

50.003: Elements of Software Construction

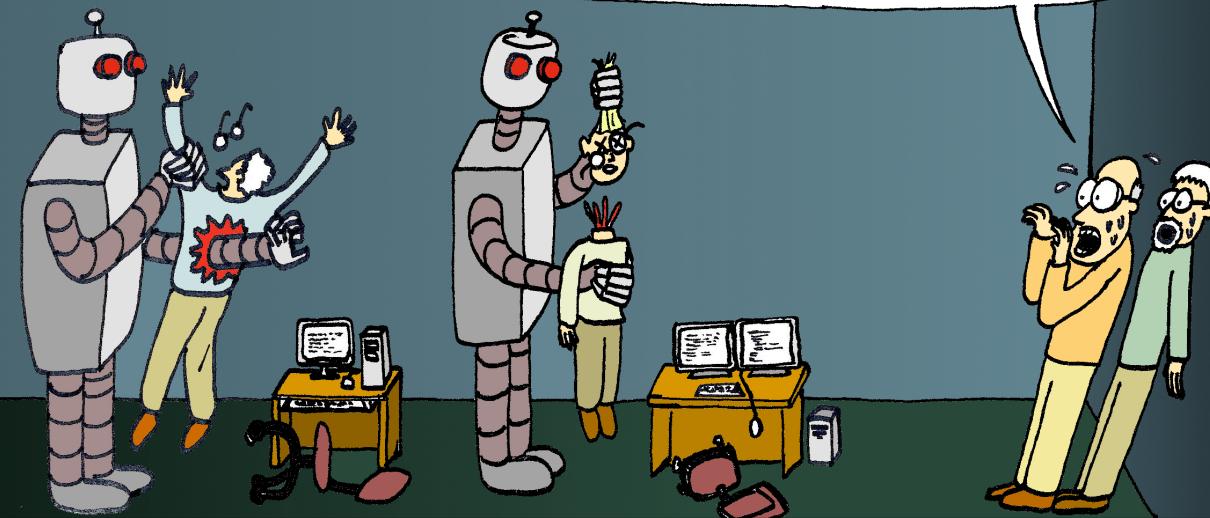
Week 3 and Week 4

Software Testing

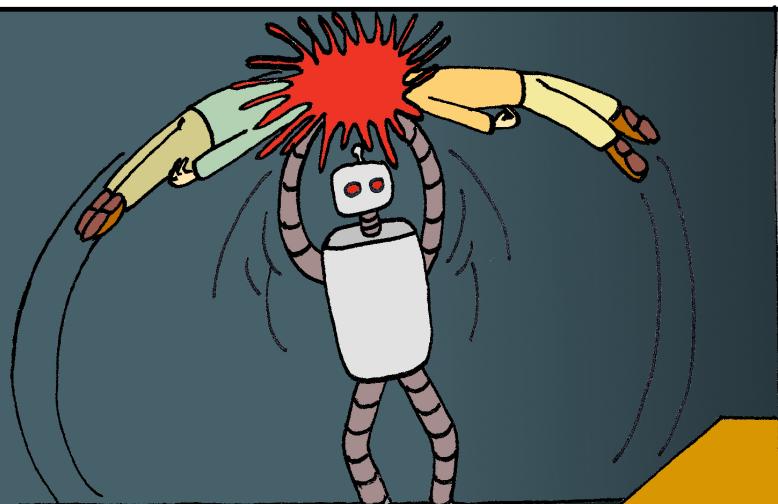
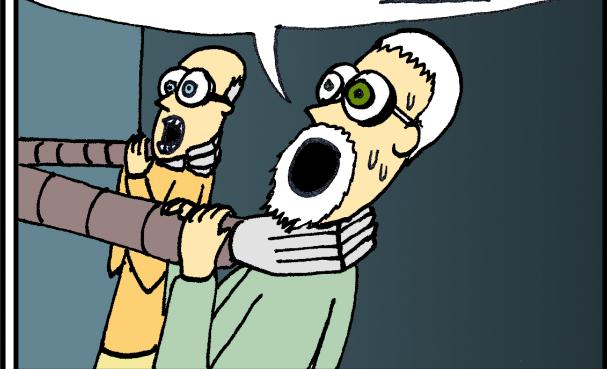
(with some slides from Darko Marinov and Andreas Zeller)

Why bother testing?

OH NO! THE ROBOTS ARE KILLING US!!!



BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!



```
static bool isCrazyMurderingRobot = false;  
  
void interact_with_humans (void){  
    if(isCrazyMurderingRobot = true)  
        kill(humans);  
    else  
        be_nice_to(humans);
```



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

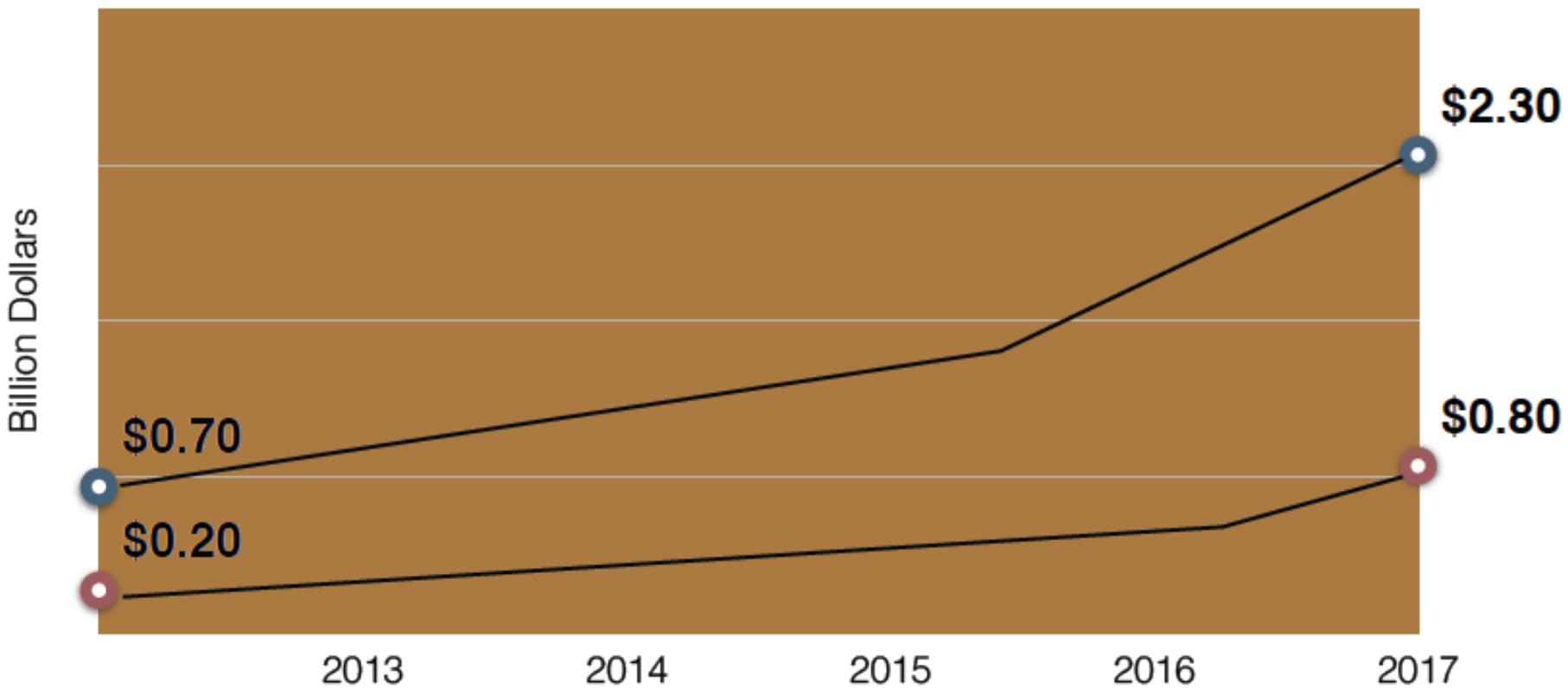


Pentium Floating point debacle
475 million USD lost (1995)

