SOFTENG 254:
Quality Assurance

Lecture 3a: Control Flow Coverage

Paramvir Singh
School of Computer Science

Potential Assessment Question

You are given the source code to a class and told that it is known there are at most 3 faults in it. You are asked to create a test suite for the class. How many JUnit test methods will you need to write?

(a) Exactly 3.
(b) Not more than 3.
(c) At least 3.
(d) It depends.

Justify your answer.

Agenda

- PAQ
- Admin
  - Schedule: Lab — More JUnit
  - Assignment 1?
- Control flow graphs as a model for code
- Other forms of coverage-based test case development
  - branch coverage
  - condition coverage
  - path coverage

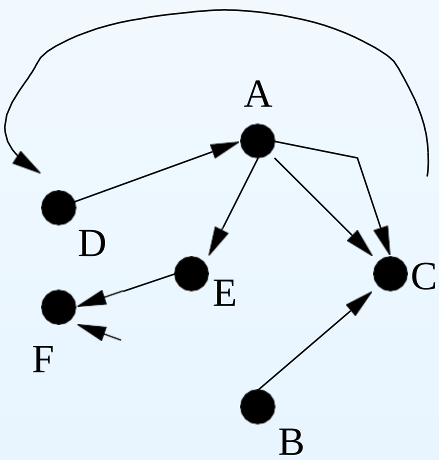
Statement coverage isn't enough

. . . so what is?

Modelling Code

- a graph is a mathematical structure (with an appealing visual representation) for representing or modelling things that are related
- consists of *vertices* (or nodes, or points), connected by *edges* (or line segments, or arcs).

  Vertices: A, B, C, D, E, F Edges: (A,C), (A,C), (A,E), (B,C), (B,F), (C,D), (D,A), (E,F),



- Also: undirected graphs, rules restricting edges between vertices, classification of vertices
- *Graph Theory* — the study of properties of graphs

Control Flow Graphs

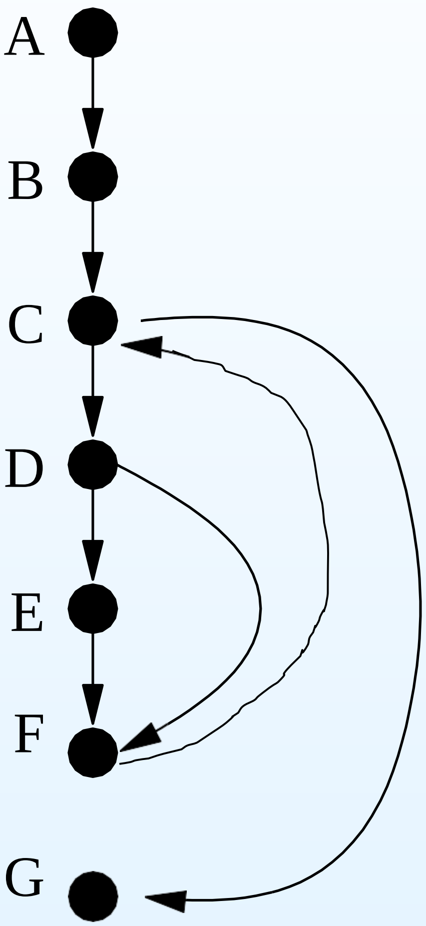
- Directed graphs can be used to model control flow of a program $\Rightarrow$ Control Flow Graph (CFG)
- *vertex* $=$ statement, *edge* $= (A, B)$ if control flows from statement A to B
- properties of CFGs may provide information about properties of the code
  - one property of interest: *how many ways can control flow through the code*

Example

```
     public static boolean isPrime(int n) {
A        boolean prime = true;
B        int i = 2;
C        while (i < n) {
D            if (n % i == 0) {
E                prime = false;
             }
F            i++;
         }
G        return prime;
     }
```

