# *Software-Testing-Module-5-Important-Topics-PYQs*

## *1. Write the difference between Regression Testing and Orthogonal Array Testing.*

### 1) Regression Testing

- **Purpose**: Regression testing ensures that any new changes or fixes made to the software (such as bug fixes or new features) do not negatively affect the existing functionalities of the application.
- **When it's performed**: It's done after changes, updates, or defect fixes are applied to the software.
- **Focus**: It focuses on verifying that the existing features still function correctly after changes.
- **Scope**: The scope is typically broad, as it tests existing functionalities across the entire system.
- **Efficiency**: May require running a large number of tests to check all aspects of the software, leading to higher test execution time.

### 2) Orthogonal Array Testing (OAT)

- **Purpose**: OAT is used to test complex functionalities or applications where it's essential to achieve maximum code coverage with a minimal number of test cases.
- **When it's performed**: It is used when there are a large number of input combinations that need to be tested.
- **Focus**: It focuses on creating test cases that cover different combinations of parameters systematically, often focusing on variations that may not be easily covered by traditional testing methods.

- **Scope**: The scope is more specific and optimized, targeting combinations of inputs that may have a significant impact on the software's behavior.
- **Efficiency**: It is highly efficient because it reduces the number of test cases required by selecting representative combinations of input values. It helps in large test data scenarios with numerous combinations.

### Summary of Differences:

| Feature | Regression Testing | Orthogonal Array Testing (OAT) |
|---|---|---|
| **Purpose** | Ensure changes don't break existing functionality | Maximize code coverage with minimal test cases |
| **When performed** | After changes or fixes in the software | When dealing with large combinations of inputs |
| **Focus** | Validating existing features | Testing different combinations of input parameters |
| **Scope** | Broad (covers entire system) | Focused (specific combinations of parameters) |
| **Efficiency** | Can be time-consuming due to large test cases | Highly efficient, reduces the number of tests |

In essence, regression testing ensures stability after changes, while orthogonal array testing is a strategic approach to efficiently cover complex test cases.

## *2. What is Parameterized Unit testing.*

To understand **Parameterized Unit Testing**, let's break it down:

1. **Unit Testing**
   1. A **unit test** is a type of software testing where you test a small part (unit) of the software, usually a function or a method, to make sure it works as expected.
   2. It's like checking if one small piece of a puzzle fits properly. For example, you might write a unit test to check if a function that adds two numbers gives the correct result.
2. **Parameterized Unit Testing**:
   1. Now, **parameterized unit testing** takes unit testing a step further. Instead of testing just one fixed set of inputs, you make the test flexible by passing in different inputs

(parameters) to the test.

2. This helps test how your code behaves with various inputs, rather than just one.
3. **Example**: Imagine a test to check if an “addition” function works correctly. In a regular unit test, you might check `2 + 3 = 5`. But in a **parameterized unit test**, you can pass different values like `4 + 6`, `7 + 8`, and so on, to see if the function handles all of them correctly.
4. **Why is it useful?** It saves you time by automatically running the test with multiple values, making sure your code works in various situations.

## *3. State the advantages of Grey box testing*

- **Improved Software Quality**:
  - Since gray box testing blends both external and internal perspectives, it helps find defects that might be missed by using only black or white box testing. This leads to **better overall quality** of the software.
- **Focus on User Perception**:
  - Gray box testers typically have some understanding of the system's internal workings, but they also focus on how users interact with the system. This **user-centric approach** helps in ensuring that the software behaves as expected from the user's point of view, improving user satisfaction.
- **More Time for Bug Fixing**:
  - In gray box testing, testers have partial knowledge of the internal workings of the system, which helps them identify problems more quickly. This leads to more **time for developers to fix bugs** before the software goes live.
- **Combines the Benefits of Black and White Box Testing**:
  - By combining the strengths of both **black box** (focusing on functionality without knowing the internals) and **white box** (having detailed knowledge of the code), gray box testing ensures that both **functional and structural issues** are addressed. It’s a balanced approach that brings the best of both worlds.
- **No Need for High Programming Knowledge**:
  - Unlike white box testing, which requires deep programming skills, gray box testing **doesn't require testers to have high-level programming knowledge**. Testers can still perform effective testing based on their understanding of the system’s internal design, making it easier for non-developers to get involved.

