50.003: Elements of Software Construction

Week 3 and Week 4

Software Testing

(with some slides from Darko Marinov and Andreas Zeller)

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

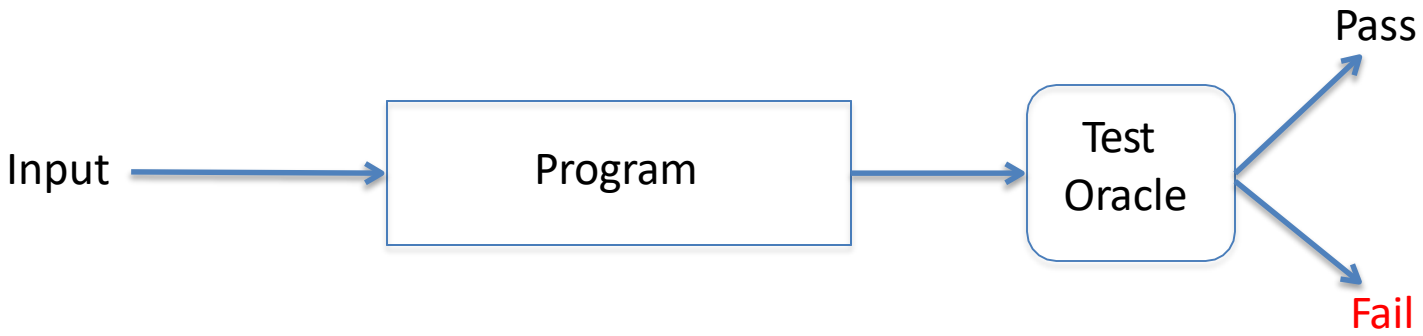
Computer Scientist

ACM Fellow, 2010

To show the presence of bugs: Use testing

BASICS OF TESTING AND JUNIT

What is a test?



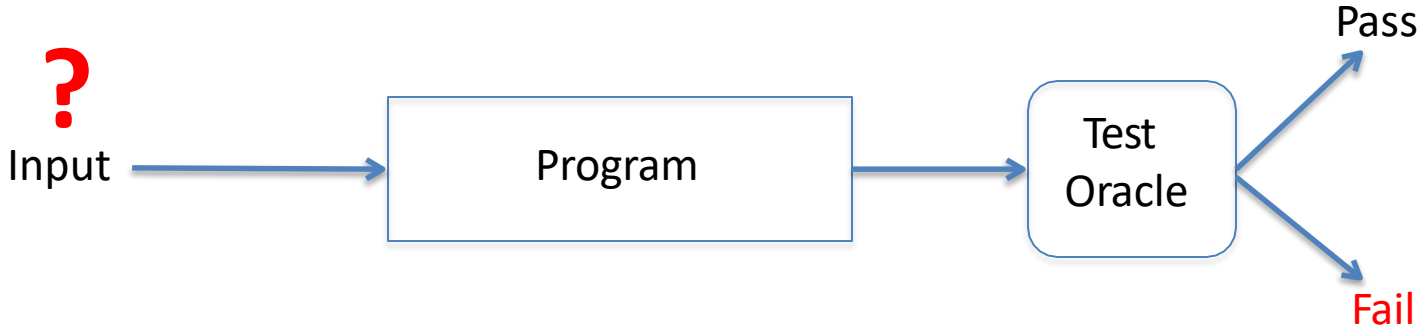
Failures *are when your test cases fail - i.e. your assertions are incorrect.*
Errors *are unexpected errors that occur while trying to actually run the test - exceptions*

Summary

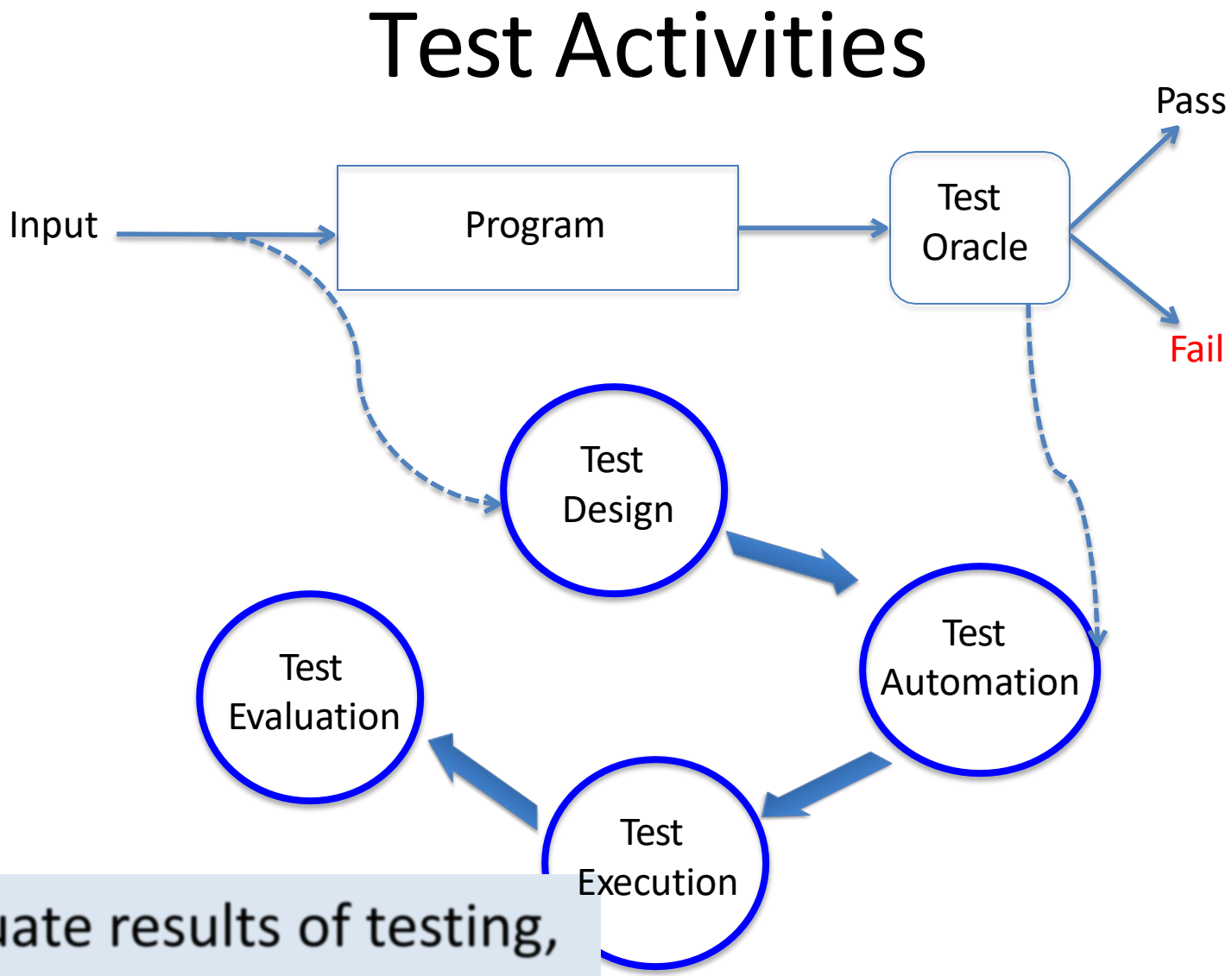
- The only way to make testing efficient as well as effective is to automate as much as possible
- JUnit provides a very simple way to automate our unit tests
- It is no "silver bullet" however … it does not solve the hard problem of testing :