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

Testing Market

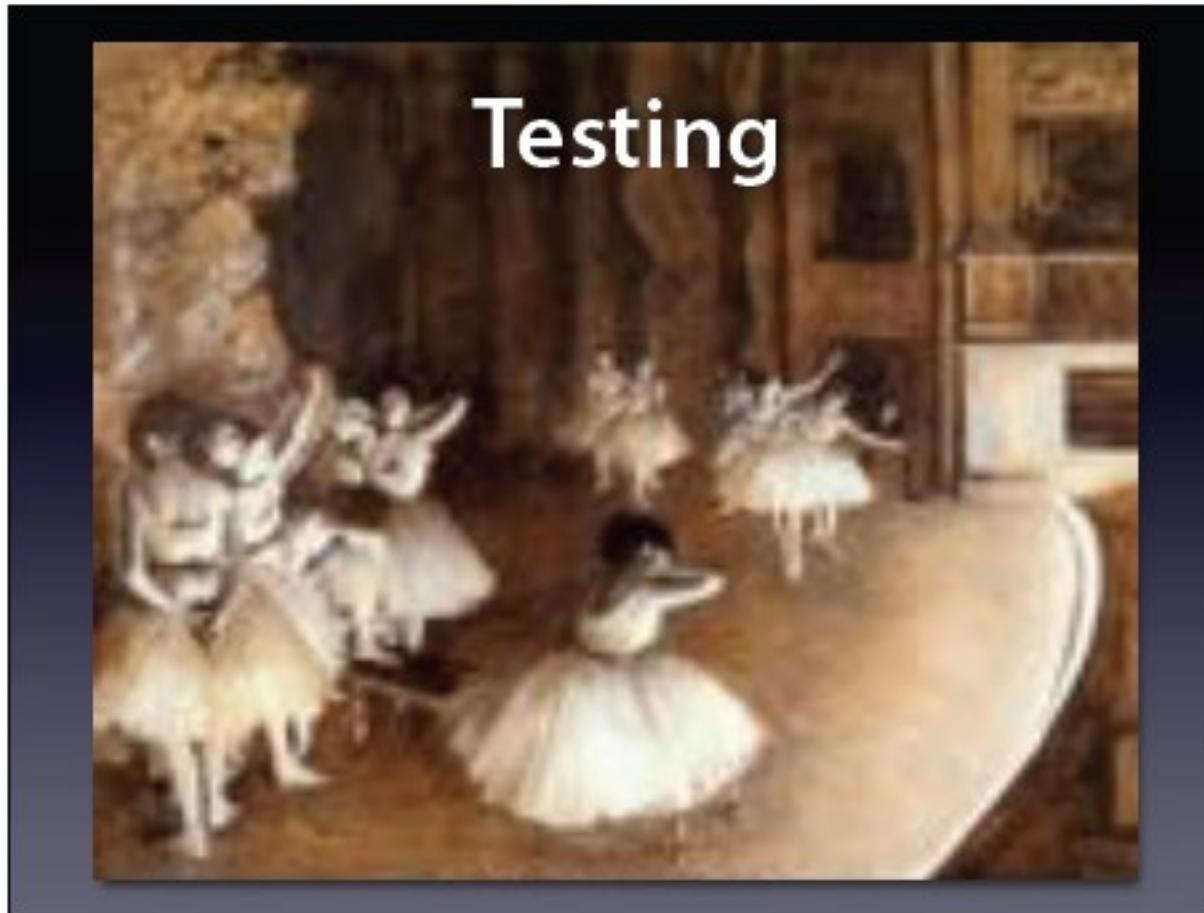
◆ Smartphone Sales ◆ Mobile App Testing Market Size



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 manifold



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

But?

Software is manifold



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)*

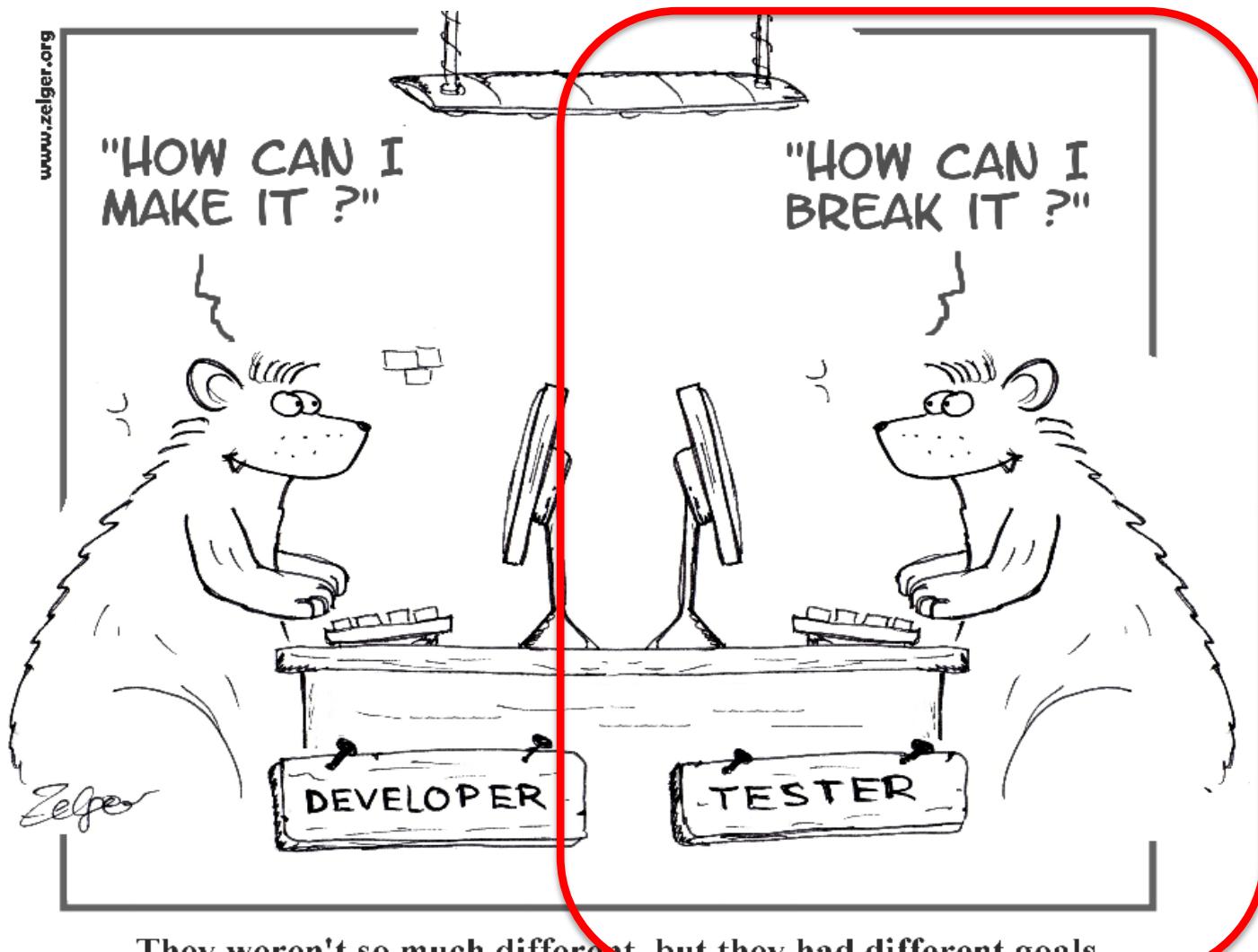
Andreas Zeller

Computer Scientist

ACM Fellow, 2010

To show the presence of bugs: Use testing

Testing is Industry Practice

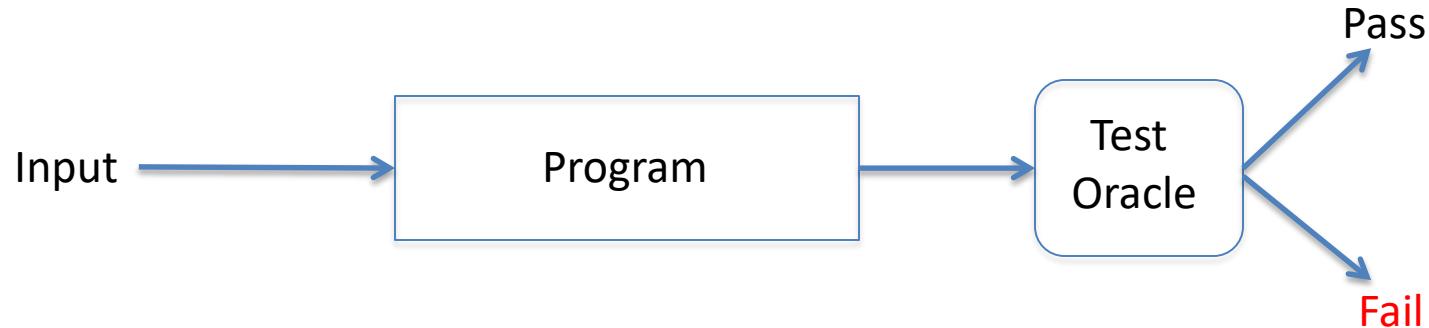


We will take
the side of
this bear in
this week

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

BASICS OF TESTING AND JUNIT

What is a test?



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



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`

fail() throws an assertion error unconditionally. This might be helpful to ensure that an expected exception has been thrown

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 *to clean up memory (eg stack)*

Stack.java; StackTest.java



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).

Failures are when your test cases fail - i.e. your assertions are incorrect.

Errors are unexpected errors that occur while trying to actually run the test - exceptions

FindMax.java; FindMaxTest.java



Exercise 2

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
- multiple objects of **ParameterizedTest** class, each object will run the @test*

```
ParameterizedTest.java
```



Exercise 3

Write a parameterized test for QuickSort.java.

```
QuickSortParameterizedTest.java
```

JUnit Tests

- JUnit can be used **to test** ...
 - ... an entire object
 - ... part of an object – a method or some interacting methods
 - ... interaction between several objects
- Each test is embedded into one **test method**
- A **test class** contains one or more test methods
- Test classes include :
 - A **test runner** to run the tests (`main()`)
 - A collection of **test methods**
 - Methods to **set up** the state before and **update** the state after each test and before and after all tests

