JUNIT - CFG TESTING

TABLE OF CONTENTS

INTRODUCTION

- Control flow graphs (CFGs) are a fundamental tool for understanding software programs and analyzing their behavior. A CFG is a graphical representation of the control flow of a program, showing how the program executes from start to finish.

- Testing of CFGs is essential to ensure that the program behaves correctly and meets the required specifications. Testing helps to identify defects, errors, and vulnerabilities in the program and to ensure that it behaves as intended.

- JUnit is a popular testing framework in Java used for unit testing. JUnit helps to automate the testing process and provides an efficient way to validate the behavior of a program.

UNDERSTANDING CONTROL FLOW GRAPHS

- A control flow graph is a directed graph that represents the control flow of a program. The nodes in the graph represent basic blocks of code, and the edges represent the control flow between these blocks

- A basic block is a sequence of statements in a program that always execute in the same order, without any jumps or branches. A basic block can be thought of as a straight line of code that has a single entry point and a single exit point

- A path in a control flow graph is a sequence of edges that starts at the entry node and ends at the exit node. A path represents a sequence of basic blocks that are executed in a particular order

- Control flow graphs are useful for understanding the behavior of a program, identifying potential defects, and optimizing the program's performance

TESTING OF CONTROL FLOW GRPAHS

- Testing of control flow graphs involves the verification of the program's behavior under different input conditions. Testing aims to identify defects, errors, and vulnerabilities in the program and ensure that it meets the required specifications.

- Different types of testing can be performed on control flow graphs, such as statement coverage, branch coverage, and path coverage.

- Statement coverage involves ensuring that every statement in the program is executed at least once during testing. Branch coverage ensures that every possible branch in the program is executed at least once. Path coverage aims to ensure that every possible path through the program is executed at least once.

- Each type of testing has its advantages and disadvantages. For example, statement coverage is easy to achieve but may miss some subtle defects. Path coverage is more thorough but can be challenging to achieve in practice.

INTRODUCTION TO JUNIT

- JUnit is a popular testing framework in Java used for unit testing. JUnit provides a simple and efficient way to test the behavior of a program by automating the testing process.

- JUnit provides a set of annotations and assertions that simplify the process of writing tests. Annotations are used to identify methods that are test cases, while assertions are used to verify the behavior of the program under test.

- JUnit is compatible with popular IDEs such as Eclipse and IntelliJ IDEA, making it easy to integrate into the development process.

TESTING CONTROL FLOW GRAPHS USING JUNIT

- Writing JUnit tests for control flow graphs involves identifying the basic blocks, branches, and paths in the program and ensuring that each one is executed at least once during testing.

- To write a JUnit test, we first identify the input conditions that we want to test. We then write a test method that sets up the input conditions, executes the program under test, and verifies the output using JUnit assertions.

- JUnit provides a set of assertion methods, such as assertEquals, assertTrue, and assertFalse, that simplify the process of verifying the behavior of the program under test.

- JUnit also provides annotations, such as @Before and @After, that are used to set up and tear down the test environment.

TECHNIQUES FOR TESTING CFG IN JUNIT

- JUnit provides several advanced techniques that can be used to test complex control flow graphs. For example, JUnit supports parameterized tests, where a single test method can be executed with multiple sets of input data.

- JUnit also supports test suites, where multiple test cases can be grouped together and executed as a single unit. Test suites can be used to test different parts of the program and ensure that they work together correctly.

- Another advanced technique is the use of mock objects, which are objects that simulate the behavior of real objects in the program. Mock objects can be used to isolate the behavior of a particular part of the program and test it independently of other parts.

GENERATING TEST CASE

- Test cases can be generated manually or automatically for control flow graphs.
- Manual test case generation involves analyzing the control flow graph and identifying test scenarios that exercise different parts of the graph.
- Automatic test case generation involves using tools such as model checkers or symbolic execution to explore the control flow graph and generate test cases that exercise different paths through the graph.

USING JUNIT TO TEST CONTROL FLOW GRAPHS

- JUnit is a popular unit testing framework for Java programs that can be used to test control flow graphs.
- To use JUnit for testing control flow graphs, you need to define test methods that exercise different parts of the graph.
- JUnit provides several assertions that can be used to check the behavior of the program, such as assertEquals() and assertTrue().

