Software Engineering Processes

Course Code: XB_0089

Claudia Raibulet & Fernanda Madeiral

Lecture: Software Testing

VU

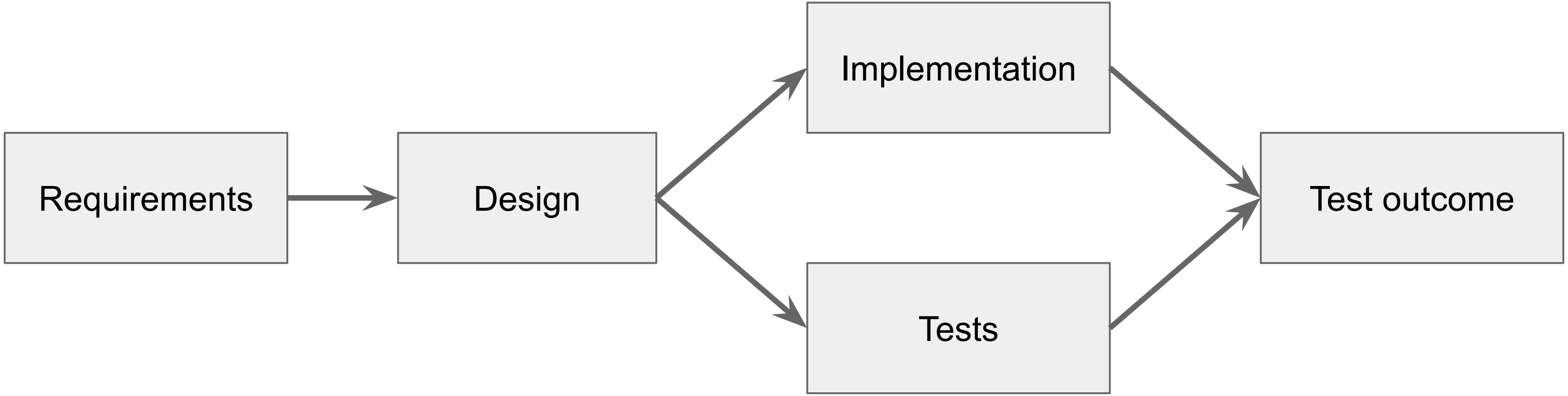
Bachelor in Computer Science, 2023/2024

Software testing

Software testing is a process in which you execute your program using data that simulate user inputs.

—Ian Sommerville

Software testing: overall idea



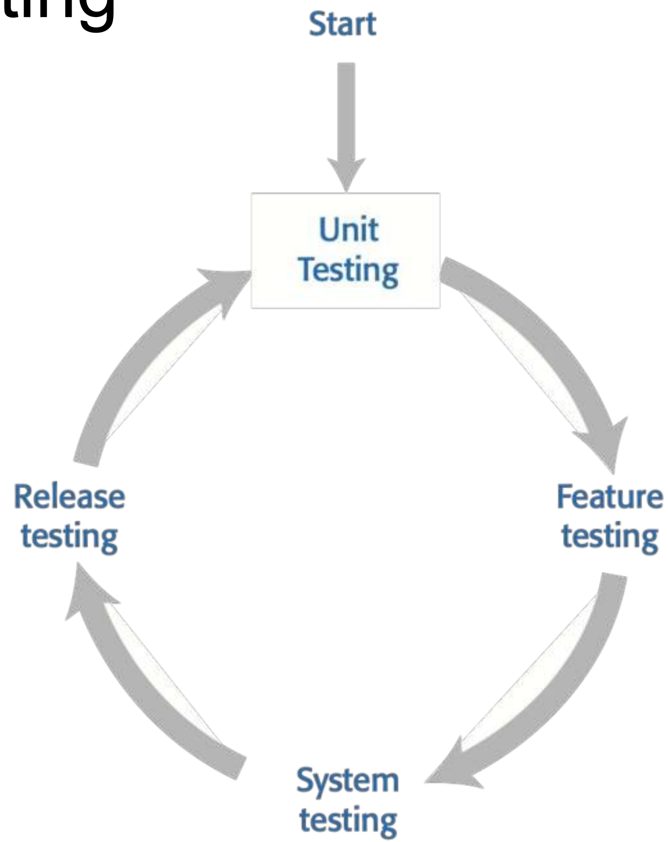
Types of testing in terms of purpose

- Functional testing
- User testing
- Performance and load testing
- Security testing

