Software-Testing-Module-4-Important-Topics-PYQs

1. What is functional testing and highlight the important steps.

What is Functional Testing?

Functional Testing is a type of software testing where we check whether each function of the application works according to the **requirements** and **specifications**.

- It is concerned with **what the system does** (not how it does it).
- It checks if the software gives the **correct output** for the given **input**.
- It mainly tests the **user interface**, **APIs**, **databases**, **security**, and **main functions** of the application.
- It can be done **manually** or using **automation tools**.

Purpose of Functional Testing

- To **verify** that each function of the application behaves as expected.
- To ensure **smooth user navigation** across the app.
- To check that **entry and exit points** in the app are working properly.
- To test **error handling**, **basic usability**, and **accessibility** features.

What do we Test in Functional Testing?

- **Basic Usability:** Can users easily use the app?
- **Main Functions:** Are the important features working?
- **Accessibility:** Is the app usable by people with disabilities?
- **Error Handling:** Are proper error messages shown when something goes wrong?

Functional Testing Process

Step	Description
Step 1: Identify Test Inputs	Find out which features or functions need to be tested.
Step 2: Compute Expected Outcomes	Based on the specifications, predict what the correct output should be.
Step 3: Execute Test Cases	Perform the tests using the identified inputs and record the actual outputs.
Step 4: Compare Actual and Expected Outputs	Compare what you got vs. what you expected, to find any mismatches or defects.

2. Preconditions are excellent sources for functionality - based characteristics. Justify

Preconditions are the conditions that must be true **before** a function or feature in software can be used.
They describe the **starting point** or **requirements** for a function to work correctly.

Before logging into a website, a user must have a registered account.
Here, **having an account** is a **precondition** for using the login function.

Justification:

- **1. Helps Identify Key Functions:**
  Preconditions tell us what is needed **before** a function starts.
  By knowing these, testers can **focus on the right features** and **important actions** to test.
- **2. Defines Valid Inputs and Scenarios:**
  Preconditions help testers **understand what inputs are valid** and **which situations are meaningful** for testing the software's functionality.
- **3. Ensures Proper Behavior:**
  If preconditions are not met, the function should either **block** the user or **show a helpful message**.
  Testing preconditions ensures that the software behaves **correctly** in both **normal** and **error** situations.
- **4. Avoids Wasting Time:**
  Without knowing the preconditions, testers might test with **wrong inputs** or **wrong scenarios**, wasting time.
  Preconditions help testers **test smarter** by focusing only on **realistic and meaningful cases**.
- **5. Improves Quality of Testing:**
  Since preconditions are directly related to **how a function should behave**, they help create **strong, functionality-based test cases** that ensure the software works properly in real-life usage.

---

3. Differentiate between structural and functional testing

Feature	Structural Testing	Functional Testing
Meaning	Testing the **internal structure or code** of the program.	Testing the **functionality** of the software based on **requirements**.
Other Name	Also called **White-Box Testing**.	Also called **Black-Box Testing**.
Focus	Focuses on **how** the software works internally (code, logic, paths).	Focuses on **what** the software does (features, behavior).
Knowledge Required	Tester must know the **code** and **program structure**.	Tester only needs to know the **requirements and functionality**, not the code.
Test Basis	Based on the **design documents**, **code structure**, or **logic flow**.	Based on the **functional specifications** or **user requirements**.
Example	Checking if all if-else conditions and loops are working properly.	Checking if the "Login" button logs the user in correctly.
Goal	To make sure all paths, conditions, and statements in the code are tested.	To make sure the application meets the expected business needs.
Tools	Examples: JUnit, NUnit	Examples: Selenium, QTP/UFT
When Used	Mainly during **development and unit testing**.	Mainly during **system testing and acceptance testing**.

4. Outline the functional testing concepts of Howden

The **Howden approach** talks about important ideas in functional testing. It mainly focuses on **how to check if a software's functions work properly** by designing good test cases.

