CPSC 310 – Software Engineering

Testing

http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf

http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf

Overview

- Introduction (Proving and Testing)
- Types of Testing
  - Unit, Integration, Regression, System, Acceptance
- Testing Tactics
  - Functional (Black Box), Structural (White Box)
- Stopping Criteria
  - Equivalence Class Partitioning, Boundary Tests, Coverage
- Who should test the software?

Learning Goals

By the end of this unit, you will be able to:

- Explain differences between **proving** and **testing**.

- Explain to a manager why testing is important and what its **limitations** are as well as **who** should be involved in testing.

- Be familiar with the **different kinds** of testing and choose which one(s) to use.

- Be able to generate test cases that provide **statement**, **branch** and **basis path coverage** for a given method.

Testing   vs. Proving

- Dynamic
    - Program is executed

- Builds confidence
    - Can only show the presence of bugs, not their absence!

- Used widely in practice

- Costly

- Static
    - Program is logically analyzed
- Theoretically could show absence of bugs
- Applicability is practically limited
- Extremely costly

Should be considered to be complementary, not competitive. Testing is by far more dominant approach to assess software products.

Proving example

```
static int someMethod(int x, int y) {
  if(x > y) {
      return x - y;
  } else if(x < y) {
      return y - x;
  } else {
      return x + y;
  }
}
```

Can this function ever return a negative number?

Who uses proofs?

Researchers prove kernel is secure

An Australian research organization says it has absolute mathematical proof of the security of an operating system core.

NFM
NASA FORMAL METHODS SYMPOSIUM

Why Test?

- Everyone should know about testing!
  - Developers test code
  - Testers test systems
  - Managers plan out the test procedures

- Industry averages
  - 1-25 errors per 1000 lines of delivered code
  - Microsoft
    - 10-20 errors/kloc internal, 0.5/kloc delivered
  - Space Shuttle
    - 0 errors for 500 kloc

- *So satellite rockets don't explode*
- *So hospital patients are not accidentally killed (Therac accident)*
- *So Apple Maps isn't terrible*
- *So credit card numbers don't get stolen*
- *So a development team has mutual trust and a fun time working together*

Testing Goals

- Verification: "Did we build the system right?"
  - Discover situations in which the behavior of the software is incorrect or undesirable

- Validation: "Did we build the right system?"
  - Demonstrate to the developer and the customer that the software meets its requirements

Testing Terminology

Testing: *Execute a program or program unit with a test set to determine if the expected output is produced*

Test Case: *A set of value assignments to each input parameter and expected values for each output*

Test Set: A set of test cases

Exhaustive testing: A test set which covers all possible inputs.
   Almost always infeasible

Exhaustive Testing

Metaphor: Looking for needle in haystack

- Searching the haystacks:
    - *Writing test cases*
    - *Each test case <u>covers</u> part of the program*

- Finding a needle
    - *Finding a bug*
    - *Showing the presence of an error*

- Searching every inch of the haystacks
    - *Exhaustive testing*
    - *Showing the absence of an error*

Exhaustive Testing

boolean and(boolean x, boolean y) {

…

}

Can we exhaustively test this function?

Exhaustive Testing

```
int hypot(int x, int y)  {
      ...
}
```
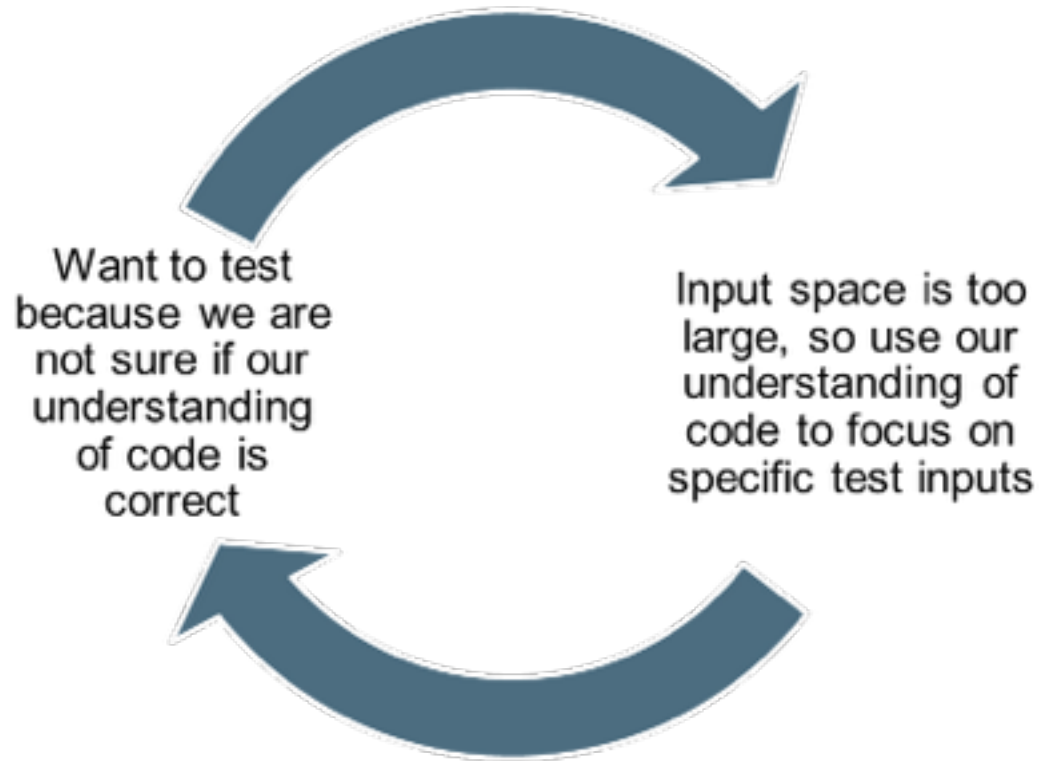
Can we exhaustively test this function?

