EXPERIMENTS, RESULT ANALYSIS AND SYSTEM TESTING

4.1 EXPERIMENTS FOR ADAPTER PATTERN

The DPI is given as an input the system which has adapter design pattern implemented in its design. This DPI should be able to extract pattern related information i.e. the name of the pattern and its components and their role names. The UML document that is provided as input signal is as shown in Fig. 4.1

![Class diagram comprising Adapter Design Pattern](image)

Figure 4.1: Class diagram comprising Adapter Design Pattern

The source code for the same is as proven in the next listing

**MyAdaptee.java**

```java
public class MyAdaptee {
    public void xyz() {
        // TODO Auto-generated method stub
    }
}
```

**MyConcreteAdapter.java**

```java
public class MyConcreteAdapter extends MyAdapter {
    private MyAdaptee adaptee = null;

    @Override
```
public void operation() {
    adaptee.xyz();
}

MyAdapter.java
public class MyAdapter {
    public void operation() {
        // TODO Auto-generated method stub
    }
}

The Results found by DPI is as shown below
UML file : C:\Users\kishor\Desktop\DPIfinalnew\res\uml\my_adapter.uml
-------------- Identifying Design Patterns --------------
Source Folder : C:\Users\kishor\Desktop\DPIfinalnew\res\source\adapter

-----------------------------------------------
Matrix
-----------------------------------------------
<table>
<thead>
<tr>
<th></th>
<th>MyConcreteAdapter</th>
<th>MyAdapter</th>
<th>MyAdaptee</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyConcreteAdapter</td>
<td>1</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>MyAdapter</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MyAdaptee</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

-----------------------------------------------
Weight
-----------------------------------------------
<table>
<thead>
<tr>
<th></th>
<th>MyConcreteAdapter</th>
<th>MyAdapter</th>
<th>MyAdaptee</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyConcreteAdapter</td>
<td>255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MyAdapter</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MyAdaptee</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-----------------------------------------------
Structural Analysis
-----------------------------------------------
Below design patterns are found :-
-----------------------------------------------
Pattern Name : ADAPTER PATTERN
   Adapter : MyAdapter
   ConcreteAdapter : MyConcreteAdapter
Adaptee : MyAdaptee

Behavioral Analysis

Below design patterns successfully passed Behavioral Analysis test :-

Pattern Name : ADAPTER PATTERN
Adapter : MyAdapter
ConcreteAdapter : MyConcreteAdapter
Adaptee : MyAdaptee

Semantic Analysis

Below design patterns successfully passed Semantic Analysis test :-

Pattern Name : ADAPTER PATTERN
Adapter : MyAdapter
ConcreteAdapter : MyConcreteAdapter
Adaptee : MyAdaptee

Done

It can be seen that in structural, behavioral and semantic analysis, Adapter design pattern is identified.

4.2 COMPOSITE DESIGN PATTERN

The DPI is given as an input the system which has composite design pattern implemented in its design. DPI should be able to extract pattern related information i.e the name of the pattern and its components and their role names.

UML document as input signal that is provided is as demonstrated in Fig 4.2

Figure 4.2: Class Diagram containing composite Design Pattern
The source code of the machine is really as given below

MyComponent.java
public class MyComponent {
    public void operation() {
    }
    public void addElement(MyComponent component) {
    }
    public void removeElement(MyComponent component) {
    }
    public MyComponent getElement(int i) {
        return null;
    }
}

MyComposite.java
import java.util.List;

class MyComposite extends MyComponent {
    List<MyComponent> components;
    @Override
    public void operation() {
        for (MyComponent component : components) {
            component.operation();
        }
    }
    @Override
    public void addElement(MyComponent component) {
        components.add(component);
    }
    @Override
    public void removeElement(MyComponent component) {
        components.remove(component);
    }
    @Override
    public MyComponent getElement(int i) {
        return null;
    }
}
MyLeaf.java
public class MyLeaf extends MyComponent {
    
    @Override
    public void operation() {
        // TODO Auto-generated method stub
        super.operation();
    }