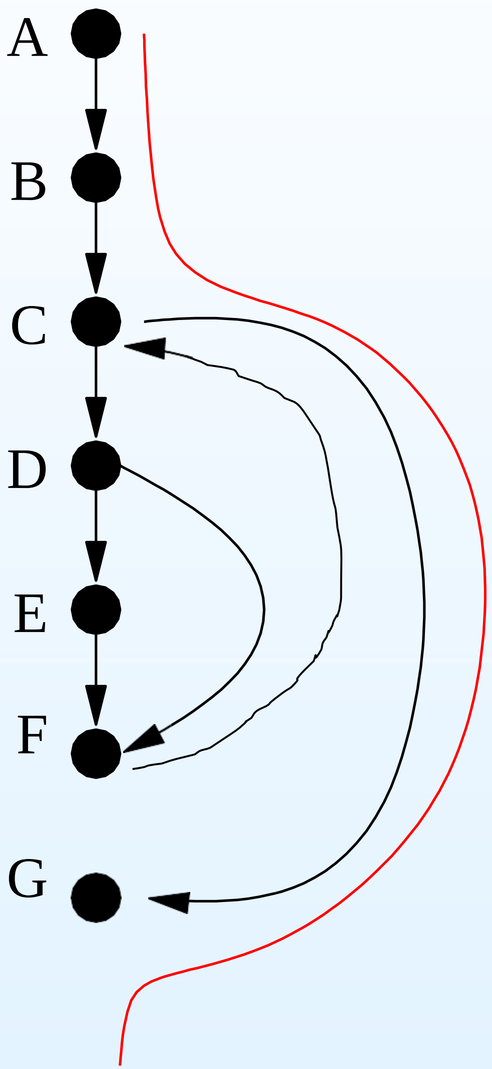
Example


```
    public static boolean isPrime(int n) {
A       boolean prime = true;
B       int i = 2;
C       while (i < n) {
D           if (n % i == 0) {
E               prime = false;
            }
F           i++;
        }
G       return prime;
    }
```



Example

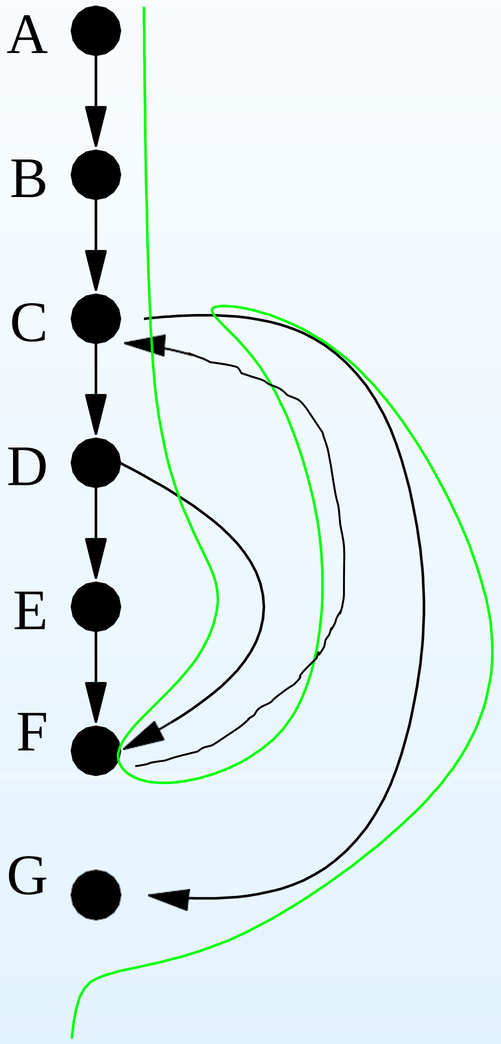
```
    public static boolean isPrime(int n) {
A       boolean prime = true;
B       int i = 2;
C       while (i < n) {
D           if (n % i == 0) {
E               prime = false;
            }
F           i++;
        }
G       return prime;
    }
```



- PAQ
- Agenda
- Statement Coverage
- Control Flow Graphs
- Coverage Criteria
- CFG Construction
- Compound Expressions
- Path Coverage
- Key Points