- **Effective in Integration Testing**:
    - Gray box testing is particularly useful in **integration testing**, where different modules or components of the system are tested together. The knowledge of the internal architecture allows testers to better understand how components should interact, making integration testing more effective.
- **Helps Avoid Conflicts Between Developers and Testers**:
    - Since gray box testers have insight into the internal structure, it helps create a **collaborative environment** between developers and testers. This reduces the chances of misunderstandings or conflicts, as testers can provide more informed feedback.
- **Can Handle Complex Applications and Scenarios**:
    - Gray box testing is well-suited for testing **complex applications or systems**. The tester's understanding of both the functionality and the architecture helps them navigate and test complex features that might be difficult to evaluate with black box testing alone.
- **Non-Intrusive**:
    - Gray box testing is considered **non-intrusive** because it doesn’t require direct changes to the source code or invasive techniques. It typically involves analyzing the system’s behavior from both the outside and inside, without disrupting the operation of the software.

## *4. Explain the use of pattern testing*

**Pattern Testing** is a software testing technique used to check how the software behaves when it encounters specific patterns or sequences of inputs. The idea is to test the software using common or expected patterns of data, actions, or behaviors that might occur in real-world scenarios.

### How Pattern Testing Works:

- **Patterns in Software**:
    - In software, "patterns" refer to **recurring sequences or behaviors**. For example, a pattern could be how a user fills out a form (e.g., entering a name, followed by an email, then a phone number) or a sequence of actions taken in an application (like opening a file, editing it, and saving it).

- **Test Cases Based on Patterns**:
  - In pattern testing, test cases are designed to follow these common patterns. By doing this, testers check whether the software handles these predictable scenarios correctly.

### Example of Pattern Testing:

Let’s say you’re testing a **login form** in a web application. The typical pattern for a user might be:

1. Entering a username.
2. Entering a password.
3. Clicking the “Login” button.

A **pattern test case** would involve checking if the system correctly handles this series of actions (inputting a username, password, and submitting the form).

Another example could be in a **shopping cart system**:

1. Adding a product to the cart.
2. Changing the quantity of the product.
3. Proceeding to checkout.

Pattern testing would check if these common sequences work smoothly.

### Why Pattern Testing is Important:

1. **Checks Common User Behaviors**: It simulates how real users might interact with the software by following typical patterns. This ensures that the most common use cases are handled correctly.
2. **Finds Bugs in Routine Actions**: By testing patterns that people often use, you can find bugs or issues that might arise during normal usage.
3. **Ensures Consistency**: It ensures the software behaves consistently when users follow certain patterns, like filling out forms or navigating through different screens.

## *5. Symbolic execution is a midway of program proving and program testing. Justify.*

To understand how **symbolic execution** is a midway between **program proving** and **program testing**, let's break it down:

## 1. What is Program Testing?

- **Program Testing** involves running a program with **concrete inputs** (like specific values, e.g., numbers or strings) and checking whether the program behaves as expected with those inputs. It's typically used to identify **bugs** or **incorrect behaviors** in the program.
- **Example**: Testing a function that adds two numbers by running it with specific inputs like `3 + 5`, `2 + 4`, etc.

## 2. What is Program Proving?

- **Program Proving** is a formal method where the program is mathematically proven to be correct. It involves analyzing the program's **structure** and **logic** to guarantee that it will always behave as expected for all possible inputs.
- This is typically done using **formal methods** and **mathematical proofs**. For example, proving that an algorithm will always terminate or that a certain property will always hold true, no matter what input is given.