}

The output of the DPI is as given below
UML file : C:\Users\kishor\Desktop\DPIfinalnew\res\uml\my_composite.uml
------------- Identifying Design Patterns ---------------
Source Folder : C:\Users\kishor\Desktop\DPIfinalnew\res\source\composite

-----------------------------------
Matrix
-----------------------------------
|               | MyComponent | MyComposite | MyLeaf |
-----------------------------------
| MyComponent   | 1           | 1           | 1      |
| MyComposite   | 187         | 1           | 1      |
| MyLeaf        | 17          | 1           | 1      |

-----------------------------------
Weight
-----------------------------------
| MyComponent  : 81 |
| MyComposite  : 15147 |
| MyLeaf       : 51   |

-----------------------------------
Structural Analysis
-----------------------------------
Below design patterns are found :-

Pattern Name : COMPOSITE PATTERN
    Component : MyComponent
    Composite : MyComposite
    Leaf : MyLeaf

Pattern Name : STRATEGY PATTERN
Context: MyComposite
Strategy: MyComponent
ConcreteStrategy: MyLeaf

--------------------------------
Behavioral Analysis
--------------------------------
Below design patterns successfully passed Behavioral Analysis test:

Pattern Name: COMPOSITE PATTERN
  Component: MyComponent
  Composite: MyComposite
  Leaf: MyLeaf

Pattern Name: STRATEGY PATTERN
  Context: MyComposite
  Strategy: MyComponent
  ConcreteStrategy: MyLeaf

--------------------------------
Semantic Analysis
--------------------------------
Below design patterns successfully passed Semantic Analysis test:

Pattern Name: COMPOSITE PATTERN
  Component: MyComponent
  Composite: MyComposite
  Leaf: MyLeaf

------------------- Done -------------------

Result: It is seen that in architectural and behavior two examples of design designs are discovered of which is false good i.e. method design as its construction and behaviour is comparable to Composite style pattern but it is removed in the semantic analysis stage and consequently this generates right results of the id.
4.3 APPLICATION CONTAINING MULTIPLE DESIGN PATTERNS IN PACKAGES

Figure 4.3: Class diagram of the program including several layout designs in bundles

The origin of the same is really as listed below program com.pkg1;

```java
public class TestComponent {
    public void operation() {
        // TODO Auto-generated method stub
    }
    public void addComponent(TestComponent tc) {
    }
    public void removeComponent(TestComponent tc) {
    }
    public TestComponent getComponent(int i) {
        return null;
    }
}
```
package com.pkg1;

import java.util.List;

public class TestComposite extends TestComponent {
    List<TestComponent> components = null;

    @Override
    public void operation() {
        for (TestComponent component : components) {
            component.operation();
        }
    }

    @Override
    public void addComponent(TestComponent tc) {
        components.add(tc);
    }

    @Override
    public void removeComponent(TestComponent tc) {
        components.remove(tc);
    }

    @Override
    public TestComponent getComponent(int i) {
        return components.get(i);
    }
}

package com.pkg1;

public class TestLeaf extends TestComponent {
    @Override
    public void operation() {
    }
}

package com.pkg2;

public class TestAdaptee {
    public void adapteeOperation() {
    }
}
Generic Algorithm with Mathematical Evaluation for Experiments, Result Analysis and Quality Assurance of Real Time Applications

package com.pkg2;

public class TestAdapter {
    public void operation() {
    }
}

package com.pkg2;

public class TestConcreteAdapter extends TestAdapter {
    private TestAdaptee adaptee = null;

    @Override
    public void operation() {
        adaptee.adapteeOperation();
    }
}

package com.pkg2;

public class AdditionStrategy implements Strategy {
    @Override
    public int executeStrategy(int a, int b) {
        // TODO Auto-generated method stub
        return a + b;
    }
}

package strategy;

public class Circumstance {
    private Strategy strategy = null;

    public Circumstance(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        // TODO Auto-generated method stub
        return a + b;
    }
}
return strategy.executeStrategy(a, b);
}
}

package strategy;

public interface Strategy {
    int executeStrategy(int a, int b);
}

Result: The result is as given below. DPI successfully retrieves all the design patterns and package software metrics

