Module - 5 (Grey Box Testing Approaches)

Introduction to Grey Box testing - Why Grey Box testing, Gray Box Methodology, Advantages and Disadvantages. Techniques of Grey Box Testing - Matrix Testing, Regression Testing, Orthogonal Array Testing or OAT, Pattern Testing. An Introduction to PEX - Parameterized Unit Testing, The Testing Problem. Symbolic Execution – Example, Symbolic execution tree. PEX application Case Study – PEX.

Grey Box Testing

- **Grey Box Testing** or Gray box testing is a software testing technique to test a software product or application with partial knowledge of internal structure of the application.
- The purpose of grey box testing is to search and identify the defects due to improper code structure or improper use of applications.
- In this process, context-specific errors that are related to web systems are commonly identified. It increases the testing coverage by concentrating on all of the layers of any complex system.

- Gray Box Testing is a software testing method, which is a combination of both White Box Testing and Black Box Testing method.
- In White Box testing internal structure (code) is known
- In Black Box testing internal structure (code) is unknown
- In Grey Box Testing internal structure (code) is partially known

- In Software Engineering, Gray Box Testing gives the ability to test both sides of an application, presentation layer as well as the code part. It is primarily useful in Integration Testing and Penetration Testing.
- **Example of Gray Box Testing:** While testing websites feature like links or orphan links, if tester encounters any problem with these links, then he can make the changes straightaway in HTML code and can check in real time.

Why Gray Box Testing

- Gray Box Testing is performed for the following reason,

➢it provides combined benefits of both black box testing and white box testing both

➢It combines the input of developers as well as testers and improves overall product quality

➢It reduces the overhead of long process of testing functional and non-functional types

➢It gives enough free time for a developer to fix defects

➢Testing is done from the user point of view rather than a designer point of view

- **Gray Box Testing Strategy**
- To perform Gray box testing, it is not necessary that the tester has the access to the source code. A test is designed based on the knowledge of algorithm, architectures, internal states, or other high -level descriptions of the program behavior.
- To perform Gray box Testing-
- It applies a straightforward technique of black box testing
- It is based on requirement test case generation, as such, it presets all the conditions before the program is tested by assertion method.

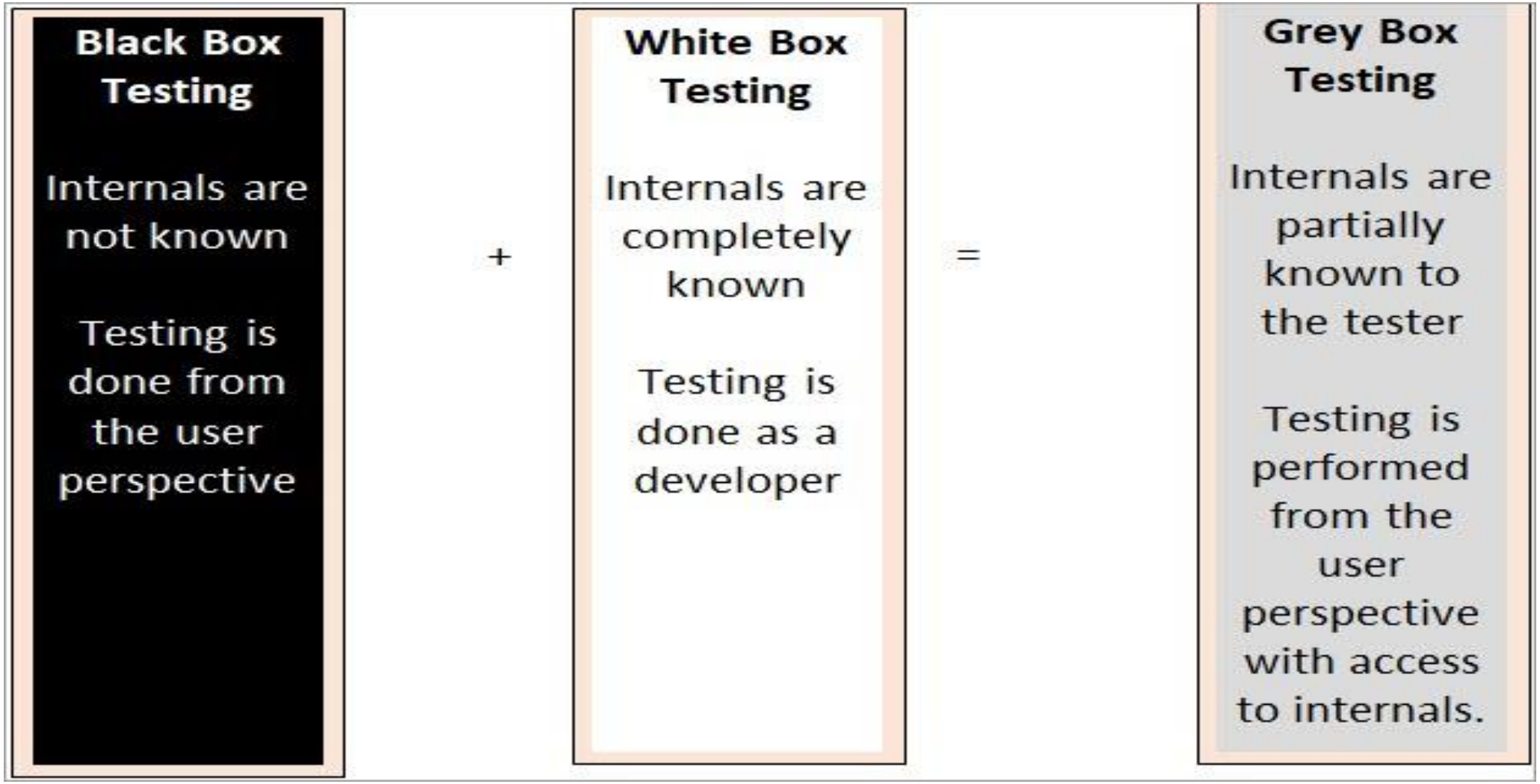
- **Techniques used for Grey box Testing are-**

