Lab II: Coverage-based Test Design and Input Domain Model
Code Coverage and Input Space Partitioning (ISP) Test Inputs

Software Testing and QA (COE891)

Week 4 and Week 5

1 Part I: Test Coverage

Test coverage is defined as a metric in Software Testing that measures the amount of testing performed by a set of test. It will include gathering information about which parts of a program are executed when running the test suite to determine which branches of conditional statements have been taken.

1.1 Objectives

- Finding the area of a requirement not implemented by a set of test cases.
- Creating additional test cases to increase code testing coverage.
- Identifying a quantitative measure of test coverage, which is an indirect method for quality check.
- Identifying meaningless test cases that do not increase coverage.

1.2 Benefits of Test Coverage

In simple terms, it is a technique to ensure that your tests are testing your code or how much of your code you exercised by running the test.

- It can assure the quality of the test
- It can help identify what portions of the code were actually touched for the release or fix
- It can help to determine the paths in your application that were not tested
- Prevent Defect leakage
- Time, scope and cost can be kept under control
- Defect prevention at an early stage of the project lifecycle
- It can determine all the decision points and paths used in the application, which allows you to increase test coverage
- Gaps in requirements, test cases and defects at the unit level and code level can be found in an easy way

1.3 Clover: A Code Coverage Tool

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage.

In most cases, code coverage system gathers information about the running program. It also combines that with source code information to generate a report about the test suite's code coverage.

1.3.1 Code Coverage Usage

Here, are some prime reasons for using code coverage:

- It helps you to measure the efficiency of test implementation.
- It offers a quantitative measurement.
- It defines the degree to which the source code has been tested.

1.3.2 How to Calculate Test Coverage?

Generally, to calculate test coverage, you need to follow the below-given steps:

1. The total lines of code in the piece of software quality you are testing.
2. The number of lines of code on which all test cases currently execute.

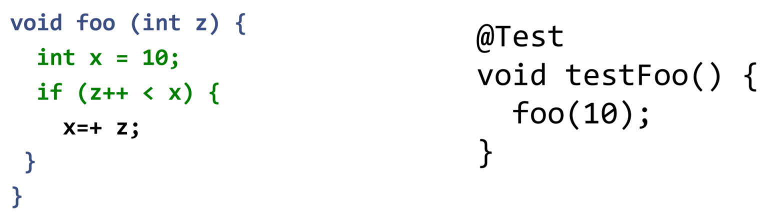
1.4 Statement Coverage

Statement Coverage is a white box testing technique in which all the executable statements in the source code are executed at least once. It is used for calculation of the number of statements in source code which have been executed. The main purpose of Statement Coverage is to cover all the possible paths, lines and statements in source code.

Statement coverage is used to derive scenario based upon the structure of the code under test.

$$Statement\ Coverage = \frac{\#Executed\ Statements}{Total\ Number\ of\ Statements} \times 100 \tag{1}$$

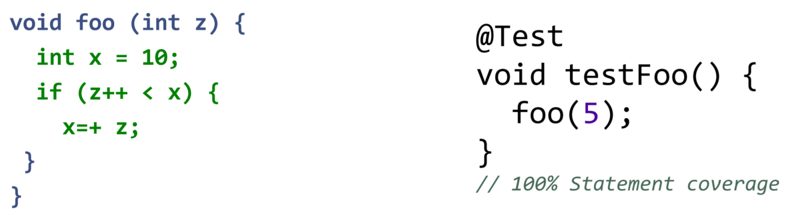
So, the adequacy criterion is that each statement (or node in the CFG) must be executed at least once:



```
void foo (int z) {
  int x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

Figure 1



```
void foo (int z) {
  int x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(5);
}
// 100% Statement coverage
```

Figure 2

Let's understand this with an example, how to calculate statement coverage. Here we are taking two different scenarios to check the percentage of statement coverage for each scenario.