Types of testing in terms of purpose

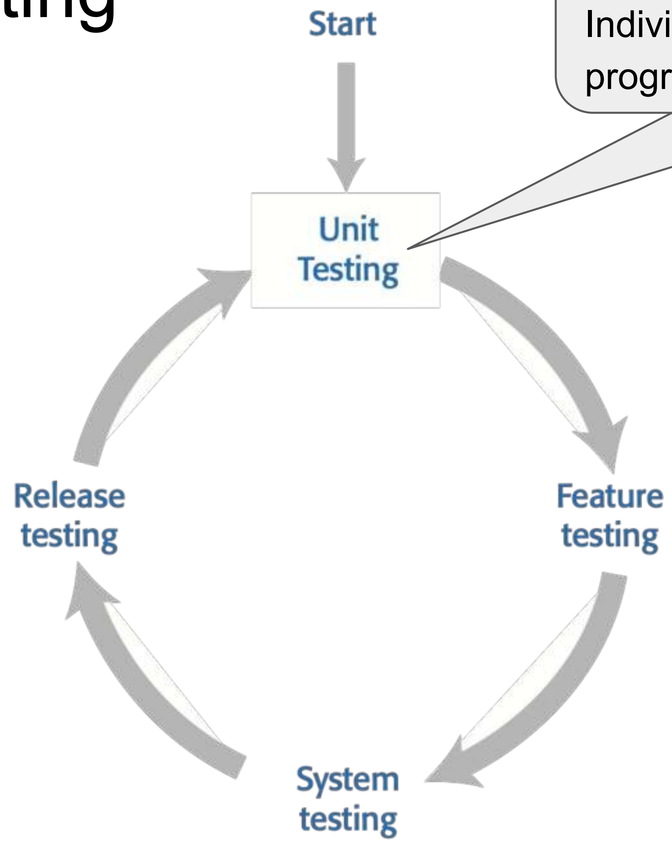
- **Functional testing**
- User testing
- Performance and load testing
- Security testing

Functional testing

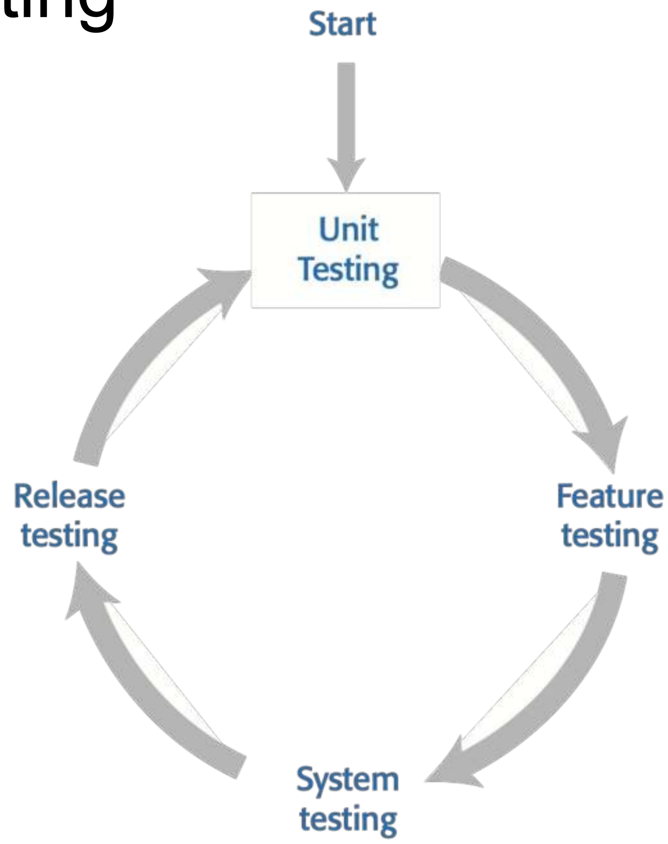


Functional testing

The aim of unit testing is to test program units in isolation.
Individual code units are tested by the programmer as they are developed.