What test values to use ?

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



TEST DESIGN

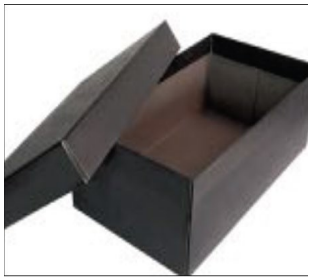


Evaluate results of testing, report to developers

Black Box Testing

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

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

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

Write tests for different Angles and Force (Test Design)
Observe behavior and spot potential bugs (Test Evaluate)

Infinite Monkey Theorem

"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".

Angle -> 2^{32} possible values
Force -> 2^{32} possible values
Assuming both Angle and Force are integers
Total possibilities = 2^{64} = 18, 446, 744, 073, 709, 551, 616 diff. executions

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, ignore how it is being written

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.

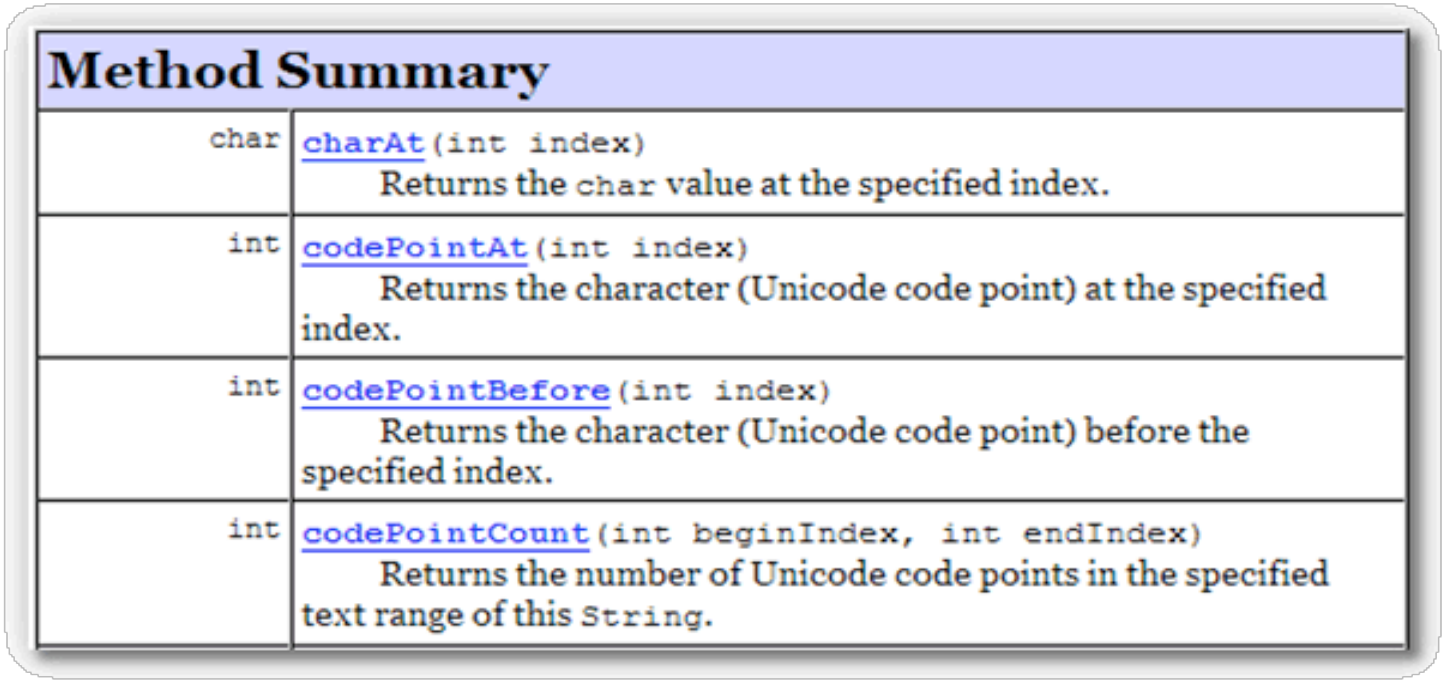
Specifications

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	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) {….}