The key concepts are:

1. Functionality-Based Testing

- Tests are created based on the **functions** (features) that the software must perform.
- Focus is on **what the system should do**, not how it is coded internally.

2. Use of Preconditions

- **Preconditions** are the conditions that must be true before a function runs.
- These preconditions help decide **what inputs** should be given for testing.
- Good preconditions lead to better test cases because they are closely tied to the expected behavior.

3. Input Domain Focus

- Input values are selected from different parts of the **input domain** (all possible valid inputs).
- By picking different kinds of inputs, we can test how the function behaves in various situations.

4. Valid and Invalid Inputs

- Howden's method suggests testing both:
  - **Valid inputs** (expected and correct inputs).
  - **Invalid inputs** (wrong inputs) to see how the system handles errors.

5. Output Verification

- After executing the function with the test inputs, the **output must be checked** carefully.
- The output is compared with the **expected result** to decide if the function works correctly.

6. Test Set Completeness

- A **complete test set** is when the designed test cases are enough to find most of the errors in the function.
- Howden emphasizes designing **good coverage** with fewer, but meaningful, test cases.

5. Give the guidelines for equivalence class partitioning.

Equivalence Class Partitioning (ECP) is a technique where **input data** is divided into different groups (**classes**) where the system should behave similarly for all values in the same group.

The idea is: **Test one value from each class**, because if one works, others are likely to work too!

Guidelines for Equivalence Class Partitioning:

1. Identify the Input Domain

- Find out **what inputs** your system accepts.
- Look for **input fields, forms, values, options** — anything the user gives.

2. Divide Inputs into Classes

- Group the inputs into **equivalence classes**.
- Inputs that should behave the **same way** go into the **same class**.
  - Example: For an age field (allowed: 18–60), inputs like 20, 30, and 50 are in one valid class.

3. Create Valid Equivalence Classes

- Create classes where inputs are **valid** (according to requirements).
- Example: Ages between 18 to 60 → Valid Class.

4. Create Invalid Equivalence Classes

- Also create classes for **invalid inputs**.
- Example: Ages less than 18 or greater than 60 → Invalid Class.

5. Choose Test Cases from Each Class

- Pick **at least one representative value** from each class.
- This saves time because you don't have to test every single input separately!

6. Cover Boundary Conditions Carefully

- Specially check **edges** like minimum and maximum values (example: exactly 18 or exactly 60).
- Sometimes, a separate boundary value analysis is needed, but ECP should consider boundaries too.

7. Do Not Overlap Classes

- Classes must be **clear and non-overlapping**.
- Each input must belong to only **one equivalence class** at a time.

8. Think About Output Conditions Too

- If **output** depends on input, make equivalence classes based on different outputs too.
- Example: Different discounts based on the price entered.

6. List the guidelines for performing Boundary value Analysis.

Boundary Value Analysis (BVA) is a testing technique where we focus on **values at the edges** (boundaries) of input ranges, because most errors usually happen there!

Guidelines for Boundary Value Analysis:

1. Identify the Input Range

- Find the **minimum** and **maximum** values that inputs are allowed to take.
- Example: If a form accepts ages from 18 to 60, then 18 and 60 are the boundaries.

2. Focus on Boundaries

- **Test at the boundary** and **just outside** the boundary.
- Typically, test these five values:
  - Minimum value (lower boundary) → (example: 18)
  - Minimum value **-** 1 → (example: 17)
  - Maximum value (upper boundary) → (example: 60)
  - Maximum value + 1 → (example: 61)
  - A value **within** the range → (example: 30)

3. Test Both Valid and Invalid Boundaries

- Check that valid boundary values **work correctly**.
- Check that slightly invalid values **fail correctly**.

4. Apply BVA for Each Input Field

- If your application has multiple input fields, **apply BVA individually** to each one.

5. Combine BVA with Equivalence Class Partitioning

- After doing BVA, also divide inputs into valid and invalid classes to cover more conditions smartly.