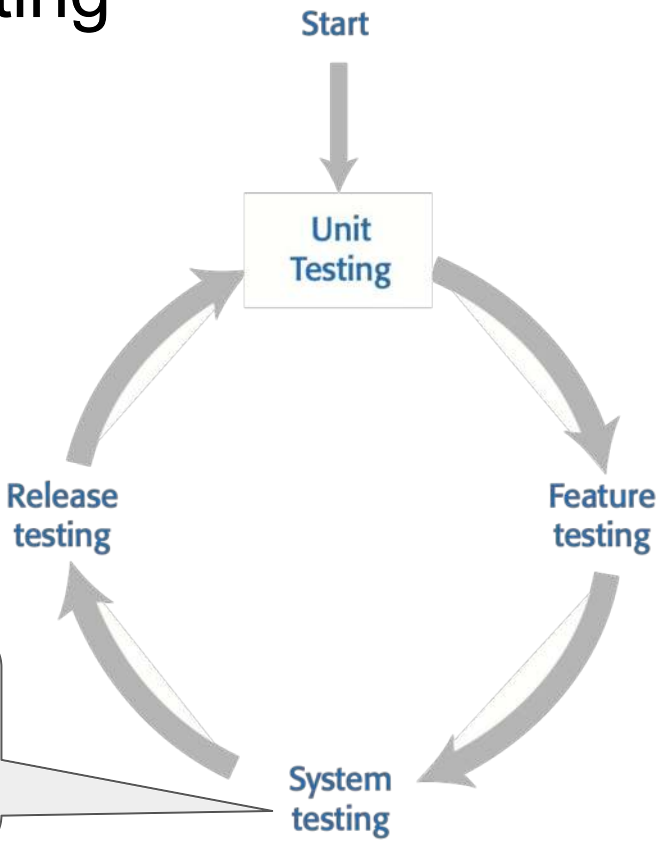


Functional testing



Code units are integrated to create features.

