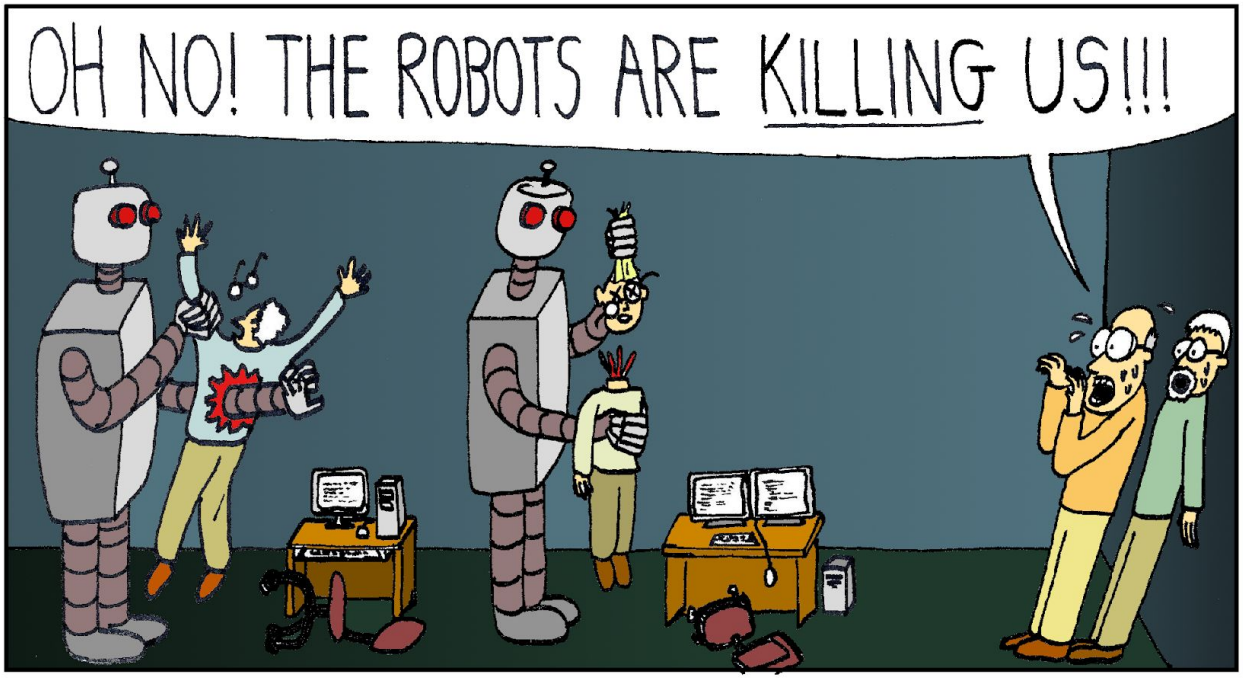
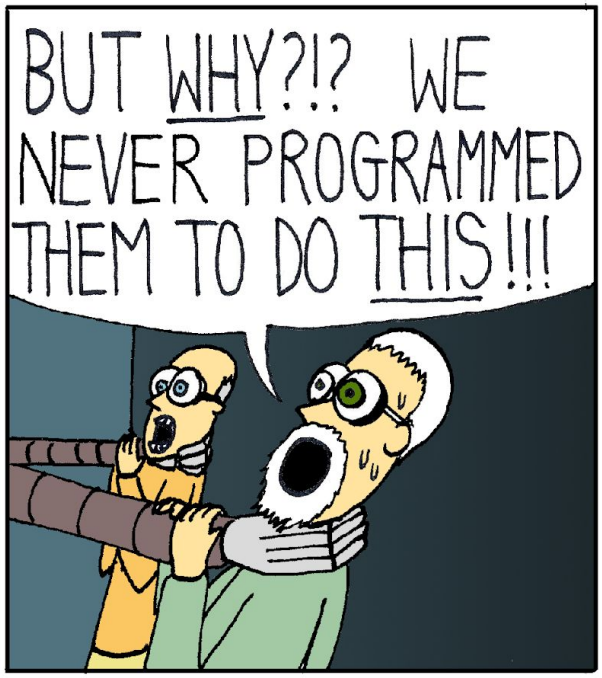
50.003: Elements of Software Construction

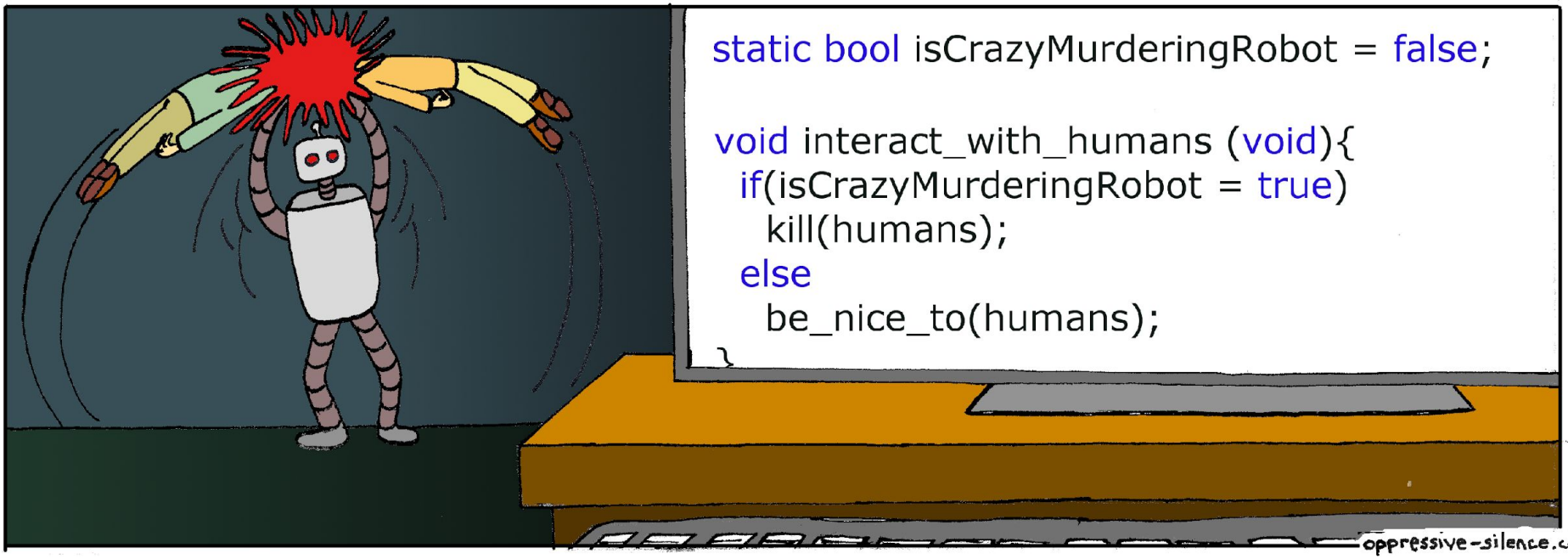
Week 6/8

Software Testing

Why bother testing?