6. Don't Ignore Complex Conditions

- If boundaries are not simple numbers (like dates, string lengths, array sizes), still apply BVA carefully.

7. Consider Multi-variable Situations

- If there are **two or more fields** together affecting output, check boundary combinations too (this is called multiple boundary testing).

7. Define call graphs and classes.

A **Call Graph** is a graphical representation of the relationships between the different methods or functions in a program. It shows how functions/methods in a software application **call** or **invoke** each other.

- **Purpose:** In software testing, call graphs help in visualizing the flow of execution within a program. They can also be used to identify **unreachable code** or **dead code** (code that never gets executed).
- **Structure:**
    - **Nodes:** Represent methods or functions.
    - **Edges:** Represent the **calls** from one function to another.
- **Usefulness:**
    1. **Test Coverage:** Helps testers identify the parts of the application that need more testing.
    2. **Code Optimization:** Detects unnecessary function calls.
    3. **Path Testing:** Supports testing all possible paths by analyzing the relationships between functions.

Example:

Suppose you have the following methods in a program:

```
function A() {
    B();
}

function B() {
    C();
}

function C() {
    // do something
}
```

In a **Call Graph**, the nodes would be A, B, and C, and the edges would show the calls from A to B, and from B to C.

Classes in Software Testing

In the context of **software testing**, a **Class** is typically associated with **object-oriented programming (OOP)**, where it is a blueprint for creating objects that contain both **data (attributes)** and **methods (functions)**.

- **Purpose:** In testing, classes are used to organize and structure the code in an application, and testing them ensures that they behave as expected.
- **Class Testing:** This refers to testing the behavior of a **class** (or its methods) to verify that it works correctly. It's a type of **unit testing**, where individual classes are tested for correctness, including all their methods and attributes.

Key Points:

1. **Class Testing** ensures that methods within a class produce the correct output based on the given inputs.
2. **Focuses on testing individual units of a class**, such as **methods** and **attributes**.
3. It also checks for **interaction between methods** of the class and ensures that methods work together properly.

---

8. Illustrate Random Testing with four steps

Random Testing is a type of software testing technique where random inputs are provided to the system in an attempt to find bugs. It does not follow a pre-defined test plan or sequence, and testers typically use random actions or inputs to evaluate the software's behavior. The main goal of random testing is to explore as many different situations as possible in order to detect unexpected or unpredictable errors.

Four Steps in Random Testing

1. **Identify the Testing Environment**:
   - **Step 1**: The first step is to understand the software or system being tested, the environment in which it operates, and its core functionalities. This can include web apps, mobile apps, or desktop applications.
   - **Action**: Set up the testing environment and tools Familiarize yourself with how the application responds to user input.
2. **Generate Random Inputs**:
   - **Step 2**: Randomly generate inputs for the software or system, without following any specific pattern or predefined test cases. The inputs can be:
     - Random numbers
     - Random characters or strings
     - Random mouse clicks, key presses, etc.
     - Random values in fields (text fields, dropdowns, etc.)
   - **Action**: Use a tool or write scripts to generate random inputs within the possible valid and invalid ranges of the software.
3. **Execute the Random Tests**:
   - **Step 3**: Execute the software by applying the randomly generated inputs and performing random actions on the software (such as clicking buttons, selecting checkboxes, or navigating between screens). This step mimics the user experience, where the software is being tested under random conditions.
   - **Action**: Observe how the software behaves under these random conditions. Ensure the software is stable, and take note of any crashes, errors, or abnormal behaviors that occur.
4. **Analyze and Report Results**:
   - **Step 4**: After executing the tests, analyze the results to identify any defects, crashes, or unexpected behavior. Record any bugs found during the testing process and analyze

them for potential causes. The goal is to spot issues that might not be caught through structured or planned testing.
  - **Action**: Report the findings, including the random inputs that caused issues, to the development team for fixing.