JUnit Resources

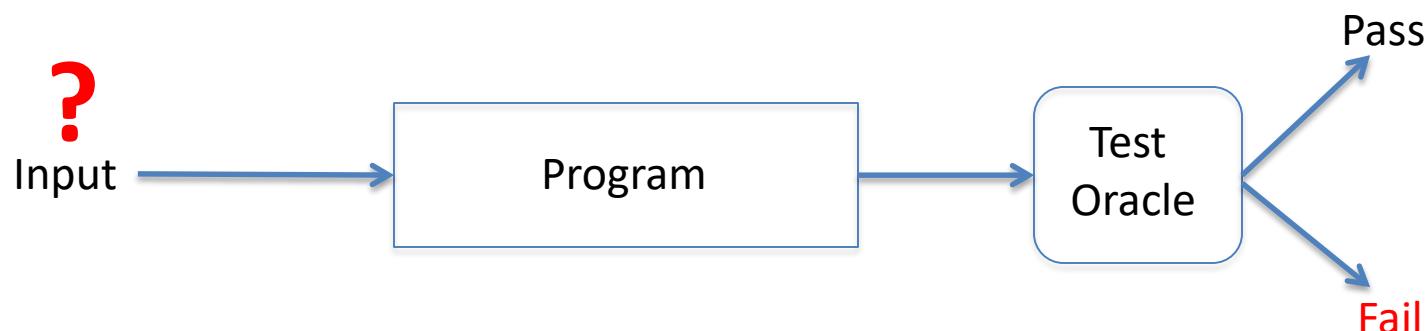
- Some JUnit tutorials
 - <http://open.ncsu.edu/se/tutorials/junit/>
(Laurie Williams, Dright Ho, and Sarah Smith)
 - <https://www.tutorialspoint.com/junit/>
 - <http://articles.jbrains.ca/JUnitAStarterGuide.pdf>
(J.B. Rainsberger)

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 :

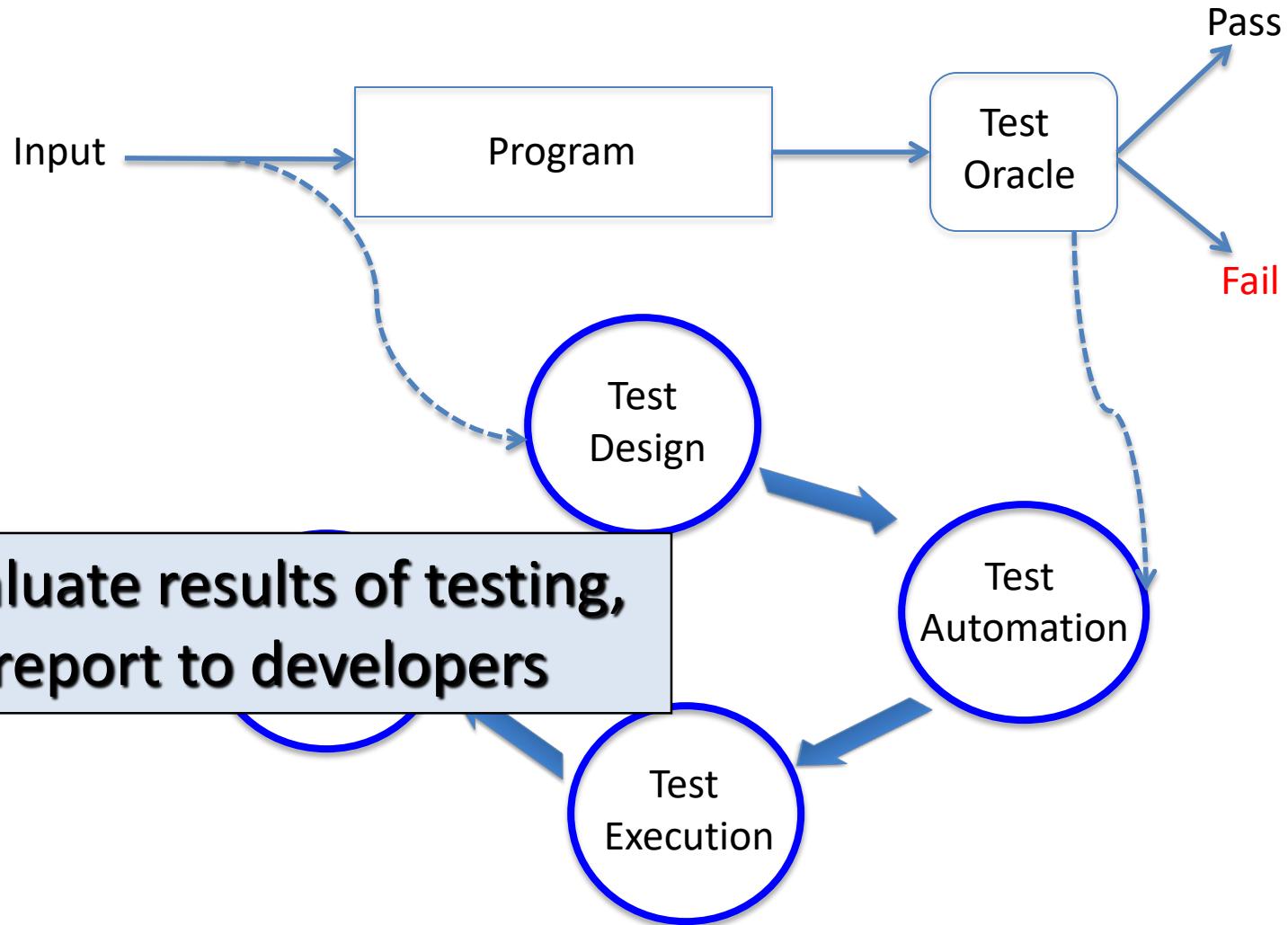
What test values to use ?

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



TEST DESIGN

Test Activities



Black Box Testing



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

Black Box Testing



*If the program is not the code, what it is?
(hint: we know what it is supposed to do)*

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:

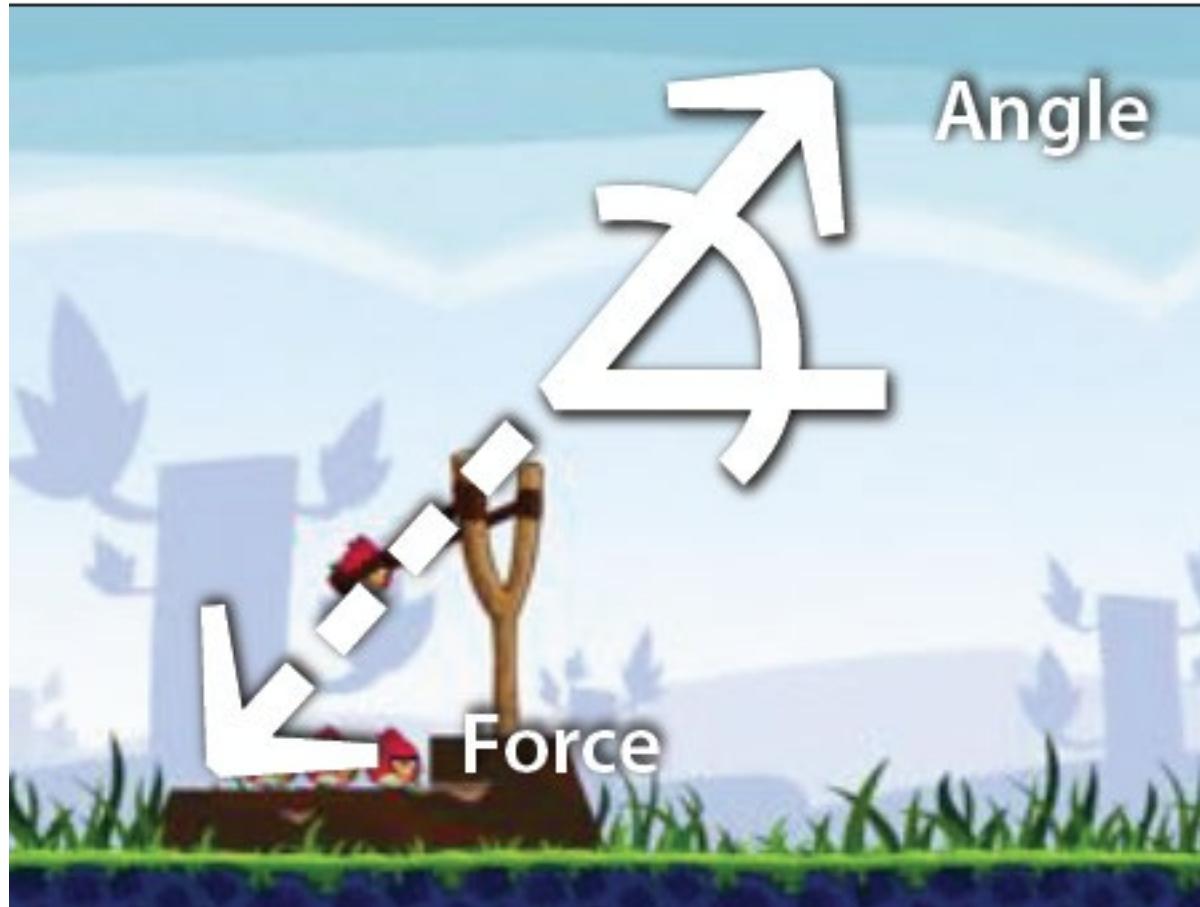
Tests based on code

Test covers as much implemented behavior as possible

Black Box Testing

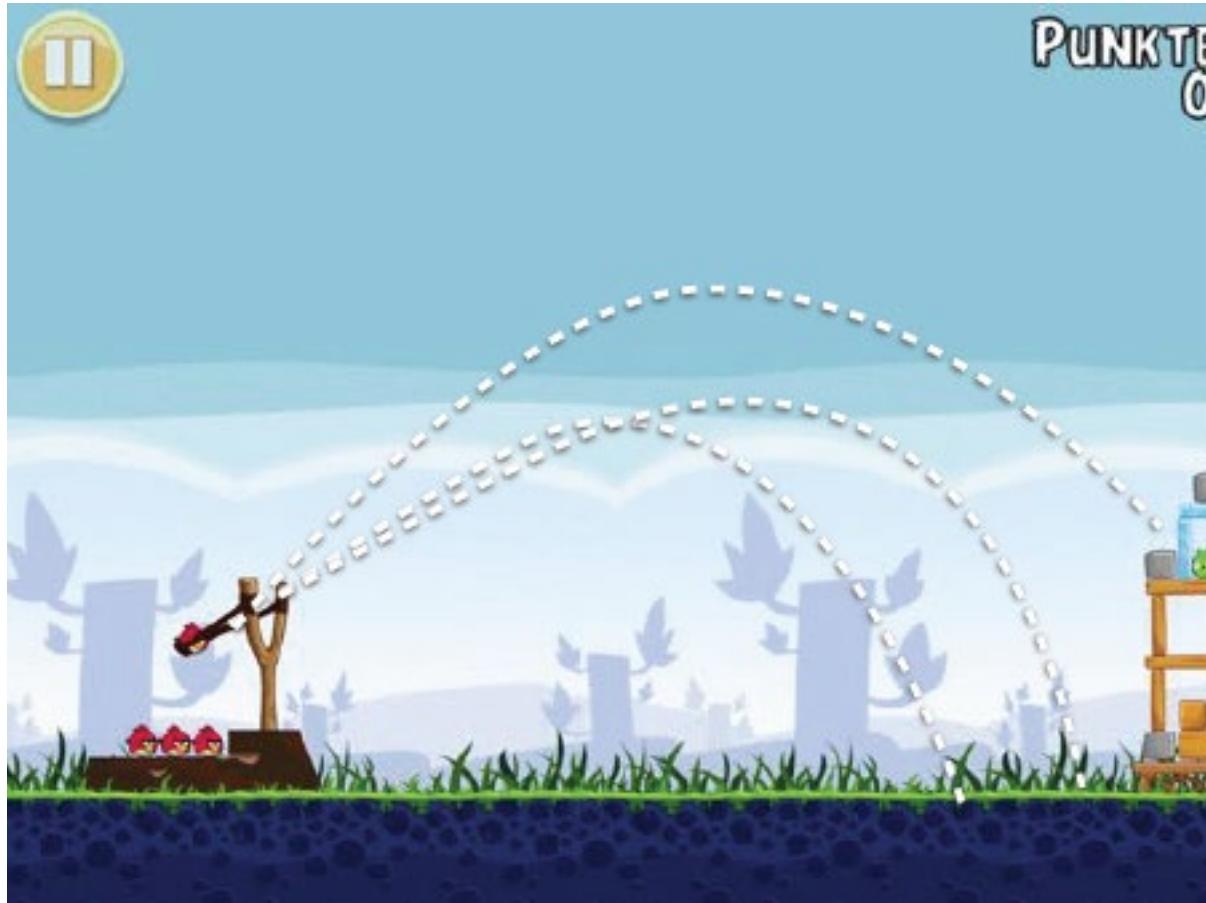


Black Box Testing



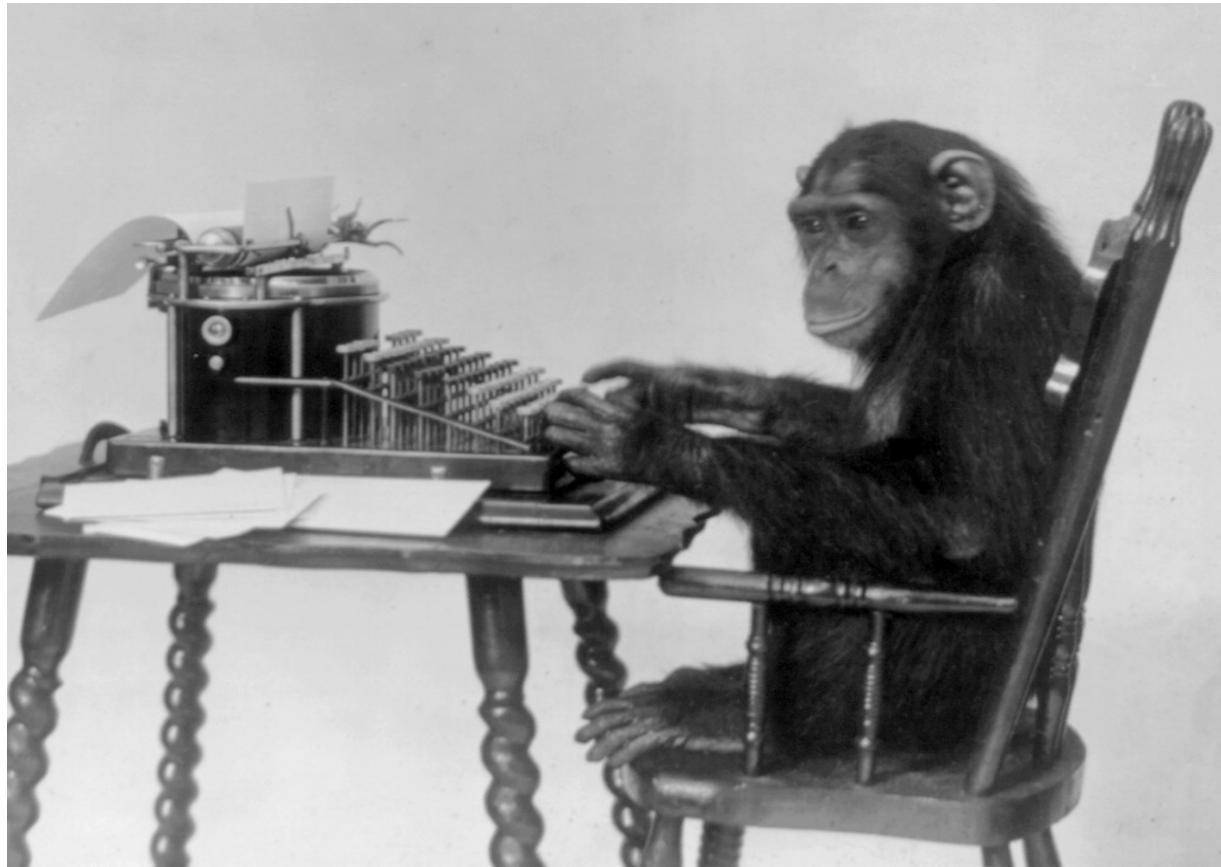
Write tests for different Angles and Force (Test Design)

Black Box Testing



Observe behavior and spot potential bugs (Test Evaluate)

Infinite Monkey Theorem

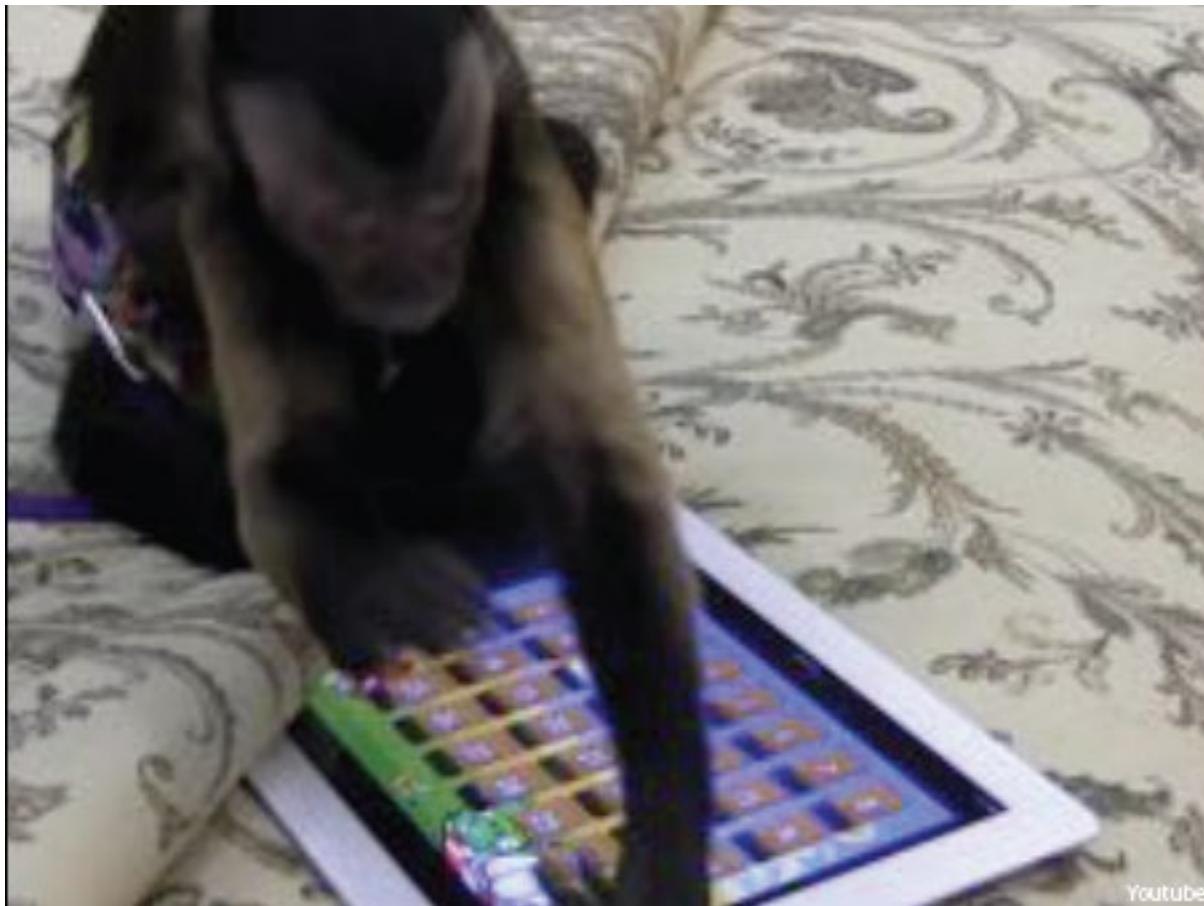


Picture:

<https://en.wikipedia.org>

“Monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type the complete works of William Shakespeare”.