}
```

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 whats the result, is it null?

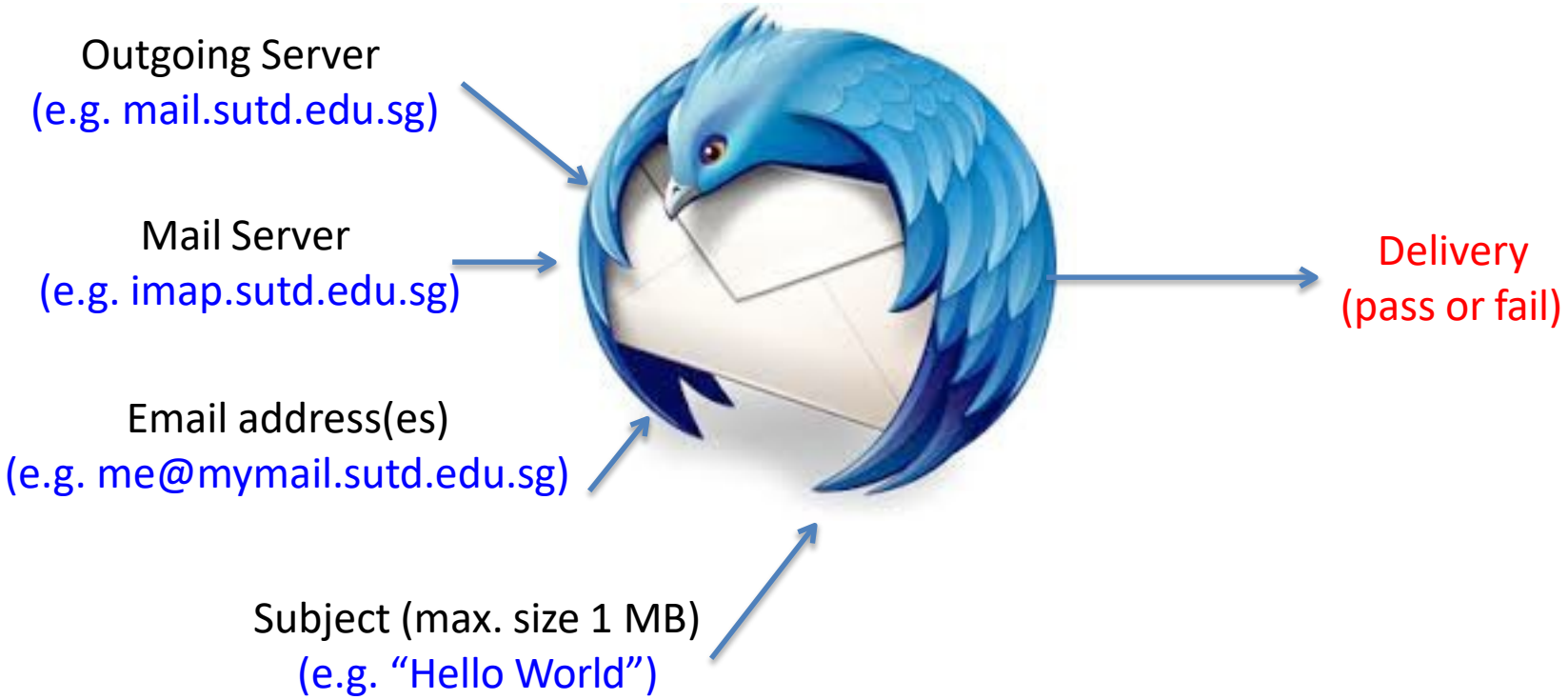
What kind of test would you design from this documentation?

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://\\$##^^☺/\\
- 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.



Email Client

Email address(es)
(valid and invalid)

