Software-Testing-Module-3-Important-Topics-PYQs

For more notes visit

https://rtpnotes.vercel.app

1. Explain structural graph coverage for Design Elements.

Structural graph coverage means **checking how well we have tested the structure (flow) of a software design**, by representing it as a **graph** and covering different parts of it during testing.

In this, **design elements** like **procedures**, **functions**, **modules**, or **states** are treated as **nodes**, and the **flow of control or data** between them is treated as **edges**.

We draw a **graph** from the design and then plan our tests so that different parts (nodes and edges) of this graph are covered properly.

Key Design Elements in Structural Graph Coverage

1. **Procedures / Modules**
   - Each function or module is a **node**.
   - If one module calls another, that's an **edge** connecting them.
2. **States and Transitions**
   - In systems like **state machines** (for example, ATMs, login systems), each **state** is a **node**, and moving from one state to another is an **edge**.

3. **Data Flow**
    - How **data moves** through the system can also be mapped — for example, from one variable to another.

Type of Coverage	What it means	Example
Node Coverage	Make sure every **node** (function/module/state) is tested.	Test each screen in an app.
Edge Coverage	Make sure every **edge** (connection or flow) is tested.	Test moving from login to home page.
Edge-Pair Coverage	Test **pairs of edges** together, checking sequences.	Test login → home → profile screens.
Path Coverage	Try to test **all possible paths** from start to end.	Test all ways to go through a checkout system.

Example (Easy)

Imagine a simple Login System design:

- Start → Login Screen → Home Screen → Logout → End

This can be drawn as a graph like this:

```
Start → Login → Home → Logout → End
```

In testing:

- **Node coverage** means: visit Start, Login, Home, Logout, End.
- **Edge coverage** means: test transitions like Start → Login, Login → Home, etc.
- **Path coverage** means: test full path like Start → Login → Home → Logout → End.

Why Structural Graph Coverage is Important?

- It **finds missing links** in the design (maybe you forgot to handle a failed login?).
- It ensures **full coverage** of how the software will behave.
- It catches **bugs early** even before coding starts!

2. Define Tour, Tour with side trips and Tour with Detours.

1. Tour (Simple Tour)

- A **Tour** is **a path through a graph** where you **visit nodes and edges** according to the normal flow.
- In testing, it means you are **following the usual designed paths** without taking any extra or unexpected steps.

Simple words:

- "Tour means walking through the normal routes exactly as they are supposed to be."
  Example:
- In an online shopping app, a tour could be:
- `Home → Select Product → Add to Cart → Checkout → Payment → Confirmation`

2. Tour with Side Trips

- A **Tour with Side Trips** is like a **normal tour**, but **sometimes you take a small extra path (side trip)** and then **come back** to the main route.
- These "side trips" check extra conditions or alternative flows but still return to the main path.

Simple words:
"Tour with side trips means you mainly follow the normal path but sometimes take small extra steps and return."

Example:
In the shopping app:

- Main tour: `Home → Select Product → Add to Cart → Checkout`
- Side trip: While selecting a product, you **view product reviews** (side trip), and then return to add it to the cart.

3. Tour with Detours

- A **Tour with Detours** allows you to **take completely different paths** — not just small side trips.
- You **leave the main flow**, explore other routes, and **may or may not** return immediately to the main path.

- It tests **more complicated behaviors**.

Simple words:
"Tour with detours means you can take big alternative paths, not just small trips."

Example:
In the shopping app:

- While selecting a product, you instead **go to "Wishlist"**, browse it for some time, and then **maybe come back** to checkout later.

Term	Meaning (easy)	Example
Tour	Follow the normal path.	Home → Cart → Checkout.
Tour with Side Trips	Follow the normal path + small extra steps and come back.	Home → View Reviews → Add to Cart → Checkout.
Tour with Detours	Leave the main path and explore other flows.	Home → Wishlist → Cart → Checkout.

3. Explain any two methods for computing the cyclomatic complexity

What is Cyclomatic Complexity?