Picture from http://oppressive-silence.com/comic/oh-no-the-robots

Mars Climate Orbiter
327.6 million USD lost (1998)

Back in school did you get “0” for not providing units to number?

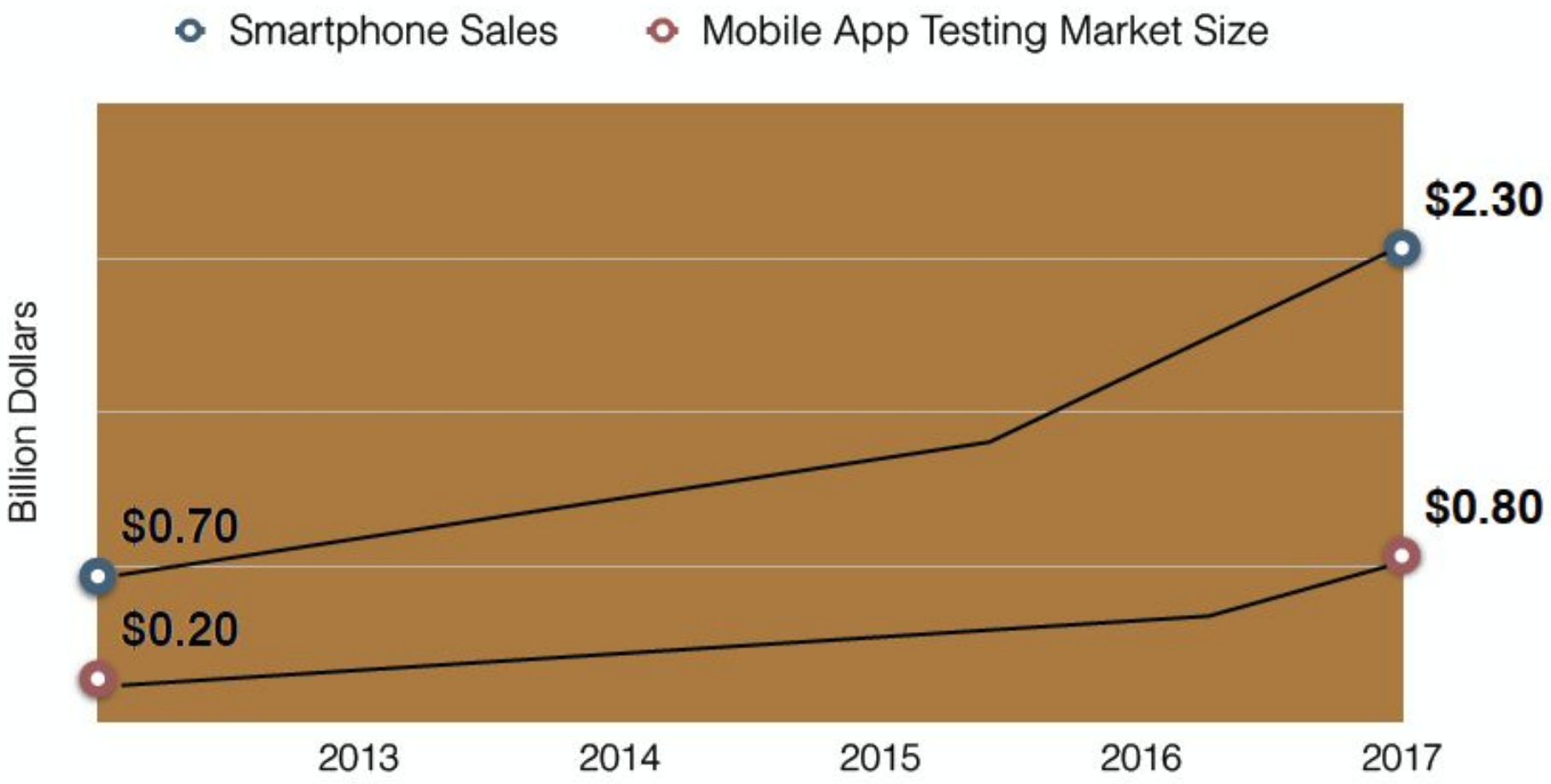
- 7 week?
- 7 month?
- 7 years?

Different engineers used different units

Pentium Floating point debacle
475 million USD lost (1995)

Missed five (5) entries while copying a lookup table having 1066 entries

Testing Market



Testing Effort

- Reported to be >50% of development cost [e.g., Beizer 1990]

- Microsoft: 75% time spent testing
  - 50% testers who spend all time testing
  - 50% developers who spend half time testing

Testing is like Rehearsal?

Edgar Degas: The Rehearsal. With a rehearsal, we want to check whether everything will work as expected.

This is a test

But?

Software is not a planned linear show. Meaning, if it works once, will it work again?

But?

And soon becomes unmanageable…..

A curse!!!

Testing can only find the presence of errors, not their absence

Edsger W. Dijkstra

Computer Scientist

ACM Turing Award Winner, 1972

To show the absence of bugs: Static analysis, theorem proving, verification

Another curse!!!

Verification can only find the absence of errors, not their presence (in general)