## 3. What is Symbolic Execution?

- **Symbolic execution** lies somewhere between program testing and program proving.
    - Instead of running the program with specific inputs (as in testing), you run it with **symbolic inputs**, which represent **arbitrary values**.
- For example, instead of testing a function with `3 + 5`, you would use symbolic values like `x` and `y` and analyze the program's behavior in terms of **expressions** involving `x` and `y`.
    - This allows you to explore how the program behaves for a wider range of inputs without actually testing them all.
- Symbolic execution builds **path conditions**—mathematical formulas that describe the **constraints** on inputs that must hold true for a particular execution path. These path conditions help you reason about the behavior of the program.

## Why is Symbolic Execution Between Program Proving and Testing?

1. **Testing** uses concrete inputs to check for **real-world bugs**. It doesn't guarantee that the program will work for **every possible input**.

- **Example**: You test a function with `x = 3` and `y = 5`, but that doesn't mean it will work for `x = 1000` and `y = -500`.

2. **Proving** guarantees that the program will behave correctly for **all possible inputs** by proving its correctness mathematically. This is usually very difficult and time-consuming.
3. **Symbolic execution** allows you to **simulate** running the program with all possible inputs by using **symbols** instead of concrete values.
    1. This gives you a deeper understanding of the program's behavior without having to manually test every possible combination of inputs, yet without needing a full mathematical proof of correctness.
    2. **Midway Point**: Symbolic execution **bridges the gap** between testing and proving because:
    3. It **tests multiple inputs** simultaneously, covering many scenarios without explicitly running the program for every input.
    4. It doesn't guarantee **full correctness** as program proving does, but it provides **much broader coverage** than traditional program testing.

---

## *6. Write the need for grey box testing and list the steps in grey box methodology.*

**Gray Box Testing** is a mix of two common testing methods: **Black Box Testing** (focusing only on what the system does) and **White Box Testing** (focusing on how the system works internally). Here's why it's needed:

- **Combines the Benefits of Both Black Box and White Box Testing**:
    - Gray box testing gives testers some understanding of the internal structure of the system, while also focusing on its external behavior.
    - This means testers can verify how the system works (like white box testing) and also check if it meets the user's requirements (like black box testing).
- **Improves Overall Product Quality**
    - By including insights from both the developers (internal knowledge) and testers (external behavior), gray box testing helps improve the quality of the product, ensuring it works well for users and is built efficiently.
- **Reduces Overhead in Testing Functional and Non-Functional Aspects**:

    - Gray box testing reduces the need for separate black box and white box testing, streamlining the process.
    - It ensures that both functional and non-functional aspects of the system are covered in fewer steps.
- **Gives Developers Time to Fix Defects**:
    - Testers have some internal knowledge of the system, so they can identify issues more quickly.
    - This gives developers enough time to fix the defects before they become bigger problems.
- **Focuses on the User's Point of View**:
    - Even though gray box testers have internal knowledge, testing is done from the user's point of view.
    - This means that testers check the functionality and performance of the system as the end user would, ensuring it meets the user's expectations.

## Steps in Gray Box Testing Methodology:

To carry out gray box testing, follow these steps:

1. **Step 1: Identify Inputs**:
    - Determine what kind of data or information will be provided to the system. These inputs will be used in the test cases to validate the system's behavior.
2. **Step 2: Identify the Outputs**:
    - Identify the expected outputs or results based on the inputs. This will help you compare the actual outputs after running the test cases to determine if the system is working correctly.
3. **Step 3: Identify the Major Paths**:
    - Identify the key paths in the system that are critical for its operation. These paths represent the most important actions the system must perform, and testing these paths ensures key functionalities are working.
4. **Step 4: Identify Subfunctions**:
    - Break down the system into smaller subfunctions or components. These are the individual parts of the system that carry out specific tasks, and each will need to be tested independently.
5. **Step 5: Develop Inputs for Subfunctions**:

- For each subfunction, design specific inputs that will help test that part of the system. This ensures that each subfunction is validated on its own.

