# Testing

Sergey Mechtaev

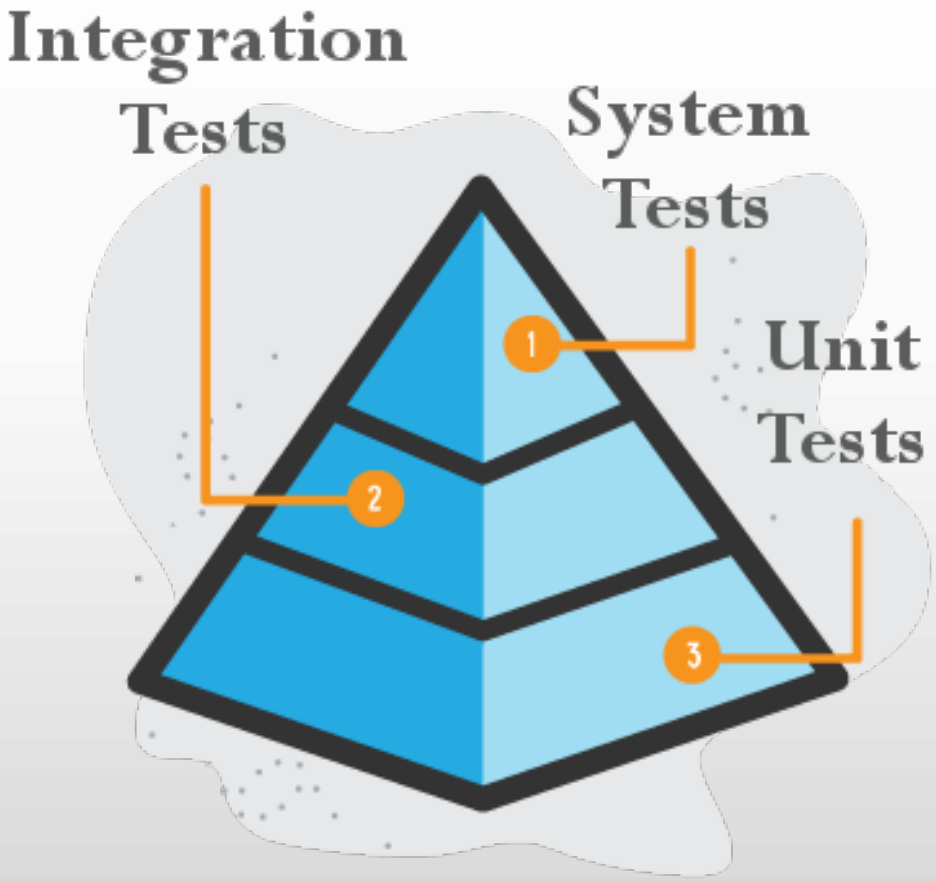
# Testing

- An **error** is a deviation of the observed behavior from the required (desired) behavior
  - **Functional requirements** (e.g., user-acceptance testing)
  - **Nonfunctional requirements** (e.g., performance testing)
- **Testing** is the process of executing a program with the intent of finding errors
  - Development testing
  - Release testing
  - User testing

# Limitation of Testing

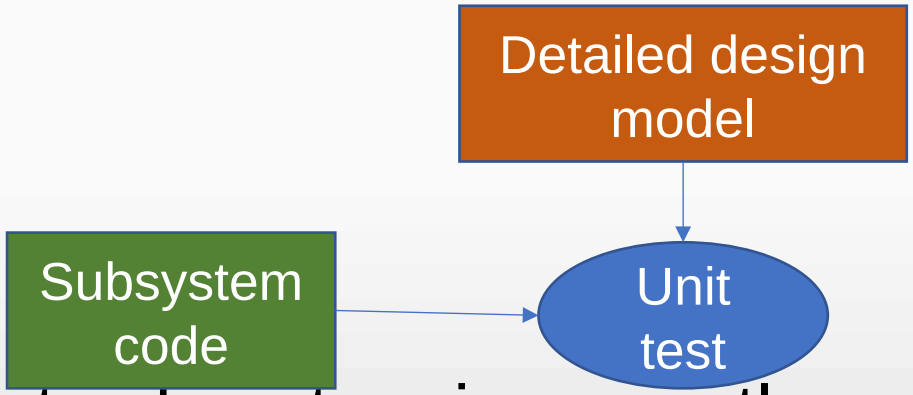
Testing can only show the presence of bugs, not their absence. – E.W. Dijkstra