UML file: C:\Users\kishor\Desktop\DPIfinalnew\res\uml\multi_pkg.uml
---------- Identifying Design Patterns ----------
Source Folder: C:\Users\kishor\Desktop\DPIfinalnew\res\source\multi_pkg

<table>
<thead>
<tr>
<th></th>
<th>Child2</th>
<th>TestComponent</th>
<th>TestConcreteAdapter</th>
<th>AdditionStrategy</th>
<th>context</th>
<th>TestParent</th>
<th>Child1</th>
<th>TestComposite</th>
<th>TestLeaf</th>
<th>Strategy</th>
<th>TestClass</th>
<th>TestAdaptee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestComponent</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestConcreteAdapter</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AdditionStrategy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Context</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestParent</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Child1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestComposite</td>
<td>1</td>
<td>187</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestLeaf</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Strategy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestClass</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TestAdaptee</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Design patterns and package software metrics

Weight

Child2 : 17
TestComponent : 81
TestConcreteAdapter : 3
AdditionStrategy : 255
Context : 33
TestParent : 1
Child1 : 17

- 66 -
TestComposite : 15147
TestLeaf : 51
Strategy : 3
TestClass : 1
TestAdaptee : 3

---------------------------------------------------
Structural Analysis
---------------------------------------------------
Below design patterns are found :-

......................................
Pattern Name : ADAPTER PATTERN
  Adapter : TestAdapter
  ConcreteAdapter : TestConcreteAdapter
  Adaptee : TestAdaptee
......................................
Pattern Name : COMPOSITE PATTERN
  Component : TestComponent
    Composite : TestComposite
    Leaf : TestLeaf
......................................
Pattern Name : STRATEGY PATTERN
  Context : Context
    Strategy : Strategy
      ConcreteStrategy : AdditionStrategy
......................................
    Context : TestComposite
      Strategy : Strategy
      ConcreteStrategy : AdditionStrategy
......................................
    Context : Context
      Strategy : TestComponent
      ConcreteStrategy : TestComposite
......................................
    Context : Context
      Strategy : TestComponent
      ConcreteStrategy : TestLeaf
......................................
    Context : TestComposite
      Strategy : TestComponent
ConcreteStrategy : TestLeaf

Behavioral Analysis

Below design patterns successfully passed Behavioral Analysis test:-

Pattern Name : ADAPTER PATTERN
    Adapter : TestAdapter
    ConcreteAdapter : TestConcreteAdapter
    Adaptee : TestAdaptee

Pattern Name : COMPOSITE PATTERN
    Component : TestComponent
    Composite : TestComposite
    Leaf : TestLeaf

Pattern Name : STRATEGY PATTERN
    Context : Context
    Strategy : Strategy
    ConcreteStrategy : AdditionStrategy

    Context : TestComposite
    Strategy : Strategy
    ConcreteStrategy : AdditionStrategy

    Context : Context
    Strategy : TestComponent
    ConcreteStrategy : TestComposite
Context : Context

Strategy : TestComponent

ConcreteStrategy : TestLeaf

..............................................
Context : TestComposite

Strategy : TestComponent

ConcreteStrategy : TestLeaf

..............................................

---------------------------------------------------
Semantic Analysis
---------------------------------------------------
Below design patterns successfully passed Semantic Analysis test:-

.................................
Pattern Name : ADAPTER PATTERN
Adapter : TestAdapter

ConcreteAdapter : TestConcreteAdapter

Adaptee : TestAdaptee

.................................

.................................
Pattern Name : COMPOSITE PATTERN
Component : TestComponent

Composite : TestComposite
Leaf : TestLeaf

.................................