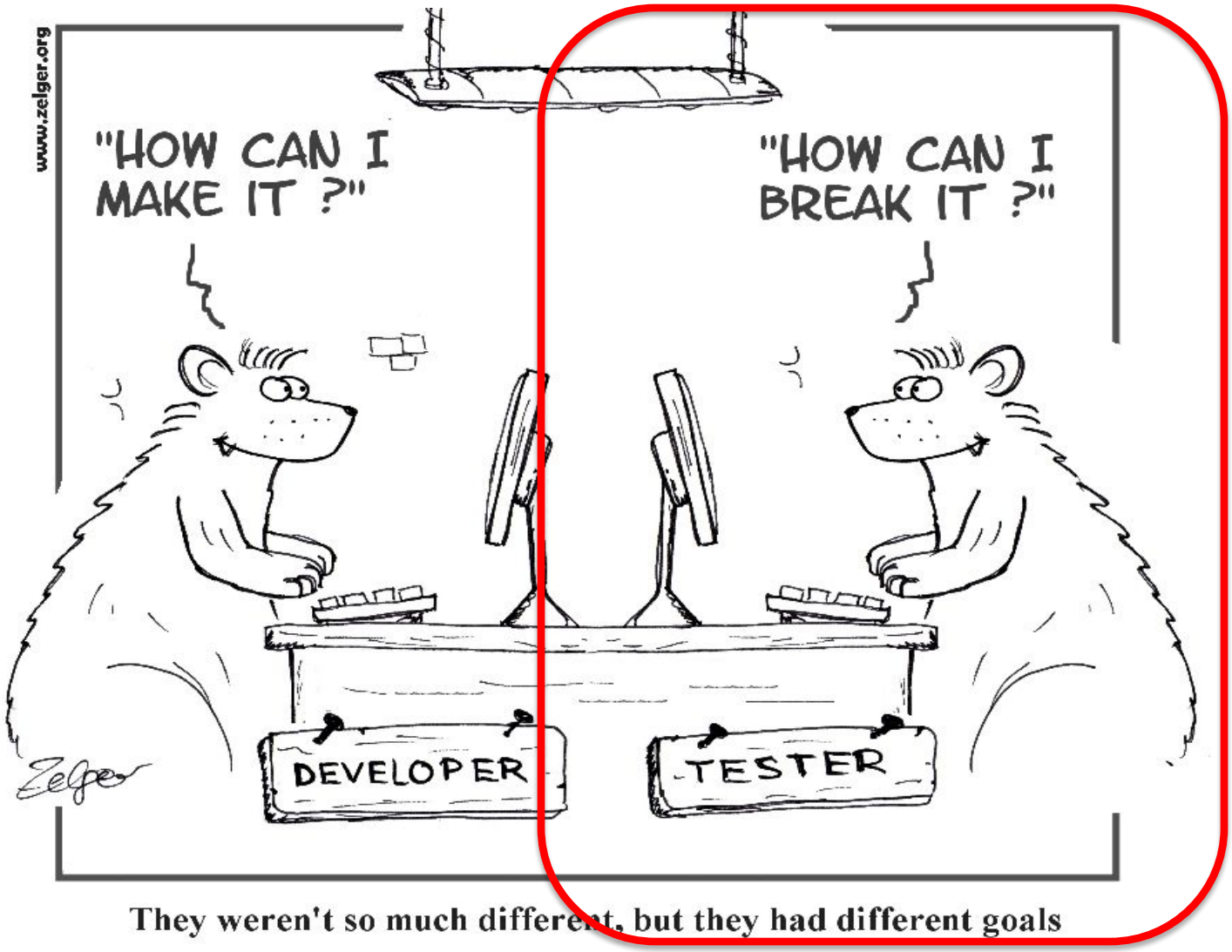
Andreas Zeller

Computer Scientist

ACM Fellow, 2010

To show the presence of bugs: Use testing

Testing is Industry Practice

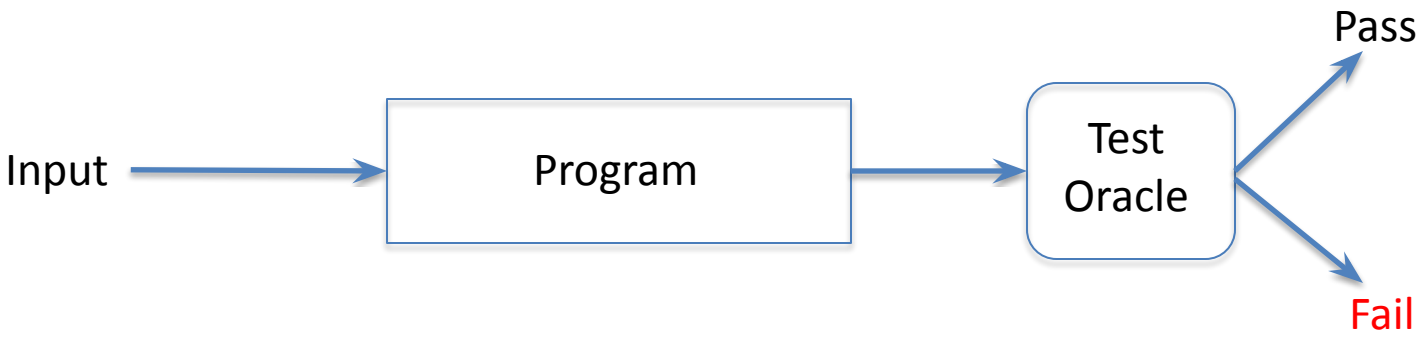


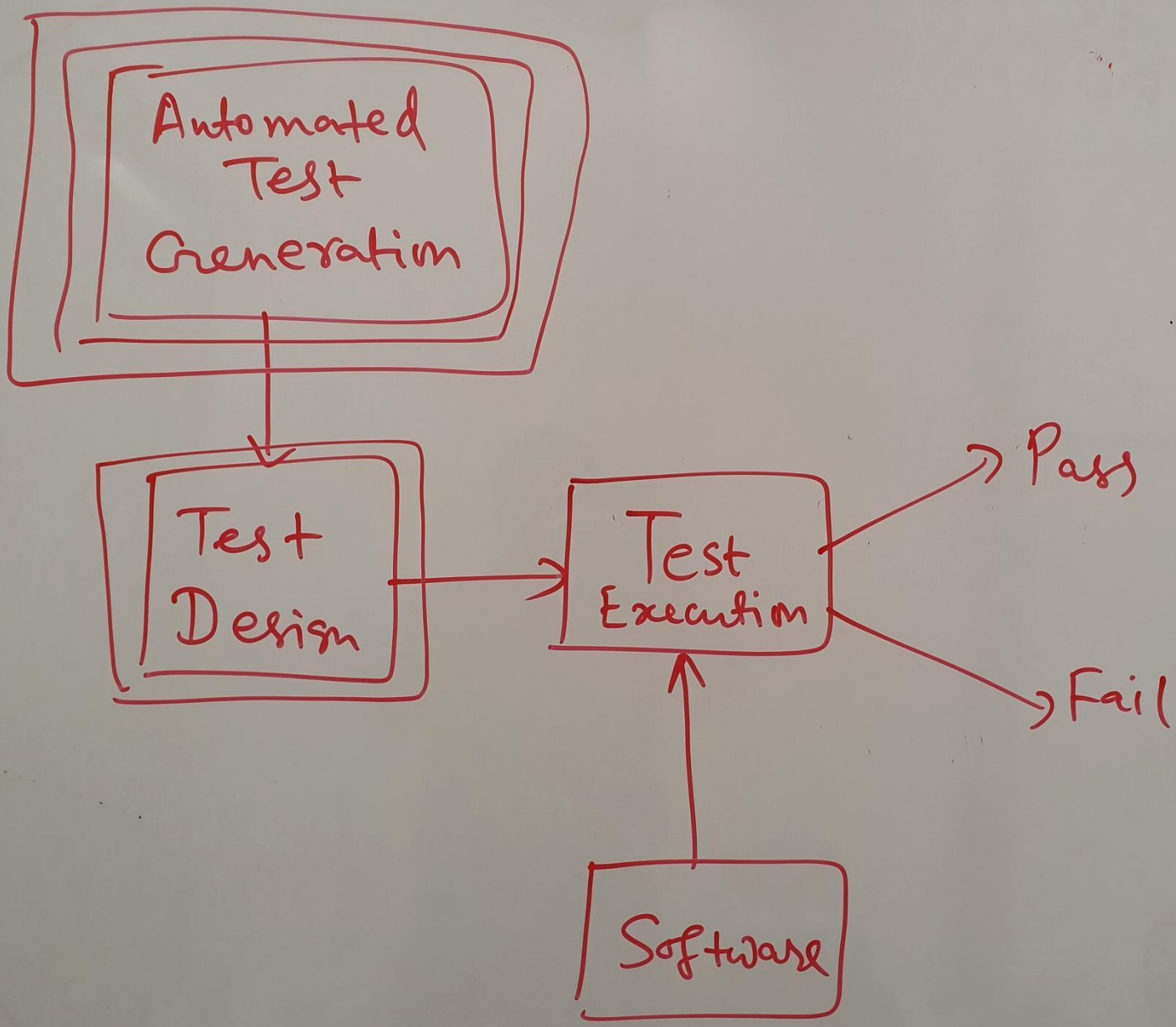
They weren't so much different, but they had different goals

We will take the side of this bear in this week

BASICS OF TESTING AND JUNIT

What is a test?