# Development Testing



# Unit Testing

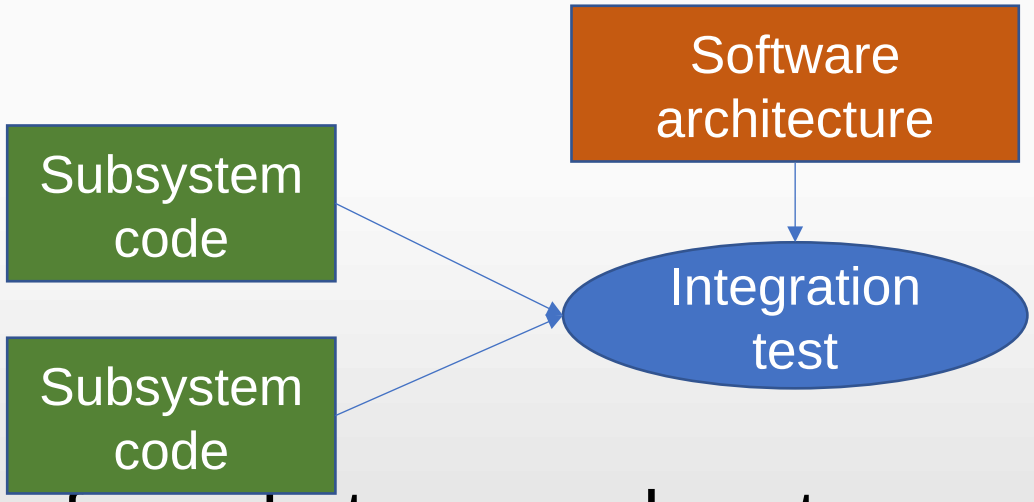
- Testing individual subsystems (collection of classes)




- Goal: Confirm that subsystem is correctly coded and carries out the intended functionalities

# Integration Testing

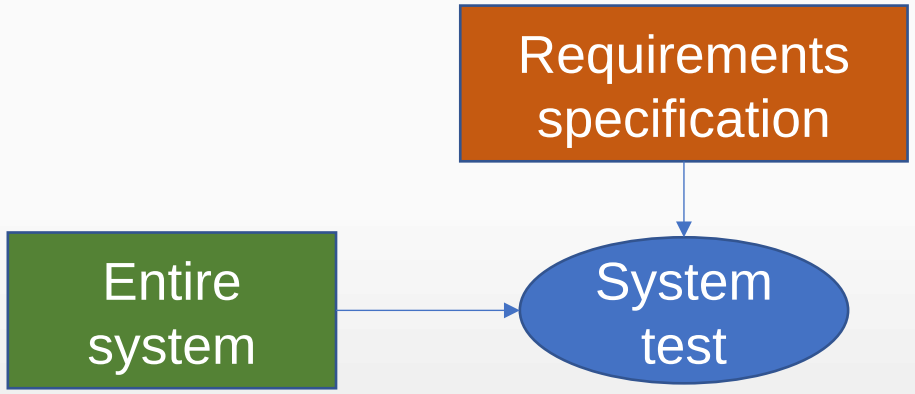
- Testing groups of subsystems and eventually the entire system




- Goal: Test interfaces between subsystems

# System Testing

- Testing the entire system




- Goal: Determine if the system meets the requirements (functional and non-functional)