Delivery
(pass or fail)

Failure Test:

- Invalid email address
- Invalid SMTP server
- Invalid IMAP server
- Large Message
- Blank recipient

Successful Test:
- Sending a valid email

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 ckass 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.

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

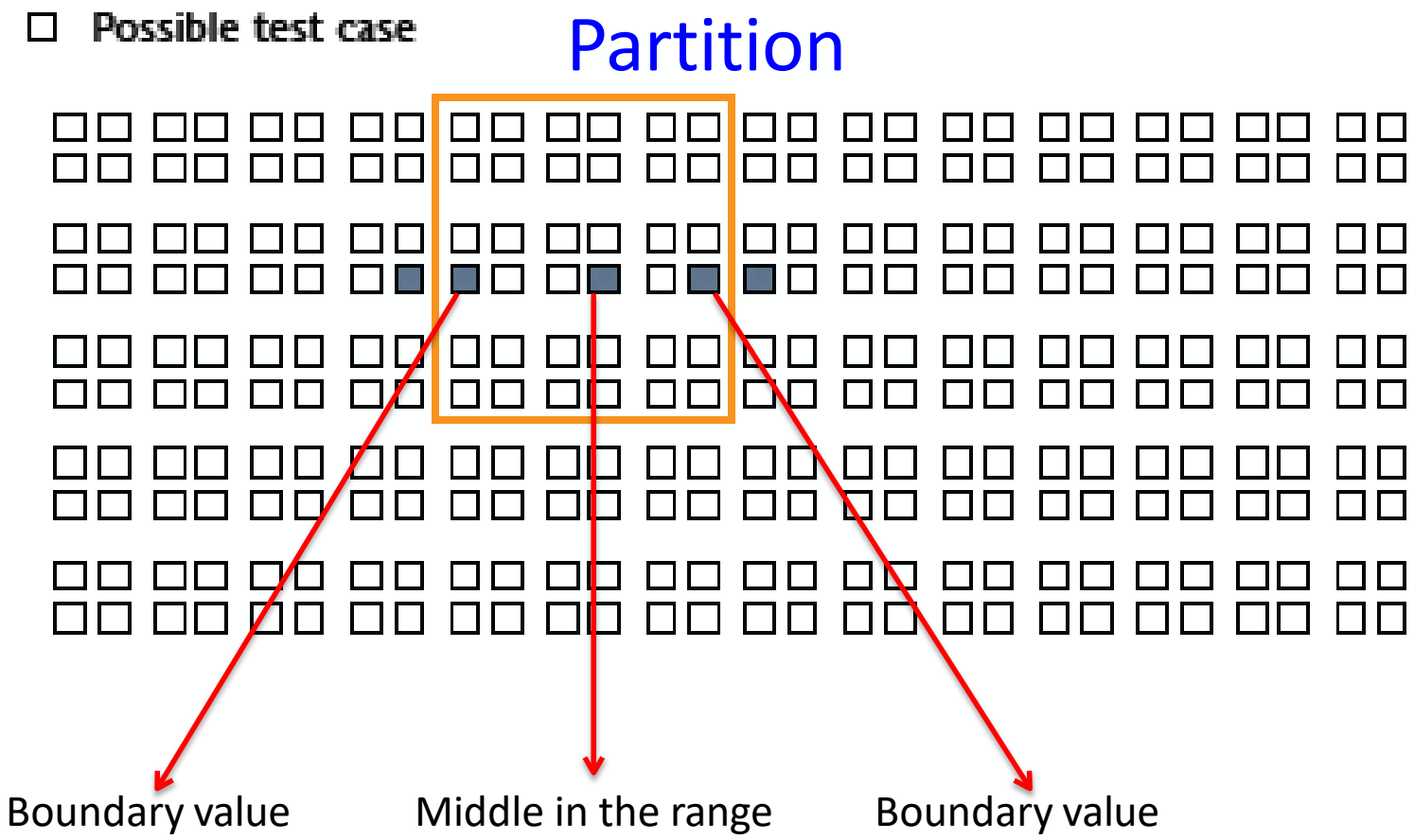
Sorted array: {0,5,7,89, 111}
Reversed array: {113, 907, 12, 1}
With negatives: {-1, -5, -6, -1111}
With duplicates: {4,6,6,6,6, 121212, 34}

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)

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.

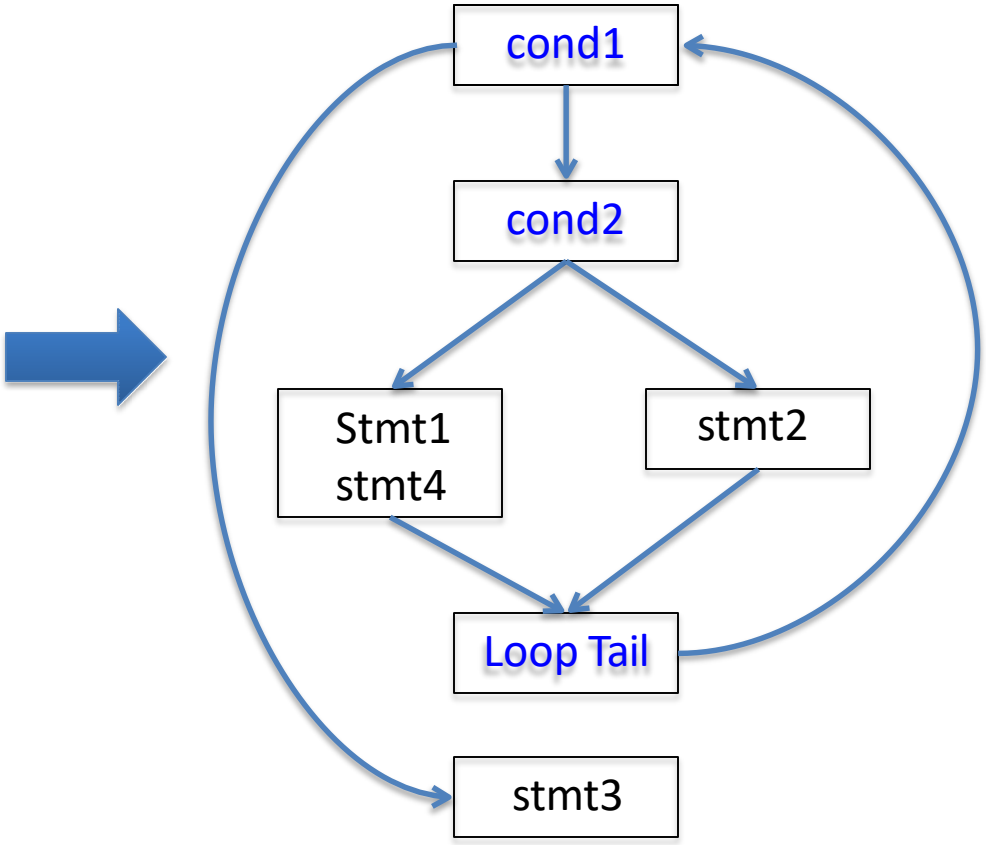
BiSectionExample.java; BiSectionTest.java