Automated Test Generation
Test Design
Test Execution
Pass
Fail
Software

JUnit

- Open source Java testing framework used to write and run repeatable automated tests
- A structure for writing test drivers
- JUnit is widely used in industry

Account.java; AccountTest.java

JUnit Test Fixtures

- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
- They should be initialized in a @Before method
  - JUnit runs them before every @Test method
- Can be deallocated or reset in an @After method
  - JUnit runs them after every @Test method

Stack.java; StackTest.java

Writing Tests for JUnit

- Need to use the methods of the junit.framework.assert class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded
  - *assertTrue (boolean)*
  - *assertTrue (String, boolean)*
  - *assertEquals (Object, Object)*
  - *assertNull (Object)*
  - *Fail (String) //fail a test with (possibly) a given message*
- All of the methods return void

Cohort Exercise 1

Given FindMax.java, write three test cases: one resulting in Failure (does not compute maximum), one resulting in Error (throws exception) and one resulting in Pass (computes maximum).

FindMax.java; FindMaxTest.java

Cohort Exercise 2

Fix the class Stack (in Stack.java) so that all tests in StackTest.java pass. Do not change the StackTest.java file.

Stack.java; StackTest.java

Cohort Exercise 3

Given the testRepOk method in StackTest.java, decompose it into multiple **equivalent** test cases.

StackTest.java; StackTestSolution.java

Parameterized Tests

How to test a function with similar values?

- Parameterized Unit Tests Call Constructor For Each Logical Set of Data Values
  - Same Tests Are Then Run On Each Set of Data Values
  - List of Data Values Identified with @Parameters Annotation

ParameterizedTest.java

Cohort Exercise 4

Write a parameterized test for QuickSort.java.

QuickSortParameterizedTest.java

JUnit Resources

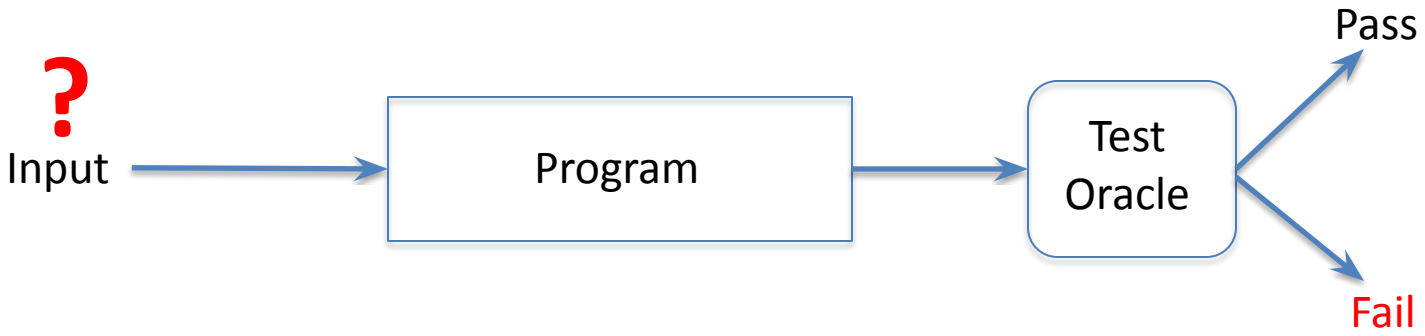
- Some JUnit tutorials
  - http://open.ncsu.edu/se/tutorials/junit/
    (Laurie Williams, Dright Ho, and Sarah Smith)
  - https://www.tutorialspoint.com/junit/
  - http://articles.jbrains.ca/JUnitAStarterGuide.pdf
    (J.B. Rainsberger)

