level 4 thinking

- Teach you to be “change agents” who advocate for Level 4 thinking

Is software “well-tested”?

- Testers use coverage criteria to measure how well-tested software is
- What are some coverage criteria that you know?

Coverage criteria: pros

- Provide a way to know when to stop testing
- Can be continuously measured during regression testing
- Maximize the "bang for the buck"
  - find the **fewest tests** that will find the **most faults**

Coverage criteria: cons

- Some criteria are “weaker” than others
- Strong criteria are harder to achieve or more expensive
- HUNDREDS of criteria have been proposed!
- Many developers are not trained in test design ☹

Discuss: how to create effective tests?

```
/**
 * Return first index of Node n in path, or
 * -1 if n is not present in the path
*/
public int indexOf (Node n, List<Node> path)
{
  for (int i=0; i < path.size(); i++){
    if (path.get(i).equals(n))
      return i;
  }
  return -1;
}
```

How would you go about producing effective test cases for this method?

"We cannot solve our problems with the same thinking that we used when we created them"

- Albert Einstein (?)
- Yogi Bera (?)

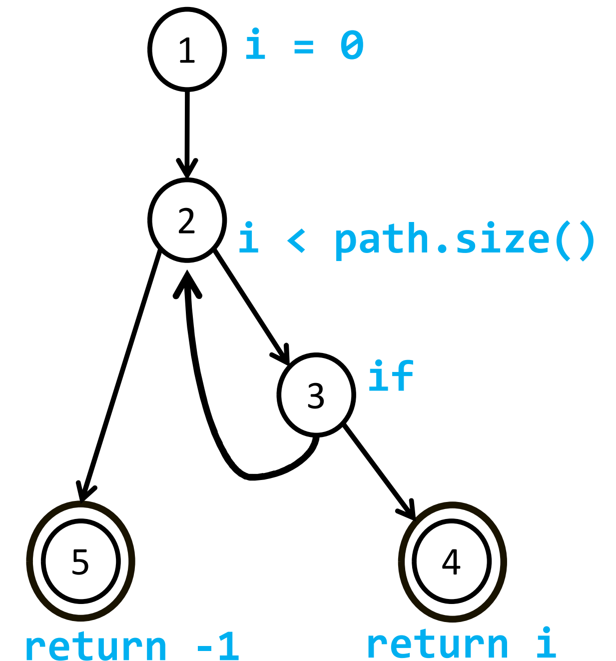
"It is difficult to create effective tests if we only look at code. We need a higher level of abstraction"

- Offutt and Ammann

Producing effective tests for indexOf (1)

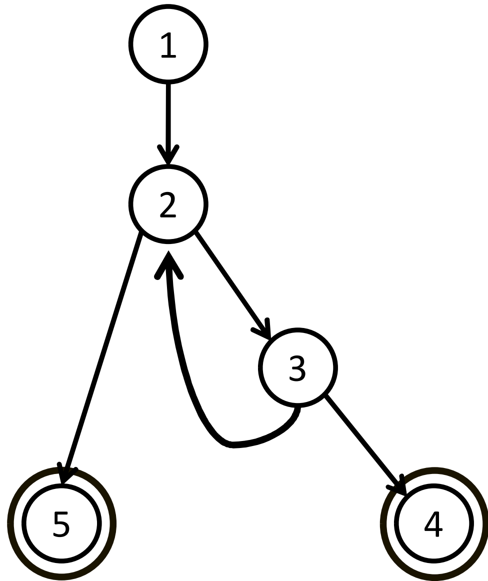
```
/**
 * Return first index of Integer n in path, or
 * -1 if n is not present in the path
*/
int indexOf (Integer n, List<Integer> path){
  for (int i=0; i < path.size(); i++){
    if (path.get(i).equals(n))
      return i;
  }
  return -1;
}
```

Control Flow Graph



Producing effective tests for indexOf (2)

Graph: abstract version



Edges
1 2
2 3
3 2
3 4
2 5
Initial Node: 1
Final Nodes: 4, 5

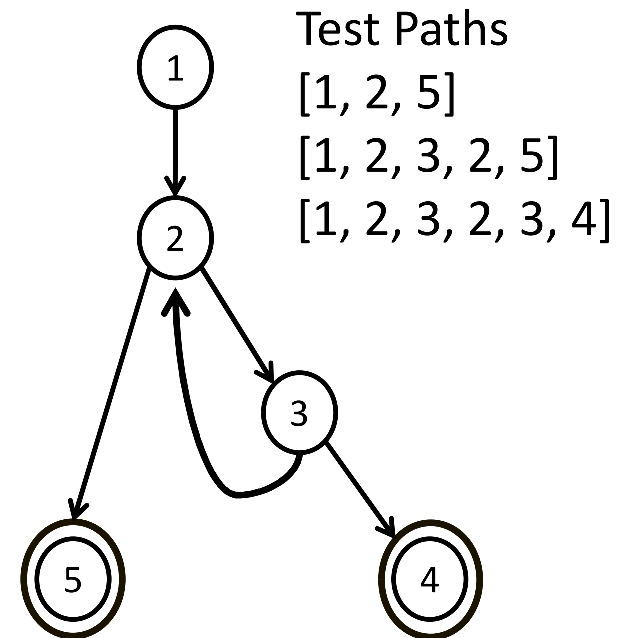
6 requirements for Edge-Pair Coverage
1. [1, 2, 3]
2. [1, 2, 5]
3. [2, 3, 4]
4. [2, 3, 2]
5. [3, 2, 3]
6. [3, 2, 5]