.................................
Pattern Name : STRATEGY PATTERN
Context : Context

Strategy : Strategy

ConcreteStrategy : AdditionStrategy

------- Software Package metrics -------
Package : com
Total Elements : 0
Generic Algorithm with Mathematical Evaluation for
Experiments, Result Analysis and
Quality Assurance of Real Time Applications

- Afferent coupling : 0
- Efferent coupling : 0
- Abstractness : 0
- Instability : 0

--------------------------------------------

Package : com.pkg2
- Total Elements : 4
- Afferent coupling : 1
- Efferent coupling : 0
- Abstractness : 50
- Instability : 0

--------------------------------------------

Package : com.pkg1
- Total Elements : 4
- Afferent coupling : 0
- Efferent coupling : 1
- Abstractness : 0
- Instability : 100

--------------------------------------------

Package : strategy
- Total Elements : 4
- Afferent coupling : 0
- Efferent coupling : 1
- Abstractness : 0
- Instability : 100

--------------------------------------------

------------------- Done -------------------
4.4 TESTING STRATEGIES

Testing is definitely an essential period within the Program Development Life Cycle. Screening must be prepared and performed methodically.

Universal areas of a test strategy

- Testing begins at the module level and works outwards.
- Different testing techniques are used at different points of time.
- Testing is done by developers and mainly for larger projects, by an independent test group.
- Testing and debugging are two different activities, but debugging should be incorporated into any testing strategy.

Different testing strategies used for system testing –

a) Unit testing
b) Integrated testing
c) System testing

4.4.1 Unit Testing

It involves testing the minimum functionality of the smallest unit of software design the modules. Using the design specification documentation as the guide, important control paths are tested to uncover errors within the boundary of the module. Module level testing involves heavy use of white box testing techniques. Specific paths are exercised in the modules control structures for complete coverage and maximum error detection. Testing can be conducted in parallel for multiple modules.

4.4.2 Integrated testing

Integration testing is a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing. The objective is to take unit tested module and build a program structure that has been dictated by design. It involves testing functionality of the modules with respect to the other modules, with which the module is supposed to interact. There is heavy use of black box testing techniques and some use of white box testing techniques to ensure coverage of major control paths.
4.5 TESTING TECHNIQUES

Computer software testing is definitely a study conducted to provide stakeholders with information regarding the standard of the item or service under evaluation. Software testing can provide a goal, independent view of the program to permit the company to comprehend and recognize the risks of software execution. Evaluation methods comprise, but are not restricted to, the procedure of running a software or application with the aim of finding computer software bugs (errors or other defects). Pc software testing can be said as the process of validating and confirming that a computer program / program / product : meets the requirements that fulfills the needs of stakeholders, works as expected, can be implemented with the same features and guided its development and design. Computer software testing, with respect to the testing procedure applied, could be implemented at any moment in the development process. Traditionally most of the test effort happens after the necessities have been defined and the coding procedure has been finished, in the Agile approaches most of the test effort is on-going. As a result, the methodology of the evaluation is governed by the preferred computer software development methodology.

The test effort will be focused by different software development models at different points in the development process. Test is often employed by newer development models, such as Agile, - driven place and development an elevated part of the testing in the hands of the programmer, before it reaches a proper team of testers. In the coding procedure has been finished and a far more conventional model, the majority of the test execution happens following the necessities have been defined.

4.5.1 Black-box testing:

Black-box testing is just a process of computer software screening that investigates the performance of an application (e.g. exactly what the software does) without peering in to its inner buildings or functions (see white-box screening). This procedure of evaluation could be employed to almost every degree of software testing : system, integration, unit and acceptance. It generally contains most or even all higher-level testing, but may also control unit screening too. Particular knowledge of the program's code/internal construction and encoding knowledge generally isn't demanded. The examiner is aware of exactly what the program is meant to do but isn't aware of how it will it. For example, the examiner is conscious that a special input
signal results a specific, invariable output but isn't aware of how the computer software creates the result in the first place. Dark box testing targets the practical needs of the applications. I.e. black box testing allows the program professional to obtain models of input signal problems which will completely workout all functional requirements to get a course. Black box testing isn't an option to white box testing methods; rather it's a contrasting strategy that's prone to discover a various course of mistakes than white box approaches. Black box testing tries to discover errors within the next classes

a) Incorrect or lacking capabilities.
b) Interface mistakes.
c) Errors in information buildings or external database entry.
d) Performance errors.
e) Initialization and termination mistakes

Unlike white box testing, that is done early within the testing procedure, black box testing is commonly used throughout later periods of testing. Because black box testing purposely disregards control arrangement, interest is concentrated about the info domain name.