- **Matrix Testing:** This testing technique involves defining all the variables that exist in their programs.
- **Regression Testing**: To check whether the change in the previous version has regressed other aspects of the program in the new version. It will be done by testing strategies like retest all, retest risky use cases, retest within a firewall.
- **Orthogonal Array Testing or OAT**: It provides maximum code coverage with minimum test cases.
- **Pattern Testing:** This testing is performed on the historical data of the previous system defects. Unlike black box testing, gray box testing digs within the code and determines why the failure happened

- Steps to perform Grey box Testing are:
- Step 1: Identify inputs
- Step 2: Identify the outputs
- Step 3: Identify the major paths
- Step 4: Identify Subfunctions
- Step 5: Develop inputs for Subfunctions
- Step 6: Develop outputs for Subfunctions
- Step 7: Execute test case for Subfunctions
- Step 8: Verify the correct result for Subfunctions
- Step 9: Repeat steps 4 & 8 for other Subfunctions
- Step 10: Repeat steps 7 & 8 for other Subfunctions

- **Gray Box Testing Challenges**
- When a component under test encounter a failure of some kind may lead to abortion of the ongoing operation
- When test executes in full but the content of the result is incorrect.

- Grey box tester has knowledge of the code, but not completely.



- **Techniques For Grey Box Testing**
- **Let’s discuss the techniques used for Grey Box testing:**

1.Matrix testing

2.Regression testing

3.Orthogonal Array testing

4.Pattern Testing

- **#1) Matrix Testing**
- Software developers provide all the variables in a program along with the technical and business risks that are linked with them. The matrix testing technique tests the risks defined by the developers.
- Matrix technique states all the used variables in a program. This technique helps to identify and remove the variables which are not being used in the program and in turn, helps to increase the speed of the software.

- **#2) Regression Testing**
- Regression testing is performed when any change is done in the software or any defect is fixed. It is done to ensure that a new change or fix done has not impacted any existing functionality of the software.
- **#3) Orthogonal Array Testing or OAT**
- This testing technique is used more for complex functionalities or applications, as this technique is utilized when maximum coverage of code is required with minimum test cases and has large test data with n number of combinations.
- **#4) Pattern Testing**
- Pattern Testing is performed based on the previous defects found in the software. Defect record is analyzed for the cause of defects and test cases are created keeping the defects and their cause in knowledge to find defect before the software goes into production.

Advantages Of Gray Box Testing

- The quality of the software gets improved.
- This technique focuses more on user perception.
- In grey box testing developers are benefitted as they get enough time for bug fixing.
- As grey box testing is a combination of both black box and white box, the benefits of both are acquired.
- Grey box testers do not require having high programming knowledge for testing the product.
- This testing technique is effective in Integration testing.
- This testing technique helps to have no clashes between the developer and the tester.
- Complex applications and scenarios can be tested effectively with this technique.
- This testing technique is non-intrusive.