Test Paths
[1, 2, 5]
[1, 2, 3, 2, 5]
[1, 2, 3, 2, 3, 4]

Work with your neighbor

- Write input values that satisfy the Edge-Pair coverage requirements

```
/**Return first index of Integer n in path, or
 * -1 if n is not present in the path */
int indexOf (Integer n, List<Integer> path){
  for (int i=0; i < path.size(); i++){
    if (path.get(i).equals(n))
      return i;
  }
  return -1;
}
```



Question: is indexOf now well-tested?

We just saw Test Design in action

- Test Design: the process of creating effective tests
- A major ingredient in level-4 thinking
- The most **mathematical** and **technically challenging** testing activity
  - Requires knowledge from discrete math: graphs, sets, relations, etc.

The 5 steps in Test Design

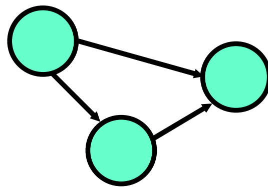
1. Do math or analysis to obtain test requirements
2. Find input values that satisfy the test requirements
3. Automate the tests
4. Run the tests
5. Evaluate the tests

In CS5154: Model-Driven Test Design

- We will do test design w.r.t. four models of software

Input Domains	Graphs	Logic Expressions	Syntax

A: {0, 1, >1}
B: {600, 700, 800}
C: {cs, ece, is, sds}



(not X or not Y) and A and B

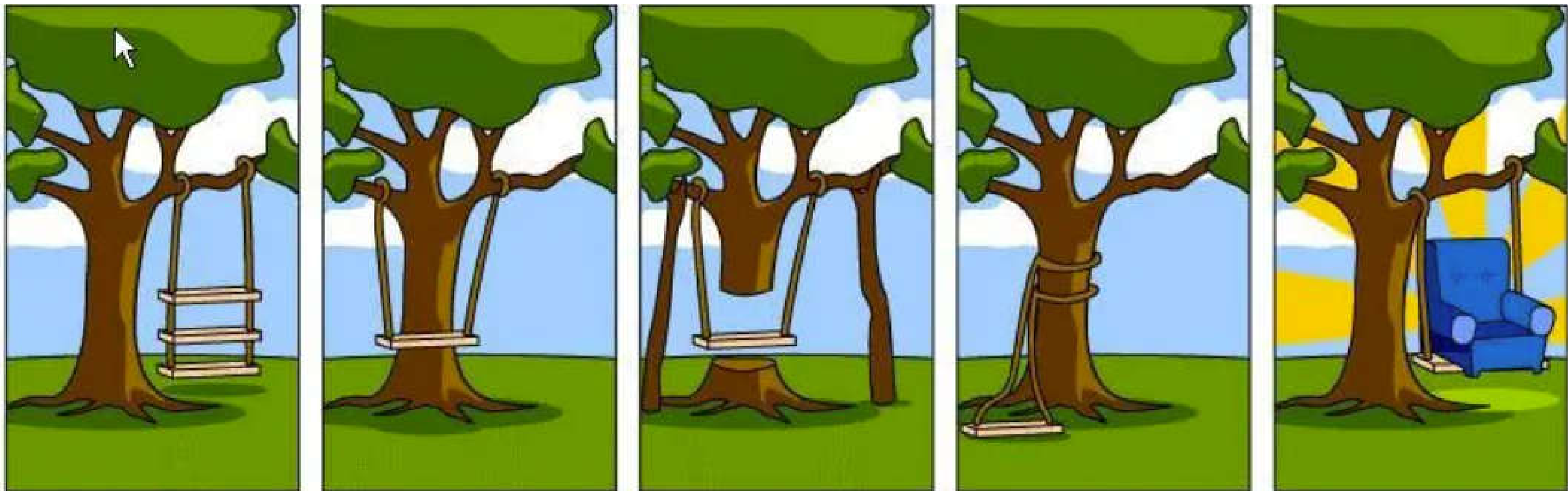
```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

- The first half of the course and the textbook cover MDTD

Why so much math in an SE class?

MDTD is about DESIGN

- Multiple test designs may exist for the same code



- Considering cost/benefit tradeoffs in designs is an essential part of SE

Image Credits: https://blogs.perficient.com/2011/07/22/how-to-build-a-tire-swing-a-case-for-agile-development/

Why should you care about MDTD?

- Develop “testing as a mental discipline” mindset (level 4)
- Organize HUNDREDs of criteria around four models of software
- Develop a disciplined approach to engineering your tests
  - What’s the difference between a programmer and a software engineer?

What we learned

- Standard testing terminology (test, fault, error, failure)
- Conditions that effective tests must meet (the RIPR model)
- Fundamental limit of software testing
- Levels of test maturity
- Introduction to model-driven test design

Next

- Test Automation