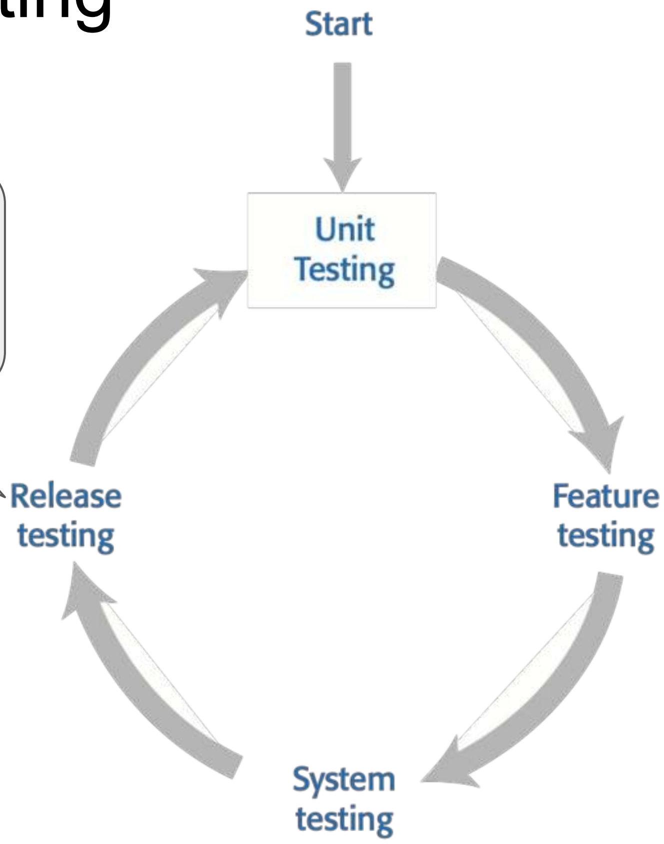
Functional testing



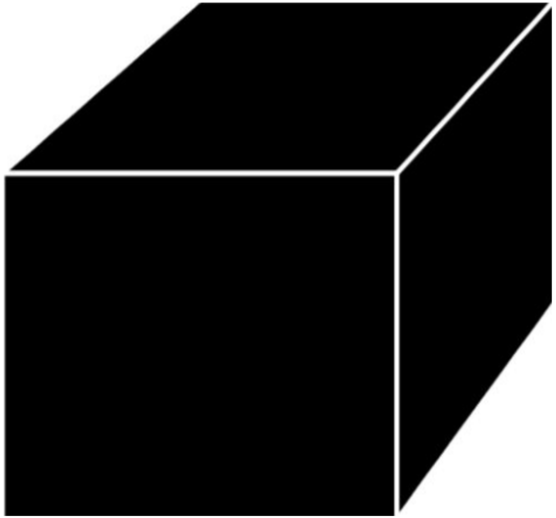
Code units are integrated to create a working version of the system. This checks the interaction of features.

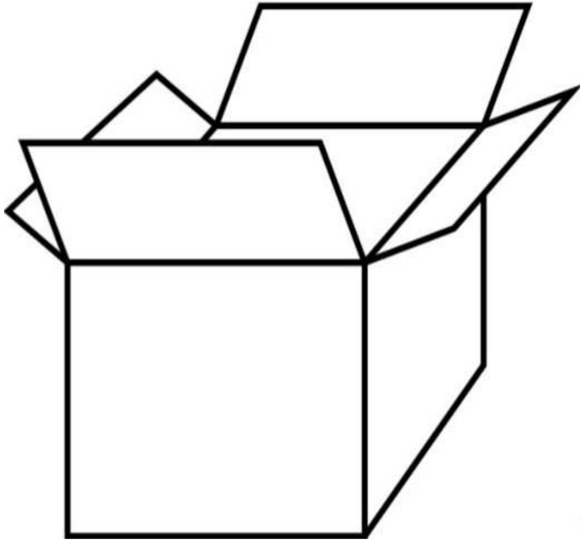
Functional testing

The system is packaged for release to customers and the release is tested to check that it operates as expected.