- **Disadvantages Of Gray Box Testing**
- **These are as follows:**
- Complete white box testing cannot be performed in grey box testing, as a source cannot be accessed.
- For a distributed system, it becomes difficult to associate defects in this testing technique.
- Test case creation for grey box testing is complex.
- Because of limited access, code path traversal access also gets limited.

An Introduction to PEX

- Pex enables parameterized unit testing, an extension of unit testing that reduces test maintenance costs.
- A parameterized unit test is simply a method that takes parameters, calls the code under test, and states assertions.
- Pex analyzes the code in the parameterized unit test together with the code-under-test, attempting to determine interesting test inputs that might exhibit program crashes and assertion violations.
- Pex learns the program behavior by monitoring execution traces, using a constraint solver to produce new test cases with different behavior.

- Parameterized unit test (PUT) is simply a method that takes parameters, calls the code under test, and states assertions.
- Given a PUT written in a .NET language, Pex automatically produces a small test suite with high code and assertion coverage.
- Moreover,when a generated test fails, Pex can often suggest a bug fix. To do so, Pex performs a systematic program analysis, similar to path bounded model-checking.
- Pex learns the program behavior by monitoring execution traces, and uses a constraint solver to produce new test cases with different behavior

- Pex, an automated test input generator, leverages dynamic symbolic execution to test whether the software under test agrees with the specification.
- As a result, software development becomes more productive and the software quality increases.
- Pex produces a small test suite with high code coverage from Parameterized Unit Tests

- we combine two kinds of testing introduced
- 1) Testing for functional properties: Just as unit tests, Parameterized Unit Tests usually serve as specifications of functional properties.
- 2) Structural testing: We analyze such tests with dynamic symbolic execution, a structural testing technique.

- What are unit tests?
- A unit test is a self-contained program that checks an aspect of the implementation under test. Here is an example of a unit test that checks the interplay among .NET's ArrayList operations. The example is written in C#, omitting the class context. We omit visibility modifiers like public for brevity.

```
void AddTest() {
  ArrayList a = new ArrayList(1);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[0] == o);
}
```

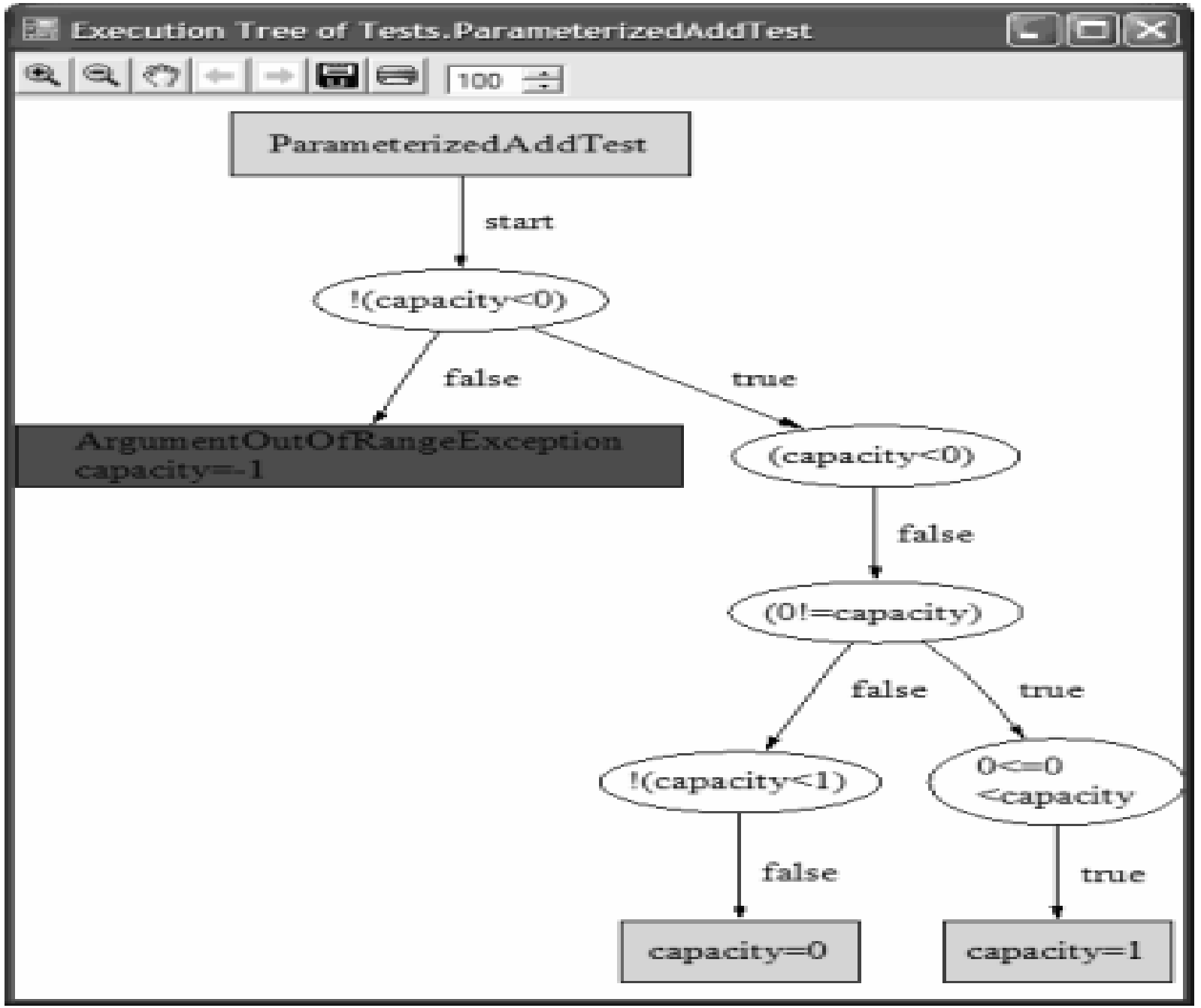
- Prameterized unit tests
- Traditional unit tests do not take inputs. A natural extension would be to allow parameters. For example, the above test could be parameterized over the initial capacity of the array list:

```
void ParameterizedAddTest(int capacity) {
  ArrayList a = new ArrayList(capacity);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[0] == o);
}
```