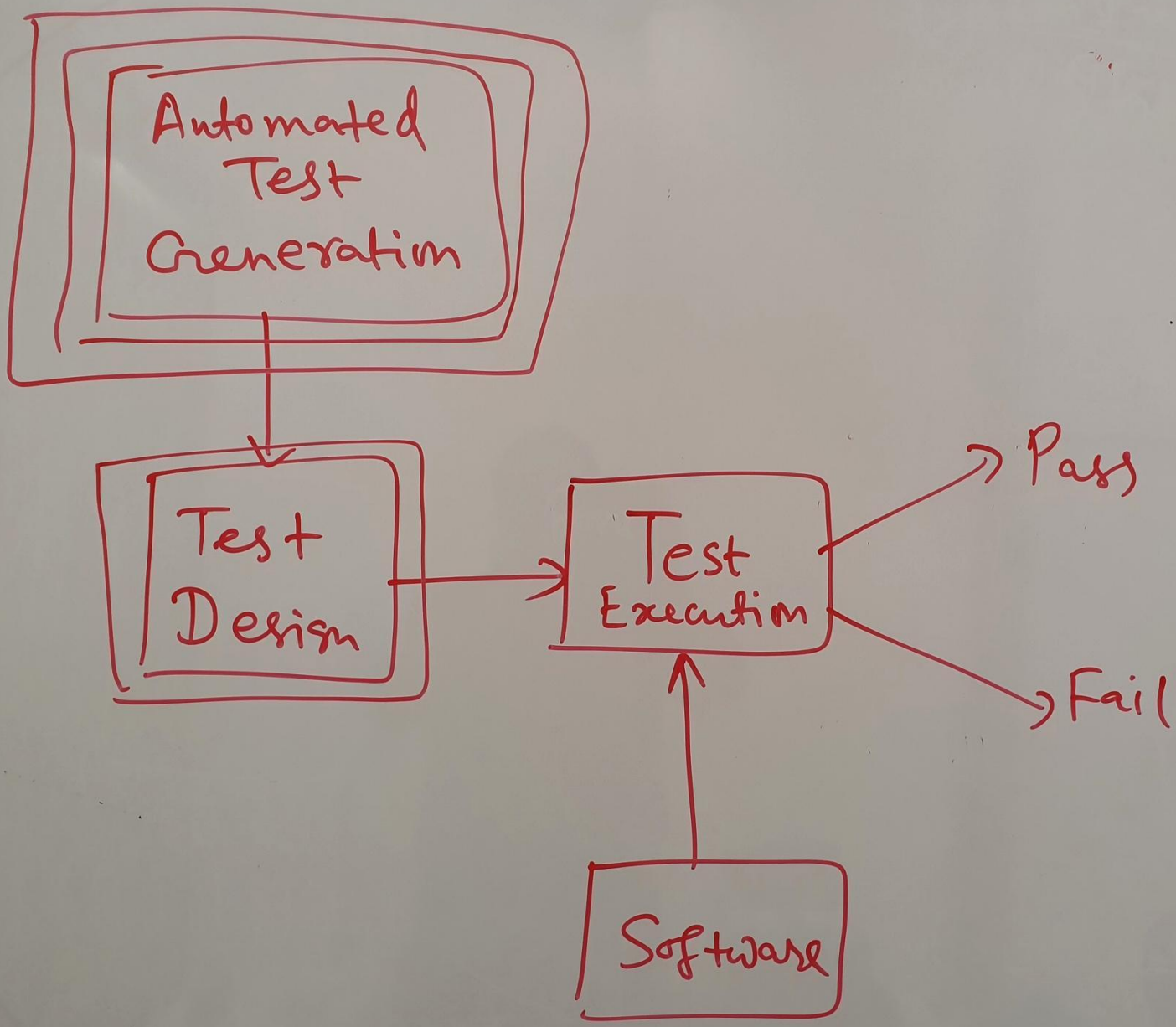
Summary

- The only way to make testing efficient as well as effective is to automate as much as possible
- JUnit provides a very simple way to automate our unit tests
- It is no "silver bullet" however … it does not solve the hard problem of testing :

What test values to use ?

- **This is test design … the purpose of test criteria**




Automated Test Generation
Test Design
Test Execution
Pass
Fail
Software

Typical Automated Testing Use Case

Example 1: A Sorting Algorithm
- Test Input? Test Oracle?

Example 2: A searching Algorithm
- Test Input? Test Oracle?

Example 3: A web page
- Test Input? Test Oracle?

Example 4: A Calculator Application
- Test Input? Test Oracle?

Example 5: Amazon Alexa Speech Recognition
- Test Input? Test Oracle?

Example 6: A Bluetooth Speaker/Headphone
- Test Input? Test Oracle?

Example 7: A C/C++ Compiler
- Test Input? Test Oracle?

Example 8: A Face recognition system/An object detection system
- Test Input? Test Oracle?

Example 9: A test generator :-)
- Test Input? Test Oracle?

TEST DESIGN

Black Box Testing

We see the program as a black box, we ignore how it is being written

White Box Testing

In White Box testing, tests are written based on the code

White Box Testing

Black box testing:

Tests based on specification

Test covers as many specification as possible

White box testing:

Tests based on code

Test covers as much implemented behavior as possible

Black Box vs. White Box Testing

Black box testing:

Tests based on specification

Better at finding whether code meets the specification or some specs are not implemented

White box testing:

Tests based on code

Better at finding crashes, out-of-bound errors, file handling errors etc.

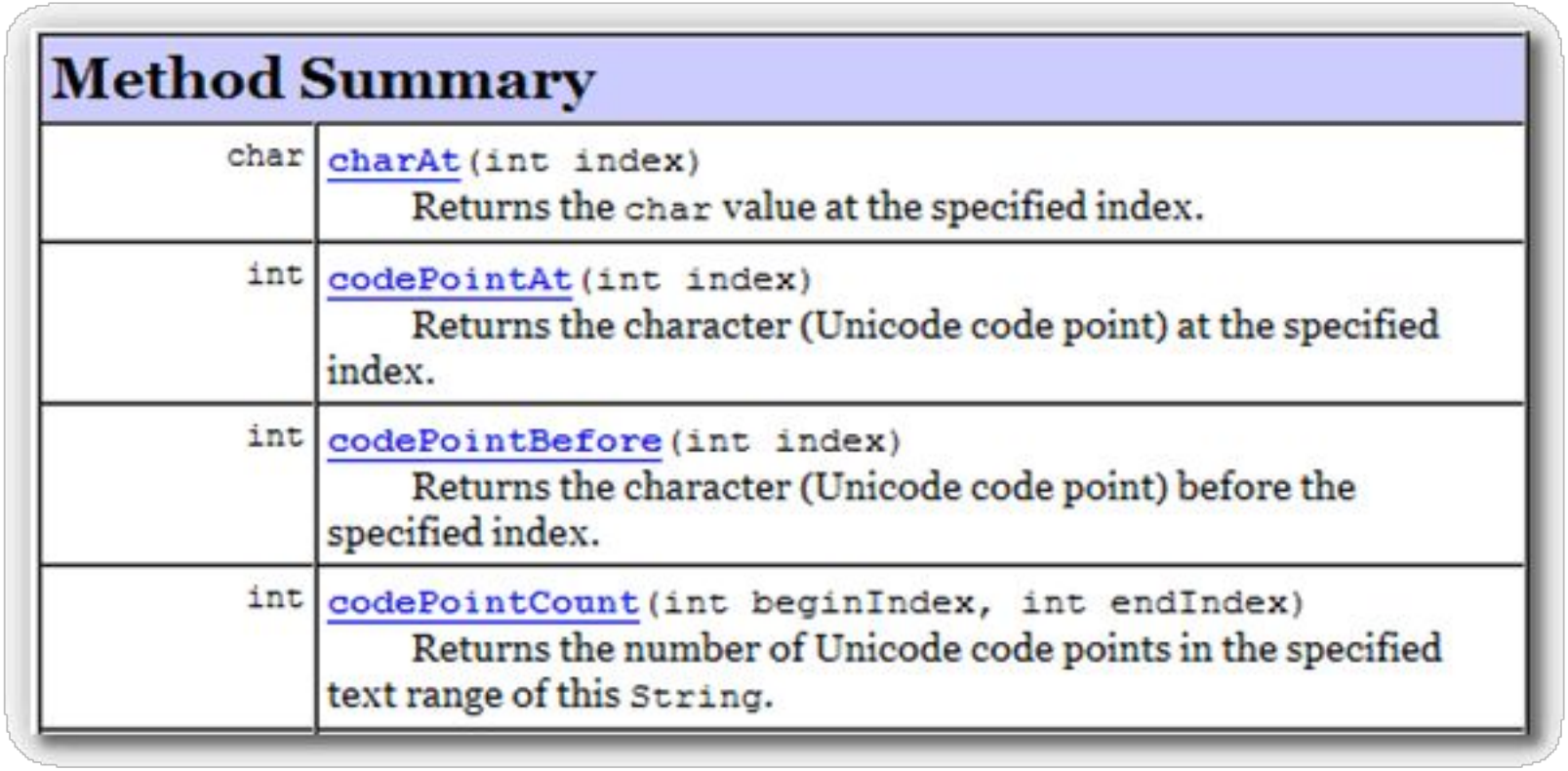
Black Box Testing

We see the program as a black box, we ignore how it is being written

Write tests from specification

JavaDoc

- Specify Java classes and methods
- JavaDoc starts out as comments in code



Method Summary	
char	`charAt(int index)` Returns the `char` value at the specified index.
int	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
int	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index.
int	`codePointCount(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this `String`.

Low-level specification

```
public class Account {
     int  balance;
     /* @invariant balance >= 0 */

    public withDraw(int amount) {….}


}
```

What kind of test would you design from this documentation?