Exhaustive Testing

int[ ] sort(int[ ] arr) {

    ...

}

Can we exhaustively test this function?

Exhaustive Testing

```
int[ ] sort(int[ ] arr)  {
        ...
}
```

Can we exhaustively test this function?

arr.length	number of inputs
0	1
1	4,294,967,296
2	18,446,744,073,709,551,616
3	79,228,162,514,264,337,593,543,950,336

Dijkstra's Law

"Testing can show the *presence*, but not the *absence* of errors"

Testing can help nonetheless!

Testing is challenging

Want to test because we are not sure if our understanding of code is correct

Input space is too large, so use our understanding of code to focus on specific test inputs

Testing is challenging

"No Silver Bullet" (Brooks '86):

must rely on combination of logical reasoning, math, experience, and creativity

Types of Testing

- Unit
- Integration
- Regression
  - Continuous integration
- System
- Acceptance

Unit Testing – Definition

- A unit is the smallest testable part of an application.

- Units may be:
    - functions / methods
    - classes
    - composite components with defined interfaces used to access their functionality

- Who does Unit Testing?
    - Software developers (i.e. software engineers, programmers)

Unit Tests

- Uncover errors at module boundaries
- Frequently programmatic

- It is important to keep specification up to date with implementation

- Developers often forget to update specification based on test results

Unit specs

Unit Test
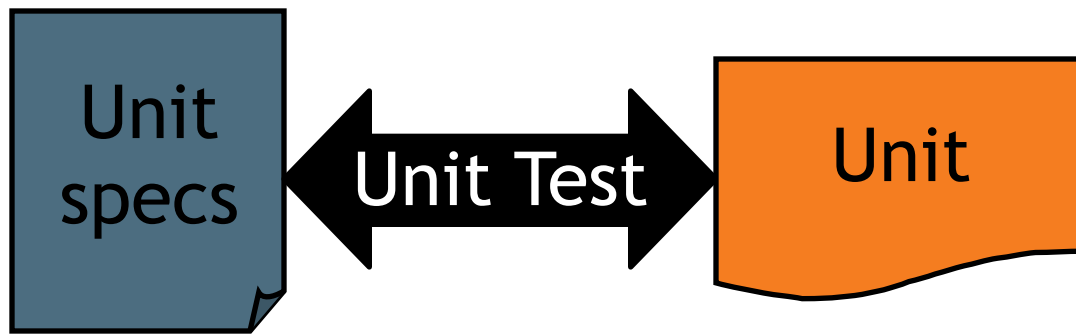
Unit

JavaDoc or
other specification

Integration Testing – Definition

- Individual units are combined and tested as a group.

- Shows that major subsystems that make up the project work well together.

- Phase of development where "Mock objects" are replaced by real implementation

- Who does Integration Testing?
  - Software development teams

Mock Objects

- Often we rely on portions of the system which are
  - non-deterministic
  - slow
  - user-driven
  - not yet built

- Mocks adhere to the contract (interface) but simulate behavior

- Real implementation is substituted in production

Mock example

```
public class MockWeatherService implements WeatherService {
        public double getCurrentTemp(String city) {
                return Math.random() * 32;
        }
}
```

- One idea here is that software developers usually should not throw up their hands and say:
  "I can't test my code until others finish their part of the system"

Regression Testing

- Testing the system to check that changes have not 'broken' previously working code

- Not new tests, but repetition of existing tests

- JUnit makes it easy to re-run all existing tests

- This is why it is so important not to write "throwaway tests"

- Who does Regression Testing?
  - Software developers (i.e. software engineers, programmers)

Continuous integration

- Everyone commits to mainline (a designated branch) every day.

- Every commit/merge results in an automated sequence of events:

  - run unit tests, run integration tests, run performance tests

- Who does continuous integration?
  - Software developers
  - and Hudson!
  - http://hudson-ci.org/

Hudson

System Testing – Definition

- Testing large parts of the system, or the whole system, for functionality.

- Test the system in different configurations.

- Who does System testing?
    - Test engineers
    - Quality assurance engineers

System Tests

- Recovery testing
  - Forces the software to fail in various ways and verifies that recovery is properly performed.

- Security testing
  - Verifies that the protection mechanism will protect the software against improper penetration

- Stress testing
  - Executes the system in a manner that demands resources in abnormal quantity, frequency or volume

System Tests

- Example system testing tools

  - Selenium
    - Web application user interface testing
    - Record and replay web user actions

  - New Relic
    - Web application server monitoring
    - https://www.youtube.com/watch?v=hDxI1jEOVzc

Acceptance Testing – Definition

- Formal testing with respect to user needs and requirements conducted to determine whether or not the software satisfies the <u>acceptance criteria</u>.

- Does the end product solve the problem it was intended to solve?