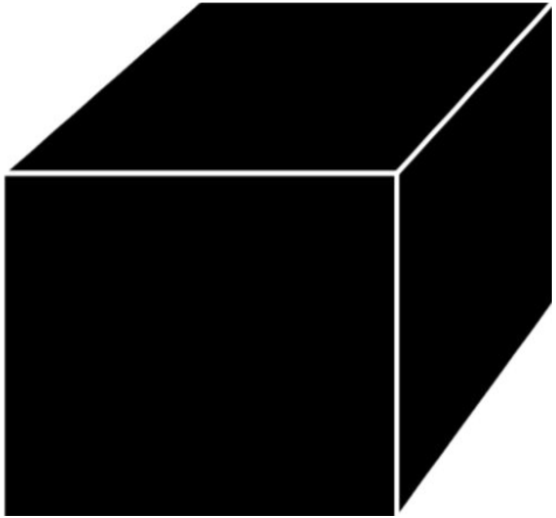


Black-box testing and white-box testing

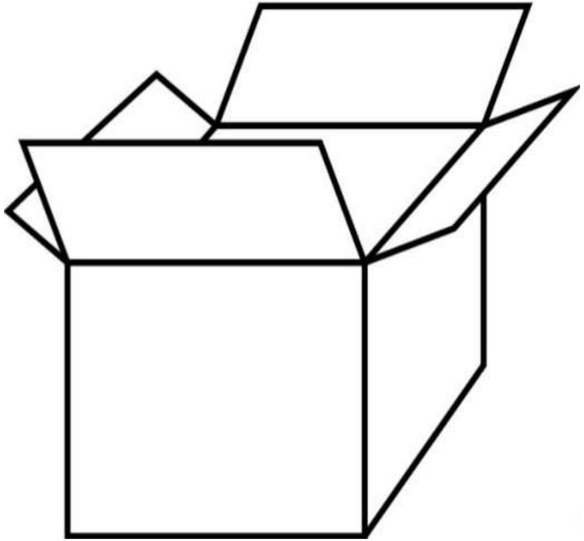




Black-box testing and white-box testing

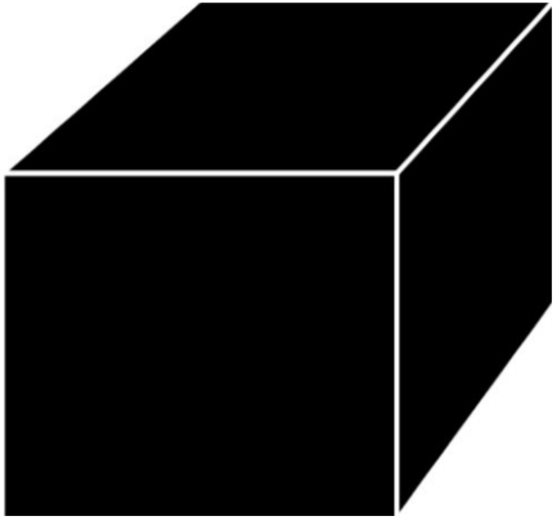


- equivalence partitioning
- boundary value analysis

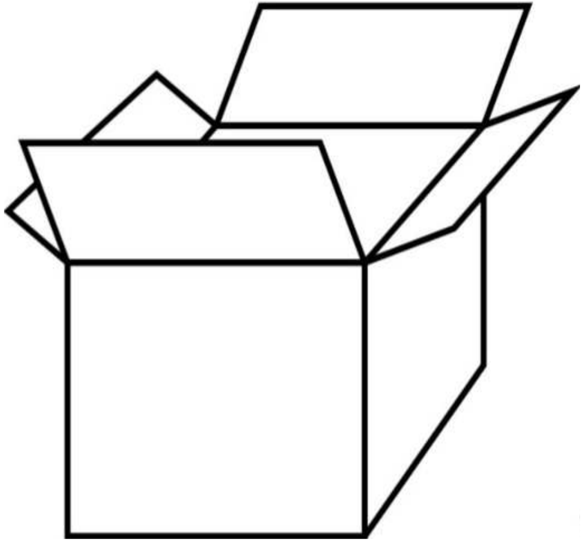


- statement coverage testing
- branch coverage testing

Black-box testing and white-box testing



- **equivalence partitioning**
- **boundary value analysis**



- statement coverage testing
- branch coverage testing

Black-box testing: equivalence partitioning

This involves the identification of sets of inputs that will be treated in the same way by the program.

Black-box testing: equivalence partitioning

Example: testing a function that checks simple names according to the following rules:

1) The length of the name should be between 2 and 40 characters
2) The only nonalphabetic separator characters allowed are hyphen and apostrophe.
3) Names must start with a letter.

Black-box testing: equivalence partitioning

<u>Correct names 1</u>: The input only includes alphabetic characters and is between 2 and 40 characters long.
<u>Correct names 2</u>: The input only includes alphabetic characters, hyphens or apostrophes and is between 2 and 40 characters long.
<u>Incorrect names 1</u>: The input is between 2 and 40 characters long but includes disallowed characters.
<u>Incorrect names 2</u>: The input includes allowed characters but is either a single character or is more than 40 characters long.
<u>Incorrect names 3</u>: The input is between 2 and 40 characters long but the first character is a hyphen or an apostrophe.