Low-level specification

```
/**
* Edit the text in the biodata
* Precondition: isEditMode()
* Postcondition: result != null
* @return
* @param name
*/

public String deliverText(String text) {

 …..
}
```

What kind of test would you design from this documentation?

High-level specification

- UI
  - e.g. tasks user can perform
  - *e.g. click->play->pause->exit*
  - *e.g. user interactions in the Angry Bird game*
- Web server
  - e.g. URLs
  - http://sudiptac.bitbucket.io
  - Whatever://\\$##^^☺/\\
- Internet server
  - e.g. Message format

Important Consideration for Black Box Test Planning

- Look at requirements/problem statement to generate.
- Test cases need to be traceable to a requirement.
- Repeatable test case so anyone on the team can run the exact test case and get the exact same result/sequence of events.

©L. Williams

Equivalence Class Partitioning

- Divide your input conditions into groups (classes).
  - Input in the same class should behave similarly in the program.

Email address(es)
(valid and invalid)

e.g. su@sutd.edu.sg
e.g. this_is_not/an_email

Delivery
(pass or fail)

Equivalence partitioning

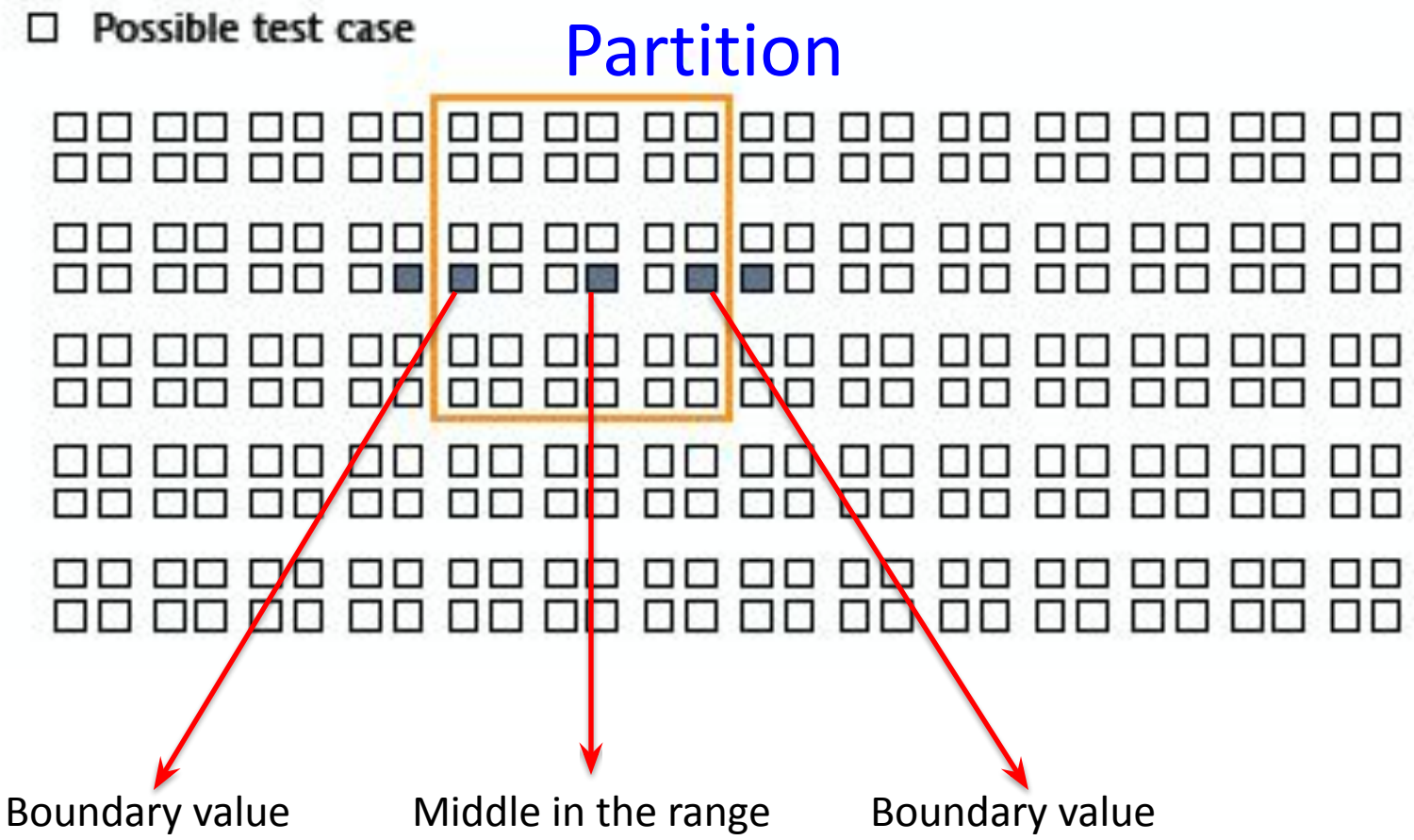
Sorted array: {0,5,7,89, 111}
Reversed array: {113, 907, 12, 1}
With negatives: {-1, -5, -6, -1111}
With duplicates: {4,6,6,6,6, 121212, 34}

Equivalence partitioning

Input list: {0,1,7,89, 111}; Output List = {111, 89, 7, 1, 0}

How do you design the partitions?

Boundary value analysis



Boundary value analysis

Email address(es)
(valid and invalid)

Delivery
(pass or fail)

(valid email)
- su@sutd.edu.sg

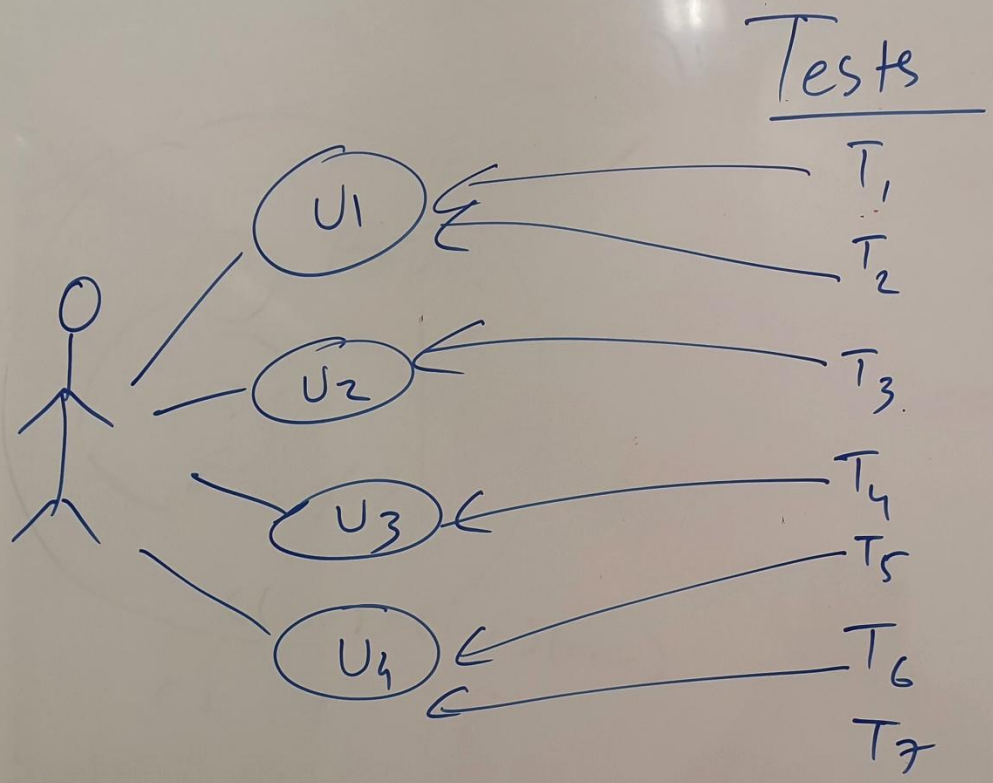
(middle value)
- blahZblahblahblah@sutd.edu.sg