Example of Random Testing

Imagine you're testing a mobile app that allows users to add items to a shopping cart and make purchases. In random testing:

- You might randomly click buttons, enter random product quantities, select random payment methods, or input random addresses without a specific test case in mind.
- You could also test by inputting incorrect data (e.g., invalid card numbers or random characters in text fields) to see how the app responds to these errors.
  The goal is to observe how the app behaves under unplanned, unpredictable conditions, which may expose bugs that would otherwise be missed.

Benefits of Random Testing:

1. **Unpredictability:** It helps uncover unexpected bugs because it tests the software under random and unpredictable conditions.
2. **Exploration:** Allows testers to explore the software without being constrained by predefined test cases, which could miss edge cases.
3. **Simplicity:** It's easy to execute because it doesn't require detailed planning or knowledge of the application's inner workings.

Drawbacks of Random Testing:

1. **Inefficiency:** It may not be very efficient in finding bugs because it lacks the structure and focus of other testing techniques.
2. **Limited Coverage:** Since the testing is random, it can miss critical areas of the application or key functional paths that need testing.
3. **Low Reproducibility:** Random testing might not be easily repeatable, as it relies on chance.

9. Explain Input Domain Modelling with examples

Input Domain Modelling (IDM) is a testing technique used to systematically define the set of all possible inputs that a software system can accept.

It helps testers identify valid, invalid, and boundary inputs to test the software more thoroughly.

The idea is to break down the entire range of possible inputs into smaller, manageable "domains" or categories, making it easier to select test cases that cover all critical scenarios.

Key Concepts of Input Domain Modelling:

1. **Input Domain**: The complete set of values or inputs that a system or application can accept. For example, if you're testing a login page, the input domain might include usernames, passwords, and other fields.
2. **Partitioning**: Dividing the input domain into smaller subsets or "equivalence classes" that represent similar values or behaviors. This helps reduce the number of test cases by focusing on representative values for each partition.
3. **Equivalence Classes**: These are the sets of input values that are treated in the same way by the system. For instance, if a form field accepts age between 18 and 100, any age in that range would be considered part of the same equivalence class.
4. **Boundary Values**: These are the values at the edges of equivalence classes, often where bugs are more likely to appear. For example, if the input range is 1 to 10, the boundary values are 1 and 10.

Steps in Input Domain Modelling

1. **Identify the Input Domain**: Determine what inputs the software can accept. For example, if you're testing a login form, the inputs could be the username, password, and remember-me checkbox.
2. **Partition the Input Domain**: Divide the input domain into equivalence classes. This could be done by considering valid inputs, invalid inputs, and boundary cases.
   - Example: If the age input should be between 18 and 100, you have the following equivalence classes:
     - **Valid range**: Age between 18 and 100 (e.g., 20, 30, 50).
     - **Invalid ranges**: Age less than 18 (e.g., 10, 15) or more than 100 (e.g., 120).
     - **Boundary values**: Age at 18 and 100.
3. **Select Test Cases**: Pick representative values from each equivalence class, including boundary values, to test. You don't need to test every possible value, just one or two

representative ones for each class.

4. **Execute the Tests**: Perform the tests using the selected test cases, observing how the system behaves with each set of inputs.

Example of Input Domain Modelling

Let's take an example of a simple software form where the user enters their **age**:

1. **Input Domain**: The valid input domain for age might be a range between 1 and 100.
2. **Partitioning the Input Domain**:
   - **Valid equivalence class**: Age between 18 and 100.
     - Example valid ages: 20, 45, 99.
   - **Invalid equivalence class**: Age less than 18 or greater than 100.
     - Example invalid ages: 10, 120.
   - **Boundary values**: The values on the edge of the valid range.
     - Boundary values: 18 (lower boundary), 100 (upper boundary).
3. **Selecting Test Cases**:
   - Valid test case: Age = 25 (from the valid equivalence class).
   - Invalid test case: Age = 10 (from the invalid equivalence class).
   - Boundary test cases: Age = 18 (lower boundary), Age = 100 (upper boundary).
4. **Execute the Tests**:
   - Test the system using the input values 25, 10, 18, and 100.
   - Observe how the system behaves when it receives these values and check if it handles valid and invalid inputs correctly.