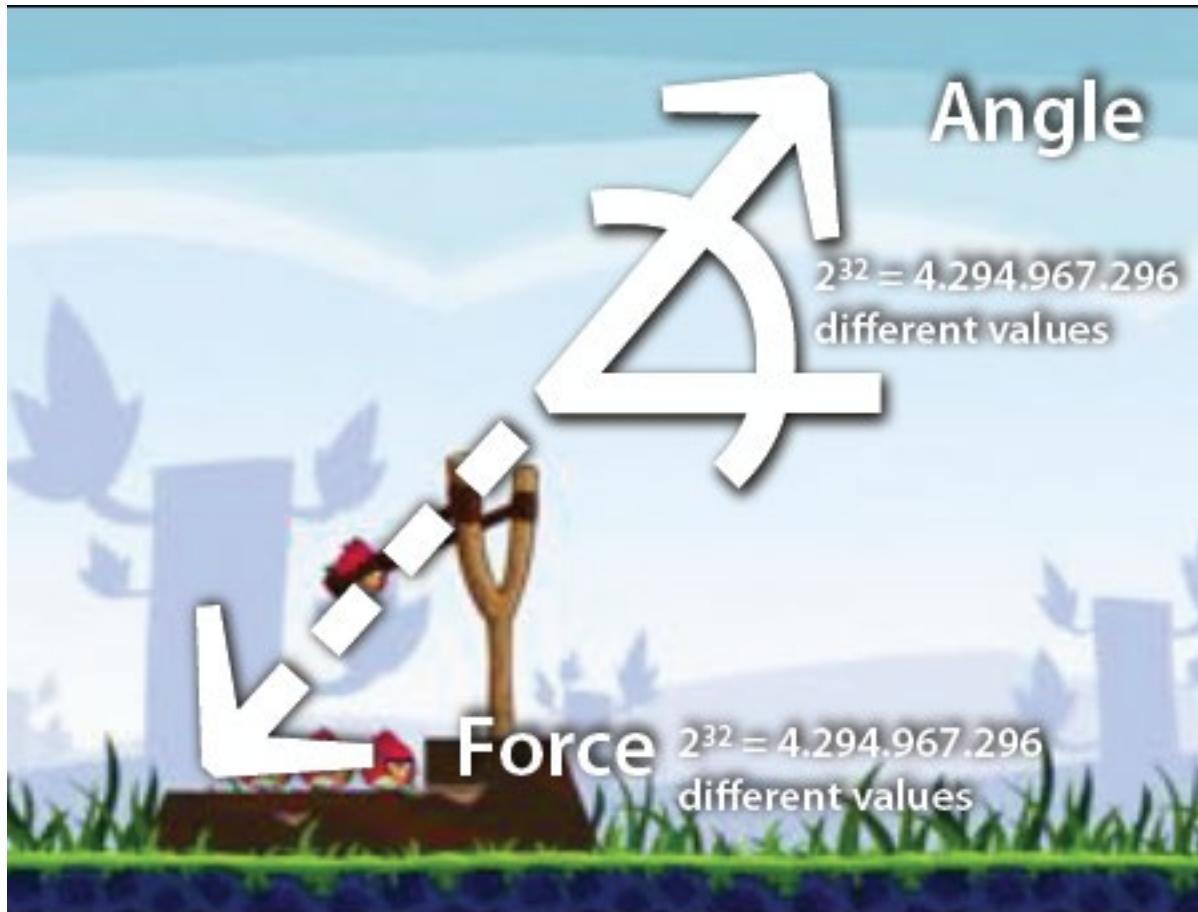
Black Box Testing



Youtube

Instead of writing novels, let us put a monkey to perform black box tests on Angry Bird

Black Box Testing



Angle -> 2^{32} possible values

Force -> 2^{32} possible values

Assuming both Angle and Force are integers

Black Box Testing

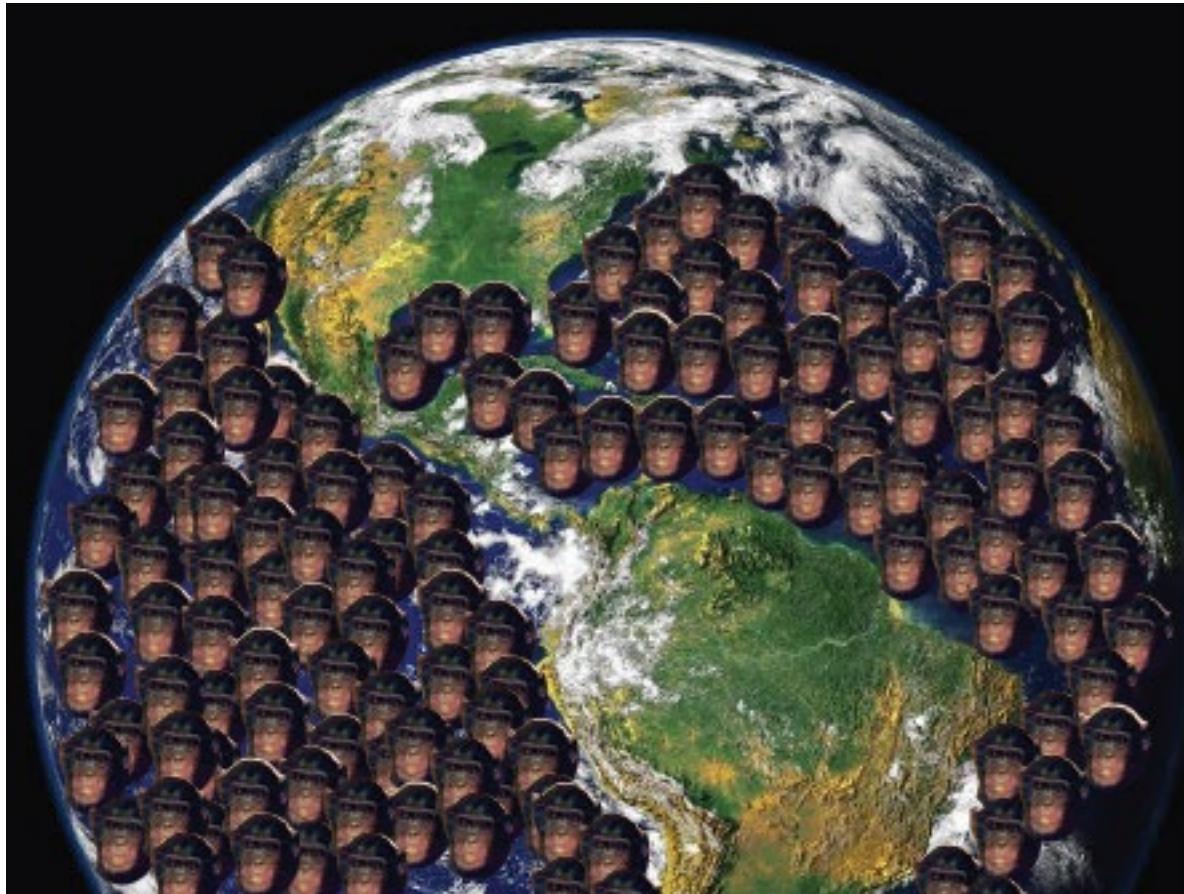


Angle -> 2^{32} possible values

Force -> 2^{32} possible values

Total possibilities = $2^{64} = 18, 446, 744, 073, 709, 551, 616$ diff. executions

Black Box Testing



Of course, if you fill the world with 18 million trillion monkeys, we can test angry bird case in one minute

Black Box Testing



*We need a systematic process to design test and evaluate them
(for all testing methodologies)*