Black-box testing: boundary value analysis

Boundary value analysis is based on testing the boundary values of valid and invalid partitions.

The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition.

Black-box testing: boundary value analysis

For each input, we check:

- Minimum value
- Just below the minimum value
- Just above the minimum value
- Maximum value
- Just below the maximum value
- Just above the maximum value

Black-box testing: boundary value analysis

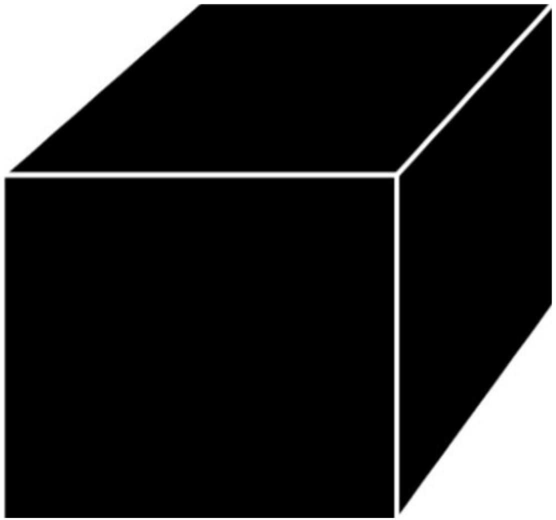
Example: consider a system that accepts ages from 18 to 56.

Black-box testing: boundary value analysis

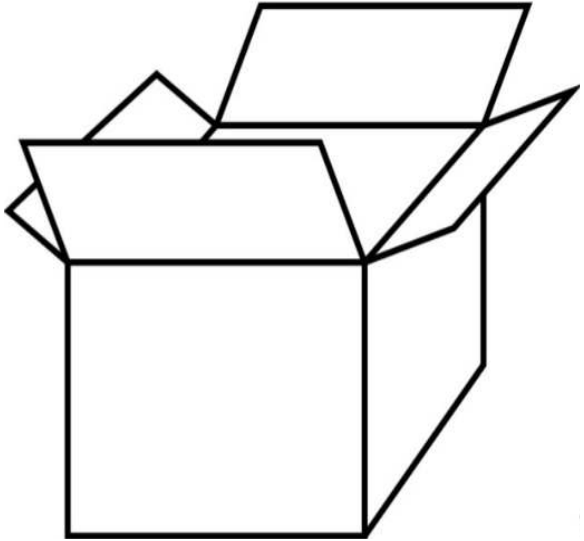
Example: consider a system that accepts ages from 18 to 56.

Minimum value: 18
Just below the minimum value: 17
Just above the minimum value: 19
Maximum value: 56
Just below the maximum value: 55
Just above the maximum value: 57

Black-box testing and white-box testing



- equivalence partitioning
- boundary value analysis



- **statement coverage testing**
- **branch coverage testing**

White-box testing: statement coverage testing

Statement coverage is a technique in which all the executable statements in the source code are executed at least once.

Statement coverage =

number of executed statements / total number of statements x 100

White-box testing: statement coverage testing

```
prints(int a, int b) {
   int result = a + b;
   if (result > 0)
      print ("Positive", result)
   else
      print ("Negative", result)
   }
```