- Who does acceptance testing?
    - Test engineers
    - and customers

Acceptance Tests

- Typically incremental
  - Alpha test : At production site
  - Beta test : At user's site

- Work is over when acceptance testing is done

Actual needs and constraints

Acceptance Test

Delivered package

Testing Tactics

Functional
"black box"

Structural
"white box"
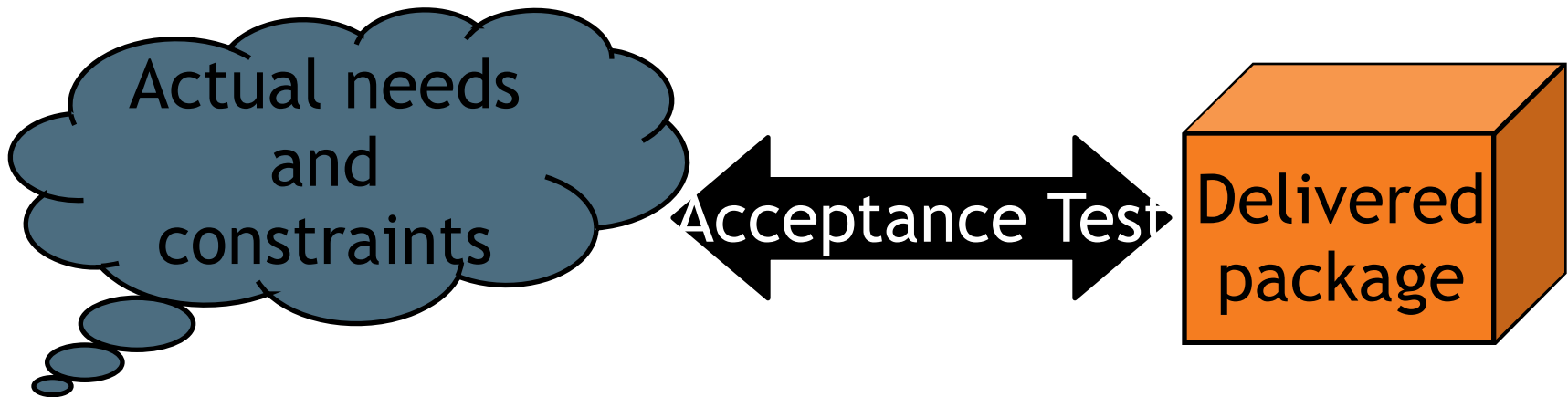
- Tests based on *spec*
- Treats system as atomic
- Covers as much *specified* behaviour as possible

- Tests based on *code*
- Examines system internals
- Covers as much *implemented* behaviour as possible

Recall Flow Graphs from 210
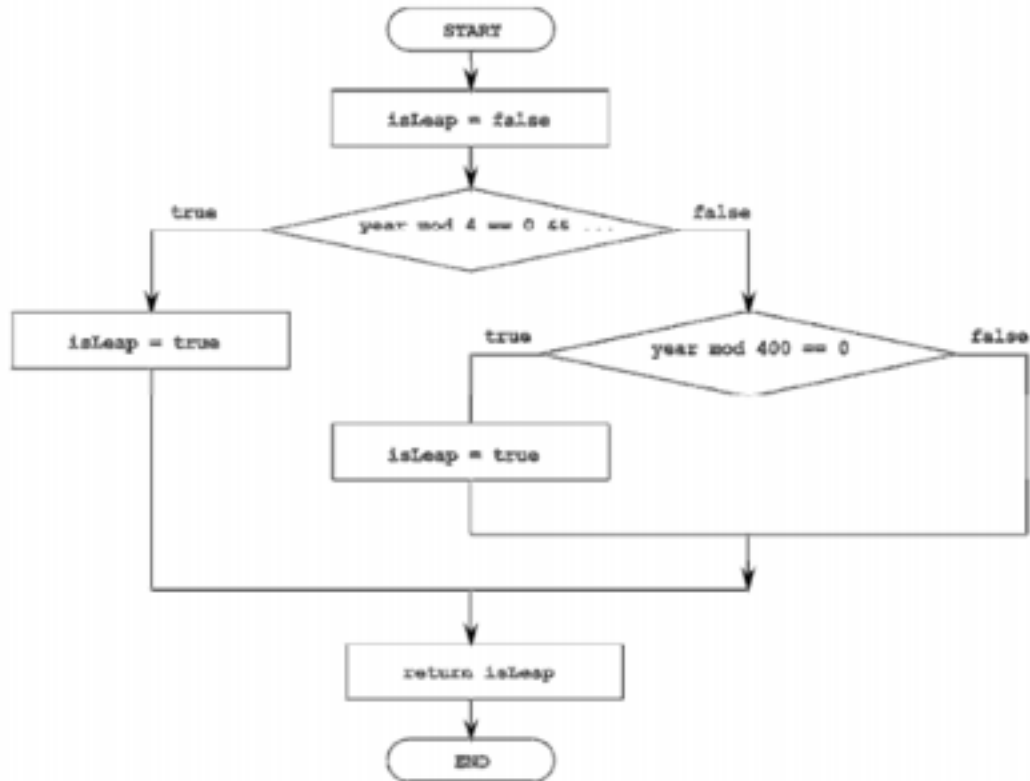
Figure 2 Flowchart for isALeapYear method

White-Box testing and Control-flow Graph

Stopping criteria based on a model of the control-flow of the program

Node: represents a statement or an expression from a complex statement
 e.g. a complex statement could be a `for` *loop which includes initializer, condition, and increment expressions*

Edge (i,j): represents transfer of control from node **i** to node **j**
 i.e. node **i** might execute immediately before node **j**

We consider single method CFGs in this course

There is no standardized syntax for drawing CFGs.
 –You will see many different syntax on the web
 –Not a design tool like UML
 –Different tools will draw them differently
 –Many tools use them internally without drawing them

Statements

```
a = b
```
↓
```
x = x + 1
```
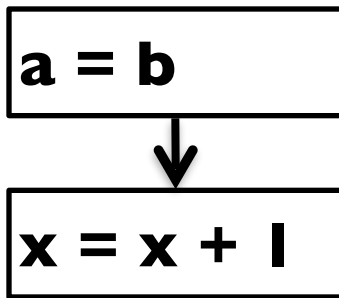
```
if ( a > b ) {
    a = b;
    x = x + 1;
}

print(x);
```