6. **Step 6: Develop Outputs for Subfunctions**:
    - Similarly, define the expected outputs for each subfunction. These outputs are used to compare the results of the tests and verify if each subfunction is working as intended.
7. **Step 7: Execute Test Cases for Subfunctions**:
    - Run the test cases for each subfunction, using the inputs and expected outputs defined in the previous steps. This helps ensure that every part of the system is tested properly.
8. **Step 8: Verify the Correct Result for Subfunctions**:
    - After executing the test cases, check if the actual results match the expected outputs. If they don't match, it means there is a problem with the subfunction that needs to be fixed.
9. **Step 9: Repeat Steps 4 & 8 for Other Subfunctions**:
    - Continue testing the other subfunctions of the system by following the same process. Break them down, define inputs/outputs, test them, and verify the results
10. **Step 10: Repeat Steps 7 & 8 for Other Subfunctions**:
    - For any remaining subfunctions, execute the test cases and verify the outputs to ensure everything is functioning correctly.

## *7. Explain symbolic testing and symbolic execution tree*

**Symbolic Execution** is a smart way of testing a program **without giving it real (concrete) inputs**.
Instead of putting numbers like 5 or 10, we give **symbols** like `x` or `y` as inputs.

- The program **runs normally**, but now **calculations happen using symbols**.
- As the program runs, it **builds conditions** that must be true for the code path taken. These are called **path conditions**.
  Think of it like:
- Instead of saying "If 5 > 0", it says "If x > 0".
- And it keeps track of "what must be true" for each way the program could run.

### Why do Symbolic Execution?

Because it helps:

- Check **all possible behaviors** of a program for **all inputs** at once.
- Find **bugs** that normal testing (with few inputs) might miss.

## Example

Suppose you have a code:

```
def check(n):
    if n > 0:
        return "positive"
    else:
        return "non-positive"
```

- In **normal execution**, you might test `n = 5` or `n = -3`.
- In **symbolic execution**, you test with `n = x`, a **symbol**.
  You get **two paths**:
- If `x > 0`: return "positive"
- If `x <= 0`: return "non-positive"
  And **path conditions** are
- `x > 0` for positive
- `x <= 0` for non-positive

## Symbolic Execution Tree

While running symbolically, a **tree** is built:

- Each **branch** in the tree happens when the code checks a condition (like an `if`).
- **Ovals** represent decision points (like `if (n > 0)`).
- **Rectangles** at the bottom are the end points (program outputs).

📖 **Tree parts:**

- **Condition** at a branch: like "is x > 0?"
- **Outgoing edges**: different possibilities (Yes or No).
- **Leaf nodes**: final result with some specific conditions.

## *8. Discuss any 2 techniques of grey box testing*

**Grey box testing** is a mix of **black box** (no code knowledge) and **white box** (full code knowledge) testing.
In grey box, the tester knows **some internal details** of the system but not everything.

The main techniques used are:

### 1. Matrix Testing

- **What it is:**
  In this method, we **list and track all the variables** in the program.
- **Why:**
  To check how variables are used and how they interact with each other.
- **Example:**
  Suppose a program has variables like `username`, `password`, and `email`.
  Matrix testing ensures we know:
  - Which functions use these variables
  - How they are changed
  - If they are properly validated

### 2. Regression Testing

- **What it is:**
  After **changing** or **fixing** something in the code, we check if **other parts** of the program got **broken** accidentally.
- **Strategies used:**
  - **Retest All:** Run **all** old tests again.
  - **Retest Risky Use Cases:** Only test the parts most likely to be broken.
  - **Retest Within a Firewall:** Only test the **affected modules**.
- **Example:**
  You fixed a bug in the "login" page.
  Now, you must check if "login" works **and** make sure that "signup" or "profile update" features didn't break because of your change.

### 3. Orthogonal Array Testing (OAT)

- **What it is:**
  It's a **smart way** to cover **maximum combinations** of inputs **with very few test cases**.
- **Why:**
  Saves time and ensures important input combinations are tested without needing to test every single one.
- **Example:**
  If you have 3 input fields, each with 5 options, testing all combinations would mean 5×5×5 = 125 tests.
  But with OAT, you might only need around **10-15 tests** to cover almost everything important!