10. Explain All Combinations Coverage (ACoC), Each Choice Coverage (ECC) with examples.

All Combinations Coverage (ACoC)

All Combinations Coverage (ACoC) is a testing technique that ensures all possible combinations of inputs are tested. It is particularly useful when testing systems that involve multiple input parameters, and each combination of input values might produce a different output.

How ACoC Works:

- In ACoC, if a system has multiple inputs, this technique will test **every single combination** of those inputs to ensure each one works as expected.
- This approach is **exhaustive**, meaning it tests all possible combinations of inputs.
- ACoC is especially useful when interactions between input variables are complex, and you need to verify that the system handles all combinations correctly.

Example of ACoC:

Imagine you have a simple login form with two inputs: **Username** and **Password**, and for simplicity, we will assume both inputs can have three possible values.

- **Username options**: `User1`, `User2`, `User3`
- **Password options**: `Pass1`, `Pass2`, `Pass3`

For ACoC, you would need to test **all combinations** of the usernames and passwords. So the test cases would be:

1. `User1, Pass1`
2. `User1, Pass2`
3. `User1, Pass3`
4. `User2, Pass1`
5. `User2, Pass2`
6. `User2, Pass3`
7. `User3, Pass1`
8. `User3, Pass2`
9. `User3, Pass3`

There are **9 test cases** in total because you need to test each combination of inputs.

Pros of ACoC:

- **Thorough**: It ensures that every possible combination of inputs is tested.
- **Effective for Complex Interactions**: It's useful when the behavior of a system depends on multiple interacting inputs.

Cons of ACoC:

- **Time-Consuming**: As the number of inputs grows, the number of combinations increases exponentially, which can lead to many test cases.
- **May not always be necessary**: In some cases, not every combination needs to be tested if the combinations are highly redundant or unnecessary.

Each Choice Coverage (ECC)

Each Choice Coverage (ECC) is a testing technique that ensures **each individual input parameter is tested with each of its possible values at least once**. Unlike ACoC, which tests every possible combination of inputs, ECC tests all the values for each input, but not necessarily in combination with all other inputs.

How ECC Works:

- ECC ensures that every possible value for each input is selected **at least once** across all test cases.
- It doesn't require testing every possible combination of inputs but ensures that every option for each individual input is considered.

Example of ECC:

Let's use the same example of a login form with two inputs: **Username** and **Password**. We have

- **Username options**: `User1`, `User2`, `User3`
- **Password options**: `Pass1`, `Pass2`, `Pass3`
  For ECC, we don't need to test all combinations. Instead, we ensure that each value for the **Username** and **Password** is tested at least once. The test cases might look like this:

1. `User1, Pass1`
2. `User2, Pass2`
3. `User3, Pass3`

Here, every possible value for **Username** (`User1`, `User2`, `User3`) and for **Password** (`Pass1`, `Pass2`, `Pass3`) is tested at least once, but not all possible combinations are tested.

Pros of ECC:

- **Less Time-Consuming**: Since it doesn't require testing all combinations, it's quicker than ACoC.

- **Good Coverage**: It ensures that every input value is tested at least once, which can help detect issues with individual inputs.

Cons of ECC:

- **Less Comprehensive**: Since combinations are not fully tested, some complex interactions between inputs might not be covered.
- **May Miss Some Issues**: If there are specific combinations of inputs that cause problems, ECC might miss those.

11. Explain pair-wise coverage and T-wise coverage with examples.

Pair-Wise Coverage

Pair-wise coverage (also called **pair testing**) is a testing technique where **all possible pairs** of input values are tested. It doesn't test every combination of inputs like ACoC (All Combinations Coverage), but it ensures that **every possible pair** of input values is included in at least one test case.