# White-box (Structural) Testing

- Statement coverage
- Branch coverage
- Path coverage

# Basic Blocks

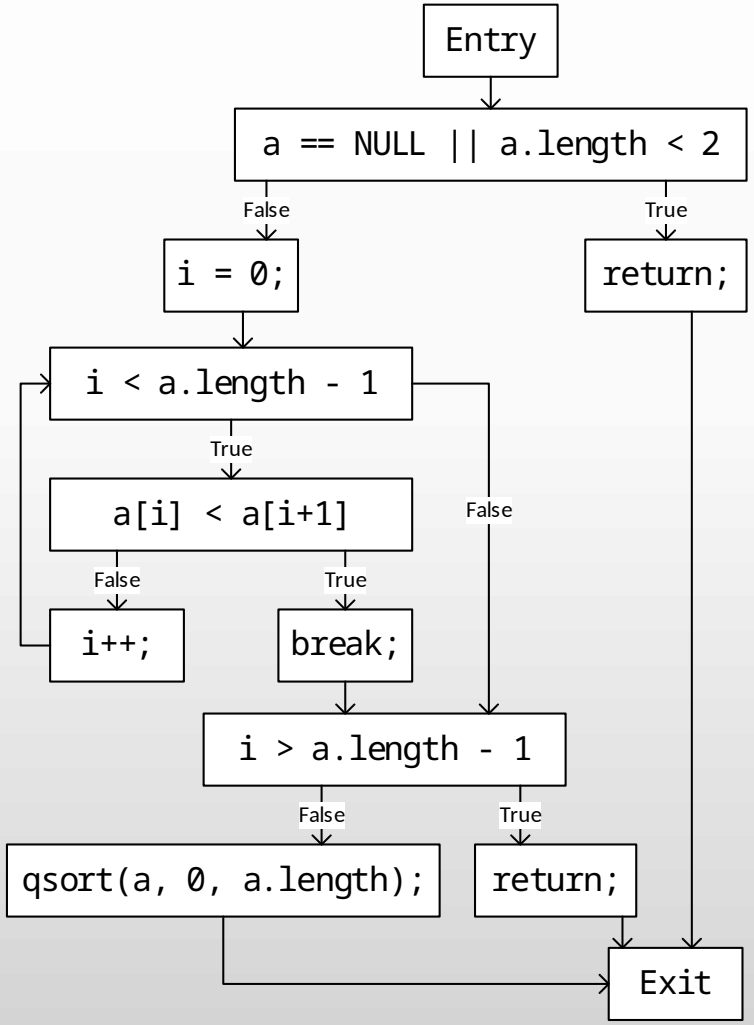
```
public void sort(int[] a) {
  if(a == null || a.length < 2)
    return;
  int i;
  for(i=0; i<a.length-1; i++) {
    if(a[i] < a[i+1])
      break;
  }
  if(i >= a.length - 1)
    return;
  qsort(a, 0, a.length);
}
```

**Definition.** Basic block is a sequence of statements that

- has one entry point
- has one exit point

# Control Flow Graph

```
public void sort(int[] a) {
  if(a == null || a.length < 2)
    return;
  int i;
  for(i=0; i<a.length-1; i++) {
    if(a[i] < a[i+1])
       break;
  }
  if(i > a.length - 1)
    return;
  qsort(a, 0, a.length);
}
```



# Statement Coverage

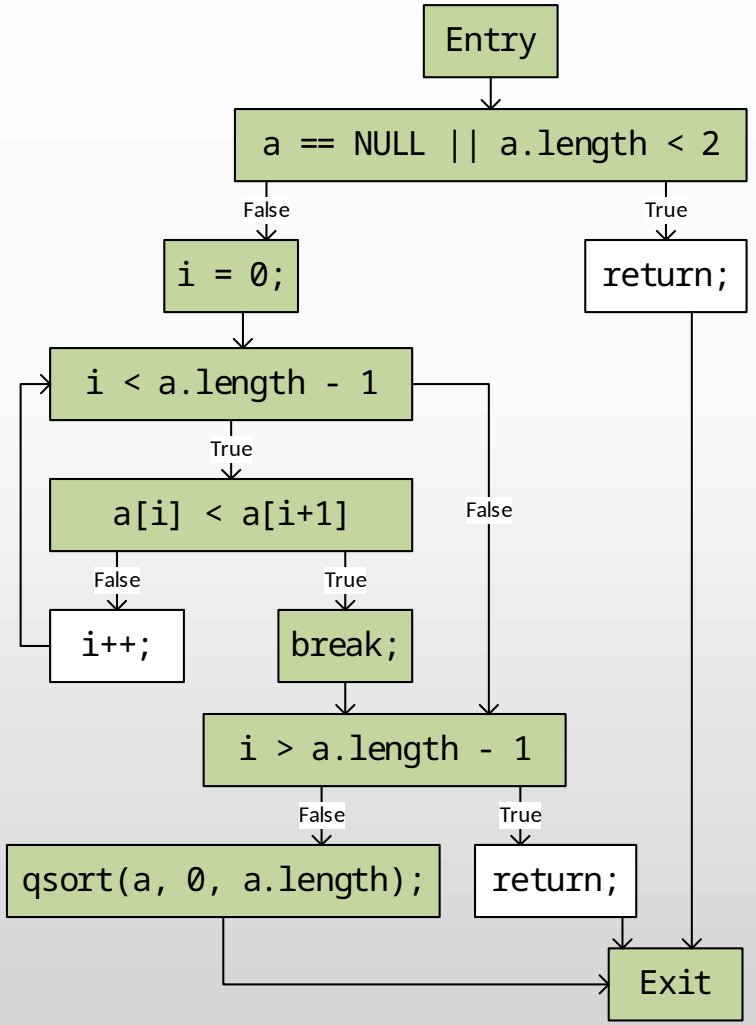
$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