### 4. Pattern Testing

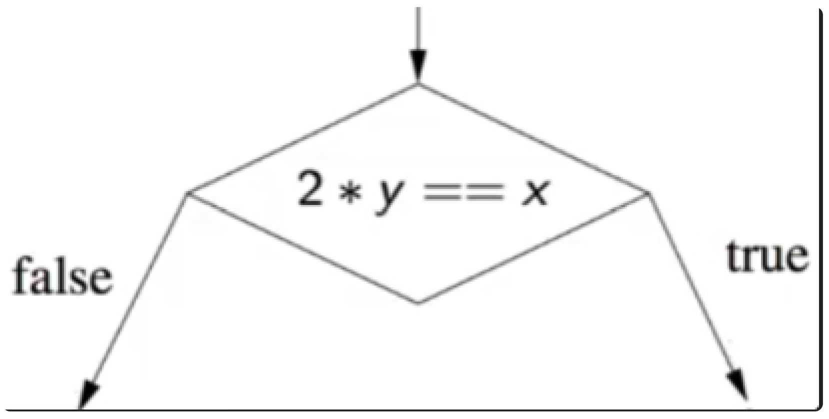
- **What it is:**
  Use **past defect data** (old bugs) to **find patterns** and **predict where bugs are likely** to happen again.
- **Unlike black box testing:**
  In grey box, we **dig inside the code** to understand **why** the old bugs happened.
- **Example:**
  If many bugs were found earlier in the payment module because of wrong calculations, then during testing, you **focus more** on that module.

## *9. Draw the symbolic execution tree for the following program code and execution of testme (α1,α2)*

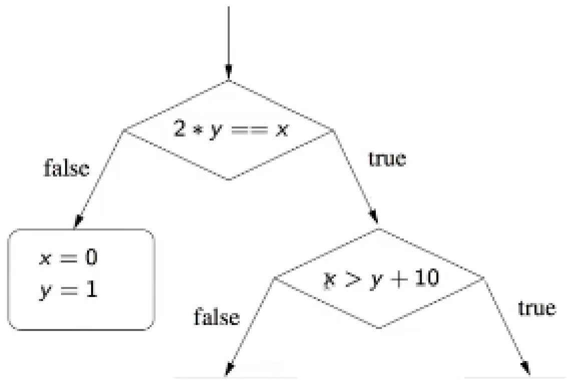
```
int twice(int v) {
    return 2 * v;
}

void testme(int x, int y) {
  int z = twice(y);
  if (z == x) {
      if (x > y + 10) {
          ERROR;
       }
   }
```

```c
12 }
13
14 int main() {
15     int x = sym_input();
16     int y = sym_input();
17     testme(x, y);
18     return 0;
19 }
```