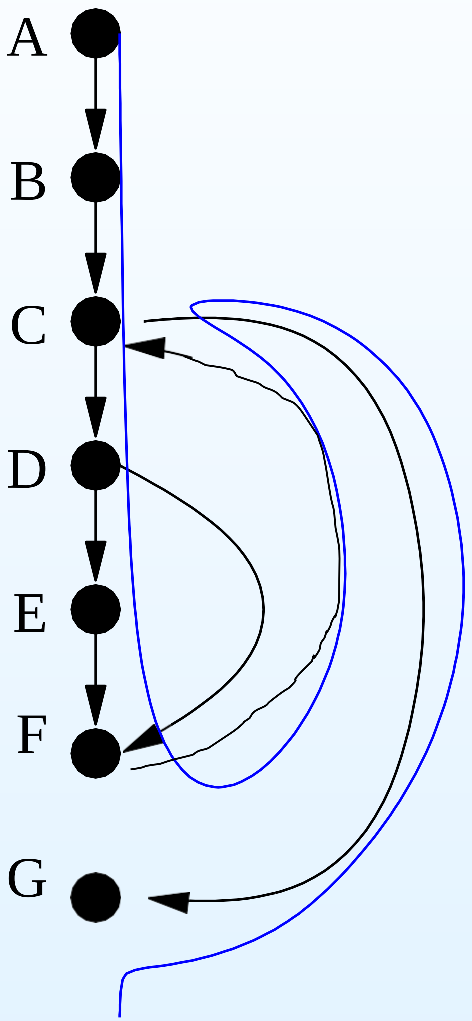
Example

```
    public static boolean isPrime(int n) {
A       boolean prime = true;
B       int i = 2;
C       while (i < n) {
D           if (n % i == 0) {
E               prime = false;
            }
F           i++;
        }
G       return prime;
    }
```



Example

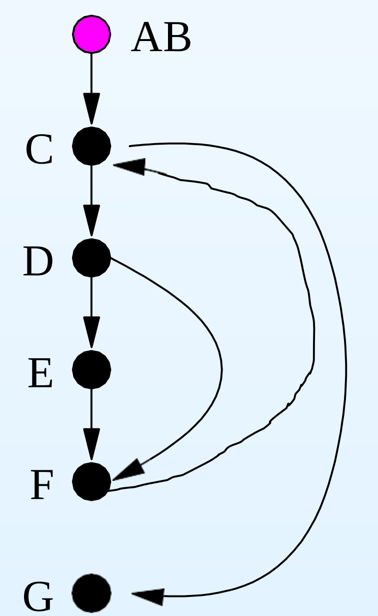
```
     public static boolean isPrime(int n) {
A        boolean prime = true;
B        int i = 2;
C        while (i < n) {
D            if (n % i == 0) {
E                prime = false;
             }
F            i++;
         }
G        return prime;
     }
```



Basic Blocks

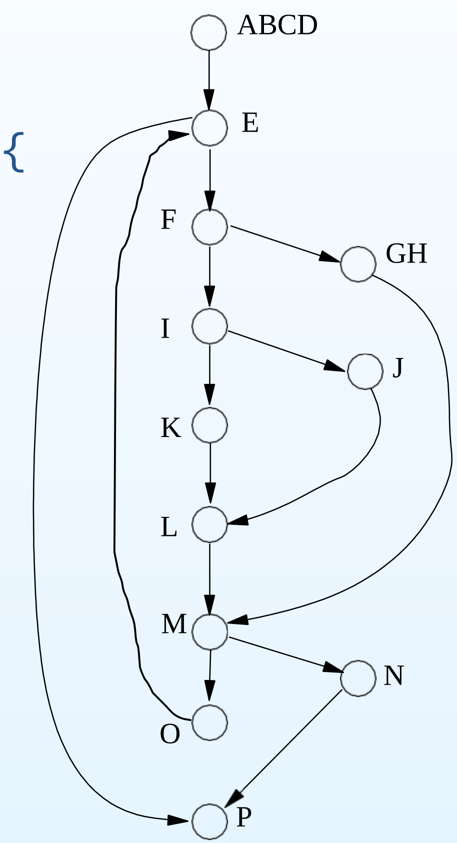
- CFGs are typically simplified by having a vertex represent a basic block
- A basic block is a sequence of statements (or instructions) with the following properties:
  - The first statement is always the first statement executed (one entry point)
  - Only the last statement can cause the program to begin executing code in a different basic block (one exit point)
  - Consequently, if the first statement is executed then all statements in the basic block are executed

```
    public static boolean isPrime(int n) {
A       boolean prime =true;
B       int i =2;
C       while (i <n) {
D           if (n %i ==0) {
E               prime = false;
            }
F           i++;
        }
G       return prime;
    }
```