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

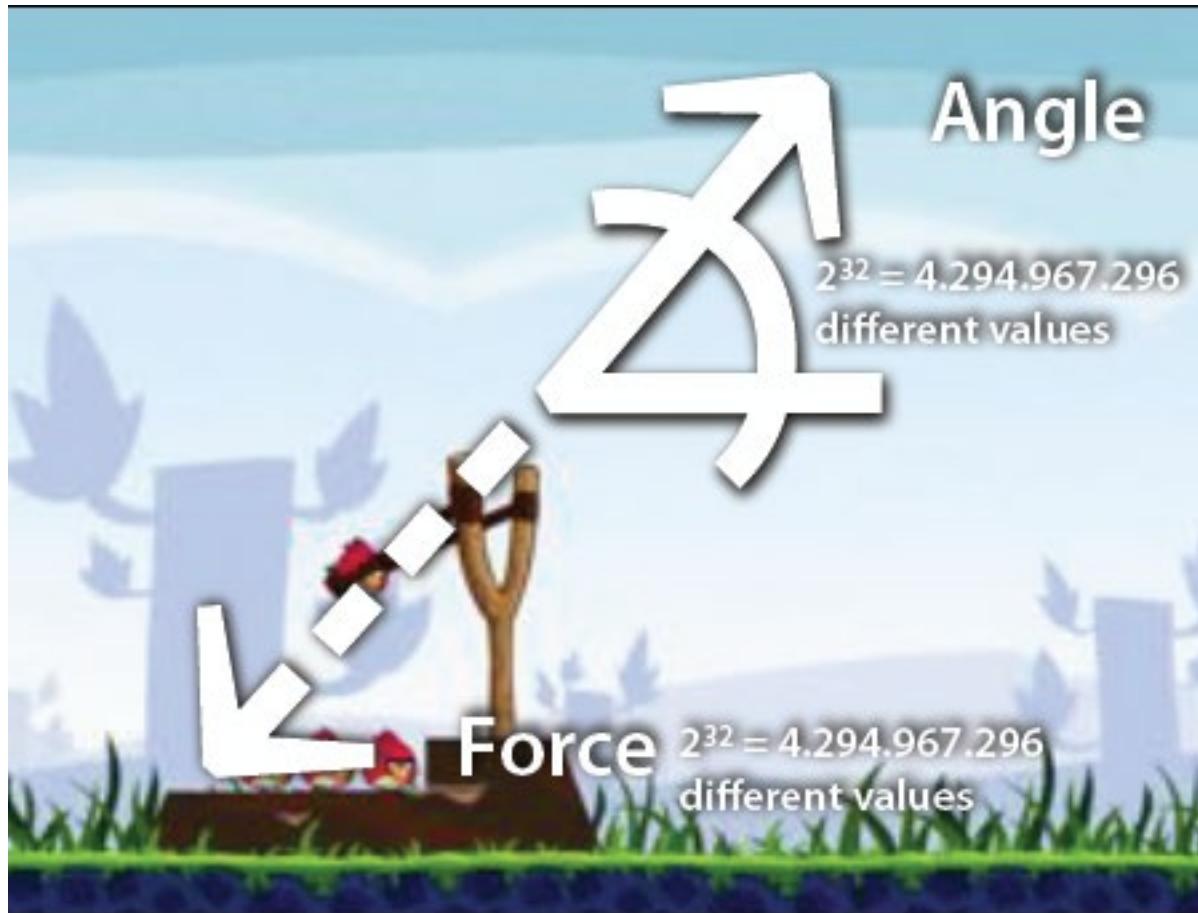
Better at finding crashes, out-of-bound errors, file handling errors etc.

Black Box Testing



*We see the program as a black box, we ignore how it is being written
Write tests from specification*

Specifications



*There will be a winner if hit with the right force and right velocity
Specifications should be unambiguous*

JavaDoc

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

Method Summary

char	<u>charAt(int index)</u> Returns the char value at the specified index.
int	<u>codePointAt(int index)</u> Returns the character (Unicode code point) at the specified index.
int	<u>codePointBefore(int index)</u> Returns the character (Unicode code point) before the specified index.
int	<u>codePointCount(int beginIndex, int endIndex)</u> 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) {...}
```

}

*test if balance can be < 0 using the function
withdraw()
- withdraw more than the balance*

*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) {
```

*deliverText(null)
pass in null and see what's the
result, is it null?*

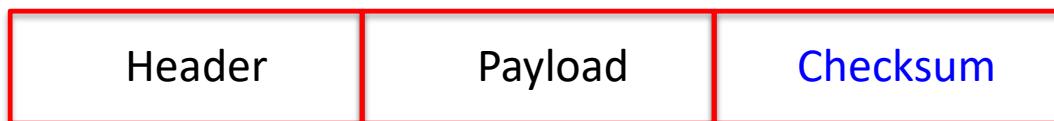
```
.....  
}
```



What kind of test would you design from this documentation?

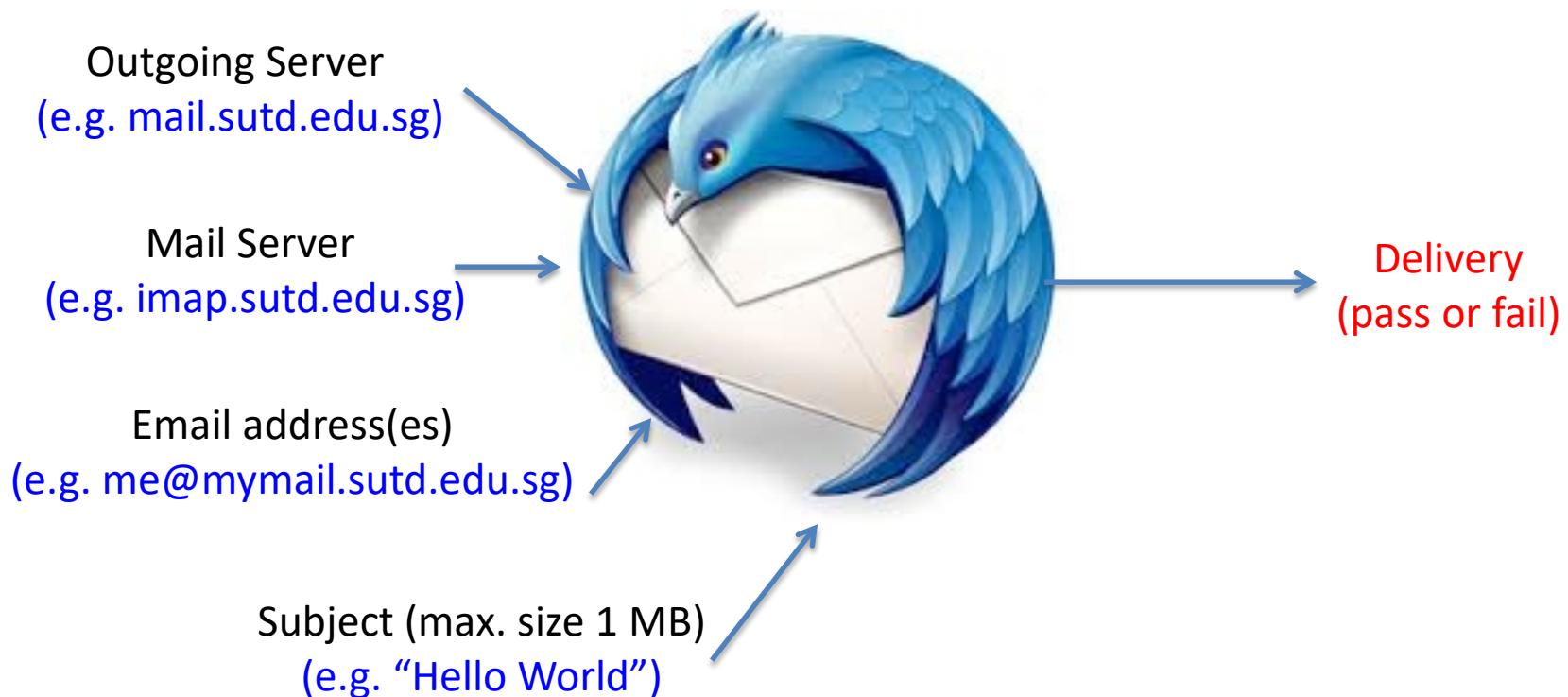
High-level specification

- 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://\\\$##^:^/\\](Whatever://\$##^:^/\\)
- Internet server
 - e.g. Message format



Exercise 6

Write down 6 test cases for an email client. Write one test for checking successful delivery and five different tests with the intention to make the delivery fail.



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.

Important Consideration for Black Box Test Planning (cont.)

- Run the **simplest** test case first – or you won’t know if the test case failed because basic functionality doesn’t work or if the hard stuff doesn’t work.
 - *Check whether the email client can send (receive) emails to (from) valid email addresses*

Important Consideration for Black Box Test Planning (cont.)

- Plan the **simplest** test to test the failure that you are looking for. If the test case fails, you don't want to have to work ***too*** hard to re-run the test when the code is (supposedly) fixed.
 - <http://sudiptac.bitbucket.io>
 - Whatever://sudiptac.bitbucket.io
 - ^^^O:/\\\$##*&*&\$%^☺/\\

BlackBox/Functional Test Diagram:



1. Check use-case diagram for every use case, there must be at least one test.
2. Every test should relate to some use case (traceability)
3. For every use case, find implicit space for the respective tests & perform equivalence class partitions
4. For each equivalence class, find middle boundary values.