4.5.2 White box testing:
White-box testing (also called clear box testing, glass box testing, clear box testing, and architectural screening) is just a procedure of testing computer software that checks internal buildings or functions of an application, instead of its performance (i.e. black-box screening). In white-box testing an inner standpoint of the machine, too as encoding abilities, are utilized to create test cases. The examiner selects inputs to exercise trails through the signal and establish the proper results. This really is similar to testing nodes in a signal, e.g. in-circuit testing (ICT). While white - box testing could be employed at the integration, unit and program degrees of the software testing procedure, it's generally completed at the unit amount. Test can be leveled by it test paths within a unit, paths between units during integration, and between subsystems during a system - -. It could not discover unimplemented areas of the standards or lacking needs, although this procedure of evaluation layout may discover several mistakes or difficulties. making use of white box screening approaches, the computer software professional can obtain test circumstances that may
Generic Algorithm with Mathematical Evaluation for Experiments, Result Analysis and Quality Assurance of Real Time Applications

- Guarantee that all dependent pathways within a component have been practiced a minimum of once.
- Workout all rational conclusions on the fake and true attributes.
- Execute all coils at their borders and within their functional bounds.
- Exercise internal info buildings to ensure their credibility.

Demand for white package testing appears due to distinct motives

- Errors tend to creep into the work when we design and implement function, conditions or controls that are out of the main stream.
- We often believe that a logical is not likely to be executed when, in fact, it may be executed on the regular basis.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.

Black Box Testing and White Box testing are utilized. It's also known as behavioral screening which concentrates on practical demands of the applications. Within this screening the program is examined like a dark container without contemplating its inner details. Required sets of input were supplied and the desired outputs are obtained.

4.6 TEST CASES OF THE SYSTEM

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case Name</th>
<th>Description</th>
<th>Steps Carried out</th>
<th>Expected Results</th>
<th>Actual Result</th>
</tr>
</thead>
</table>
| TC1          | Structural Analysis| Calculate the matrix and weight of the users class diagram by parsing the file and match against the design pattern definition | 1. select a correct file in which UML class diagrams are stored and click start button.  
2. select an incorrect file in which UML class diagrams are stored and click start button | Show the design pattern passed the structural analysis test.  
Error Message = “does not contain valid data” is displayed. | Calculated the matrix and weight of the users class diagram by parsing the file and match against the design pattern definition  
Error Message = “does not contain valid data” is displayed. |
<p>| TC2          | Behavioral Analysis| Input the path of the source folder for                                      | 1. type the path of the source folder and                                     | Design pattern found if it is implemented in                                  | Design pattern found                                                        |</p>
<table>
<thead>
<tr>
<th>TC3</th>
<th>Semantic Analysis</th>
<th>To check whether the name of the class contain pattern related information</th>
<th>1. call the semantic analyzer method</th>
<th>Filer the false positives from the previous phases</th>
<th>False positives are filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC4</td>
<td>Package software metrics</td>
<td>To calculate package software metrics</td>
<td>1. Input more than one packages in the UML class diagram and select the package metrics option in the GUI form</td>
<td>Calculate the software package metrics for all packages found in the Class diagram</td>
<td>Calculated the software package metrics for all packages found in the Class diagram</td>
</tr>
<tr>
<td>TC5</td>
<td>GUI</td>
<td>Alignment of Controls</td>
<td>1. Textboxes should be properly aligned. Should be uniform Should be aligned</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Color of all buttons should be uniform.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All textboxes should be aligned in a straight line</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Test Cases