Example Variation B

```
     public static int leadingSpacesCount(String text, int tabstop) {
A      int index = 0;
B      int count = 0;
C      int interTab = 0;
D      char[] chars = text.toCharArray();
E      while (index < chars.length &&
             Character.isWhitespace(chars[index])) {
F        if (chars[index] == '\t') {
G          count += tabstop - interTab;
H          interTab = 0;
         }else {
I          if (interTab == tabstop - 1) {
J            interTab = 1;
           }else {
K            interTab++;
           }
L          count++;
         }
M        if (index == chars.length-1) {
N          break;
         }
O        index++;
       }
P      return count;
     }
```

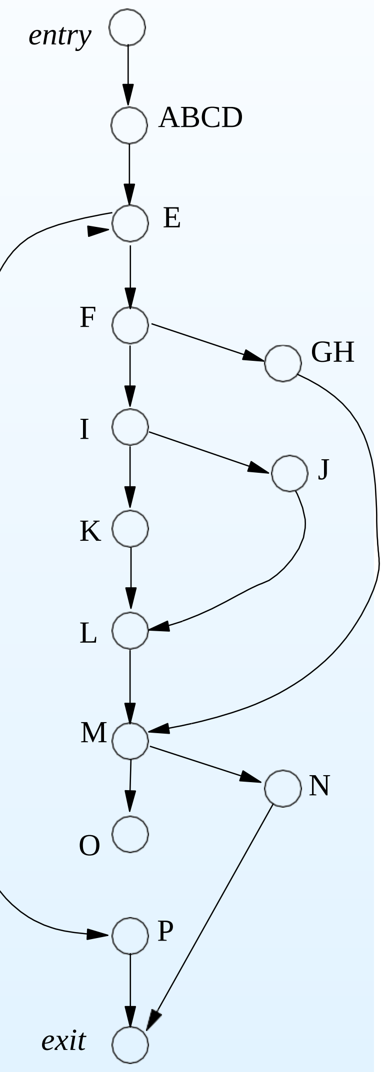


Modelling Code continued

- Need some way to represent entry point to code being modelled
  - e.g. to represent assignment of actual parameters to formal parameters
- Need some way to represent exit point of code being modelled
  - e.g. to represent multiple exists from code

Example Variation C

```
     public static int leadingSpacesCount(String text, int tabstop) {
A      int index = 0;
B      int count = 0;
C      int interTab = 0;
D      char[] chars = text.toCharArray();
E      while (Character.isWhitespace(chars[index])) {
F        if (chars[index] == '\t') {
G          count += tabstop - interTab;
H          interTab = 0;
         }else {
I          if (interTab == tabstop - 1) {
J            interTab = 1;
           }else {
K            interTab++;
           }
L          count++;
         }
M        if (index == chars.length-1) {
N          return count;
         }
O        index++;
       }
P      return count;
   }
```