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)

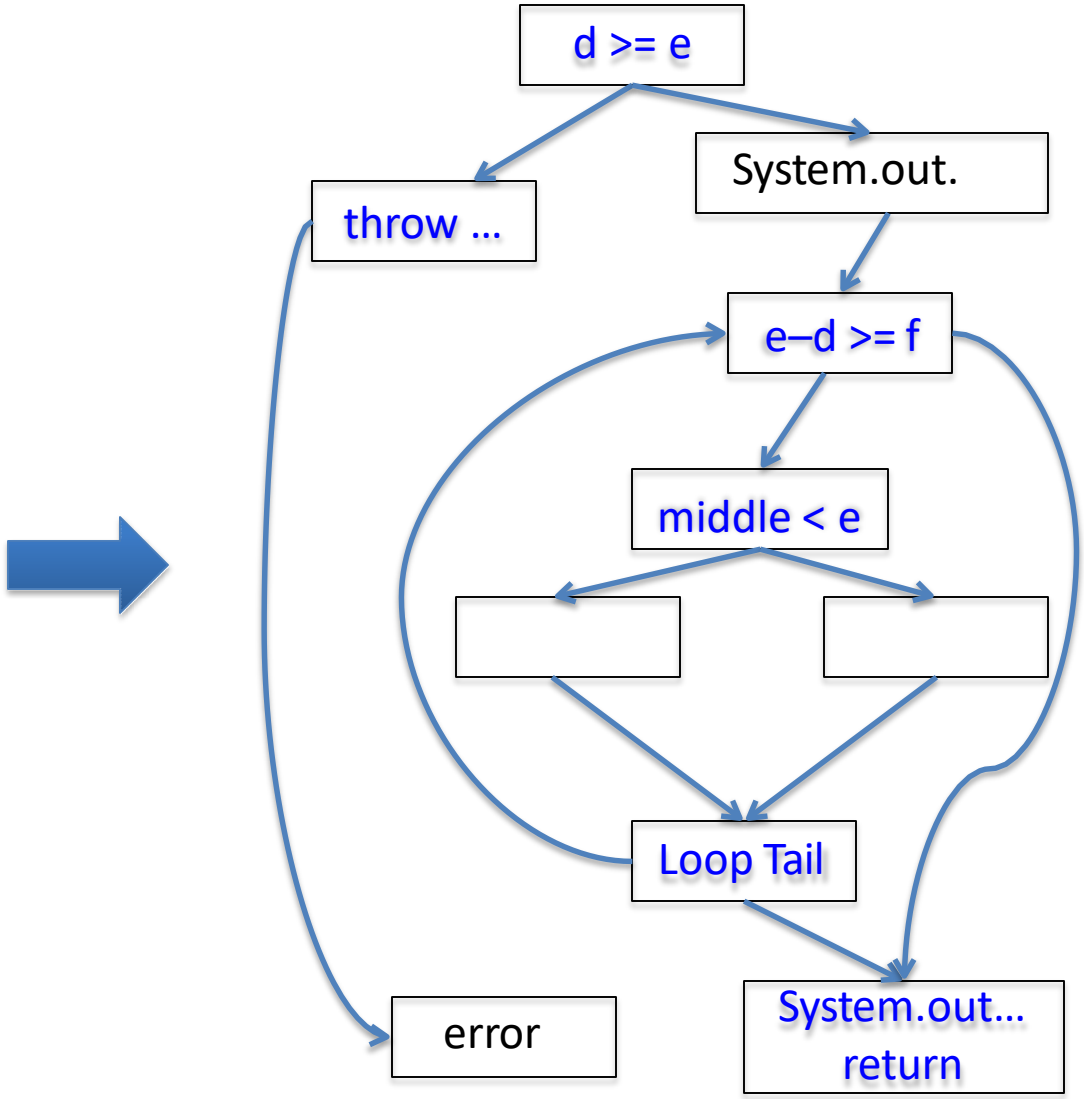
Control Flow Graph

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



Control Flow Graph

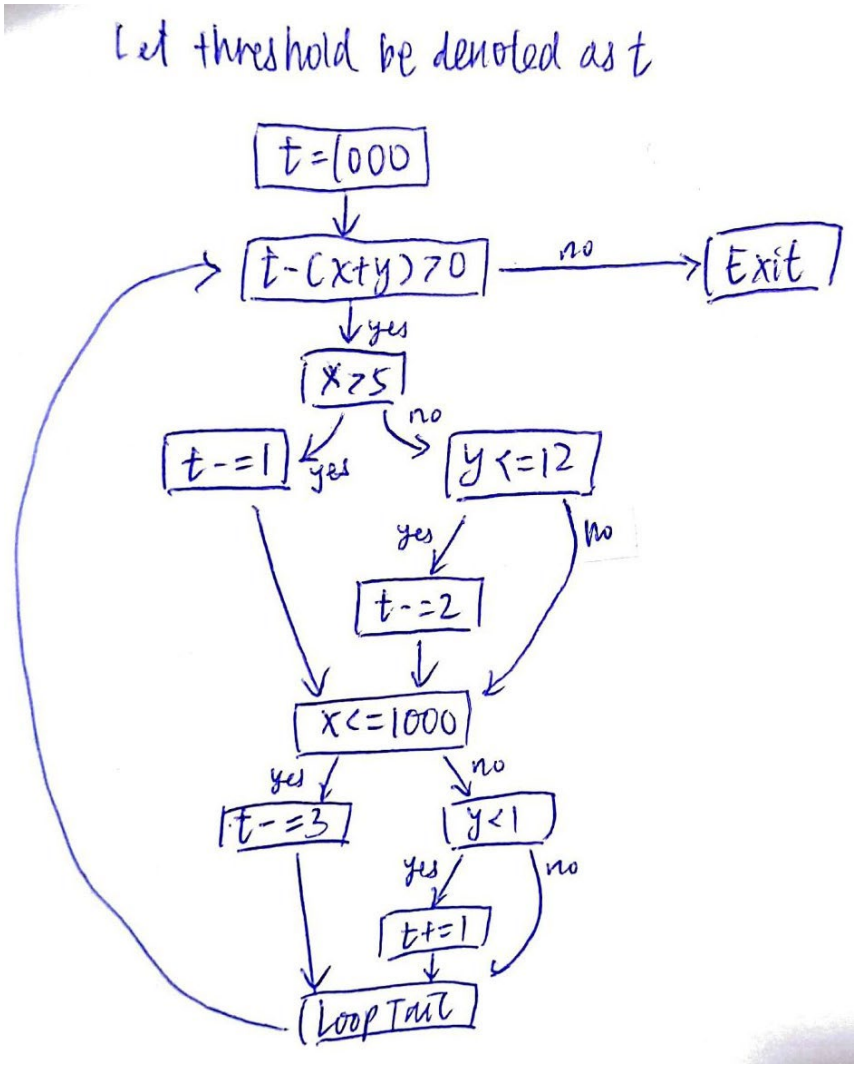
```
BiSectionExample.root()
```



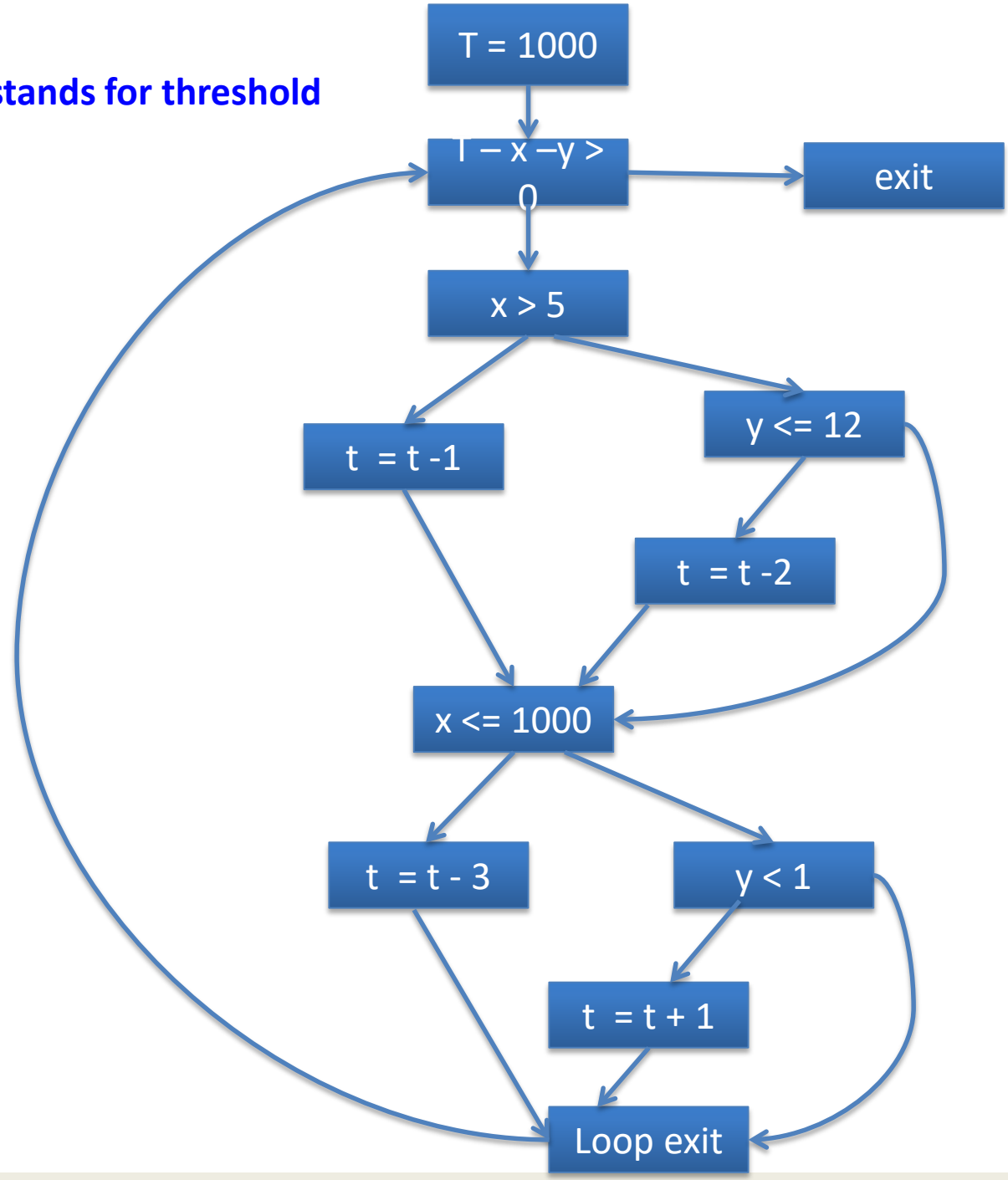
Exercise 7

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