Conditional (`if`)

Conditionals have outgoing arcs labeled `true` or `false`
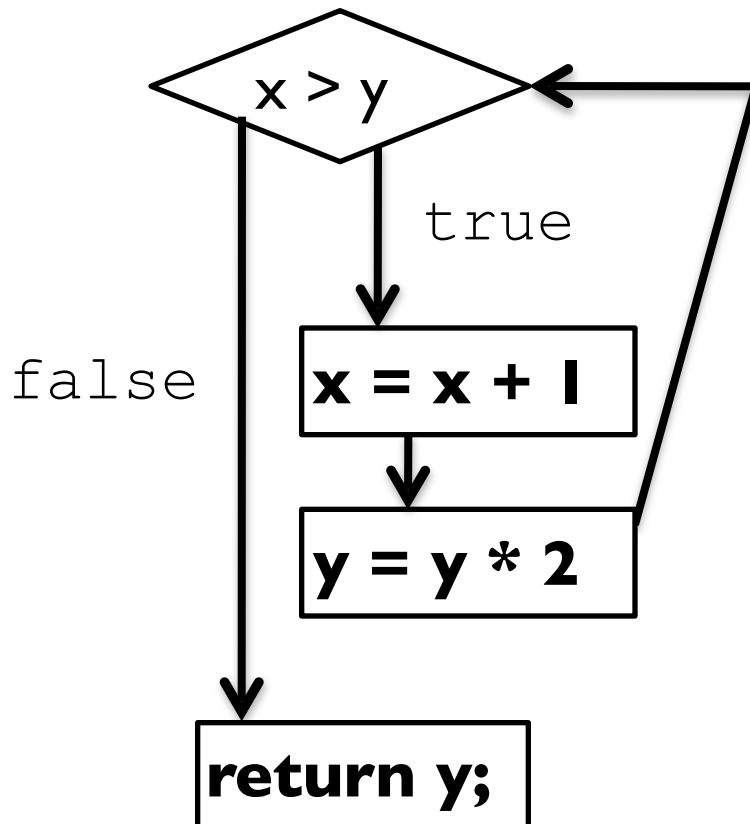
Example:

```
if ( a > b ) {
   a = b;
   x = x + 1;
}
print(x);
```
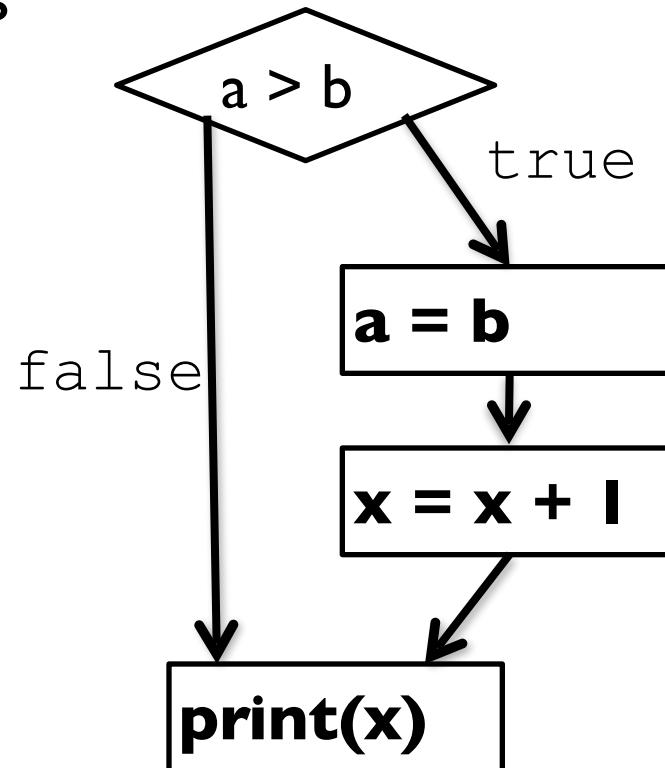
While Loop

Last statement in loop has a <u>back edge</u> to loop condition


```
while (x > y) {
    x = x + 1;
    y = y * 2;
}
return y;
```