- Cyclomatic Complexity measures **how complex** a program or a flowchart is.
- It tells **how many independent paths** exist in the code.
- More paths = more complex code = more testing needed.

Method 1: Using Nodes and Edges

Formula:

$$\text{Cyclomatic Complexity (V)} = E - N + 2P$$

Where:

- **E** = Number of edges (arrows/lines)
- **N** = Number of nodes (circles/blocks)
- **P** = Number of connected parts (for one program, P = 1)

✅ **Steps:**

1. Draw the control flow graph (blocks and arrows).
2. Count total **edges** and **nodes**.
3. Use the formula.

👉 **Example:**

Suppose you have a flowchart with:

- Nodes (blocks) = 6
- Edges (arrows) = 8
  Then,

$$V = 8 - 6 + 2(1) = 4$$

Cyclomatic Complexity = **4**

Method 2: Using Decision Points (Conditions)

Formula:

$$\text{Cyclomatic Complexity (V)} = D + 1$$

Where:

- **D** = Number of decision points (like if, while, for, switch)

✅ **Steps:**

1. Look at the code or flowchart.
2. Count all the **decision points** (where choices are made).
3. Add 1.

👉 **Example:**

Suppose a program has:

- 3 if-conditions
- 1 while-loop

Total decision points = 4
So,

$$V = 4 + 1 = 5$$

Cyclomatic Complexity = **5**

4. Define Node coverage and prime path coverage in a control flow graph

Node Coverage (also called Statement Coverage)

- **Definition**:
  In **Node Coverage**, we make sure **every node** (which represents a statement or a block of code) in a **control flow graph (CFG)** is **visited at least once** during testing.

- **Goal**:
  To **ensure that every part of the program** (every statement) has been executed.
- **Example**:
  Imagine your program has 3 steps:

```
A → B → C
```

Prime Path Coverage

- **Definition**:
  A **prime path** is a path that is:
  - **Simple** (does not repeat nodes — except maybe at the start and end),
  - **Not a part of any longer simple path**.
    In **Prime Path Coverage**, we try to create test cases that **execute all prime paths** in the control flow graph.
- **Goal**:
  To **test more complicated paths** (including loops and branches), making the testing **more thorough**.
- **Example**:
  Suppose you have:

```
A → B → C → B → D
```

- Here, a prime path could be: **B → C → B** (because it loops back but doesn't repeat inside).
  Your tests should be designed to **cover such special paths** that involve branches and loops, not just simple straight-line code.