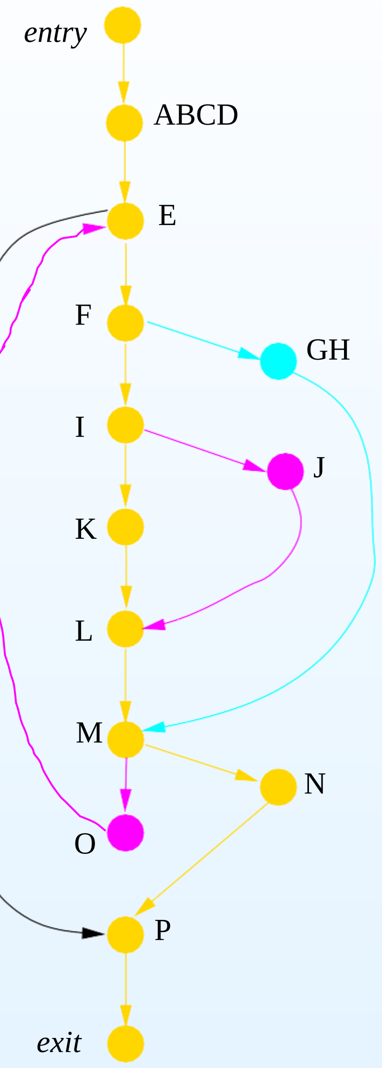
Test Suite for Variation B

- Input = (" ", 4), Expected Output: 1
- Input = ("   ", **1**), Expected Output: 2
- Input = ("\t", 4), Expected Output 4
- 100% statement coverage

- PAQ
- Agenda
- Statement Coverage
- Control Flow Graphs
- Coverage Criteria
- CFG Construction
- Compound Expressions
- Path Coverage
- Key Points

Statement coverage as CFG coverage

- Input = (" ", 4), Expected Output: 1 (yellow)
- Input = ("   ", **1**), Expected Output: 2 (magenta)
- Input = ("\t", 4), Expected Output 4 (cyan)
- 100% statement (= vertex) coverage
- but not 100% edge coverage



Coverage Criteria for Testing

- A **test requirement** is a specific element of a software artifact that a test case must satisfy (or "cover")
  - E.g. "statement A must be executed"
- A **test requirement set** is a set of test requirements
- A **coverage criterion** is a rule or set of rules that impose test requirements on a test suite.
  - E.g. The "statement executed" criterion imposes, for every statement x, the test requirement that "statement x must be executed"
- Given a test requirement set determined by a coverage criterion and a test suite, the **Coverage Level** is the ratio of the number of test requirements satisfied by the test suite to the total number of test requirements
  - E.g. 15 statements means 15 different test requirements. If there are 3 statements that are not executed by any of the tests in the test suite, then the coverage level is $\frac{12}{15}$.

Coverage Criteria Example

Coverage criterion Call Constructor: If $\mathrm{X}()$ is a constructor then it must be called

Test requirement set For all classes, for all constructors of those classes, the constructor must be called

Coverage Level For a given test suite, the proportion of constructors that get called

CFG-based Test Requirements

- Test requirements can come from anywhere
- For example, model the code with a control flow graph, develop test requirements based on the CFG
- A **test path ‡** in a CFG is a sequence of vertices in the CFG such that the first is the entry vertex, the last is the exit vertex, and every pair of adjacent vertices is connected by an edge in the CFG
  - a test path describes a potential path through the code that the CFG models
  - each test case has a corresponding test path
- A test path **visits** a vertex *V* if *V* is one of the vertices in the test path.
- A test path **visits** an edge (*V*,*W*) if *V* and *W* are adjacent in the test path (and in that order)
- vertex coverage criterion: for every vertex *V* in the CFG, the test requirement is "*V* is visited" statement coverage
- edge coverage criterion: for every edge (*V*,*W*), the test requirement is "(*V*,*W*) is visited branch or decision coverage

(‡ different people use different terminology)

More CFG Construction

- if we change how the CFG is constructed (change how we model the code) we get different test requirements
- Some language features complicate CFG construction. For example:
- for loops — one statement or 3 (or 4)?
- try/catch blocks
- conditional expressions — e.g. `(a>b)?missile.launch():missile.destruct()`
- compound expressions — e.g. `a && b || (c > 5)`

Compound Expressions

- Suppose statement is:
  `if (a > 0 && b > 0) {...}`
  but should have been:
  `if (a > 0 || b > 0) {...}`
- This test suite gives full branch coverage, but does not detect the fault

a	a > 0	b	b > 0	Expected	Actual
10	true	30	true	true	true
-1	false	-10	false	false	false

- Need to test other possible combinations of sub-expressions

a	a > 0	b	b > 0	Expected	Actual
10	true	-10	false	true	false
-1	false	30	true	true	false