Equivalence Class Partitioning

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



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



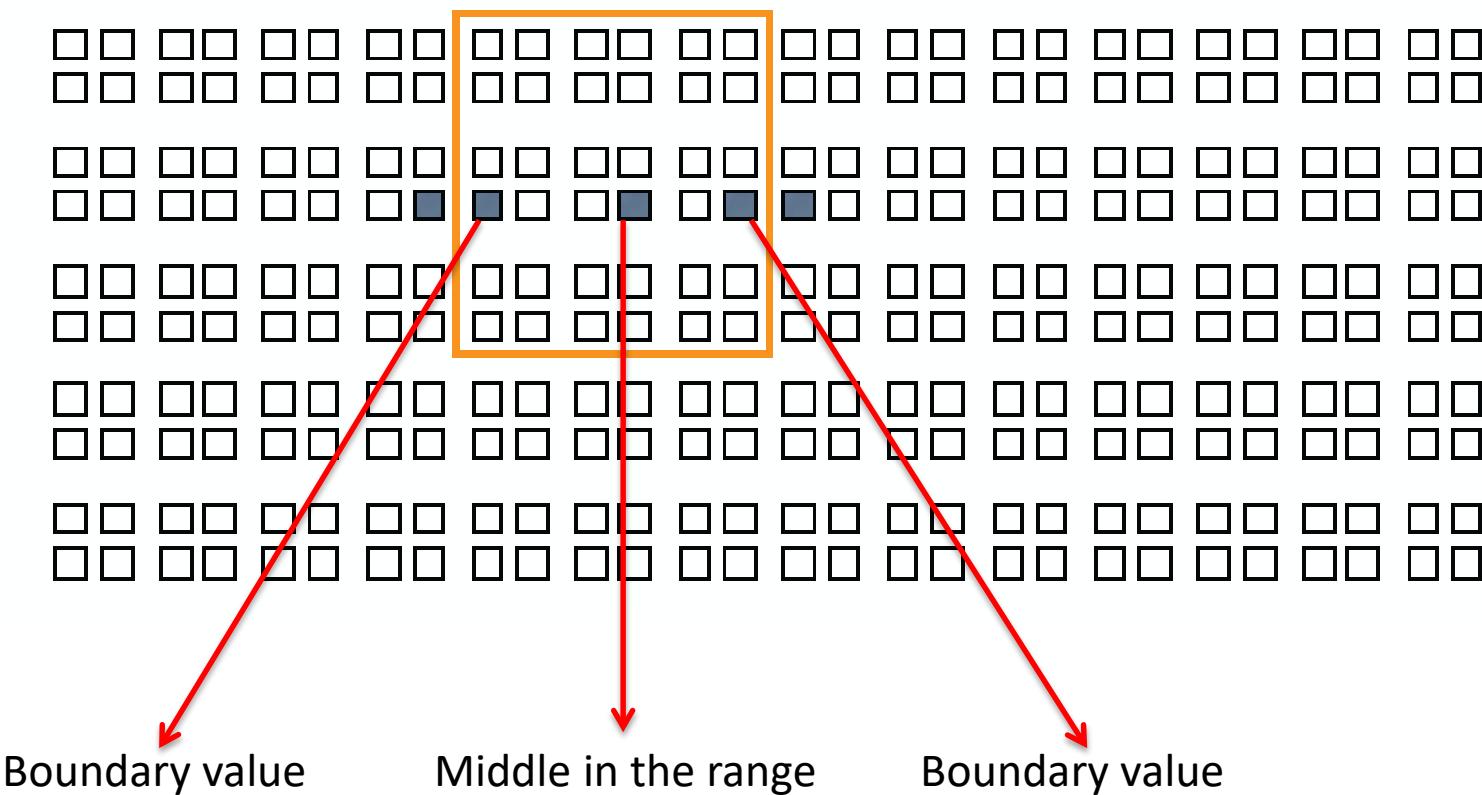
Input list: {0,1,7,89, 111}; Output List = {111, 89, 7, 1, 0}

How do you design the partitions?

Boundary value analysis

Possible test case

Partition



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](mailto:$%6**@I.do.not.know)

(middle value)

- su@@sutd.edu.sg

(only one extra @, **boundary**)

Dirty/Failure Test Cases

- Can something cause division by zero?
- What if the input type is wrong
 - *Expected Integer, Test = 2.345*
- What if the customer takes an illogical path through your functionality?
 - *Enter invalid email address or URL*
- What if mandatory fields are not entered?
 - *Do not enter recipient's email to send email*
- What if the program is aborted abruptly or input or output devices are unplugged?
 - *Quit a game suddenly in the middle, does it save the state?*

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 method
 - Every statement
 - Every branch
 - Every path
 - Every condition
 - Every loop
- Measures the quality of tests

Every Method

- For each method, there must be one test case which executes it.

BiSectionExample.java; BiSectionTest.java

Every Loop

- For every loop, there is a test case which executes the loop zero time, 1 time, etc., if feasible.

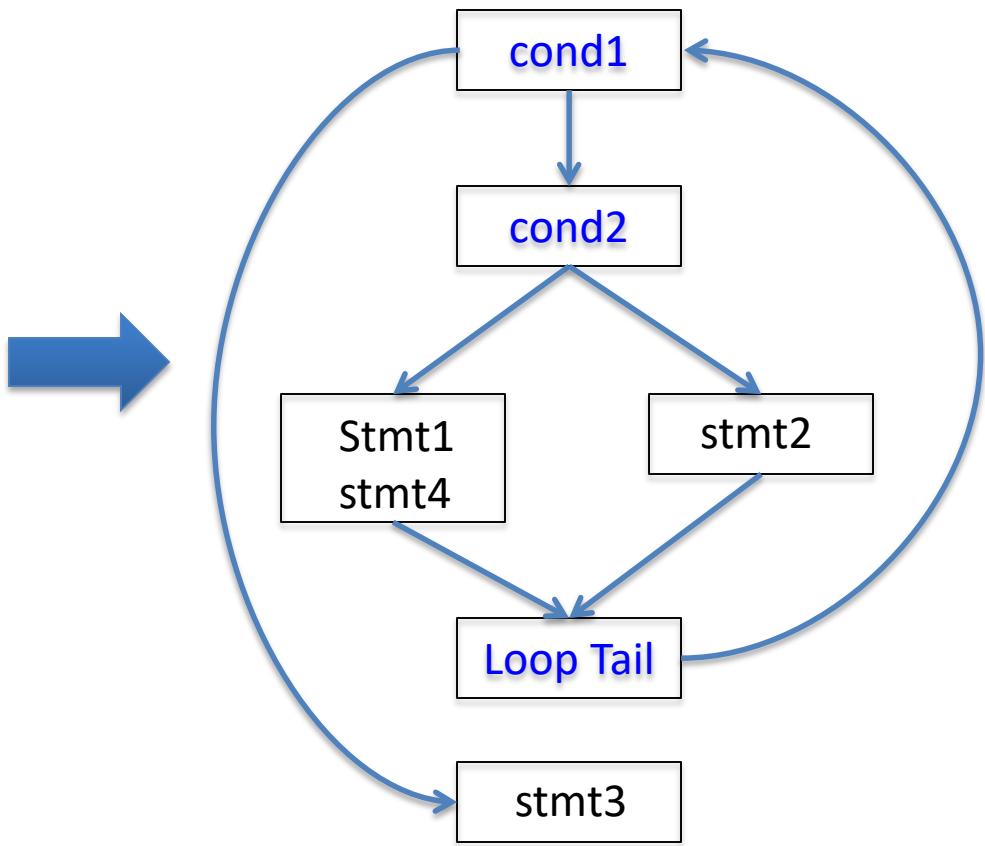
BiSectionExample.java; BiSectionTest.java

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)

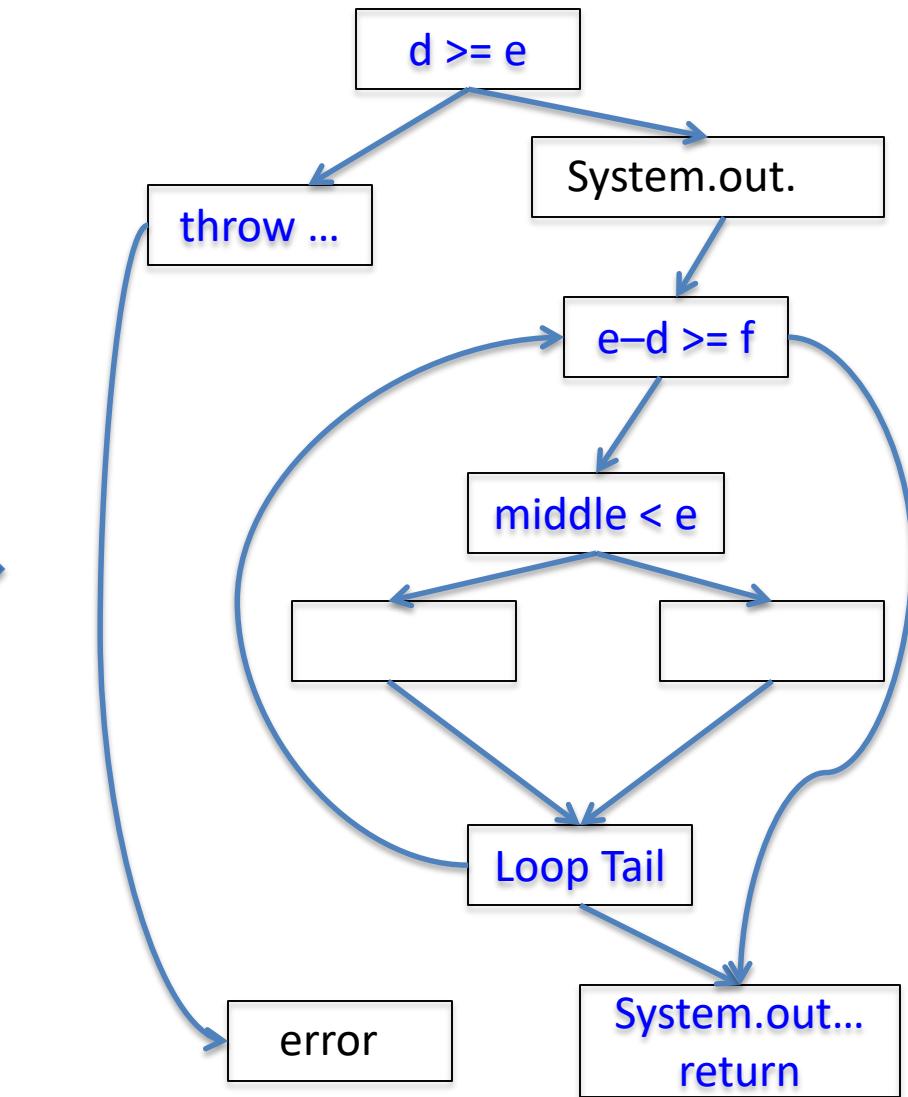
Control Flow Graph

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



Control Flow Graph

BiSectionExample.root()



Exercise 7

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

Disk.java

Every Statement

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

Exercise 8

- Write a set of tests to cover each statement 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, there must be a test case which executes it.