How Pair-Wise Coverage Works:

- In pair-wise testing, we focus on combinations of **two input parameters** at a time.
- For example, if an application takes three inputs (let's call them **A**, **B**, and **C**), pair-wise testing ensures that every possible pair of these inputs is tested.

Example of Pair-Wise Coverage:

Let’s assume a system with two inputs:

- **Input 1 (A)**: `A1`, `A2`
- **Input 2 (B)**: `B1`, `B2`
- **Input 3 (C)**: `C1`, `C2`

In pair-wise testing, we need to test **all combinations of pairs** of inputs. Here’s how we might approach it:

- Pair (A1, B1) — This combination is tested together.

- Pair (A1, C1) — This combination is tested together.
- Pair (B1, C1) — This combination is tested together.
- Pair (A2, B2) — This combination is tested together.
- Pair (A2, C2) — This combination is tested together.
- Pair (B2, C2) — This combination is tested together.

Now, the idea of **pair-wise testing** is to cover **all these pairs** with the minimum number of test cases.
One way to do this is by selecting the following **3 test cases**:

1. `A1, B1, C1`
2. `A1, B2, C2`
3. `A2, B1, C2`

This way, we ensure that:

- The pair (A1, B1) is covered by the first test case.
- The pair (A1, B2) is covered by the second test case.
- The pair (A1, C1) is covered by the first test case.
- The pair (A1, C2) is covered by the second test case.
- The pair (A2, B1) is covered by the third test case.
- The pair (A2, B2) is covered by the third test case.
- The pair (A2, C1) is covered by the third test case.
- The pair (B1, C1) is covered by the first test case.
- The pair (B1, C2) is covered by the second test case.
- The pair (B2, C1) is covered by the second test case.
- The pair (B2, C2) is covered by the second test case.
- With **just 3 test cases**, we ensure that **every pair** of input values is tested, without needing to test all possible combinations (which would be 8 test cases in this case).
- The key is that **each pair** of inputs should appear together at least once in the selected test cases.

Advantages of Pair-Wise Testing:

- **Reduces the number of test cases**: Instead of testing every possible combination, pair-wise testing significantly reduces the number of test cases.

- **Efficient**: It still covers the important interactions between input parameters.
- **Time-saving**: Because the number of test cases is reduced, it saves time and resources compared to testing all combinations.

T-Wise Coverage

T-Wise Coverage is a more generalized version of pair-wise testing, where **all combinations of input values** for **T parameters** are tested. Here, **T** refers to the number of input parameters involved in the test cases. So, **T-Wise coverage** means covering all combinations of **T parameters** at a time.

How T-Wise Coverage Works:

- **T-Wise testing** ensures that for every possible combination of **T parameters**, the system is tested.
- For example, **pair-wise testing** is a specific case of T-wise testing where T = 2, meaning we test all pairs.
- If T = 3, we test every combination of three parameters.
- T-Wise testing can be applied for any value of **T** (2, 3, 4, etc.), depending on how many input parameters we want to focus on.

Example of T-Wise Coverage (T = 3):

Let's assume we have three inputs for a system

- **Input 1 (A)**: `A1`, `A2`
- **Input 2 (B)**: `B1`, `B2`
- **Input 3 (C)**: `C1`, `C2`
  In **T-wise testing** for **T = 3**, we need to test **all possible combinations** of **three parameters** at a time. So, we would test:
- `A1, B1, C1`
- `A1, B1, C2`
- `A1, B2, C1`
- `A1, B2, C2`
- `A2, B1, C1`
- `A2, B1, C2`
- `A2, B2, C1`

- `A2, B2, C2`

These 8 test cases cover **all possible combinations of three parameters**.

Test Cases for T-Wise Coverage (T = 3):

1. `A1, B1, C1`
2. `A1, B1, C2`
3. `A1, B2, C1`
4. `A1, B2, C2`
5. `A2, B1, C1`
6. `A2, B1, C2`
7. `A2, B2, C1`
8. `A2, B2, C2`

Advantages of T-Wise Testing:

- **Higher Coverage**: T-Wise testing provides broader coverage of interactions, not just pairs, but also combinations of three or more inputs.
- **Flexible**: The value of **T** can be adjusted based on the complexity of the system and the number of input parameters.
- **Efficient**: Like pair-wise testing, it reduces the number of test cases when compared to exhaustive testing of all combinations.