```
public void Prints(int a, int b) {
    int result = a + b;
    if (result > 0)
        Print("Positive", result)
    else
        Print("Negative", result)
    }
```

Scenario 1:
If $a = 3$, $b = 9$:
Number of executed statements = 5, Total number of statements = 7
Statement Coverage: $\frac{5}{7} = 71\%$

Scenario 2:
If $a = -3$, $b = -9$:
Number of executed statements = 6, Total number of statements = 7
Statement Coverage: $\frac{6}{7} = 85\%$

But overall if you see, all the statements are being covered by 2nd scenario's considered, so we can conclude that overall statement coverage is 100%.

1.4.1 What is covered by Statement Coverage?

- Unused Statements
- Dead Code
- Unused Branches
- Missing Statements

1.5 Branch Coverage

Branch Coverage is a white box testing method in which every outcome from a code module(statement or loop) is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once. It helps to measure fractions of independent code segments and to find out sections having no branches. For example, if the outcomes are binary, you need to test both True and False outcomes. The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{\#Executed\ Branches}{Total\ Number\ of\ Branches} \times 100 \tag{2}$$

To learn branch coverage, let's consider the same example used earlier:

```
public void demo(int a) {
    If (a > 5)
        a = a * 3
    Print(a)
    }
```

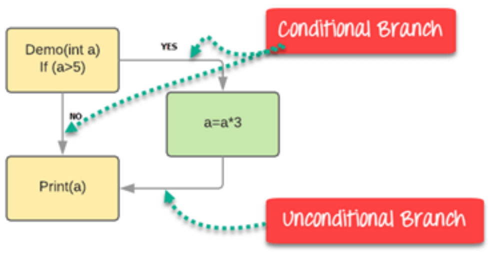
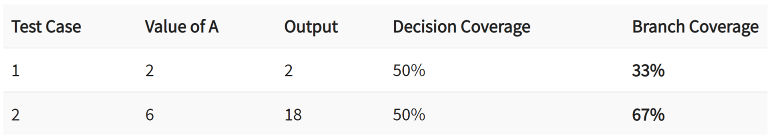


Figure 3: Branch Coverage graph

Branch Coverage will consider unconditional branch as well.



Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	33%
2	6	18	50%	67%

Figure 4: Branch coverage percentage per test case

1.5.1 Advantages of Branch Coverage:

- Allows you to validate-all the branches in the code
- Helps you to ensure that no branched lead to any abnormality of the program's operation
- Branch coverage method removes issues which happen because of statement coverage testing
- Allows you to find those areas which are not tested by other testing methods
- It allows you to find a quantitative measure of code coverage
- Branch coverage ignores branches inside the Boolean expressions

1.6 Installing The Plugin

Select from the menu "Help → Install new software". Click "Add" button and enter ´http://openclover.org/update´then click OK. Select "Clover 4" and "Clover 4 Ant Support" features. Disable the "Contact all update sites..." checkbox (for faster installation). Click "Next" button twice. Accept license agreement, click "Finish", click "OK" for warning about unsigned content, click "Restart now".

To Enable Clover for Java project, Right click on a project in "Package Explorer" view, select "Clover → Enable on this Project". If you wish to enable Clover for multiple projects at once, choose "Enable/Disable on...".

Four Clover views will be opened automatically:

- Coverage Explorer
- Test Run Explorer
- Clover Dashboard
- Test Contributions

You can always open them from "Window → Show View → Other ... → Clover". In your Java Editor (Eclipse), in order to view coverage information on a line-by-line basis, Clover adds colored annotations to your project's Java source code editors.

2 Part II: Input Space Partitioning (ISP)

Practically, due to time and budget considerations, it is not possible to perform exhausting testing for each set of test data, especially when there is a large pool of input combinations. We need an easy way or special techniques that can select test cases intelligently from the pool of test-case, such that all test scenarios are covered. We use two techniques – Equivalence Partitioning and Boundary Value Analysis testing techniques to achieve this.

2.1 Objectives

- Checking whether the software application accepts inputs within the acceptable range and delivers required output.
- Getting familiar with one of the most important White Box Testing methods.
- Verifying that the system should not accept inputs, conditions and indices outside the specified or valid range.

2.2 Equivalence Class Partitioning

Equivalence partitioning is a Test Case Design Technique to divide the input data of software into different equivalence data classes. Test cases are designed for equivalence data class. The equivalence partitions are frequently derived from the requirements specification for input data that influence the processing of the test object. A use of this method reduces the time necessary for testing software using less and effective test cases.

$$EquivalencePartitioning = EquivalenceClassPartitioning = ECP$$

It can be used at any level of software for testing and is preferably a good technique to use first. In this technique, only one condition to be tested from each partition. Because we assume that, all the conditions in one partition behave in the same manner by the software. In a partition, if one condition works other will definitely work. Likewise we assume that, if one of the condition does not work then none of the conditions in that partition will work. So, equivalence partitioning is a testing technique where input values set into classes for testing.