Single Method Control-Flow

Use two special nodes to denote entry and exit of method

Start points to first statement

```
Start
```

```
Exit
```
Exit

All return statements point to Exit
Last statement in method points to

Method CFG

Start
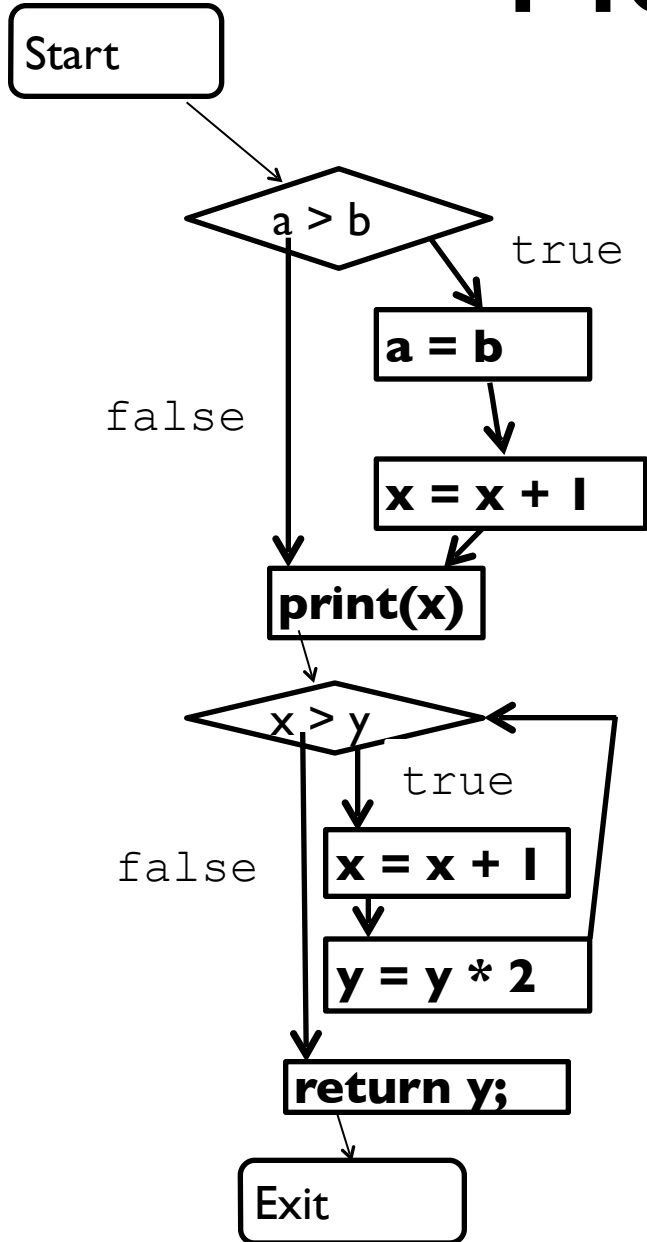

```
int testMethod(int a, int b, int x, int y) {

  if ( a > b ) {
    a = b;
    x = x + 1;
  }

  print(x);

  while (x > y) {
    x = x + 1;
    y = y * 2;
  }
  return y;

}
```
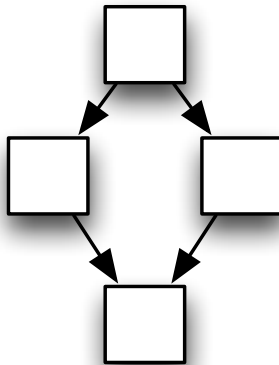
Discussion: How should we model a `for` `loop` in a CFG?

```
int y = getCount();
int x = 1;
for(int i = 0; i < y; i++) {
    x = x * 2;
}
return x;
```
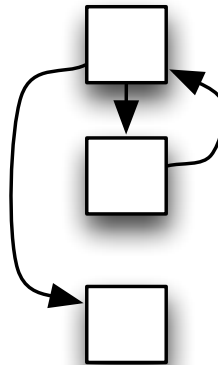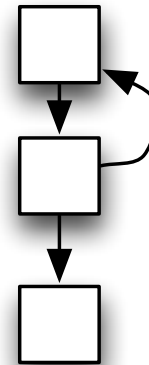
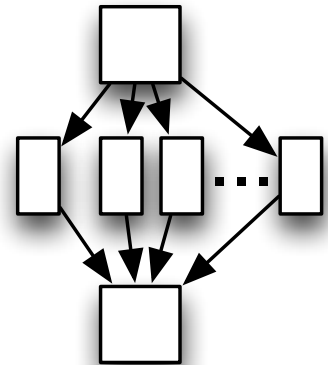sequence   if   while   until   case

White-box Coverage Criteria

- Statement Coverage (SC)
  - Synonym for node coverage
- Branch Coverage (BC)
  - Synonym for edge coverage
- Path Coverage
  - Full Path Coverage
  - Basis Path Coverage
- There are many more (that are rarely used)…

http://en.wikipedia.org/wiki/Code_coverage

White-box Coverage Criteria

•A chosen criteria, "X coverage", is covered if execution of a <u>test set</u> covers all X's in the unit

e.g. statement coverage is satisfied if all statements in the unit are executed

•For a given criteria:
- –Desirable to choose the minimum number of test cases for a given criteria