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```

Input: a = 3, b = 9

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```

Input: a = 3, b = 9

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
          print ("Positive", result)
    else
          print ("Negative", result)
    }
```

Input: a = 3, b = 9

Statement coverage: 5/7 = 71%

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```

Input: a = -3, b = -9

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```

Input: a = -3, b = -9

White-box testing: statement coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```

Input: a = -3, b = -9

Statement coverage: 6/7 = 85%

White-box testing: branch coverage testing

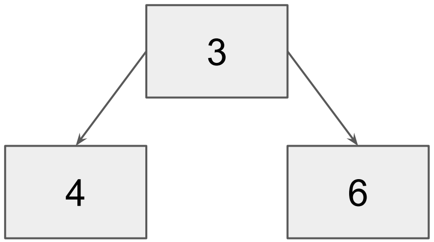
A branch is the outcome of a decision (e.g., an if statement and a loop control statement), so branch coverage simply measures which decision outcomes have been tested.

Branch coverage =

number of executed branches / total number of branches x 100

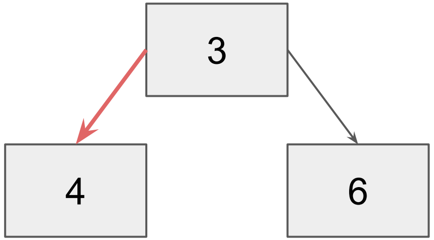
White-box testing: branch coverage testing

```
prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        print ("Positive", result)
    else
        print ("Negative", result)
    }
```



White-box testing: branch coverage testing

```
1 prints(int a, int b) {
2     int result = a + b;
3     if (result > 0)
4         print ("Positive", result)
5     else
6         print ("Negative", result)
7     }
```

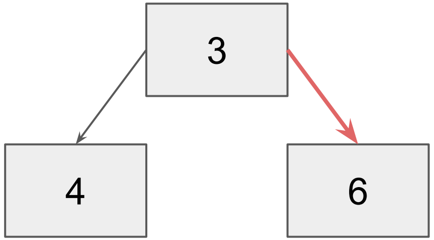


Input: a = 3, b = 9

Branch coverage: 1/2 = 50%

White-box testing: branch coverage testing

```
prints(int a, int b) {
   int result = a + b;
   if (result > 0)
      print ("Positive", result)
   else
      print ("Negative", result)
   }
```



Input: a = -3, b = -9

Branch coverage: 1/2 = 50%

Statement and branch coverage

If statement coverage is 100%, is branch coverage 100%?

If branch coverage is 100%, is statement coverage 100%?

Statement and branch coverage

```
if (passwordEnteredOK()) {
    enterSystem();
}
```

A given test entered the if block.

Statement coverage = 100%
Branch coverage = 50%

Statement and branch coverage

```
if (passwordEnteredOK()) {
    enterSystem();
}
/* Invisible else part
else {
  // do nothing
}
*/
```

A given test entered the if block.

Statement coverage = 100%
Branch coverage = 50%

Statement and branch coverage