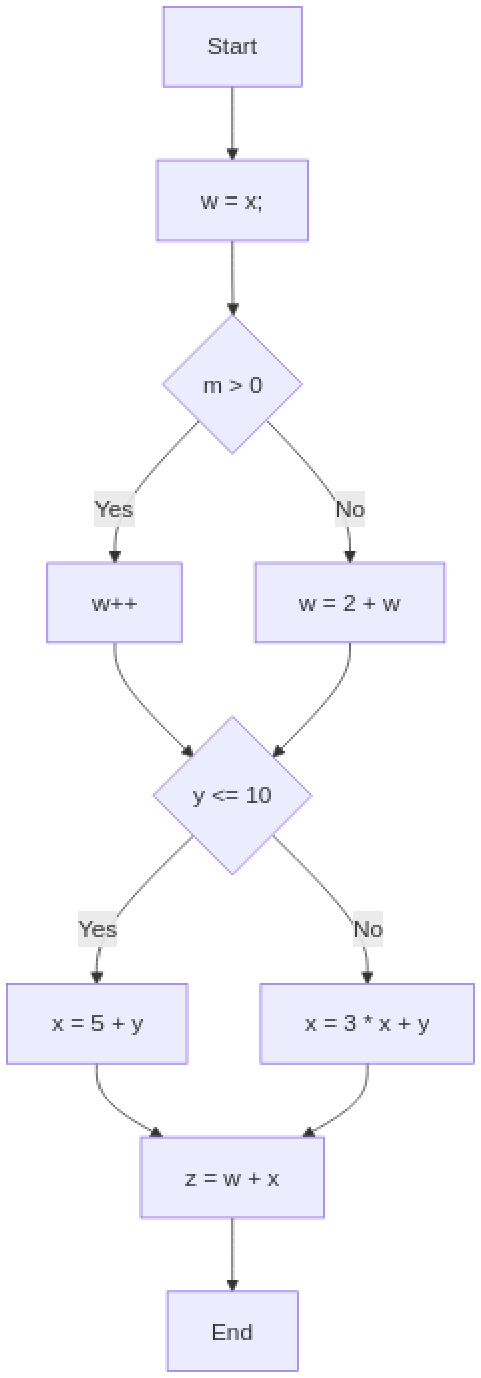
---

5. Draw control flow graph for the code fragment given below

```
w = x;
if(m>0){
    w++;
}
else{
        w = 2+w
}

if(y<=10){
    x=5+y
}
else{
    x = 3*x + y;
}

z = w+x;
```

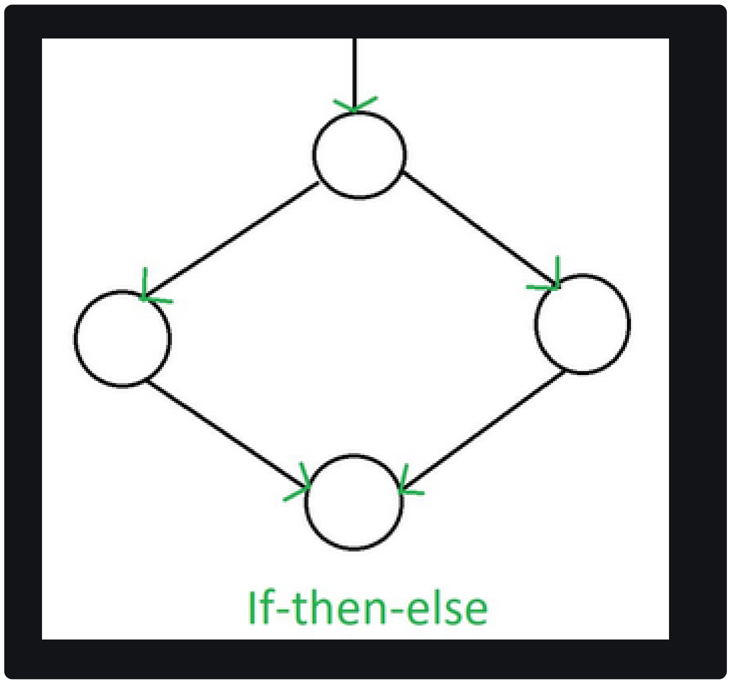


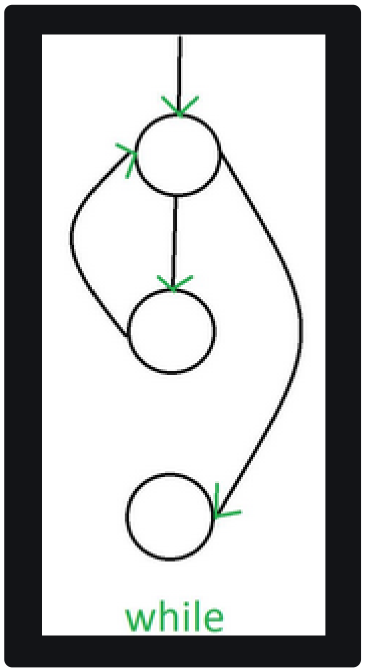
Explanation:

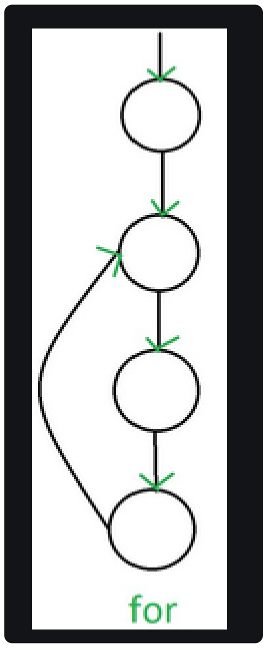
- **A**: The program starts.
- **B**: Assign the value of `x` to `w`.
- **C**: Check if `m > 0`.
    - If **Yes**, execute `w++` (D).
    - If **No**, execute `w = 2 + w` (E).
- **F**: Check if `y <= 10`.