if the branch can be executed, if not then nvm

Every Branch

`BiSectionExample.root()`



A test where $d \geq e$

A test where $d < e$

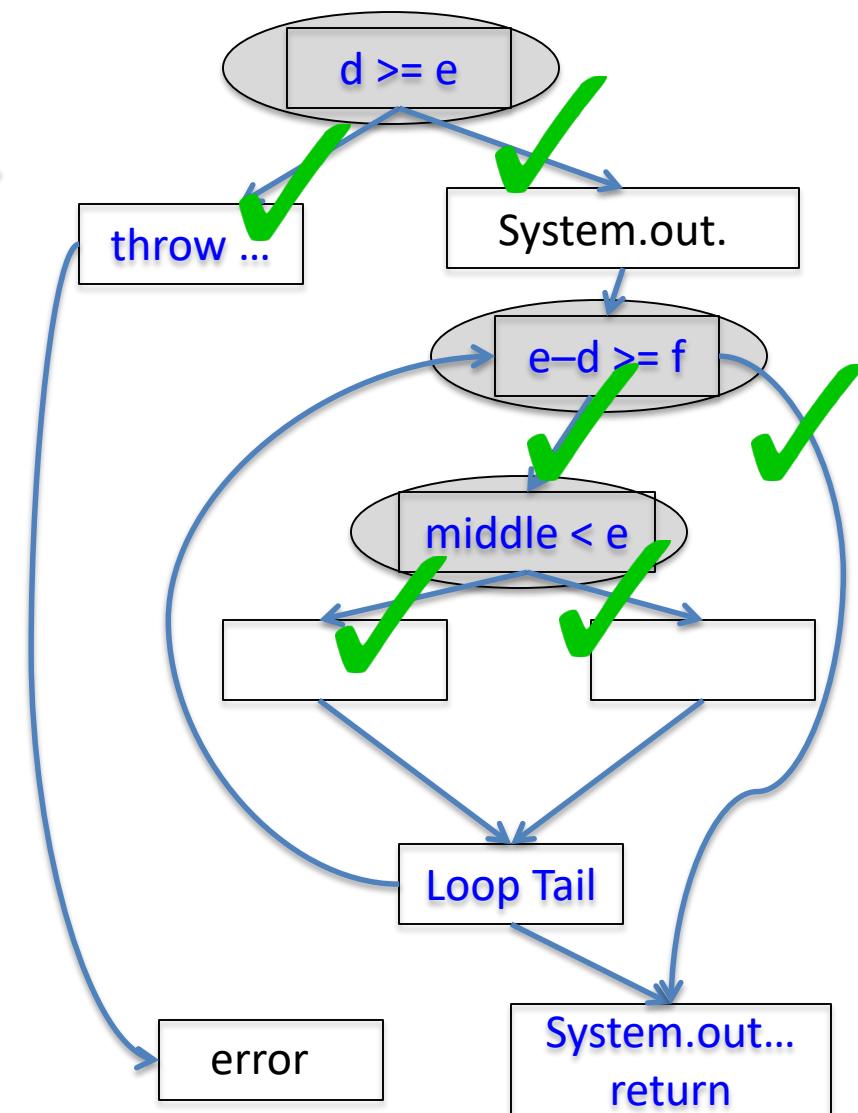
A test where $e-d \geq f$

A test where $e-d < f$

A test where $middle < e$

A test where $middle \geq e$

Note: A single test may cover multiple branches



Exercise 9

- Write a set of tests to cover each branch of the manipulate() function. 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)
 - ?

*100% condition coverage
but not 100% branch
coverage*

If the tests have complete branch coverage then we can say it also has complete statement coverage, but not the vice versa.

100% branch coverage => 100% statement coverage

100% statement coverage does not imply 100% branch coverage

Exercise 11

- 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

Every Path

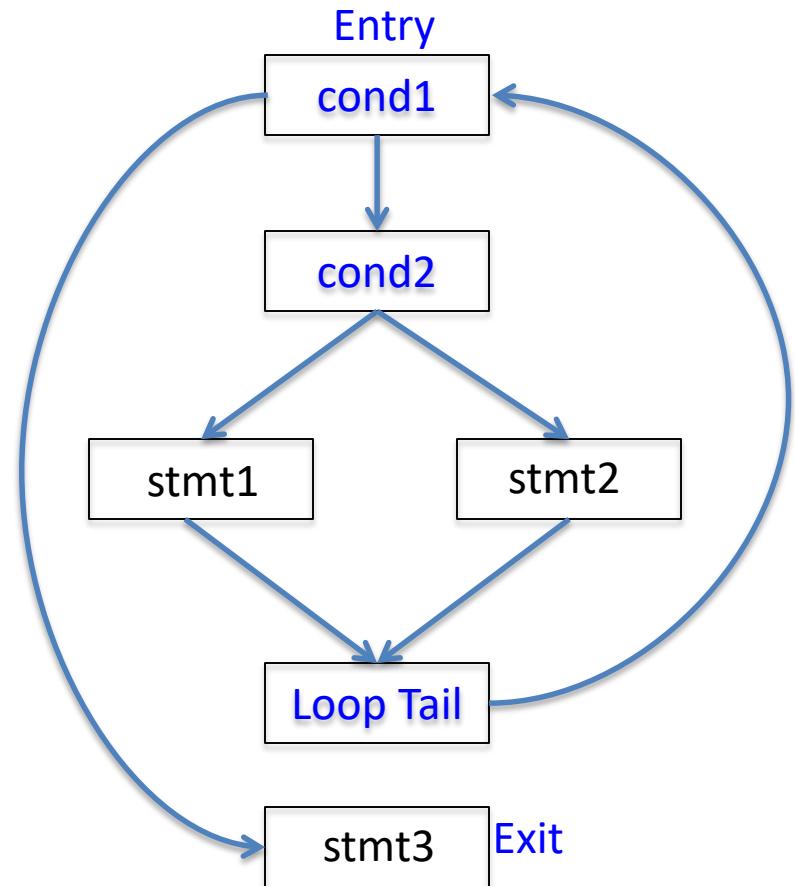
- A path is defined as a sequence of **executed** nodes in the control flow graph between the entry node of the graph and the exit node

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

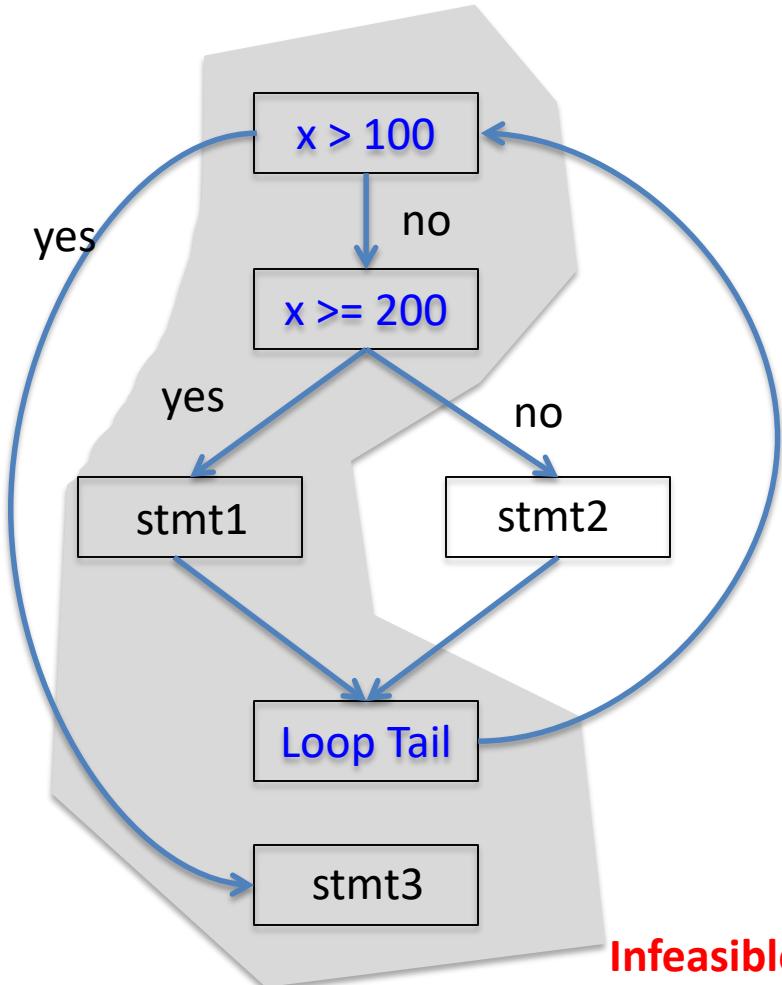
cond1->cond2->stmt1->loop tail->cond1->cond2->stmt2->loop tail->cond1->stmt3 is also a path

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

*How many paths in total?
(assuming the loop is executed exactly 100 times)*



Every Path



Remember some paths may be infeasible

How many paths in total?

Infeasible

Exercise 10

- Assume that the loop in the manipulate() function executes **at most** 100 times. How many paths are there in the manipulate() function? Explain your answer.

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

Devising a Prudent Set of Test Cases

- Equivalence Class/Boundary Value Analysis
 - Still applies!
- A metric for assessing how good your test suite is.
 - Method Coverage
 - Statement Coverage
 - Decision/Branch Coverage
 - Condition Coverage

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

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;  
}
```

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.

try to find a condition that will not terminate-> fault



DiskFaultTest.java

Exercise 13

Given method multiply Russian.java,

- Create a test suite for black-box testing.
- Create a test suite for white-box testing
 - With 100% branch coverage
- Create a test suite for fault-based testing

Russian.java; RussianTest.java