- Also known as condition coverage

Modelling compound expressions

- One approach to the compound expression problem is to change how CFGs are created for such cases

```
if (a > 0 && b > 0) {
  // if clause
}else {
  // else clause
}
```

- restructure as simple conditions

```
if (a > 0) {
  if (b > 0) {
    // if clause
  }else {
    // else clause
  }
}else {
    // else clause
}
```

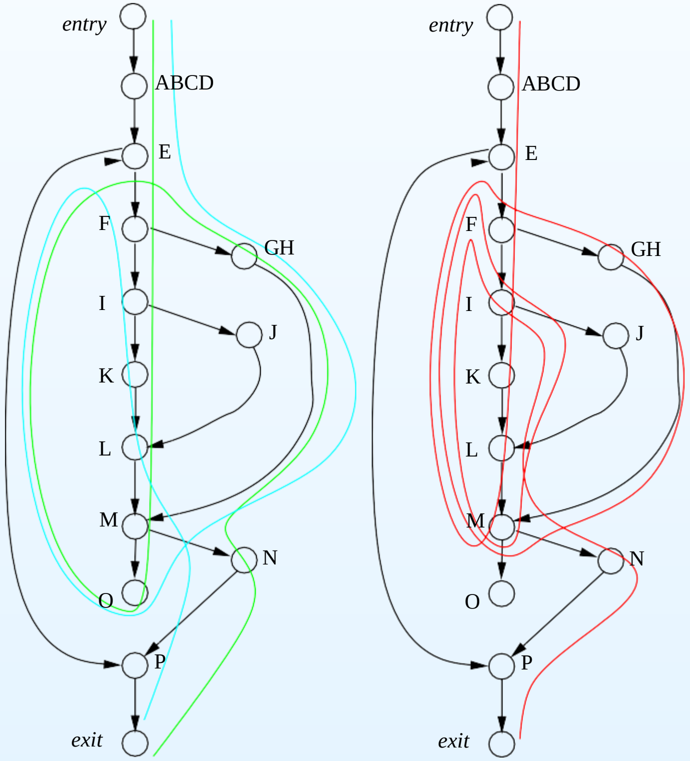
- Create CFG from restructured version
- *the restructuring is done only as part of the modelling, the actual code does not change*
- Other forms of coverage can be used to achieve the same result

Path Coverage

- Coverage criterion: any test path
- Intuition: any execution flow is represented by a path, so any missed paths may correspond to incorrect execution

Example paths

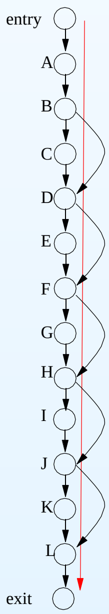
- There are many different paths through a CFG. Any path can potentially identify a fault.
- There are potentially a lot of paths!



The Map is not the Territory

- A CFG is a model of the code, therefore there can (and usually will) be some differences between the CFG and the code
- Some of those differences can lead to test requirements that can never be met in reality

```
    public String grade(int mark) {
A     String result = "";
B     if (mark > 90) {
C       result = "Adequate";
      }
D     if (mark > 75 {
E       result = "Ok";
      }
F     if (mark > 50 {
G       result = "Pass";
      }
H     if (mark > 40 {
I       result = "More effort required";
      }
J     if (mark <= 40 {
K       result = "Oh dear";
      }
L     return result;
  }
```



- some test paths are infeasible
- some test paths correspond to dead code

- [PAQ]
- [Agenda]
- [Statement Coverage]
- [Control Flow Graphs]
- [Coverage Criteria]
- [CFG Construction]
- [Compound Expressions]
- [Path Coverage]
- [Key Points]

Key Points

- There are a potentially an unlimited number of paths $\Rightarrow$ impractical number of test requirements to be met
- Sometimes, not all paths can be executed (infeasible paths, dead code) $\Rightarrow$ not all test requirements can be met
- 100% edge (branch) coverage is a minimal level, but more is typically needed
- These test requirements are based on *control flow*