Module 5

farisbashatm@gmail.com

Introduction to Grey Box Testing

What is Grey Box Testing?

Grey Box Testing is a software testing technique that combines elements of both Black Box Testing and White Box Testing. In Grey Box Testing, the tester has partial knowledge of the internal workings of the application. This approach allows testers to design test cases based on an understanding of the internal data structures and algorithms, while still focusing on the external functionality of the application.

Why Grey Box Testing?

Grey Box Testing is performed for several reasons:

- **Improved Product Quality**: By combining the insights of both developers and testers, Grey Box Testing enhances the overall quality of the product.
- **Balanced Approach**: It leverages the strengths of both Black Box and White Box Testing, providing comprehensive coverage.
- **Efficient Defect Identification**: It helps in identifying defects related to improper code structure or misuse of applications.
- **User-Centric Testing**: Testing is done from the user's perspective, ensuring that the application meets user expectations.

Grey Box Methodology

The methodology for Grey Box Testing involves the following steps:

1. **Identify Inputs**: Determine the inputs that the software program will accept.
2. **Identify Outputs**: Define the expected outputs for the given inputs.
3. **Identify Major Paths**: Map out the primary pathways through which data flows in the application.
4. **Identify Subfunctions**: Break down the application into smaller subfunctions or modules.
5. **Develop Inputs for Subfunctions**: Create inputs for each subfunction.
6. **Develop Outputs for Subfunctions**: Define the expected outputs for each subfunction.
7. **Execute Test Cases**: Run the test cases for each subfunction.
8. **Verify Results**: Check the results to ensure they match the expected outputs.

9. **Repeat**: Repeat the process for other subfunctions and major paths.

Advantages and Disadvantages

Advantages

- **Combined Benefits**: Offers the combined benefits of Black Box and White Box Testing.
- **Improved Quality**: Enhances overall product quality by combining developer and tester inputs.
- **Efficient Defect Fixing**: Provides developers with more time to fix defects.
- **User-Centric**: Testing is done from the user's point of view, ensuring better user experience.

Disadvantages

- **Limited Code Access**: Complete White Box Testing cannot be performed due to limited access to the source code.
- **Complex Test Design**: Designing test cases can be challenging due to partial knowledge of the internal structure.
- **Not Suitable for Distributed Systems**: Difficult to associate defects in distributed systems.

Techniques of Grey Box Testing

1.Matrix Testing

Matrix Testing is a technique used to evaluate all the variables in a program, considering both business and technical risks. This method helps prioritize testing based on the importance and risk associated with each variable.

Key Points:

- **Identification of Variables**: All variables in the program are identified and listed.
- **Risk Assessment**: Each variable is assessed for business and technical risks.
- **Test Case Design**: Test cases are designed based on the matrix structure, which represents different combinations of inputs, conditions, or variables.
- **Execution and Evaluation**: Test cases are executed, and the results are evaluated to ensure the correct and efficient utilization of variables.

Advantages:

- **Risk-Based Prioritization**: Focuses on testing high-risk variables, ensuring critical parts are thoroughly tested.

- **Comprehensive Coverage**: Ensures all variables and their combinations are considered, leading to more thorough testing.
- **Structured Approach**: Provides a clear, organized method for designing test cases, which can improve test planning and execution.

Disadvantages:

- **Complexity**: Can become complex and time-consuming to identify and assess all variables in large applications.
- **Resource Intensive**: Requires significant effort and resources to create and maintain the matrix, especially for large systems.

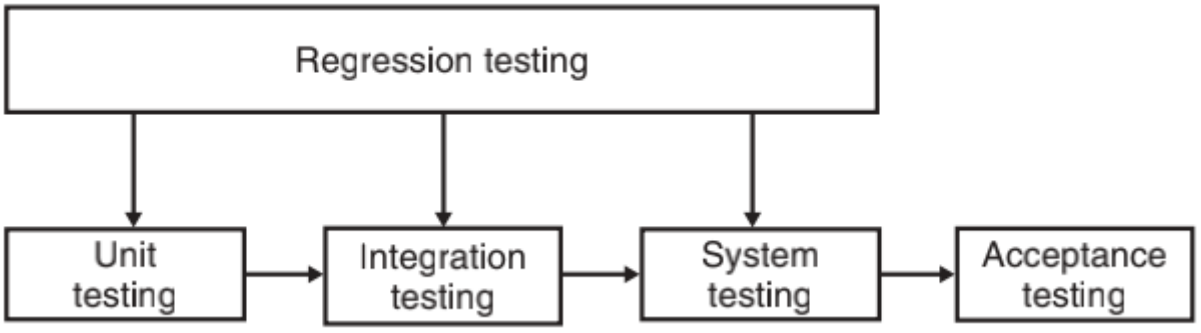
Example:

Consider a login module with variables such as username, password, and security questions. The matrix might look like this:

Test Case	Username	Password	Security Question	Expected Outcome
TC1	Valid	Valid	Valid	Success
TC2	Valid	Invalid	Valid	Failure
TC3	Invalid	Valid	Valid	Failure
TC4	Valid	Valid	Invalid	Failure

2.Regression Testing

Regression Testing ensures that new changes or updates do not adversely affect the existing functionalities of the software. It involves retesting the software after modifications to verify that new changes have not introduced new defects.



Key Points:

- **Retest All**: All test cases are re-executed to ensure no new defects are introduced.
- **Retest Risky Use Cases**: Only the test cases related to the modified areas are re-executed.

- **Regression Test Selection**: A subset of test cases is selected based on the impact analysis of the changes.

Advantages:

- **Stability Assurance**: Ensures new changes don't break existing functionalities, maintaining software stability.
- **Continuous Quality**: Facilitates continuous integration and delivery by ensuring ongoing quality with each release.
- **Defect Identification**: Helps identify defects introduced by recent code changes, improving overall software reliability.

Disadvantages:

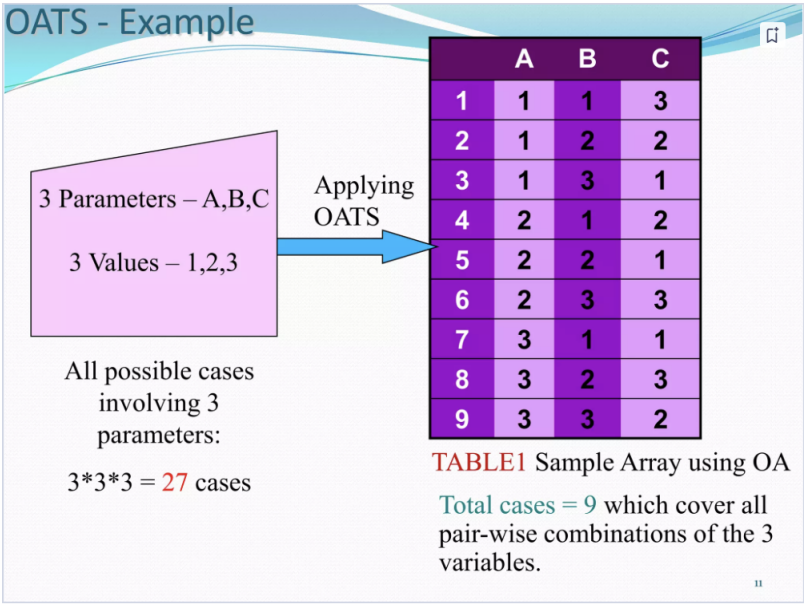
- **Resource Intensive**: Can be time-consuming and resource-heavy, especially for large applications with extensive test suites.
- **Redundancy**: May lead to redundant testing if not managed properly, causing unnecessary repetition.

Example:

If a new feature is added to a shopping cart application, regression testing would involve retesting the existing functionalities like adding items to the cart, removing items, and checking out to ensure they still work correctly.

3.Orthogonal Array Testing (OAT)

Orthogonal Array Testing (OAT) is a statistical testing technique that aims to provide maximum coverage with a minimum number of test cases. It is particularly useful for testing complex applications with numerous permutations and combinations.



Key Points:

- **Systematic Selection**: Specific combinations of inputs are systematically selected from an orthogonal array.
- **Pairwise Testing**: Ensures that all pairs of input combinations are tested at least once.
- **Efficiency**: Reduces the number of test cases needed while maximizing coverage.

Advantages:

- **Efficiency**: Achieves high coverage with fewer test cases, saving time and resources.
- **Systematic Approach**: Ensures all pairs of input combinations are tested, increasing the likelihood of uncovering defects.
- **Scalability**: Can be applied to a wide range of applications, making it versatile for different testing scenarios.

Disadvantages:

- **Limited Interaction Testing**: May not cover all possible interactions between variables, potentially missing some defects.
- **Initial Setup**: Requires careful planning and setup to create the orthogonal array, which can be complex.

