CHAPTER 4

DESIGN PATTERNS DETECTION AND VISUALIZATION ANALYSIS

4.1 Introduction

This first part of this chapter presents the unique rules of each of the structural design patterns. Analysis on the detection and visualization techniques is also discussed here. Besides, this chapter also includes the system analysis that entails the functional and non-functional requirements of the DPDV system.

4.2 Rules of the structural design patterns

Each structural pattern has a unique structure that differentiates it from the other structural patterns. The structures of all the structural patterns are analyzed and their important characteristics are represented as rules. Having a good knowledge of the structure of the patterns and consequently the rules are essential in building DPDV system.

4.2.1 Adapter design pattern rules

Adapter pattern can be object or class adapter. Furthermore, the object adapter can be implemented using class inheritance or interface implementation. The rules for each of the variations of adapter design pattern will be discussed in this section.

4.2.1.1 Object Adapter design pattern rules

This section explains the rules for object adapter design pattern implemented using class inheritance and interface implementation. Figure 4.1 shows the structure for the object adapter design pattern. The corresponding Java codes using interface implementation and class inheritance are shown in Figure 4.2 and Figure 4.3 respectively.

The rules of object adapter design pattern using interface implementation and class inheritance are as follow. Please take note that 'Interface' refers to interface implementation and 'Inheritance' refers to class inheritance. If no indication is given, the rule is applicable for both interface implementation and class inheritance.

Rule 1: **Interface:** A class (Adapter) that implements another interface (Target)

or

Inheritance: A class (Adapter) that inherits from another class (Target)

Rule 2: **Interface:** (Adapter) class must implement a method (Request) in the (Target) class.

or

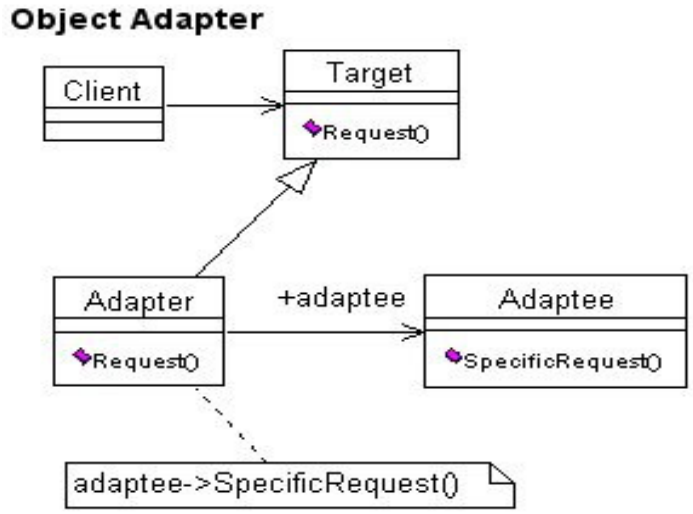
Inheritance: (Adapter) class must override a method (Request) inherited from (Target) class.

Rule 3: (Adapter) class must contain a reference/variable of type (Adaptee) class.

Rule 4: **Interface:** When (Adapter) class implements (Request) method, it uses the reference to the (Adaptee) class to call (SpecificRequest) method.

or

Inheritance: When (Adapter) class overrides (Request) method, it uses the reference to the (Adaptee) class to call (SpecificRequest) method



Figures 4.1 Object Adapter design pattern structure diagram (Gamma et al., 1995)

```
class Adapter implements Target {
Adaptee adaptee = new Adaptee();
      void Request()
      {
      adaptee.SpecificRequest();
      }
}
```

Figures 4.2: Object Adapter design pattern code (using interface implementation)

```
class Adapter extends Target {
Adaptee adaptee = new Adaptee ();
      void Request()
      {
      adaptee.SpecificRequest();
      }
}
```

Figures 4.3: Object Adapter design pattern code (using class inheritance)

4.2.1.2 Class Adapter design pattern rules

This section explains the rules for class adapter design pattern. Figure 4.4 shows the structure for the class adapter design pattern. The corresponding Java is shown in figure 4.5.