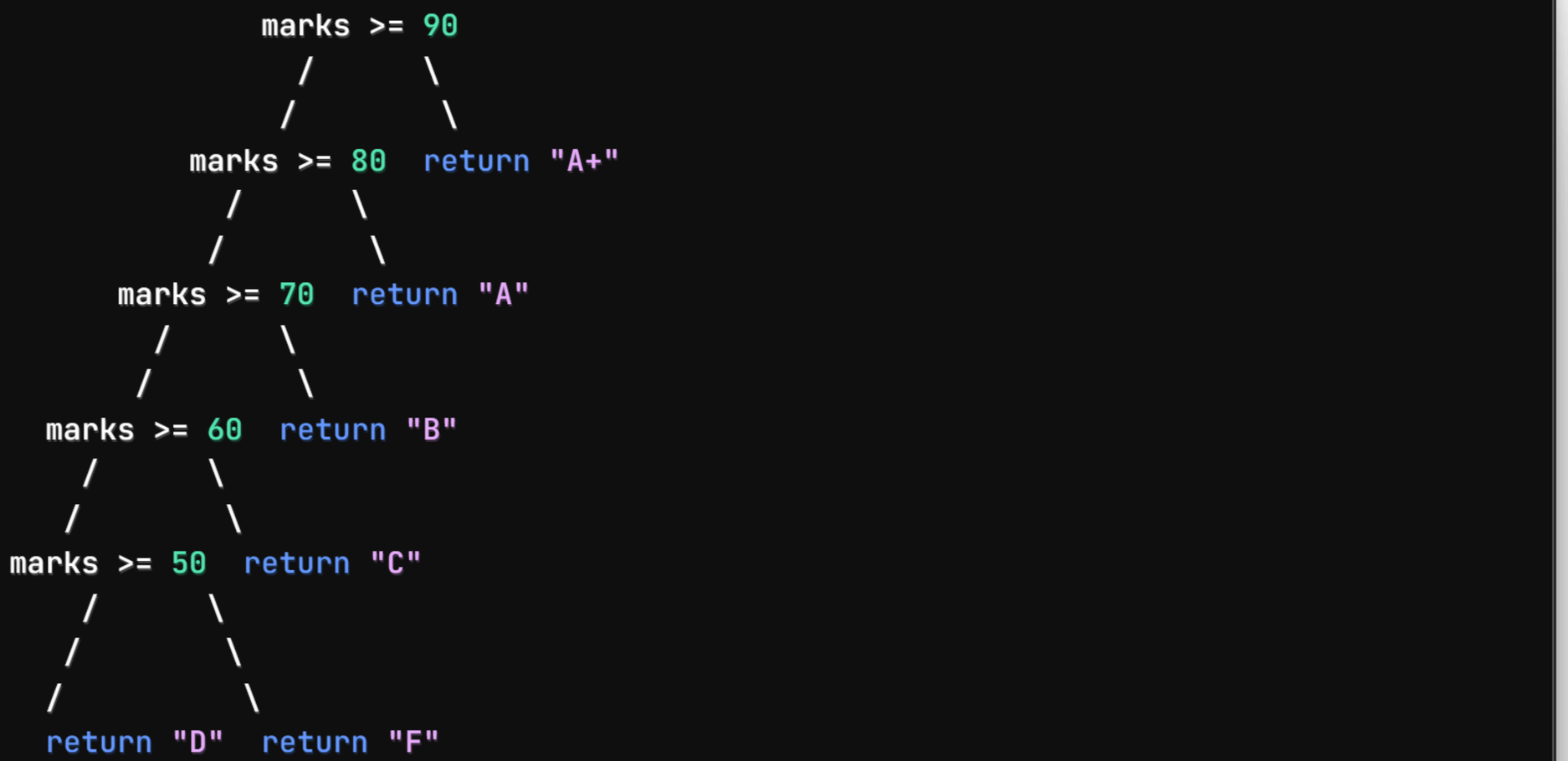
EXAMPLE

Grade - Java

```java
public static String getGrade(int marks) {
    if (marks >= 90) {
        return "A+";
    } else if (marks >= 80) {
        return "A";
    } else if (marks >= 70) {
        return "B";
    } else if (marks >= 60) {
        return "C";
    } else if (marks >= 50) {
        return "D";
    } else {
        return "F";
    }
}
```

EXAMPLE

Grade - Java



EXAMPLE

Grade - Java

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class GradeTest {

    @Test
    public void testAPlus() {
        String grade = Grade.getGrade(95);
        assertEquals("A+", grade);
    }

    @Test
    public void testA() {
        String grade = Grade.getGrade(85);
        assertEquals("A", grade);
    }

    @Test
    public void testB() {
        String grade = Grade.getGrade(75);
        assertEquals("B", grade);
    }

    @Test
    public void testC() {
        String grade = Grade.getGrade(65);
        assertEquals("C", grade);
    }

    @Test
    public void testD() {
        String grade = Grade.getGrade(55);
        assertEquals("D", grade);
    }

    @Test
    public void testF() {
        String grade = Grade.getGrade(45);
        assertEquals("F", grade);
    }
}
```

THANK YOU