    - If **Yes**, execute `x = 5 + y` (G).
    - If **No**, execute `x = 3 * x + y` (H).
- **I**: Assign `w + x` to `z`.
- **J**: End of the program.

6. Draw CFG fragment for a) Simple if b)Simple while loop c)Simple for loop







7. Explain path selection criteria with reference to

1. All path coverage criteria
2. Statement Coverage Criteria
3. Branch Coverage Criteria
4. Predicate Coverage Criteria

In software testing, path selection criteria help to determine which paths in the program's control flow should be tested to ensure that the software behaves as expected under various conditions. Path selection criteria focus on covering different execution paths or combinations of conditions within the code. Below is an explanation of common path selection criteria

1. All Path Coverage Criteria

All Path Coverage is a testing criterion that aims to execute every possible path through a program. It ensures that all possible combinations of decisions and conditions are tested.

- **Explanation**: A path in a program is a unique route that the control flow takes from the start to the end of a function. Each decision or loop in the code can create different paths, and the goal of all path coverage is to test every single path in the code.
- **Example**: Consider a program with two `if` conditions:

```
if (x > 5) {
    if (y < 10) {
        // Do something
    }
}
```

- The possible paths here are:
    1. `x > 5` and `y < 10`
    2. `x > 5` and `y >= 10`
    3. `x <= 5` and `y < 10`
    4. `x <= 5` and `y >= 10`
    All of these paths need to be tested in All Path Coverage.
- **Pros**: It provides the most comprehensive testing by covering every possible route.
- **Cons**: This method can be impractical for programs with many paths due to combinatorial explosion, making it hard to test every possible path.

2. Statement Coverage Criteria

Statement Coverage is the simplest path coverage criterion, focusing on ensuring that every statement in the code is executed at least once.

- **Explanation**: In statement coverage, you don't need to test every possible path or decision. The goal is just to make sure that each individual line of code (or statement) is executed during testing.
- **Example**: Consider the following code:

```
if (x > 5) {
    a = 10;
}
b = 20;
```

For statement coverage, the only requirement is that both a = 10 and b = 20 should be executed at least once.

Pros: Easy to implement and understand. It provides basic coverage of all statements.

Cons: It doesn't guarantee that the logic or decisions in the code are thoroughly tested, as it might miss testing conditions or combinations of paths.

3. Branch Coverage Criteria

Branch Coverage ensures that every possible branch (or decision) in the program is executed at least once. A branch is created when there is an `if`, `else`, or any conditional operator.

- **Explanation**: In branch coverage, the goal is to make sure that all the possible outcomes of each decision are tested. For example, an `if` statement has two branches: one for `true` and one for `false`. Each of these branches needs to be executed during testing.
- **Example**: For the following code:

```
if (x > 5) {
    a = 10;
} else {
    b = 20;
}
```

- The branches are:
  - One where `x > 5` (true branch) and `a = 10`
  - One where `x <= 5` (false branch) and `b = 20`
  - Branch coverage ensures that both branches are tested.
- **Pros**: Ensures that both sides of each decision point are tested.
- **Cons**: It may still not test every possible combination of conditions, just the outcomes of individual branches.

4. Predicate Coverage Criteria

Predicate Coverage focuses on testing all the possible outcomes of logical expressions (predicates) in a program. A predicate is a condition that evaluates to either `true` or `false` (like in `if`, `while`, or `for` loops).

- **Explanation**: The goal of predicate coverage is to ensure that each boolean expression (predicate) in the code is tested for both its `true` and `false` outcomes. Unlike branch coverage, which only checks whether each branch is taken, predicate coverage checks whether each condition is evaluated as true or false.