Example:

Consider a web page with three sections (Top, Middle, Bottom) that can be shown or hidden. The orthogonal array might look like this:

Test Case	Top	Middle	Bottom
TC1	Hidden	Hidden	Hidden
TC2	Hidden	Visible	Visible
TC3	Visible	Hidden	Visible
TC4	Visible	Visible	Hidden

This ensures that all combinations of visibility states are tested with just four test cases instead of the six required by conventional testing.

4.Pattern Testing

Pattern Testing involves analyzing historical data of previous defects to identify common failure patterns. This proactive approach helps in designing test cases that can detect similar defects in the current software.

Key Points:

- **Historical Analysis**: Previous defects are analyzed to identify recurring patterns.

- **Proactive Testing**: Test cases are designed based on identified patterns to detect similar defects.
- **Focus on High-Risk Areas**: Testing is focused on areas with a high likelihood of defects based on historical data.

Advantages:

- **Proactive Defect Detection**: Identifies and targets recurring defect patterns, improving defect detection rates.
- **Focused Testing**: Directs testing efforts towards areas with a high likelihood of defects based on historical data, improving efficiency.
- **Continuous Improvement**: Uses historical data to refine and enhance the testing process over time.

Disadvantages:

- **Dependency on Historical Data**: Requires accurate and comprehensive historical defect data, which may not always be available.
- **Potential Bias**: May lead to biased testing if over-reliance on past patterns occurs, potentially missing new or unique defects.

Example:

If historical data shows that most defects in a project management application occur in the scheduling module, pattern testing would involve designing test cases specifically targeting the scheduling functionality to detect similar defects.

==========

An Introduction to PEX - Parameterized Unit Testing

Parameterized Unit Testing

Parameterized unit testing is a technique that allows testing a function or method with a variety of inputs and expected outputs. This approach helps identify bugs and edge cases that may not be caught by testing a function with a single input. By testing a function with a range of inputs, including edge cases, we can ensure that the function behaves correctly for all possible inputs. Parameterized unit testing also saves time and effort by automating the testing process, making it particularly useful for functions with a large number of possible inputs

Overview of PEX

Pex is a tool developed by Microsoft Research for parameterized unit testing in .NET applications. It generates test cases automatically based on the function or method signature and the input constraints specified by the developer. Pex can also generate code coverage reports and identify code paths that are not covered by the tests. It uses a technique called symbolic execution to generate test cases, which involves analyzing a program by executing it with symbolic values instead of concrete values. This allows Pex to explore all possible code paths and generate test cases that cover all possible inputs and edge cases

How to Write Parameterized Unit Tests with PEX

1. **Define the function or method to be tested**: Start by identifying the function or method that needs to be tested.
2. **Add input constraints**: Specify the range of values or types that the inputs can take using Pex attributes or custom code.
3. **Generate tests**: Use Pex to generate tests by running the Pex explorer or using the Pex API in your code.
4. **Review and refine the tests**: After generating the tests, review and refine them to remove any redundant or unnecessary tests and add additional tests to cover edge cases.
5. **Run the tests**: Execute the tests using Visual Studio's Test Explorer or the Pex API. The results will be displayed in the test explorer.
6. **Analyze the results**: Analyze the test results to ensure that the function or method behaves correctly for all possible inputs. Use the code coverage reports generated by Pex to identify code paths that are not covered by the tests

Example of Parameterized Unit Testing with PEX

Here is an example of a parameterized unit test using Pex:

```
public static int Add(int a, int b)
{
    return a + b;
}

public void TestAdd(int a, int b)
{
    PexAssume.IsTrue(a >= int.MinValue && a <= int.MaxValue);
    PexAssume.IsTrue(b >= int.MinValue && b <= int.MaxValue);
    int result = Add(a, b);
    PexAssert.AreEqual(result, a + b);
}
```

In this example, the `Add` function is tested with a range of integer values for `a` and `b`. The `PexAssume` statements specify the input constraints, and the `PexAssert` statement checks that the result of the `Add` function is as expected

The Testing Problem

Testing software is a complex task that involves ensuring that the software behaves correctly for all possible inputs and scenarios. Traditional unit testing involves writing individual test cases for specific inputs, which can be time-consuming and may not cover all possible edge cases. Parameterized unit testing, as enabled by Pex, addresses this problem by automating the generation of test cases and ensuring high code coverage

---

==========

Symbolic Execution

Symbolic execution analyzes programs using symbolic inputs instead of concrete data, allowing multiple execution paths to be explored simultaneously.