```
public void manipulate () {
    int threshold = 1000;
    while ((threshold - (x + y)) > 0) {
        if (x > 5) {
            threshold = threshold - 1;
        }
        else if (y <= 12) {
            threshold = threshold - 2;
        }
        if (x <= 1000) {
            threshold = threshold - 3;
        }
        else if (y < 1) {
            threshold = threshold + 1;
        }
}
```



Ans: T stands for threshold



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?

Test1 => {X = 1010, y = -20}
Test2 => {X = 5, y = -20}

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

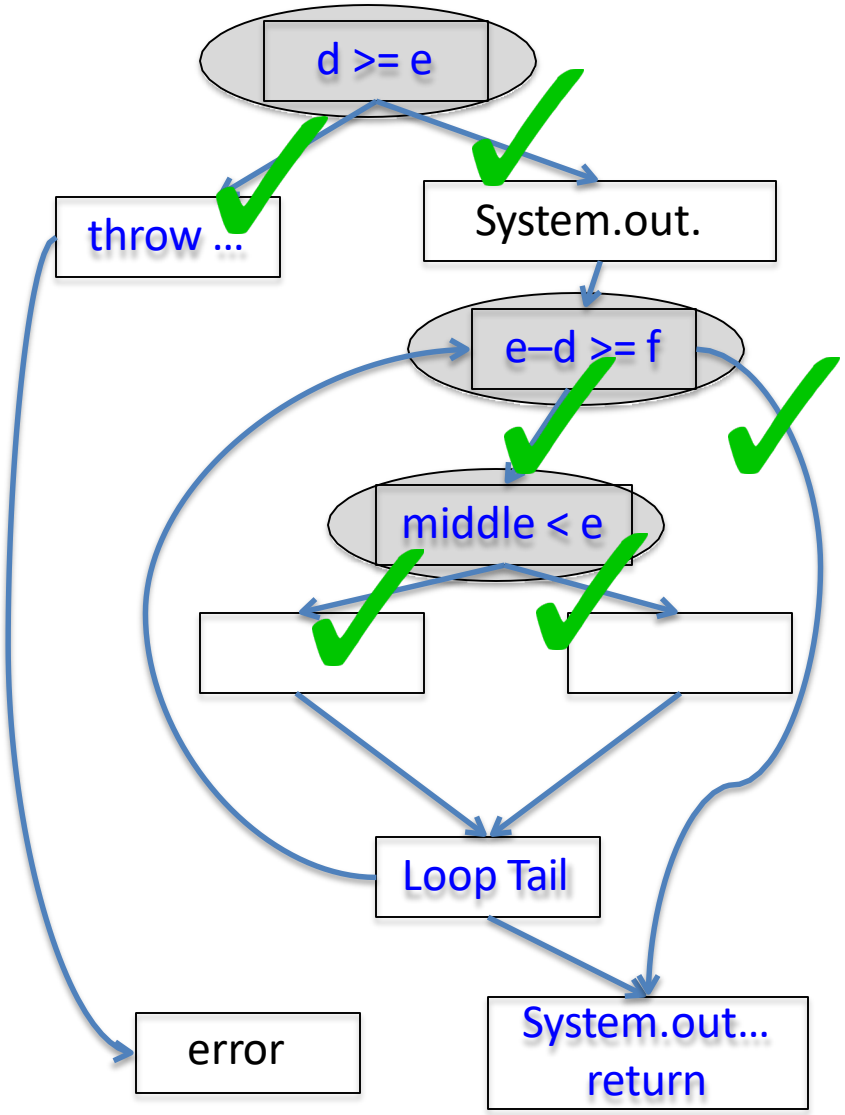
`BiSectionExample.root()`

A test where d >= e
A test where d < e

A test where e-d >= f
A test where e-d < f

A test where middle < e
A test where middle >= e

Note: A single test may cover multiple branches

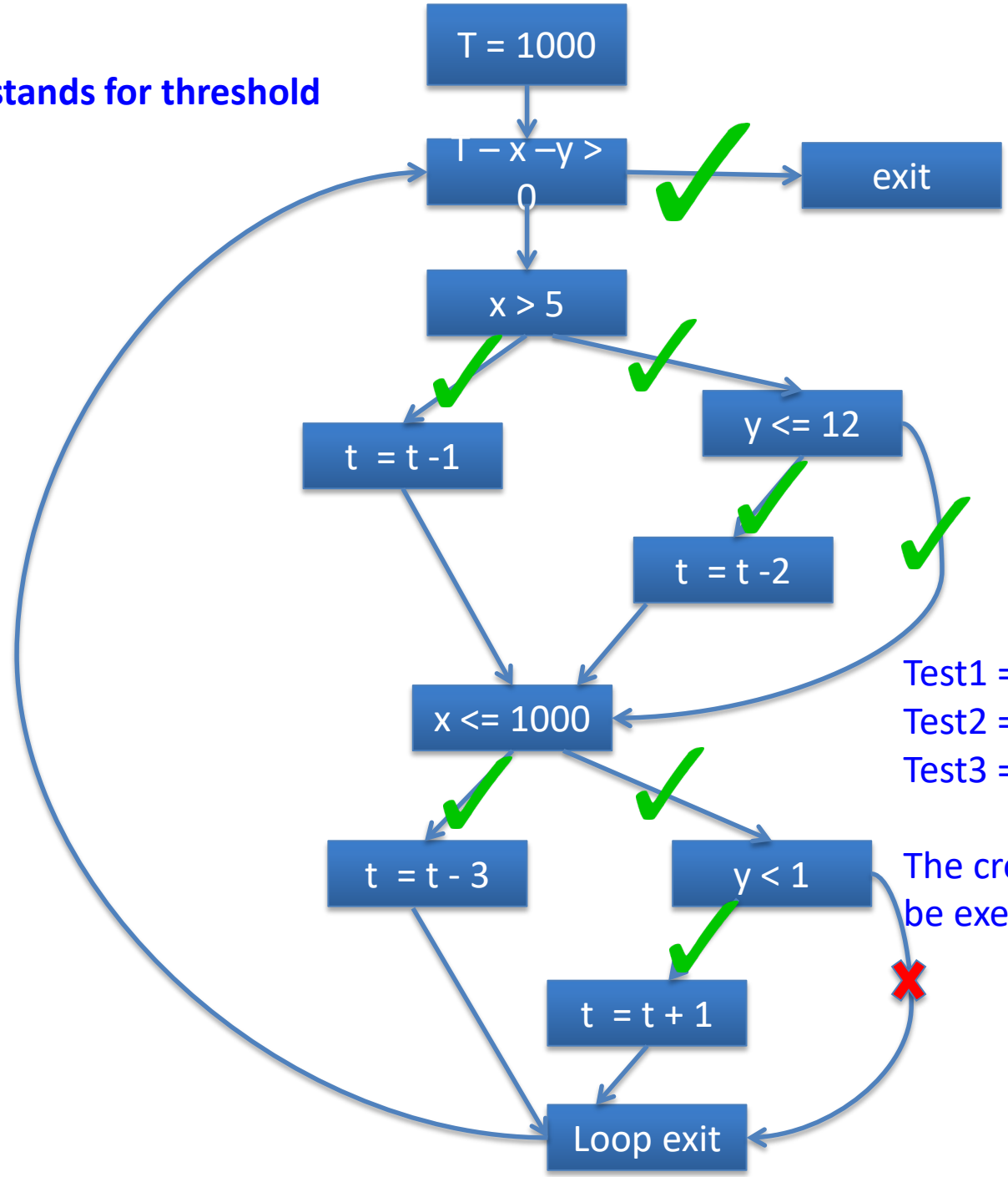


Exercise 9

- 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

Ans: T stands for threshold



Test1 => {X = 1010, y = -20}
Test2 => {X = 5, y = -20}
Test3 => {X = 5, y = 15}