# Statement Coverage

Coverage for the input

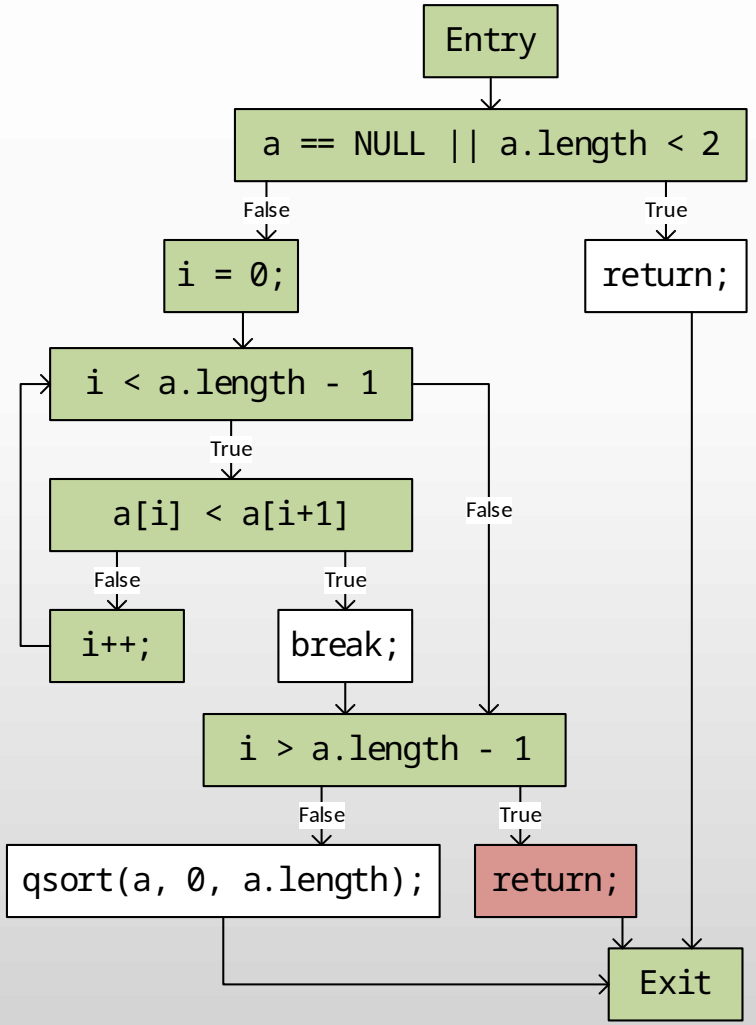
a = [ 3, 7, 5 ]

Executes 7 out of 10 blocks, so statement coverage is 70%



# Statement Coverage

- We can achieve 100% statement coverage with three test cases
  - a = [ 1 ]
  - a = [ 5, 7 ]
  - a = [ 7, 5 ]  (bug)