Statement Coverage Tools

covered

not covered

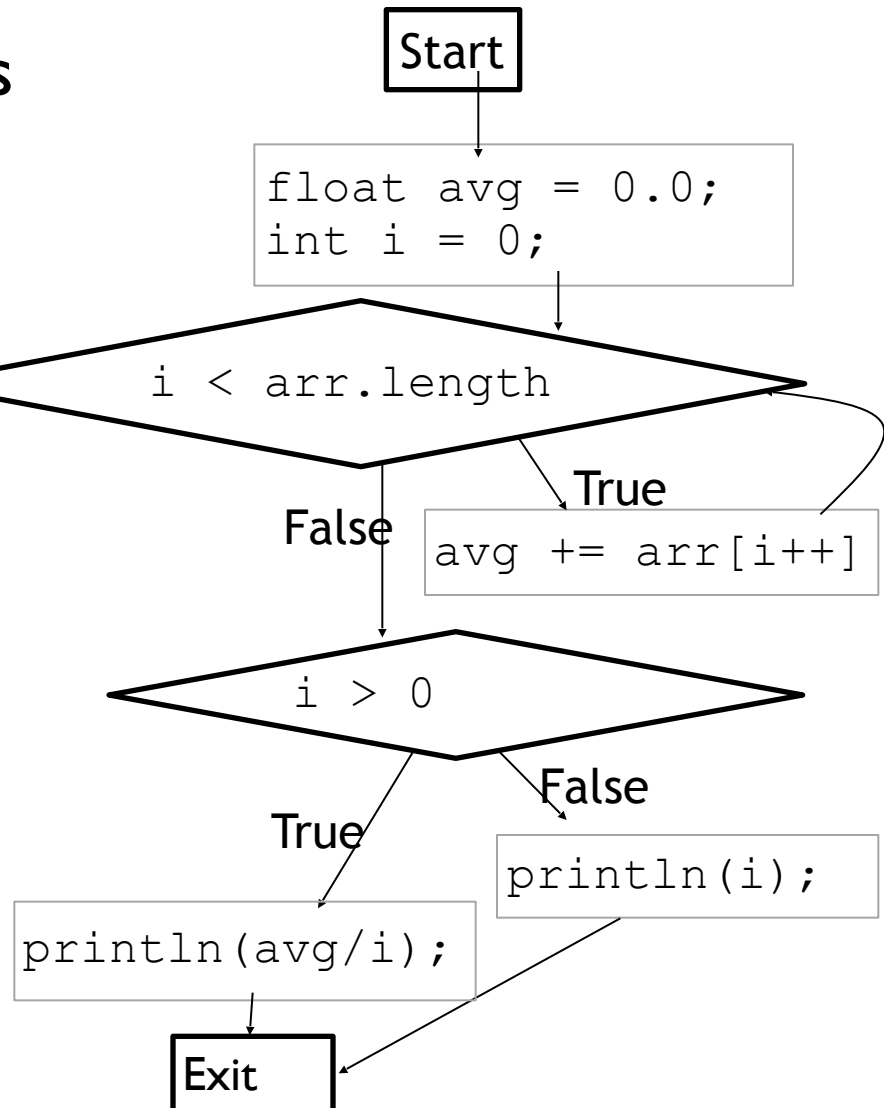# 1: Statement Coverage

Statement coverage requires that each statement (node) in the CFG is covered

What test set should we choose to achieve statement coverage of this method:

```
void avg(int[] arr)
```

?

Start

```
float avg = 0.0;
int i = 0;
```

i < arr.length

True

False

```
avg += arr[i++]
```

i > 0

True

False

```
println(i);
```

```
println(avg/i);
```

Exit

Statement Coverage

•with pen/paper, people often write test sets as:
  –Set of test cases
  –Test case as a set of assignments to input variables
  –e.g. { { x = 0, y = 0}, { x = 1, y = 1 }, { x = -1, y =1 }, { x = 1, y = -1} }

For example on previous slide:

{ {arr = [1]}, { arr = [1,2,3] } }

Criteria Rankings

Statement Coverage < Branch Coverage <
Basis Path Coverage < Path Coverage < Exhaustive Testing

If a test set satisfies some criteria then it also satisfies all lower ranked criteria

• Red are used most often in practice

2: Branch (edge) Coverage

What test set should we choose to achieve branch coverage of the `avg` method?

`void avg(int[] arr)`?

```
Start
```

```
float avg = 0.0;
int i = 0;
```

```
i < arr.length
```

True → `avg += arr[i++]`

False ↓

```
i > 0
```

True → `println(avg/i);`

False → `println(i);`

```
Exit
```
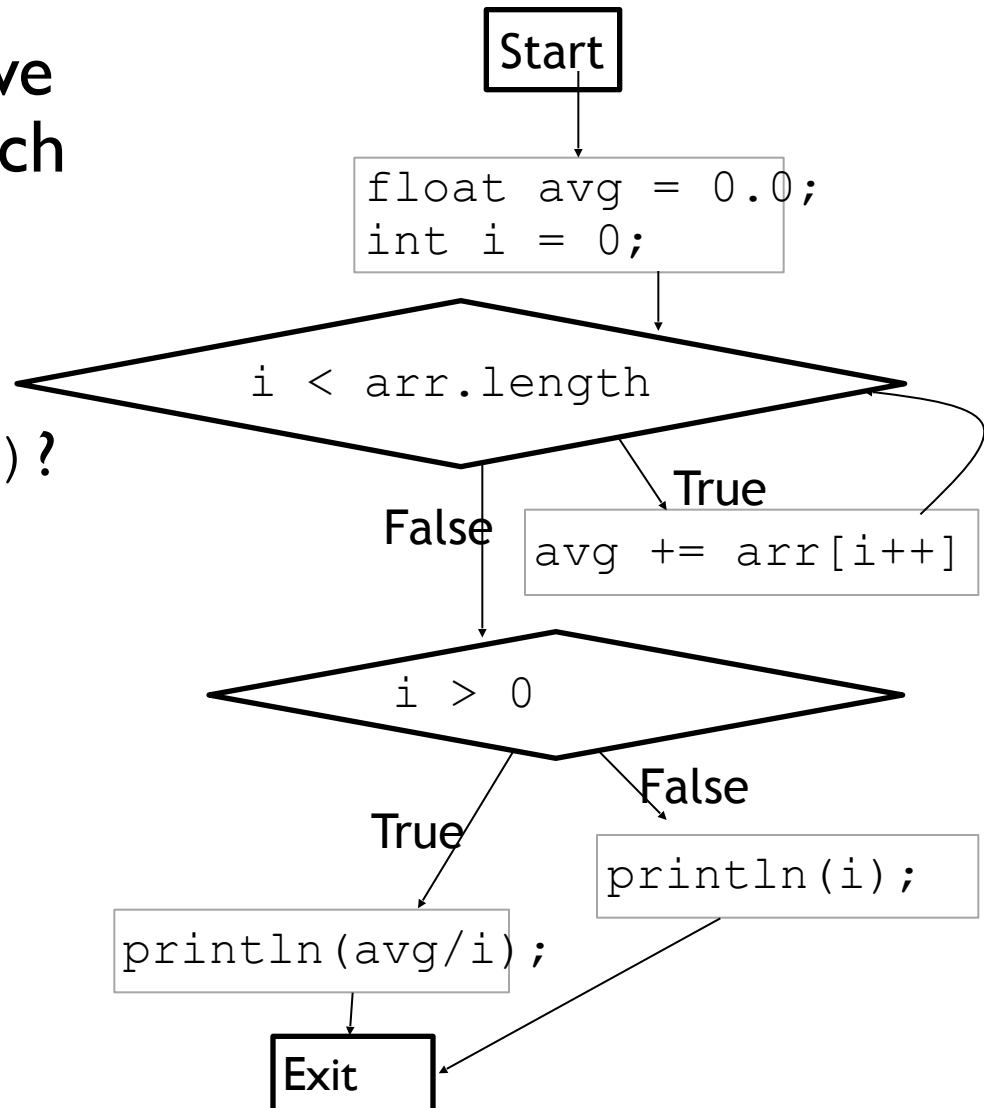
Criteria Ranking as Guidelines