The crossed branch can never be executed by any test.

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

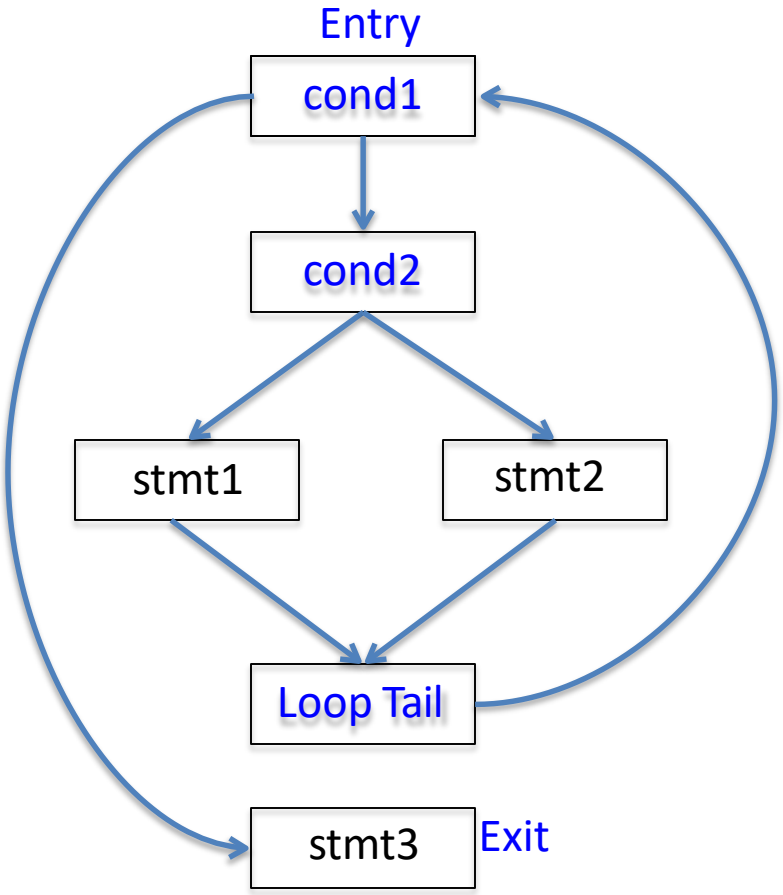
- Consider your test cases that obtain branch coverage in the manipulate() function. Argue whether the test suite also obtains the condition coverage.

Ans: Yes, it also obtains condition coverage, as all the branch conditions are atomic conditions in the manipulate() function.

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



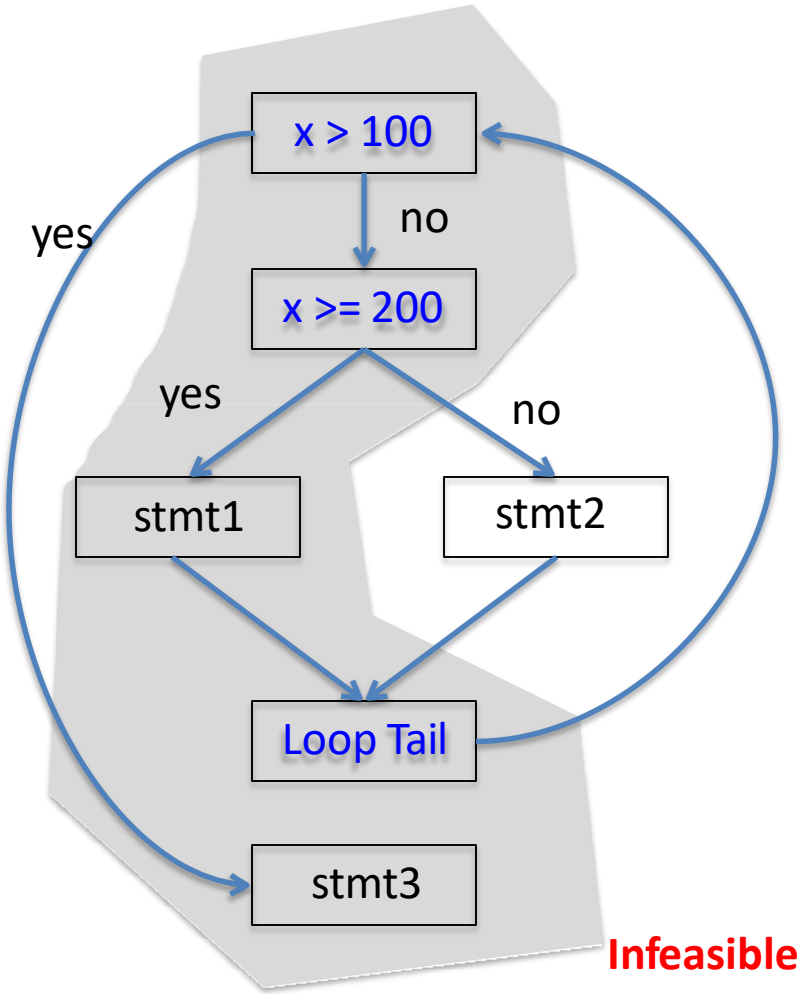
cond1->cond2->stmt1->loop tail->cond1->stmt3 is a path

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?

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.

Ans. 201 paths. There are three feasible paths inside the loop, but only two among them can terminate after at most 100 iterations. Since the loop can execute 0 to 100 iterations, the total number of paths is 201.

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

TestFault => {X = 1010, y = -20}

The changes to “t” cancels out
and the loop never terminates.

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