- **Example**: Consider the following code:

```
if (x > 5 && y < 10) {
    a = 10;
}
```

- The condition `x > 5 && y < 10` is a predicate. Predicate coverage ensures that this condition is tested both when:

1. The condition `x > 5 && y < 10` is `true`
2. The condition `x > 5 && y < 10` is `false`

- **Pros**: It provides better coverage of complex conditions and ensures that both `true` and `false` outcomes of conditions are considered.
- **Cons**: It might not cover all the paths through the program and can become complex when dealing with multiple predicates.

Criteria	What It Covers	Focus	Pros	Cons
All Path Coverage	All possible execution paths in the program.	Thorough testing of all possible combinations of decisions.	Comprehensive testing, very thorough.	Computationally expensive and impractical for large programs.
Statement Coverage	Every statement in the program.	Ensuring that every line of code is executed.	Easy to implement and understand.	Doesn't test logical decisions, can miss important paths.
Branch Coverage	Every branch (decision point) in the program.	Ensuring all branches of conditional statements are covered.	Ensures both outcomes (true/false) are tested.	Doesn't test combinations of conditions (e.g., multiple conditions in an `if`).
Predicate Coverage	Each condition or logical expression (predicate).	Testing both true and false outcomes for each condition.	Ensures that all conditions (predicates) are evaluated.	Can become complex and still doesn't cover all paths.

8. Explain inheritance graph and coupling du-pairs with examples

In software testing and design, various metrics and tools help measure and analyze the relationships between different components or classes in a software system. Two such important concepts are the **Inheritance Graph** and **Coupling DU-pairs**.

1. Inheritance Graph

An **Inheritance Graph** is a type of diagram used in object-oriented programming (OOP) to visualize the relationships between different classes in a system based on inheritance. It shows which classes inherit from others, forming a tree-like structure.

- **Explanation**: In object-oriented programming, inheritance allows one class (child or subclass) to inherit properties and methods from another class (parent or superclass). The inheritance graph represents this relationship visually by showing arrows pointing from the subclass to the superclass. This helps to understand the class hierarchy and how classes are related.

```
class Animal {
    public void makeSound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

In this example:

- `Animal` is the **base class** (or superclass).
- `Dog` and `Cat` are **subclasses** that inherit from `Animal` and override the `makeSound` method.

Inheritance Graph:

The inheritance graph visually represents the relationships:

```
      Animal
      /    \
   Dog      Cat
```

Explanation: The `Dog` and `Cat` classes are subclasses of `Animal`. Both override the `makeSound` method to provide their own behavior, while inheriting other properties of `Animal`.

2. Coupling DU-pairs

- **Coupling DU-pairs** refers to the concept of **Data Coupling** in software testing, which is a type of **coupling** metric used to analyze the interaction between functions or modules based on data flow.
- In particular, **DU-pairs** stand for **Definition-Use pairs**. These are the pairs of statements where a variable is defined (assigned a value) and later used (read or referenced) in the code.
- **Explanation**: A **DU-pair** consists of two key operations:
  - **Definition (D)**: A variable is defined (assigned a value).
  - **Use (U)**: The variable is later used (read or referenced).
- In software testing, coupling DU-pairs are important because they help identify how closely related different modules or functions are through shared data.
- The more closely data is shared, the more tightly coupled the functions or modules are.
- **Example**:
  Consider the following code snippet:

```
public class CouplingExample {
    public static void main(String[] args) {
```

```java
        int x = 10;  // Definition of x

        // Some operation that uses x
        if (x > 5) {
            x = x * 2;  // Definition of x again
        }

        // Later use of x
        System.out.println("The value of x is: " + x);  // Use of x
    }
}
```

In this code:

- `x = 10;` is a **Definition** of the variable `x`.
- `x = x * 2;` is a **Re-definition** of `x`.
- `System.out.println("The value of x is: " + x);` is a **Use** of `x`.

DU-pairs

- The first DU-pair is `(x = 10, x)` — where `x` is defined and later used.
- The second DU-pair is `(x = x * 2, x)` — where `x` is redefined and used.

Explanation:

- **Definition** means when a variable is assigned a value.
- **Use** means when the variable is read or referenced in the program.

In the above example, the two DU-pairs show how the variable `x` is defined and then used, which can help in understanding the flow of data.

9. Using the code snippet given in below. Perform data flow analysis and find all valid DU Pairs