• Criteria rankings are general guidelines

• e.g. Achieving branch coverage is frequently better than achieving statement coverage

A test set which achieves a particular coverage criteria may find less bugs than some other test set which achieves only a lower ranked criteria

Example

Assume we want to avoid divide by zero in our program:

```
double test(double x, double y) {
     if(x < y) {
          x = x + y;
     }
     return x/y;
}
```

{ { x = 1, y = 2}, { x = 2, y = 1} } achieves BC and <u>finds no bug</u>
{ { x = -1, y = 0} } achieves only SC and <u>finds a bug</u>

3: Path Coverage

• Path: a sequence of edge-connected nodes
    from Start to End
• Feasible Path:
    a path along which execution can actually flow
• The concept of feasible paths help to highlight the distinction between program syntax (structure) and program semantics (behavior)
• "Path coverage" really means "feasible path coverage"

Where is the infeasible path?
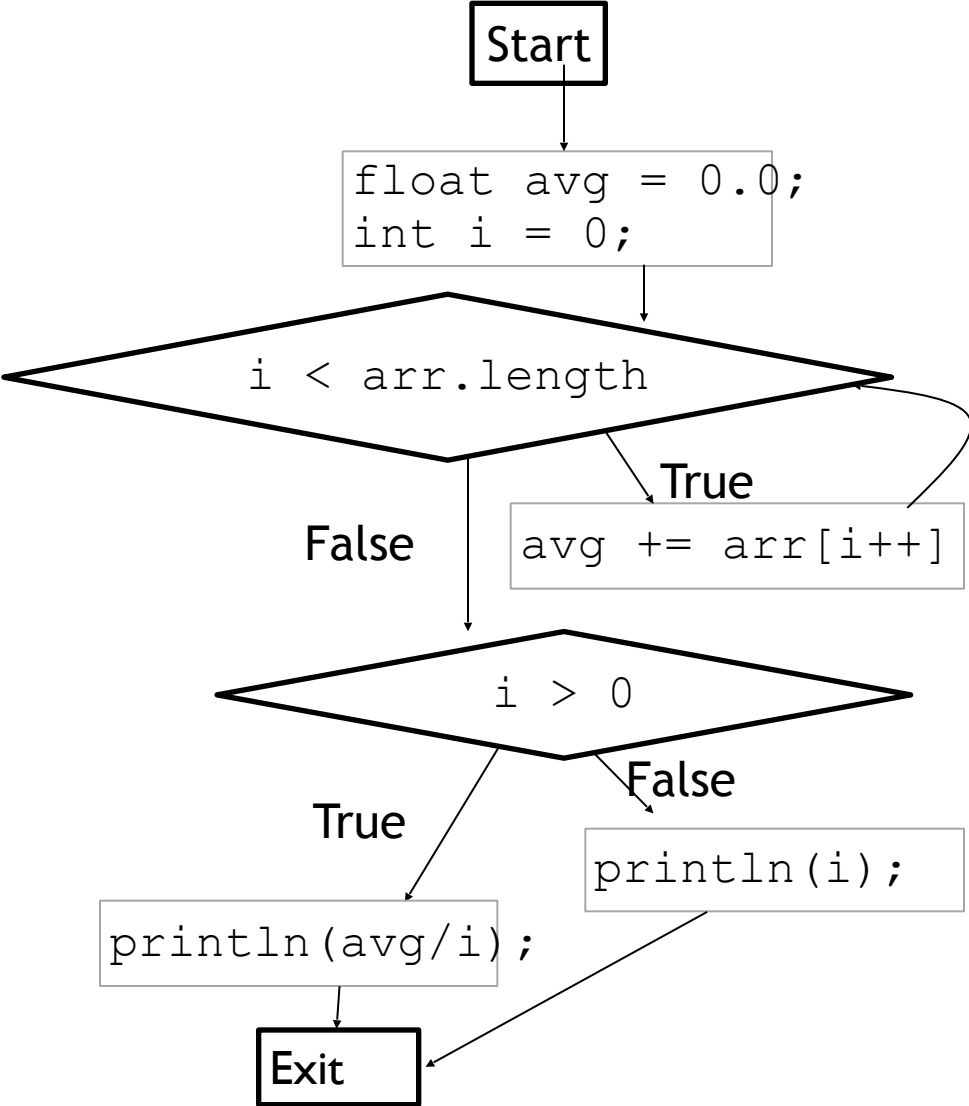
Why path coverage?