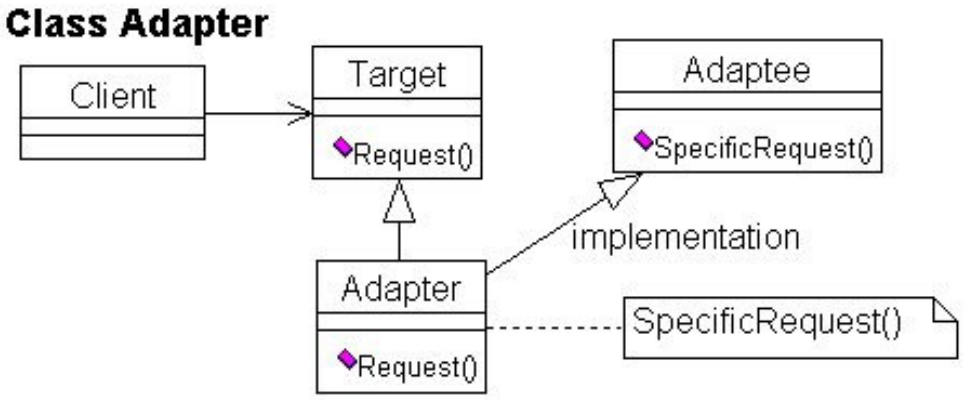
The rules of class adapter design pattern are as follow

Rule 1: A class (Adapter) that implements another interface (Target).

Rule 2: (Adapter) class must inherit from the (Adaptee) Class

Rule 3: (Adapter) class must implement a method (Request) in the (Target) class.

Rule 4: When (Adapter) class implements (Request) method in the (Target), the (Request) method of the (Adapter) class inherits the implementation of (SpecificRequest) method in the Super Class (Adaptee).



Figures 4.4: Class Adapter design pattern structure diagram (Gamma et al., 1995)

```
class Adapter extends Adaptee implements Target{
      void Request()
      {
            SpecificRequest();
      }
}
```

Figures 4.5: Class Adapter design pattern code

4.2.2 Bridge design pattern rules

This section explains the rules for bridge design pattern implemented using class inheritance and interface implementation. Figure 4.6 shows the structure for the bridge design pattern. The corresponding Java codes using interface implementation and class inheritance are shown in Figure 4.7 and Figure 4.8 respectively.

The rules of bridge design pattern using interface implementation and class inheritance are as follow.

Rule 1: **Interface:** A class (Abstraction) which has a reference to the interface (Implementer).

Or

Inheritance: A class (Abstraction) which has a reference to the class (Implementer).

Rule 2: **Interface:** The method (Operation) of the class (Abstraction) implements the method of the referenced interface (Implementer).

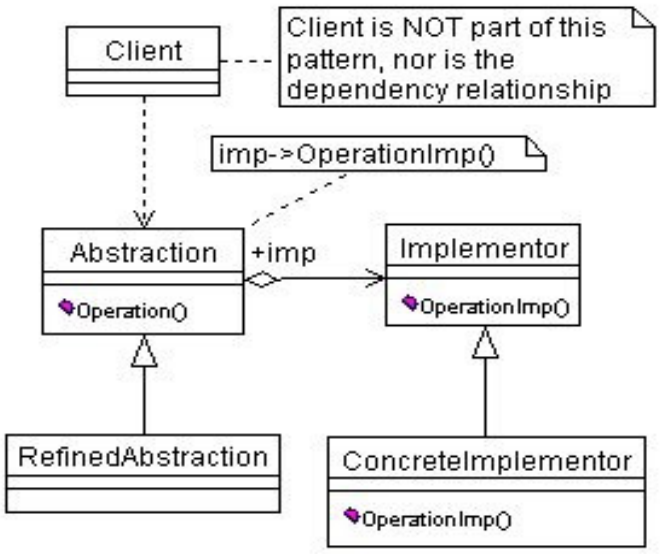
Or

Inheritance: The method (Operation) of the class (Abstraction) implements the method of the referenced class (Implementer).