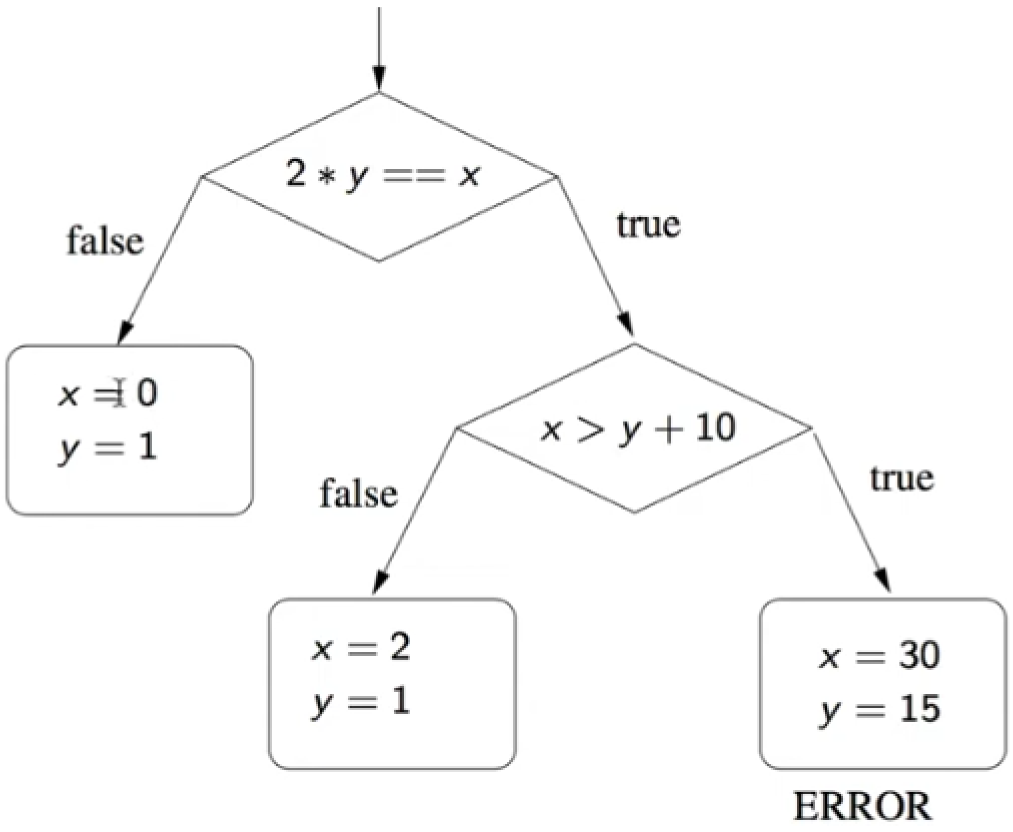


- In line no 7
  1. This is the first place where the decision takes place
  2. It checks if z == x
     1. If its true then goes to next if condition at line 8
     2. otherwise it exists the program
  3. So in the above tree you can see 2 * y == x
     1. Here 2 * y is z



- If the condition is true then as per line 8 we check x > y + 10
- If the condition is false, then we exit the program
  - An example where this can fail is

- x = 0
- y = 1
- 2 * y == x
  - 2 * y = 2
  - x = 0
  - 2 !=0
- So its false
- We draw the same in the tree, an example where it fails (x = 0, y=1)



- In line no 8
  - if x > y + 10 is true
    - An example of when this can happen is
      - x = 2
      - y = 1
  - If x > y +10 is false
    - An example of when this can happen is
      - x = 30
      - y = 15

- This returns an error

## *10. What is parameterized unit testing? How Pex is useful in generating parameterized unit tests?*

Normally, a **unit test** is a small piece of code that tests a specific part of a program, **without any input**.

**Parameterized Unit Testing** is a **smarter version**:

- Instead of hardcoding inputs, the test **accepts parameters**.
- Then, **different inputs** can be **automatically passed** to the test.
- You can check **many different cases** with the **same test code**.

### Why Parameterized Unit Tests are Useful?

- Save Time: You don't have to write the same test again and again for different inputs.
- Better Coverage: You can test with many inputs and catch more bugs.
- Easy to Maintain: Changing the test once automatically covers all inputs.

### What is Pex and How is it Useful?

**Pex** is a tool made for **.NET languages** that makes **parameterized unit testing automatic** and **very powerful**.
**How Pex Helps:**

- Pex **analyzes** your **Parameterized Unit Test** and the **code you are testing**.
- It uses **Dynamic Symbolic Execution** (smart execution tracking) to:
  - Watch how your code behaves.
  - Find important and interesting test cases (especially where bugs might happen).
- Pex then **automatically creates test inputs** that cover:
  - Different code paths
  - Edge cases
  - Potential bugs
- If a generated test **fails**, Pex can sometimes **suggest what caused the bug**.
- Pex ensures you get **very high code coverage** with **only a small number of tests**.

- It helps in **both functional testing** (what the program should do) and **structural testing** (how the program behaves internally).

## *11. Discuss orthogonal array testing and explain how it reduces the number of test cases with a suitable example.*

**Orthogonal Array Testing (OAT)** is a **smart testing technique** used when:

- There are **lots of possible input combinations**.
- We want to **test efficiently** without checking every single combination.
- **Goal:** Achieve **maximum coverage** with **minimum number of test cases**.