Finds more *non-localized* bugs:
- When dependence between particular statements causes bug
- Find a test set which has edge coverage but doesn't execute divide by zero

```
int test2(int x, int y) {
    int i = -1;
    if(x > 5)
        i = 0;
    if(y > 5)
        x = 10/i;
    return x * i;
}
```

we might try x=4, y=6.
or x=6, y=4

Problems with Path Coverage

1. Loops induce a potentially infinite number of paths

2. Number of paths can be as large as 2^c
   where c is the number of conditionals (`ifs`)
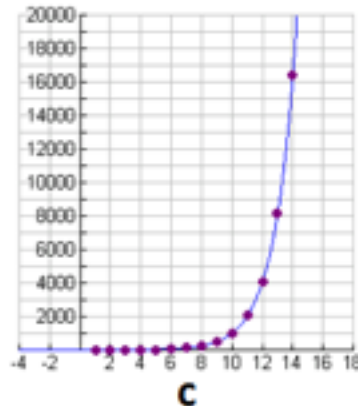
Basis Path Coverage

- Better than branch coverage (not as strong as path coverage)
- Does not consider additional cycles caused by repeating loops multiple times
- Ensures that *pairs* of conditionals/loops tested independently
  - Still doesn't test <u>all</u> paths
  - The outcome of one conditional/loop isn't always the same and isn't always different from another

Creating a Basis Set

1. Choose a "baseline" path, add it to the basis set
    –   Any path developer thinks is a useful starting point
2. Repeat:
    Add path to the basis set with one edge not already covered
    (i.e. flip one previously executed decision)
3. Write test cases which execute along those paths

Assuming all paths are feasible,
minimum number of paths in basis set is:
    N + 1, where N is number of conditionals/loops

TTT
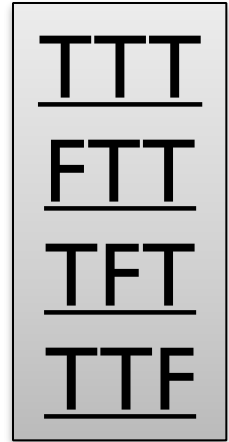FTT
TFT
TTF

Cyclomatic Complexity:
= #Edges - #Nodes + #terminal vertices (usually 2)
also = **#Predicate Nodes + 1 (used above)**
also = Number of regions of flow graph.

Class Activity:
Create a Basis Set

```
int test3(int x, int y) {
    int i = -1;
    if(x > 5)
        i = 0;
    if((x % 2) == 0)
        i++;
    if(y > 5)
        x = 10/i;
    return x * i;
}
```

TTT
FTT
TFT
TTF

Equivalence Class Partitioning

- *Identify* which types of inputs are likely to be processed in a similar way (equivalent behaviour)
  - Coverage, disjointedness, representation

- This creates **equivalence partitions**

- Each test should exercise **one and only one** equivalence partition

Equivalence Class Partitioning

Example of ECP

- System asks for numbers between 100 and 999

- Equivalence partitions:
  - ❑ Less than 100
  - ❑ Between 100 and 999
  - ❑ More than 999

- Three tests:
  - ❑ 50, 500, 1500

Boundary Testing

Test inputs starting from known **good** values and progressing through **reasonable but invalid** to known **extreme and invalid**

e.g. max/min, just inside/outside boundaries, typical values and error values

A <u>corner case</u> is a problem or situation that occurs only outside of normal operating parameters—specifically one that manifests itself when multiple environmental variables or conditions are simultaneously at extreme levels, even though each parameter is within the specified range for that parameter.

Boundary testing is like ECP but looking specifically at edge (corner) cases. Imagine testing the method: **getDaysInMonth(int month, int year)**

It would be reasonable to test with: 3,2008, 2,2002, 2,2000
It would *not* be reasonable to test with: -1 MaxInt MinInt 0

For example, if an input field is meant to accept only integer values 0–100, entering the values -1, 0, 100, and 101 would represent the boundary cases. A common technique for testing boundary cases is with three tests: one on the boundary and one on either side of it. So for the previous example that would be -1, 0, 1, 99, 100, and 101.

How good is ECP and Boundary Testing?

- Better than random testing
- But only as good as your partitions!
- No exploration of combinations of inputs.
- Unfortunately, guessing is the best you can do if you are using **black box** testing

Summary

- Path coverage is a structural criteria weaker than Exhaustive Testing but still impractical

- Many tools tell when you have achieved:
  - Statement coverage
  - Branch coverage
  - Basis path coverage

- *But in general they don't tell you how to achieve them*
  - *Newer tools can suggest inputs for simple cases*

- Coverage criteria gives you different goals to strive for

Study Activity

Write a black-box test set for the ArrayList<E>.get(int index) method

Specification:
public E get(int index) Returns the element at the specified position in this list.

Parameters:index - index of the element to return

Returns:the element at the specified position in this list

Throws: IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

Study Activity

Write a black-box test set for the
Comparable<Integer>.compareTo(Integer o) method

Specification: Compares this object with the specified object for
order. Returns a negative integer, zero, or a positive integer as this
object is less than, equal to, or greater than the specified object
Parameters: o - the object to be compared.
Returns :a negative integer, zero, or a positive integer as this object
is less than, equal to, or greater than the specified object.
Throws: NullPointerException - if the specified object is null
        ClassCastException - if the specified object's type
        prevents it from being compared to this object.

Study Activity

Considering everything you've learned about testing. Write a comprehensive test set for the method on the next slide, and write a brief description of how you decided which test cases to include.

```java
/**
 * Attempt to rename a tag. newName must not be null.
 * Returns true/false
 */
public boolean renameTag(String oldName, String newName) {

if(oldName.equals(newName))
        return false;

Tag tag = tags.get(oldName);
if(tag == null)
        return false;

if(tags.containsKey(newName))
        return false;

tags.remove(oldName);
tag.setName(newName);
tags.put(newName, tag);
return true;
}
```