Ideal Assumptions:

1. **Integer-Only Programs**: Assumes programs use only integers of arbitrary magnitude, ignoring register overflows.
2. **Infinite Execution Tree**: Acknowledges that the execution tree from symbolic execution can be infinite.
3. **Theorem Proving**: Recognizes the need for theorem proving for IF statements, which is mechanically challenging.

Path Conditions:

- **Path Condition (pc)**: A Boolean expression over symbolic inputs representing constraints for an execution path.
- **Forking Execution**: Occurs when both branches of an IF statement are possible, leading to parallel symbolic executions.

Example:

Procedure SUM:

A simple program calculates the sum of three integers:

```
1 SUM: PROCEDURE (A, B, C);
2 X := A + B;
3 Y := B + C;
```

```
4 Z := X + Y - B;
5 RETURN (Z);
6 END;
```

Symbolic execution with inputs α1, α2, α3 shows the program computes the sum α1 + α2 + α3.

Fig. 2. Execution of SUM(1, 3, 5). A dash represents unchanged values, i.e. the same values as those given in the line above; the question mark represents undefined (uninitialized) values.

After statement	X	Y	Z	A	B	C
			Values would be			
1	?	?	?	1	3	5
2	4	—	—	—	—	—
3	—	8	—	—	—	—
4	—	—	9	—	—	—
5			(Returns 9)			

Fig. 3. Symbolic execution of SUM (α_1, α_2, α_3). Path condition is abbreviated *pc*.

After statement	X	Y	Z	A	B	C	*pc*
			Values would be				
1	?	?	?	α_1	α_2	α_3	*true*
2	$\alpha_1+\alpha_2$	—	—	—	—	—	—
3	—	$\alpha_2+\alpha_3$	—	—	—	—	—
4	—	—	$\alpha_1+\alpha_2+\alpha_3$	—	—	—	—
5			(Returns $\alpha_1+\alpha_2+\alpha_3$)				

Applications / Advantages of Symbolic Execution

- Detect infeasible paths
- Generate test inputs
- Find bugs and vulnerabilies
- Generating program invariants
- Prove that two pieces of code are equivalent
- Debugging
- Automated program repair

Symbolic Execution Tree

A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a procedure.

- **Nodes**: Each node represents a statement execution, labeled with the statement number.

- **Arcs**: Directed arcs connect nodes, representing transitions between statements. For forking IF statements, nodes have two arcs labeled "T" (True) and "F" (False) for the THEN and ELSE parts, respectively.
- **Execution State**: Each node includes the current execution state, i.e., variable values, statement counter, and path condition (pc).

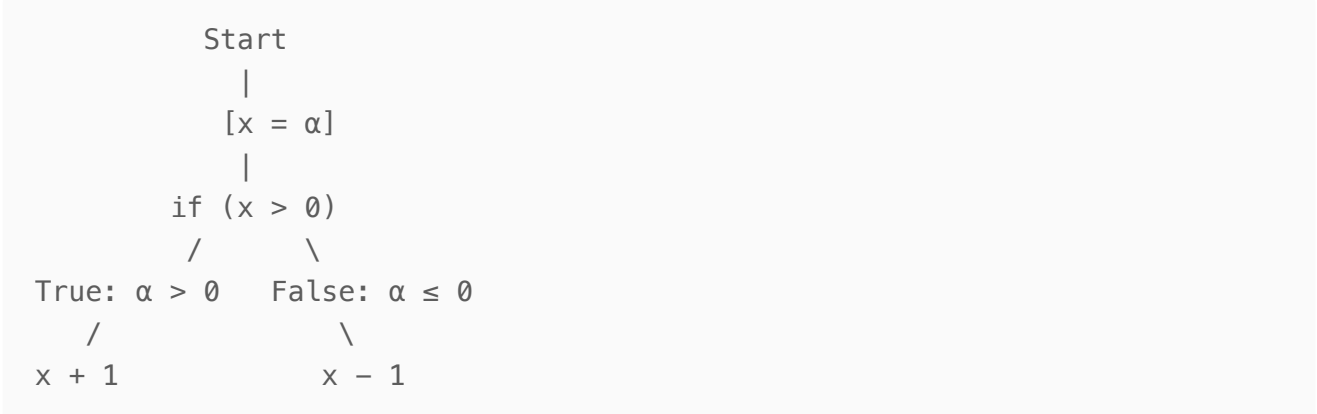
Steps to Construct a Symbolic Execution Tree:

1. **Initialization**: Start with the initial state where all inputs are symbolic, and the path condition is true.
2. **Execution**: Execute the program symbolically, updating the symbolic state and path condition at each step.
3. **Forking**: At each conditional statement, fork the execution into two paths:
    - **True Path**: Add the condition to the path condition and continue execution.
    - **False Path**: Add the negation of the condition to the path condition and continue execution.
4. **Node Creation**: Create a node for each executed statement, and connect nodes with directed edges representing the flow of control.
5. **Path Conditions**: Accumulate path conditions along each path to represent the constraints on inputs for that path.
6. **Termination**: Continue until all paths are explored or a specified limit is reached.