```java
1  public int gcd(int x, int y){
2    int tmp;
3    while(y!=0){
```

```
4         tmp = x%y;
5         x = y;
6         y=tmp;
7     }
8     return x;
9 }
```

In **Data Flow Analysis**, we analyze how variables are defined and used in the code. Specifically, **DU (Definition-Use) pairs** refer to the points where a variable is defined and later used in the program. Here's how we can perform Data Flow Analysis for the provided `gcd` function.

Steps for Data Flow Analysis:

1. **Variables**: We have the following variables in the code:
   - `x` (input variable)
   - `y` (input variable)
   - `tmp` (local variable)
2. **Definition of Variables**:
   - **tmp** is defined at Line 2 (initialization) and again at Line 4.
   - **x** is defined at the beginning (via function argument) and redefined at Line 5.
   - **y** is defined at the beginning (via function argument) and redefined at Line 6.
3. **Use of Variables**:
   - **tmp** is used at Line 6 (used as the new value of `y`).
   - **x** is used at Line 8 (return statement).
   - **y** is used at Line 3 (condition of `while` loop).

DU Pairs

- **(tmp = x % y, tmp)**: The value of `tmp` is defined at Line 4 (tmp = x % y) and later used at Line 6 (y = tmp).
- **(x = y, x)**: The value of `x` is defined at Line 5 (x = y) and used later at Line 8 (return x).
- **(y = tmp, y)**: The value of `y` is defined at Line 6 (y = tmp) and used at Line 3 (while condition `y != 0`).

Summary of DU Pairs:

- **DU Pair 1**: (Line 3: `tmp = x % y`, Line 5: `y = tmp`)
- **DU Pair 2**: (Line 4: `x = y`, Line 6: `return x`)
- **DU Pair 3**: (Line 5: `y = tmp`, Line 2: `while(y != 0)`)

10. List and explain any three path selection coverage criteria

In software testing, **path selection coverage criteria** help ensure that important flows (or "paths") through a program are tested. Here are three important criteria:

1. Statement Coverage (also called C0 Coverage)

What it means:

- Test enough paths so that **every statement** in the code runs at least once.
  Example:
  Imagine this simple code:

```
if (x > 0) {
    System.out.println("Positive");
}
System.out.println("Done");
```

- To achieve **statement coverage**, we must run the code so that:
  - The `if` condition is checked.
  - The `System.out.println("Positive");` runs **or** is skipped.
  - The `System.out.println("Done");` always runs
  Goal: Every line (statement) should be executed at least once.

2. Branch Coverage (also called C1 Coverage)

What it means:

- Test enough paths so that **every decision (if/else, switch case, loops)** is taken both **true** and **false**.
  Example:

```
if (x > 0) {
    System.out.println("Positive");
} else {
    System.out.println("Not positive");
}
```

- For **branch coverage**, we must have:
  - One test where `x > 0` is **true** (go inside `if`).
  - Another test where `x > 0` is **false** (go inside `else`).

Goal: Make sure every possible branch (true/false) of decisions is taken at least once.

3. Path Coverage

What it means:

- Test **all possible paths** through the program at least once.
  Example: If there are two decisions in a program:

```
if (x > 0) {
    if (y > 0) {
        System.out.println("Both positive");
    }
}
```

There are **four possible paths**:

1. `x > 0` is **true** and `y > 0` is **true**.
2. `x > 0` is **true** and `y > 0` is **false**.
3. `x > 0` is **false** (skip the second `if` completely).
4. No need to check `y` if `x` is false.

- Path coverage means you need test cases to cover **all** these possible paths.

Goal: All possible paths in the code should be tested. (This can become very large for complex programs.)

Coverage Type	What It Tests	Goal
Statement Coverage	Every line of code	Run all statements
Branch Coverage	Every true/false decision	Test all decision outcomes

Coverage Type	What It Tests	Goal
Path Coverage	Every possible execution path	Test all paths