# Branch Coverage

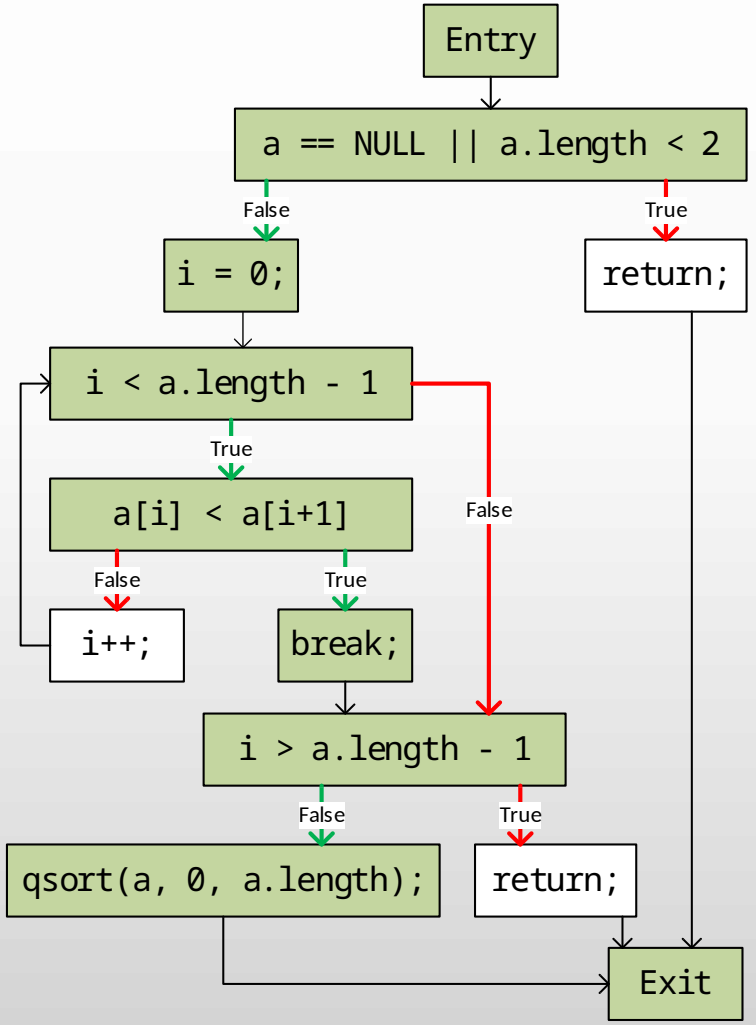
$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

# Branch Coverage

- Consider the input

a = [ 3, 7, 5 ]

- This single test case executes 4 out of 8 branches
- Branch coverage: 50%



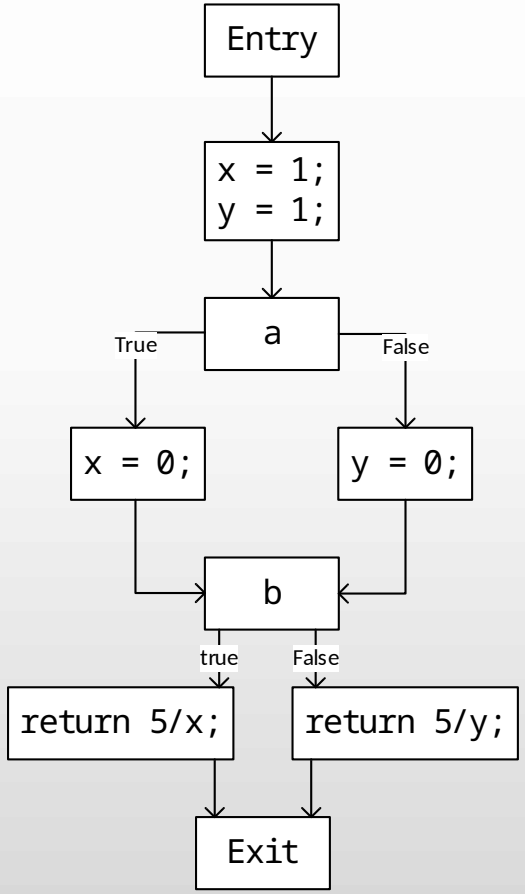
# Statement vs Branch Coverage

Which of the following statements are true?