(max length email, boundary)

(Invalid email)
- $$6**@I.do.not.know

(middle value)
- su@@sutd.edu.sg

(only one extra @, boundary)



Black Box Testing :-

1. Refer to the use case diagram
2. For each use case, find the space of input
3. For the input space for a use case, do equivalence part.
4. For each partition, choose a boundary & a middle value

White Box Testing

We see the program as code, we test as many implemented features as possible

Coverage

- Make sure tests *cover* each part of program
  - Every statement
  - Every branch
  - Every path
  - Every condition

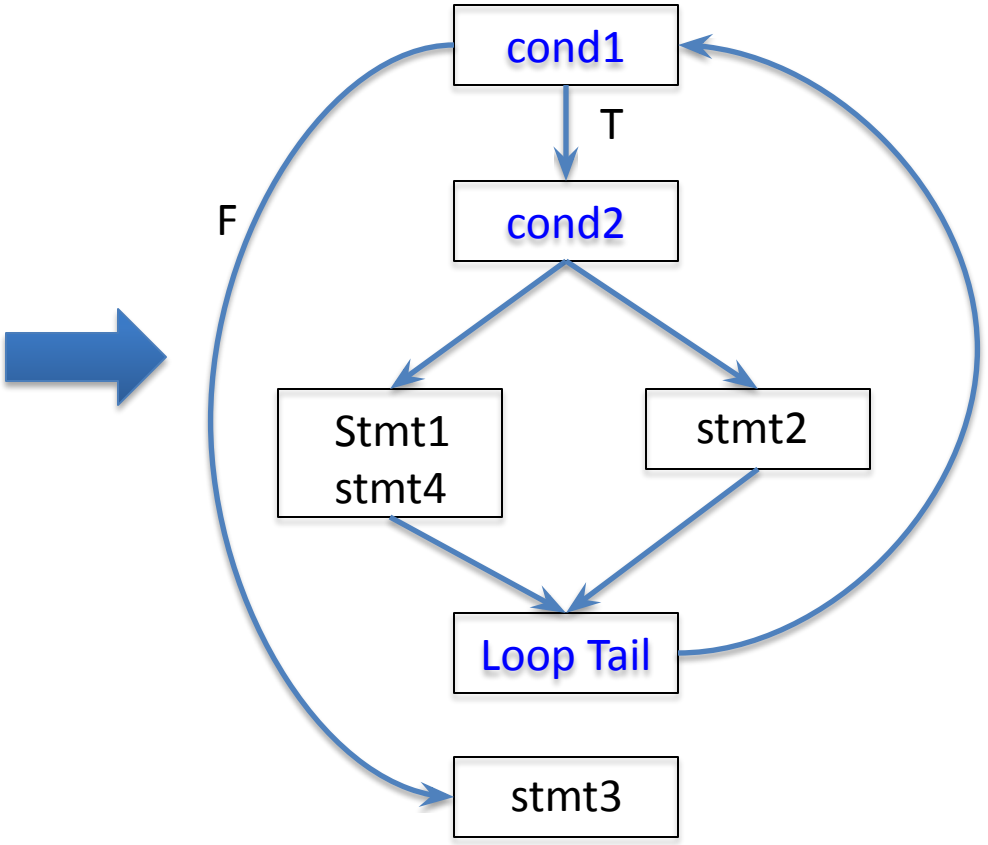
- Measures the quality of tests

Control Flow Graph

- For advanced white-box testing, it is often important to capture the control flow of the program
- For this, we have control flow graph (CFG)

Control Flow Graph

```
while (cond1)  {
      if (cond2)
    //stmt1
    //stmt4
      else
        //stmt2
}
//stmt3
```



Outline Today

- Make sure tests *cover* each part of program
  - Every statement
  - Every branch
  - Every condition
  - Every path
- Fault-based Testing

Cohort Exercise 5

- Open Disk.java. Draw the control flow graph of the function manipulate().

Disk.java

Every Statement

- For each statement that can be executed, there must be one test case which executes it, if feasible.

Cohort Exercise 6

- Write a set of tests to cover each statement (if feasible) of the manipulate() function. How many tests did you write? Is it the minimum number of tests to cover all the statements?

Disk.java

Every Branch

- For each branch (if the branch can be executed), there must be a test case which executes it.

(Branch ≠ Statement) coverage

```
if (x > 0)
   Stmt1;
x++;
```

A test input x = 5 will cover all the statements, but it will not cover all the branches.

Cohort Exercise 7

- Write a set of tests to cover each branch of the manipulate() function, if feasible. How many tests did you write? Is it the minimum number of tests to cover all the branches?

Disk.java; DiskBranchCoverage.java

Branch Coverage

```
if ((x == 0) || (y > 0))
    y = y/x;
else
    x = y++;
```

Test 1: {x = 5, y = -5}

Test 2: {x = 7, y = 5}

Obtains branch coverage, but does not expose the bug.

*How about covering true and false outcome of **each condition**?*

Every Condition

- For each condition, there must be one test case which satisfies it and one which dissatisfies it.
- Question: how many test cases we need?
  - if (A && B)
    - {A = true, B = false}, {A = false, B = true}
  - if ((j>=0) && salary[j] > 10000)
    - ?

Cohort Exercise 8

- Consider your test cases that obtain branch coverage in the manipulate() function. Argue whether the test suite also obtains the condition coverage.

Disk.java; DiskBranchCoverage.java

Content to be covered in Week 10

Path Coverage, is it Real?

- Microsoft **SAGE**
- **KLEE** https://klee.github.io/
- **JPF** Symbolic Path Finder
  https://github.com/SymbolicPathFinder/jpf-symbc
- **DART**
  https://web.eecs.umich.edu/~weimerw/590/reading/p213-godefroid.pdf

January 11, 2012
Volume 10, issue 1

PDF

SAGE: Whitebox Fuzzing for Security Testing

SAGE has had a remarkable impact at Microsoft.