```
if (passwordEnteredOK()) {
    enterSystem();
}
/* Invisible else part
else {
  // do nothing
}
*/
```

With statement coverage you just check that with a correct password one can use the system.
With branch coverage you also test that with an incorrect password one will not enter the system.

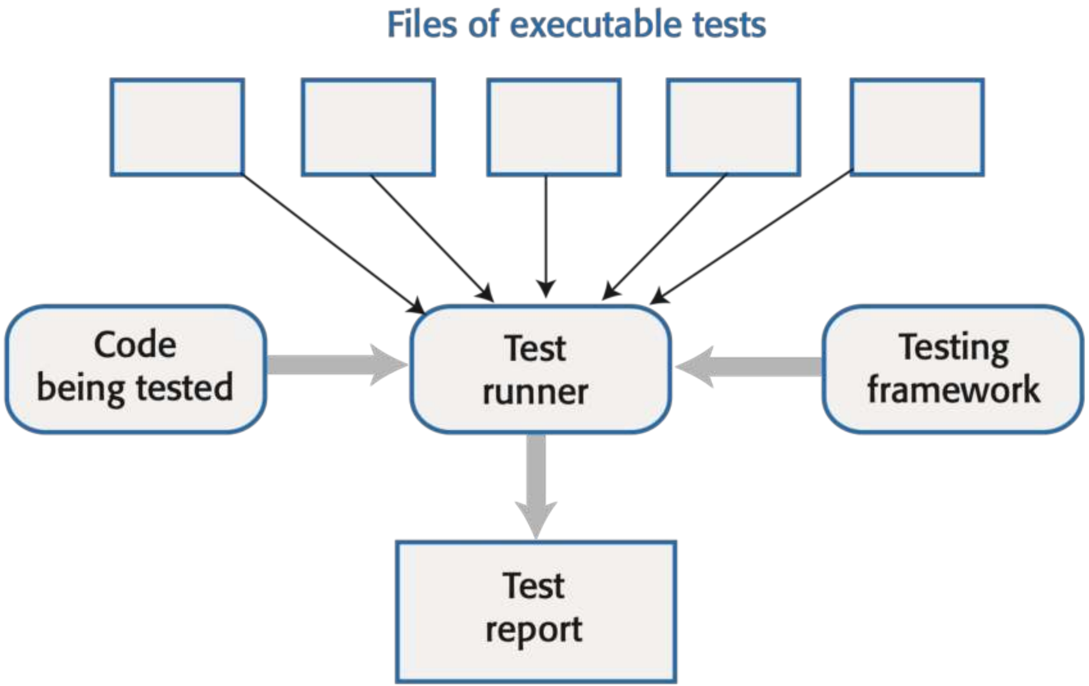
Test automation

Automated testing is based on the idea that tests should be executable.

An executable test includes <u>the input data</u> to the unit that is being tested, <u>the expected result</u>, and <u>a check</u> that the unit returns the expected result.

You run the test and the test passes if the unit returns the expected result.

Test automation



Automated tests

It is good practice to structure automated tests into three parts:

<u>Arrange</u>: You set up the system to run the test. This involves defining the test parameters.

<u>Action</u>: You call the unit that is being tested with the test parameters.

<u>Assert</u>: You make an assertion about what should hold if the unit being tested has executed successfully.

Automated tests

```
class TestInterestCalculator(unittest.TestCase):

    def test_zeroprincipal(self):
        #Arrange
        p = 0; r = 3; n = 31
        result_should_be = 0
        #Action
        interest = interest_calculator(p, r, n)
        #Assert
        self.assertEqual(result_should_be, interest)
```

Test-driven development

Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.

Test-driven development works best for the development of individual program units.

Test-driven development steps

Given a functionality the system should have:

Identify partial implementation: Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.

Write mini-unit tests: Write one or more automated tests for the mini-unit that you have chosen for implementation.

Write an incomplete code that will fail test: Write incomplete code that will be called to implement the mini-unit. You know this will fail at first.

Test-driven development steps

Given a functionality the system should have (continuation):

Run all existing automated tests: All previous tests should pass. The test for the incomplete code should fail.

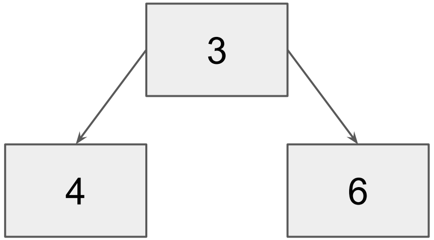
Implement code that should cause the failing test to pass: Write code to implement the mini-unit, which should cause it to operate correctly.

Rerun all automated tests: If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.

Discussion

How would you instrument this code to measure branch coverage?

```
1 prints(int a, int b) {
2     int result = a + b;
3     if (result > 0)
4         print ("Positive", result)
5     else
6         print ("Negative", result)
7     }
```



For reflection: statement and branch coverage

If both statement coverage and branch coverage, and all tests pass, can we say that the program is fully correct?

To keep in mind

No single test case will be perfect: multiple tests should be created.

Both valid and invalid inputs should be considered.

Test cases without assertions do not really check if the program is correct.

TODO

Reading

Exam material:

- Ian Sommerville, "Engineering Software Products", 2020: Chapter 9 - “Testing”

Recommended material:

- Maurício Aniche, “Effective Software Testing: A Developer's Guide”, 2022

Takeaways?

?