1. 100% statement coverage implies 100% branch coverage
2. N% statement coverage implies at least N% branch coverage
3. 100% branch coverage implies 100% statement coverage
4. N% branch coverage implies at least N% statement coverage

# Path Coverage

```
int foo(boolean a,
        boolean b) {
  int x = 1;
  int y = 1;
  if (a)
    x = 0;
  else
    y = 0;
  if (b)
    return 5 / x;
  else
    return 5 / y;
}
```

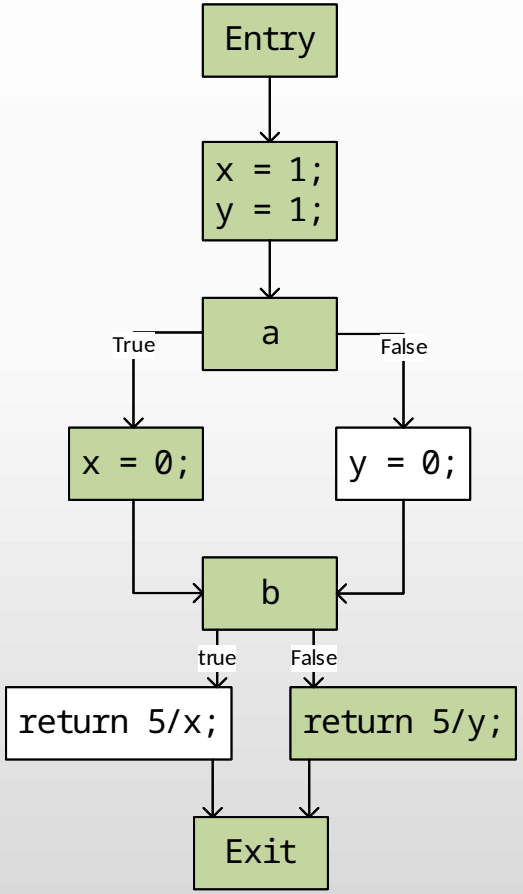


# Path Coverage

We can achieve 100% branch coverage with two test cases:

- a = true, b = false
- a = false, b = true

The test cases do not detect the bug!

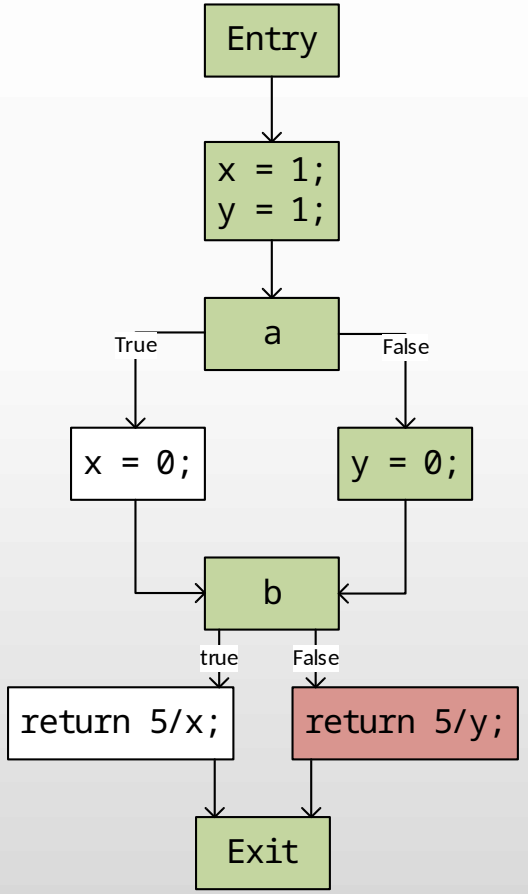


# Path Coverage

We can achieve 100% path coverage with four test cases:

- a = true, b = false
- a = false, b = true
- a = true, b = true
- a = false, b = false

The two additional test cases detect the bugs



# Mutation Testing

Learning from mistakes:

**Problem.** How good is the program tested?

**Technique.** Simulate earlier mistakes and see whether the resulting defects are found

# Mutation Testing

Original

```
int f(int x, int y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

```
int a = f(5, 10);
assertEquals(a, 15);
```

✔

Mutant

```
int f(int x, int y) {
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

```
int a = f(5, 10);
assertEquals(a, 15);
```

✖

Mutant killed