So, in Boundary Testing, Equivalence Class Partitioning plays a good role Boundary Testing comes after the Equivalence Class Partitioning. Equivalence Partitioning or Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. In this technique, input data units are divided into equivalent partitions that can be used to derive test cases which reduces time required for testing because of small number of test cases. It divides the input data of software into different equivalence data classes. You can apply this technique, where there is a range in the input field.

2.3 Boundary Value Analysis

Boundary Value Analysis (BVA) is a technique employed for black-box testing (also called Dynamic Testing). As a software tester, you always want to maximize the probability of finding errors. The test cases designed with boundary input values have high chances to find errors. This is how BVA can be a good choice to test software.

In other words, boundary value analysis is a test case design technique to test boundary value between partitions (both valid boundary partition and invalid boundary partition). A boundary value is an input or output value on the border of an equivalence partition, includes minimum and maximum values at inside and outside boundaries. Normally Boundary value analysis is part of stress and negative testing.

Using Boundary Value Analysis technique tester creates test cases for required input field. For example; an Address text box which allows maximum 500 characters. So, writing test cases for each character once will be very difficult so that will choose boundary value analysis.

Therefore, Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values. So, these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called “boundary testing”. The basic idea in normal boundary value testing is to select input variable values at their:

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

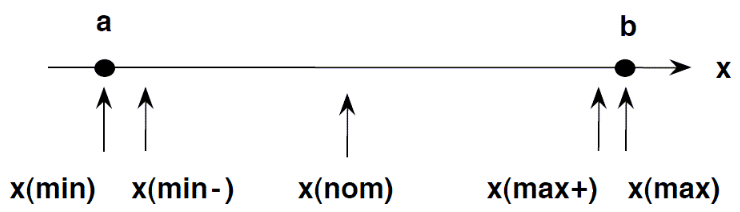


Figure 5: Boundary Value Analysis

Let’s consider the behavior of Ordering a Pizza. Suppose pizza values 1 to 10 is considered valid. While value 11 to 99 are considered invalid for order and an error message will appear, “Only 10 Pizza can be ordered” (the maximum value should be under 100). So, here is the test condition.

Any Number greater than 10 entered in the Order Pizza field (let say 11) is considered invalid. Any Number less than 1 that is 0 or below, then it is considered invalid. Numbers 1 to 10 are considered valid. Any 3 Digit Number say -100 is invalid. We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.

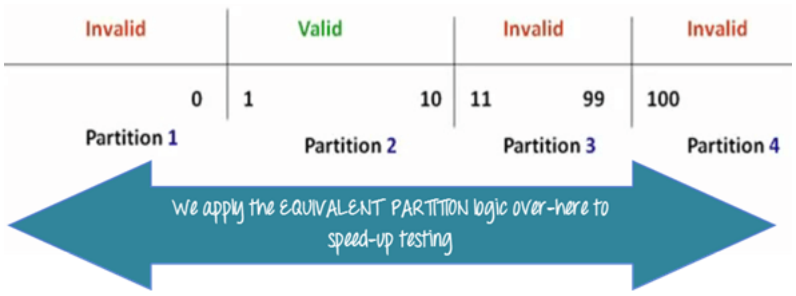


Figure 6: Boundary Value Analysis

The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is that if one condition/value in a partition passes all others will also pass. Likewise, if one condition in a partition fails, all other conditions in that partition will fail. In Boundary Value Analysis, you test boundaries between equivalence partitions.

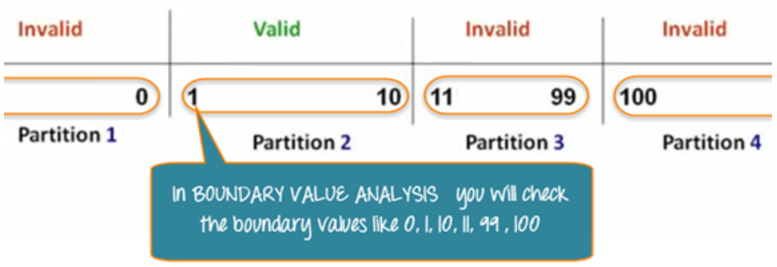

Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100

Figure 7: Boundary Value Analysis

In our earlier equivalence partitioning example, instead of checking one value for each partition, you will check the values at the partitions like 0, 1, 10, 11, and so on. As you may observe, you test values at both valid and invalid boundaries. Boundary Value Analysis is also called range checking. Equivalence partitioning and boundary value analysis (BVA) are closely related and can be used together at all levels of testing. There are some benefits in using BVA:

- This testing is used to reduce a very large number of test cases to manageable chunks.
- Very clear guidelines on determining test cases without compromising on the effectiveness of testing.
- Appropriate for calculation-intensive applications with a large number of variables/inputs.

For example, suppose a password field accepts minimum 6 characters and maximum 10 characters. That means results for values in partitions 0-5, 6-10, 11-14 should be considered.

3 Assignments

Q1: Using the Java code files provided for you in the zip folder "Money.rar", run the code and its testing programs to see and analyze the test coverage of the code for each testing components. Use Clover tool to analyze the coverage of the code by different test cases.

- Submit the pictures/snapshots of Clover test coverage results.
- If you do not achieve 100% code coverage by existing test cases, try to arbitrarily modify (add/remove) test cases for better and more code coverage (close to 100% as much as possible).

Q2: Calculate the statement coverage and branch coverage based on the following program for the following scenarios and answer the corresponding questions:

```java
public void func(int a, int b) {
    if (b > a) {
        b = b - a;
        System.out.println(b); }
    else if (a > b) {
        b = a - b;
        System.out.println(b); }
     else
        System.out.println(0);
}
```

- $a = 2$ and $b = 3$.
- $a = 3$ and $b = 2$.
- Could you cover all the statements? If not, add a few inputs required to cover 100% statement coverage. Submit all the calculations and justifications.

Q3: Write a triangle classification program with the class name *Triclass* which contains a public static method named *classify* that inputs three integers representing the lengths of the three sides of a triangle named X, Y, and Z, each one in a range between 1 to 10, and outputs a String which is showing the type of the triangle: scalene, equilateral, isosceles, and invalid ("not a triangle"). You should consider the following items and instructions:

- For invalid or any kinds of valid triangle, you should check "triangle inequality" for every two sides of a triangle. Then, write a Java test program with the class name *TriclassTest* to test your program.
- In your test program, it should print a one-time message at first which is "Testing started" and a one-time message at last which is "Testing is finished".
- Before and after running each test, it should print messages "#Test_Number - started" and "#Test_Number - finished" containing a number associated with each test method, respectively.
- Verify the classification results of your program with different boundaries of input values (zero is not allowed as the length of a side).
- Each time, you should fix two sides to the value 5, and change the third side to different boundaries.
- First, you should implement the test methods and select different inputs to test the inputs that are classified as a triangle. Then, design the tests containing the inputs that are NOT classified as a triangle (invalid).

4 Submission and Marking Instructions

4.1 Submissions

Once all required tasks are completed, you should submit your lab assignment. Follow the instructions below for a valid submission:

- After checking the accuracy and completeness of your assignment tasks, you should export and submit the FULL Eclipse project/folder as a ZIP file containing packages and source code files (for implementation and coding tasks/questions).

  - Individual files (e.g. java or class files) will NOT be accepted as a valid submission since your submitted package or Java project should be run completely and successfully by itself.
- You MUST submit a single PDF file containing required description or explanation for each of the assignment tasks/questions separately including any required justification, graphs, diagrams, test requirements/cases, calculation, pictures/snapshots from tools or IDE, test results, and so forth.
- The **submission deadline** for this lab (Lab II) is the corresponding lab session in **Week 6**.
- The lab demo and questioning-answering will be held during the lab sessions of the corresponding submission weeks.

4.2 Marking Scheme

- This lab (Lab II) constitutes 4% of your entire grade for this course.
- All assignment tasks/questions in each lab have the same grade.
- The grade for each lab is constituted from 50% for the lab submissions, 10% for the lab attendance, and 40% for demo and questioning-answering during the lab session.
- Note that all the labs constitute 25% of your entire grade for this course.