### Why use Orthogonal Array Testing?

When an application has **many inputs**, each with **different possible values**,
testing **every possible combination** becomes **impossible** because the number of combinations grows very large.

- **Example problem:**
  If you have 4 fields (A, B, C, D), and each field can take 3 different values (1, 2, 3),
  Total combinations = 3 × 3 × 3 × 3 = **81 test cases!***
  **Testing 81 combinations would take** too much time and effort**

### How Orthogonal Array Testing helps?

- OAT helps to **pick a smart subset** of test cases.
- This subset **covers all important combinations** without testing all 81 possibilities.
- OAT uses special mathematical tables called **Orthogonal Arrays** to select these smart combinations.
- In the previous example, **using OAT**, you may only need **9 test cases** instead of 81! (And still cover all important variable interactions.)

### Simple Example

Suppose we are testing a **smartphone** with the following options:

| Feature | Options |
|---|---|
| Screen Size | Small, Medium, Large |
| Battery Type | Li-ion, NiMH, NiCd |
| Color | Black, White, Gold |

**Without OAT:**

Total test cases = 3 × 3 × 3 = 27

**With OAT:**

Using a simple Orthogonal Array, you may only need around **9 test cases**!

Here's an example set of smart test cases

| Test No | Screen Size | Battery Type | Color |
|---|---|---|---|
| 1 | Small | Li-ion | Black |
| 2 | Small | NiMH | White |
| 3 | Small | NiCd | Gold |
| 4 | Medium | Li-ion | White |
| 5 | Medium | NiMH | Gold |
| 6 | Medium | NiCd | Black |
| 7 | Large | Li-ion | Gold |
| 8 | Large | NiMH | Black |
| 9 | Large | NiCd | White |

Notice:

- Each option (Small, Medium, Large, etc.) appears equally and in different combinations.
- We **still cover** all possibilities in an intelligent way.

## *12. Write a short note about regression testing*

### Introduction

**Regression Testing** is a software testing technique that ensures that recent code changes have **not adversely affected** the existing features and functionalities of a software application. Whenever a change like a **bug fix**, **enhancement**, or **new feature** is made, regression testing is performed to check that the system continues to behave correctly.

The **main goal** is to **catch unintended side effects** introduced by new code modifications.

## Why is Regression Testing Important?

- When software evolves, even a small change can **break** previously working features.
- Regression testing **protects stability** and **ensures reliability** of the system over time.
- It helps detect **unexpected bugs** that may arise **indirectly** due to changes.
- It is a **critical activity** for maintaining the **quality of software** during maintenance and updates.

## When is Regression Testing Performed?

Regression testing is usually carried out:

- After **bug fixes**.
- After **new features** are added.
- After **code optimization** (like performance improvement).
- After **environment changes** (e.g., database upgrade, OS upgrade).
- Before a **major release** to make sure everything is still working.

## Process of Regression Testing

1. **Identify** test cases that may be affected by the change.
2. **Select** relevant test cases (from previous tests) for re-execution.
3. **Prioritize** test cases based on critical functionalities.
4. **Update** old test cases if the software behavior has changed.
5. **Execute** the selected test cases.
6. **Report** any new defects found during testing.
7. **Repeat** whenever new changes are introduced.

## Advantages of Regression Testing

- **Early detection** of defects caused by code changes.

- Maintains **software quality** during continuous development.
- Helps developers feel **more confident** about releasing software.
- Saves **cost** and **effort** in the long term by avoiding bigger failures later.
- Supports **continuous integration and deployment** (CI/CD) practices.

## Example of Regression Testing

Suppose an e-commerce app adds a new feature for applying coupon codes at checkout.
In regression testing, apart from testing the new coupon feature, testers will **retest**:

- The cart functionality
- The payment gateway
- The product quantity update
- The overall checkout flow

This ensures that the new code **has not broken** anything in the existing checkout process.