Symbolic execution

- Symbolic execution is a way to analyze the behavior of a program for all possible inputs.
- Instead of supplying the normal inputs to a program (e.g. concrete numeric values) one supplies symbols that represent arbitrary values.
- Symbolic execution proceeds like normal execution except that the values computed may be expressions over the input symbols.
- Symbolic execution builds a path condition over the input symbols.
- A path condition is a mathematical formula that encodes data constraints that result from executing a given code path.

- Symbolic execution systematically unfolds loops and recursion.
- The resulting number or length of paths might be too large to analyze (or even infinite).
- In general the existence of unbounded numbers of execution paths due to loops and recursion is a limitation of symbolic execution. However, approximation techniques can be applied.
- However, approximation techniques can be applied. For example, it is possible to analyze loops and recursion up to a fixed number of iterations


Execution Tree of Tests.ParameterizedAddTest
100
ParameterizedAddTest
start
!(capacity<0)
false
true
ArgumentOutOfRangeException
capacity=-1
(capacity<0)
false
(0!=capacity)
false
true
!(capacity<1)
0<=0
<capacity
false
true
capacity=0
capacity=1

- Figure 3. UnitMeister’s visualization of possible execution paths for for ParameterizedAddTest.
- In the tree, the condition (0 != capacity) is the condition of the above if-condition in terms of the test’s symbolic input capacity.
- The ovals show the conditional branch points encountered on each execution path.
- The outgoing edges represent possible evaluations of the conditions. The rectangles represent path terminations with exemplary concrete assignments

- Deciding feasibility
- A path is infeasible if no concrete inputs exist that would cause this path to be taken.
- In such cases the path condition is self-contradicting. Such infeasible paths can be pruned away during symbolic execution.
- Constraint solving and automatic theorem proving techniques can often decide whether a path condition is feasible.
- For example, efficient decision procedures for linear arithmetic problems exist.
- If the implemented decision procedures cannot decide the feasibility of a path condition , we can either prune the path, which leads to an under-approximation of the possible behaviors of the program (not all possible behaviors will be found) or we can include it in the exploration, which leads to an over-approximation (execution paths are considered which cannot arise in reality).

- Reusing parameterized unit tests in symbolic execution
- Decision procedures often simplify expressions according to certain rules in order to better reason about them.
- For example, a + 0 simplies to a. Parameterized unit tests can be interpreted as rules, too!
- We can rewrite our unit test mathematically as a universally quantified conditional expression.
- A universally quantified expression says that for all x some Boolean condition p(x) holds.
- For example, the AddSpec says that for every ArrayList a and object o, we have (a == null or let len be a.Count in (a.Add(o) followed by a[len] == o)). Once the validity of a parameterized unit test has been established we can add it as a rule.
- As a consequence, symbolic execution does not have to execute the code of the Add method and the index operator anymore, but it can treat the array list operations like integer operations when building up and reasoning about constraints.
- When software is designed as a layered system, this technique can help make symbolic execution scale