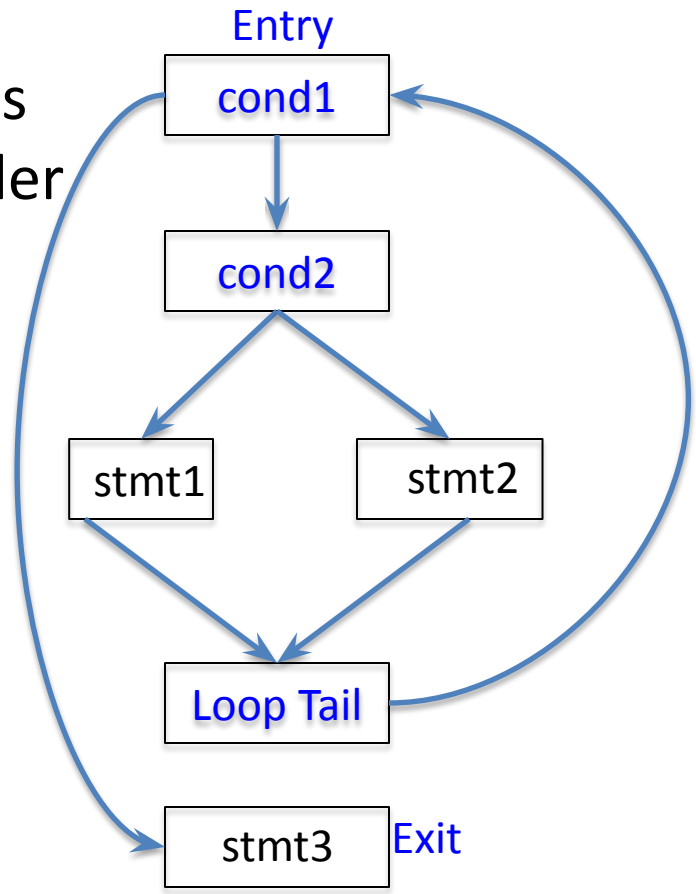
Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

Most *ACM Queue* readers might think of "program verification research" as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like 93–plus percent of PC users—that is, more than a billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

Tools targeting path coverage had been extremely successful in industry scale in finding new bugs in the last two decades

Every Path

- A **(terminating)** path is defined as a sequence of **executed** nodes in the control flow graph between the entry node of the graph and the exit node
- There are also non-terminating paths
- If not stated explicitly, we will consider only terminating paths



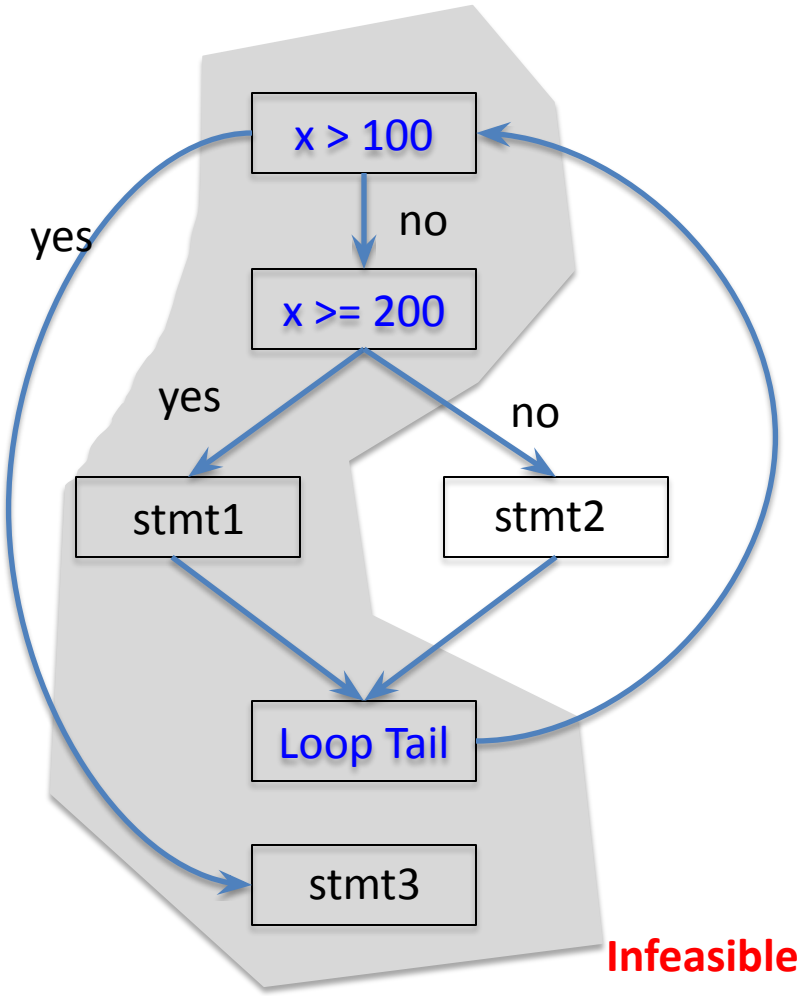
cond1->cond2->stmt1->loop tail->cond1->stmt3 is a path

cond1->cond2->stmt1->loop tail->cond1->cond2 ->stmt1->loop tail->cond1->stmt3 is also a path

cond1->cond2->stmt1->stmt2->loop tail->cond1 ->stmt3 is **not** a path

How many paths in total?
(assuming the loop terminates after 100 iterations)

Every Path



Remember some paths may be infeasible

Exercise (How many paths?)

```
int a = input(); // a is an input
int x, i = 0; // x and i are not inputs

while (i < 100) {
    if (a < 5)
        x++;
    else
        x--;
     i = i + 1;
}
```

Exercise (How many paths?)

```
int a[100] = input(); // a[100] is an input array
int x, i = 0; // x and i are not inputs

while (i < 100) {
    if (a[i] < 5)
        x++;
    else
        x--;
     i = i + 1;
}
```

Cohort Exercise 9 (Homework)

- Assume that the loop in the manipulate() function is terminated after at most 100 iterations (i.e., after 0 iteration, 1 iteration, …., 100 iterations etc.). Based on this assumption, compute the possible number of executed paths in the manipulate() function. Explain your answer.
  Note: You must discount paths that never terminate within 100 iterations.

Disk.java

White-box tests

- Purpose: exercise all the code
- Large number - take a long time to write
- Good for finding run-time errors
  - Null object, array-bounds error
- In practice, coverage can be better for evaluating tests than for creating them

Techniques for writing tests

- Black-box (from specifications)
  - Equivalence partitioning
  - Boundary value analysis
- White-box (from code)
  - Example: Branch coverage
- **Fault-based testing (from common errors)**
  - Think diabolically
  - Honestly, we just touch upon this topic to show there are more to whitebox testing.

Fault-based Testing

```
float foo (int a, b, c, d, e) {
        if (a == 0) {
           return 0.0;
        }
        int x = 0;
        if ((a==b) OR ((c==d) AND bug(a) )) {
              x +=1;
        }
        e = 1/x;
        return e;
  }
```

Ans: Any test where a != b and c != d so that x cannot be incremented.

What test cases would you like to create?

Exercise 12

Consider Disk.java. Assume that specification requires all the functions in the Disk class to be terminating. Write a Junit test that potentially reveals a bug in the manipulate() function.

DiskFaultTest.java