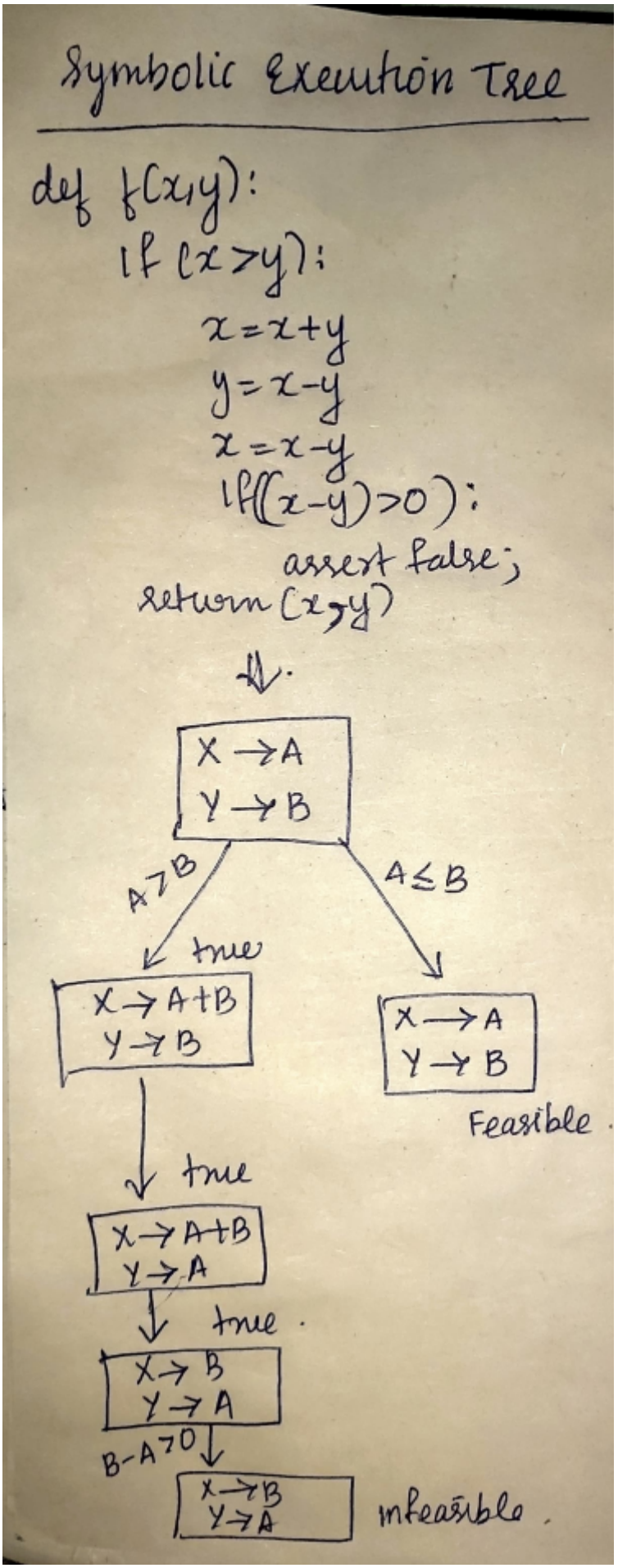
Simple Example

Consider the below Program:

```
int f(int x) {
    if (x > 0) {
        return x + 1;
    } else {
        return x - 1;
    }
}
```



Advanced Example



Symbolic Execution / Tree Problem

Checkout youtube or this paper https://madhu.cs.illinois.edu/cs598-fall10/king76symbolicexecution.pdf

Question & Answer

Regression Testing	Orthogonal Array Testing
Ensures code changes don't break existing func.	Systematically selects tests covering interactions
Focuses on retesting post-code modifications.	Efficiently covers parameter combinations.
Validates previous functionalities after changes.	Planned for comprehensive parameter interaction.
Reruns or creates test cases based on changes.	Selects cases using orthogonal arrays.
Detects regression bugs caused by code changes.	Identifies interactions leading to potential defects.