Disadvantages of T-Wise Testing:

- **Complexity increases with higher T**: As **T** increases (for example, testing combinations of 4 or more parameters), the number of test cases increases significantly.
- **May miss complex interactions**: In some cases, testing only specific combinations of parameters might miss issues that arise from less common combinations.

Feature	Pair-Wise Coverage (T = 2)	T-Wise Coverage (T > 2)
Number of Parameters	Tests all combinations of two input parameters.	Tests all combinations of **T** parameters (T can be 3, 4, etc.).
Test Cases	Fewer test cases than exhaustive testing, but covers all pairs.	More test cases as **T** increases.

Feature	Pair-Wise Coverage (T = 2)	T-Wise Coverage (T > 2)
Efficiency	Efficient and reduces the number of test cases significantly.	More efficient than exhaustive testing but less efficient than pair-wise.
Use Case	Use when interactions between **two** parameters matter.	Use when interactions between more than two parameters need testing.

12. Explain three different types of Black Box testing

1. Functional Testing

- **What it means**:
  This type checks **whether the software behaves as expected** based on the requirements.
- **What is tested**:
  Features, actions, and operations of the software — **what it should do**.
- **Example**:
  Suppose you have a login page. You check if entering the correct username and password allows you to log in successfully.
  You are **not** worried about how the login system is built inside — only if it **works correctly**.

2. Boundary Value Testing

- **What it means**:
  This type tests **values at the edge or boundary** of input fields, because errors often happen at extreme values.
- **What is tested**:
  Minimum, maximum, just below, just above, and exact boundary values.
- **Example**:
  If an age input accepts values between **18 and 60**:
  - You will test 17, 18, 60, and 61.
  - Check whether 18 and 60 are accepted and 17 and 61 are rejected.

3. Equivalence Partitioning

- **What it means**:
  This type divides inputs into **groups (partitions)** where the system should behave similarly for all values in a group.
- **What is tested**:
  Instead of testing all possible inputs, you test **one value from each group** to save time.
- **Example**:
  Again for an age field (valid range 18–60):
  - Group 1: Values below 18 (invalid) → e.g., 15
  - Group 2: Values between 18–60 (valid) → e.g., 30
  - Group 3: Values above 60 (invalid) → e.g., 65 You test with **one value** from each group.

Type	Purpose	Example
Functional Testing	Check if software features work correctly	Test login page
Boundary Value Testing	Check edges of input ranges	Test ages 17, 18, 60, 61
Equivalence Partitioning	Divide inputs into valid/invalid groups	Test age 15, 30, 65

13. Discuss the two approaches for input space modelling. Also write pros and cons of each method

In software testing, **Input Space Modeling** means thinking about **all possible inputs** a program can receive and **organizing** them in a smart way to create good test cases.

There are **two main approaches**:

1. Input Condition Approach

What it means:

- You focus directly on **individual input fields** and **conditions** separately.
- You look at each input (like a form field or function parameter) and decide what values it can have.

Example:
Suppose you have a login form:

- **Username**: Non-empty string
- **Password**: Minimum 8 characters

You would separately consider:

- What happens if the username is empty or filled?
- What happens if the password is short, exact, or long?

Pros of Input Condition Approach:

- Simple and easy to apply when there are few inputs.
- Easy to understand for beginners.
- Good for small applications.

Cons of Input Condition Approach:

- Not very systematic for complex programs.
- Can miss important combinations between multiple inputs.

2. Input Domain Approach

What it means:

- Instead of looking at inputs separately, you think about the **entire input space together** — how inputs combine.
- You partition the input domain into classes (valid and invalid) and consider combinations.

Example:
In the same login form

- Username = Valid or Invalid
- Password = Valid or Invalid

You would think of combinations like:

- (Valid Username, Valid Password)
- (Valid Username, Invalid Password)

- (Invalid Username, Valid Password)
- (Invalid Username, Invalid Password)

You cover all meaningful combinations.

Pros of Input Domain Approach:

- Very systematic — less chance of missing cases.
- Better for complex applications with many inputs.
- Helps create stronger and more complete test cases.

Cons of Input Domain Approach:

- A bit complex and time-consuming.
- Needs careful planning when inputs are many.

---

14. Discuss the following coverage criteria in equivalence class portioning a) All combinations coverage (ACoC) b) Each choice coverage (ECC) c) Pair-wise coverage d)Base Choice Coverage

When we use **Equivalence Class Partitioning (ECP)** in testing, we divide inputs into **groups** (classes) that behave similarly.

After creating classes, we have to decide **how deeply we want to test** these groups — and that's where different **coverage criteria** come in

Here are four important ones:

a) All Combinations Coverage (ACoC)

Meaning:
Test **every possible combination** of choices from all the input classes.
Simple Example:
Imagine you are testing a login screen:

- Username: Valid / Invalid

- Password: Valid / Invalid
  All combinations would mean you test all 4 possibilities:

1. Valid Username, Valid Password
2. Valid Username, Invalid Password
3. Invalid Username, Valid Password
4. Invalid Username, Invalid Password

- This is very **thorough** — you test every situation

Drawback:

- If there are too many fields, combinations grow very fast and testing becomes too much!

b) Each Choice Coverage (ECC)

Meaning:
Test **each choice** (valid and invalid) **at least once**, but **not necessarily all combinations**.
Simple Example:
From Username and Password again:

- Test some cases where Username is valid
- Test some cases where Username is invalid
- Test some cases where Password is valid
- Test some cases where Password is invalid
- You make sure every individual option is tested **at least once**.
- But you might **miss combinations** of choices together.
- **Short-cut version** of All Combinations.

c) Pair-wise Coverage

Meaning:
Test **all possible pairs** of input choices together.

Simple Example:
If there are three inputs: A, B, and C (each with 2 options), instead of testing all 8 combinations, you make sure **every pair** (A-B, A-C, B-C) appears at least once.

- Saves time compared to All Combinations.

- Still finds many hidden bugs.
- Might miss rare problems involving 3 or more fields at once.

d) Base Choice Coverage

Meaning:
Choose a **base set** (default valid inputs), and then **change one input at a time**.

Simple Example:
Let's say base case is:

- Valid Username
- Valid Password

Then you create other tests:

- Invalid Username, Valid Password (change username)
- Valid Username, Invalid Password (change password)

Very **organized** and **simple** to design.
Focuses on **important deviations** from normal case.

15. Differentiate between boundary value analysis and equivalence class portioning

Aspect	Boundary Value Analysis (BVA)	Equivalence Class Partitioning (ECP)
Main Idea	Focuses on **testing values at the boundaries** (edges) of input ranges.	Focuses on **dividing inputs into groups (classes)** that behave similarly.
Goal	To catch errors that happen **at the edges** (smallest and largest valid/invalid inputs).	To **reduce the number of test cases** by picking one value from each group.
What you test	Minimum, just below minimum, maximum, just above maximum values.	One representative value from each valid and invalid group.
Example	Input age must be between 18 and 60.	Age input: - Valid class: 18–60

Aspect	Boundary Value Analysis (BVA)	Equivalence Class Partitioning (ECP)
	Test: 17, 18, 60, 61.	- Invalid classes: less than 18, greater than 60.
Best used when	When input values are **numerical ranges**.	When there are **different categories** of inputs.
Number of Test Cases	More focus on fewer but critical values around limits.	Often fewer test cases overall, one from each class.
Focus	**Boundary values** where most bugs hide.	**Groups of values** that are treated similarly by the program.