Rule 3: **Interface:** A class (ConcreteImplementer) implements the interface (Implementer).

Or

Inheritance: A class (ConcreteImplementer) extends the class (Implementer).



Figures 4.6: Bridge design pattern structure diagram (Gamma et al., 1995)

```
class ConcreteImplementer implements Implementer {
      void Operation ()
      {
      }
}

class Abstraction {
      Implementer imp;

      void Operation ()
      {
            imp.OperationImp();
      }
}
```

Figures 4.7: Bridge design pattern code (using interface implementation)

```
class ConcreteImplementer extends Implementer {
      void Operation ()
      {
      }
}

class Abstraction {
      Implementer imp;

      void Operation ()
      {
            imp.OperationImp();
      }
}
```

Figures 4.8: Bridge design pattern code (using class inheritance)

4.2.3 Composite pattern rules

This section explains the rules for composite design pattern implemented using class inheritance and interface implementation. Figure 4.9 shows the structure for the composite design pattern. The corresponding Java codes using interface implementation and class inheritance are shown in Figure 4.10 and Figure 4.11 respectively.

The rules of composite design pattern using interface implementation and class inheritance are as follow.

Rule 1:Interface: A class (Composite) which implements the (Component) interface.

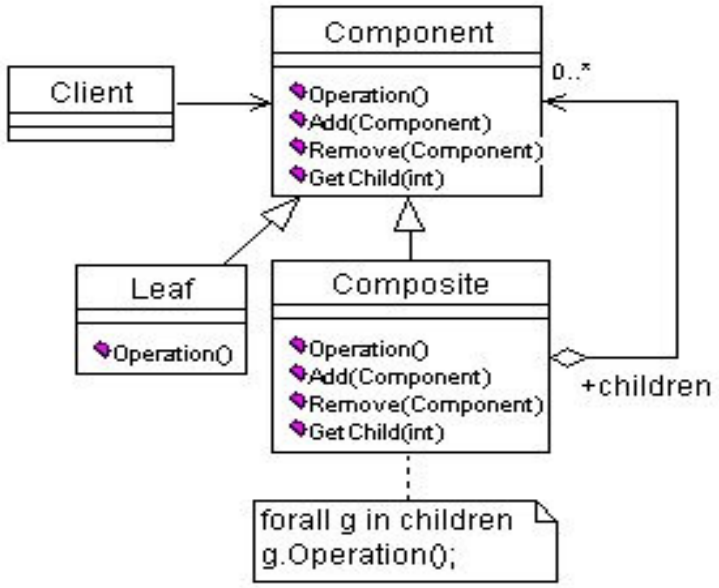
Or

Inheritance: A class (Composite) which extends the (Component) class.

Rule 2:Interface: The (Composite) class has a reference of multiple copies of the interface (Component).

Or

Inheritance: The (Composite) class has a reference of multiple copies of the class (Component).



Figures 4.9: Composite design pattern structure diagram (Gamma et al., 1995)

```
class Composite implements Component {
Component children[];
      void Operation ()
      {
            // for each Component object
            foreach (Component c in children) {
                  children[c].Operation();
            }
      }
}
```

Figures 4.10: Composite design pattern code (using interface implementation)

```
class Composite extends Component {
Component children[];
      void Operation ()
      {
            // for each Component object
            foreach (Component c in children) {
                  children[c].Operation();
            }
      }
}
```

Figures 4.11: Composite design pattern code (using class inheritance)

4.2.4 Decorator design pattern rules

This section explains the rules for decorator design pattern implemented using class inheritance and interface implementation. Figure 4.12 shows the structure for the decorator design pattern. The corresponding Java codes using interface implementation and class inheritance are shown in Figure 4.13 and Figure 4.14 respectively.

The rules of decorator design pattern using interface implementation and class inheritance are as follow.

Rule 1:Interface: A class (Decorator) which implements the (Component) interface.

Or

Inheritance: A class (Decorator) which extends the (Component) class.

Rule 2:Interface: The (Decorator) has a reference of the interface (Component).

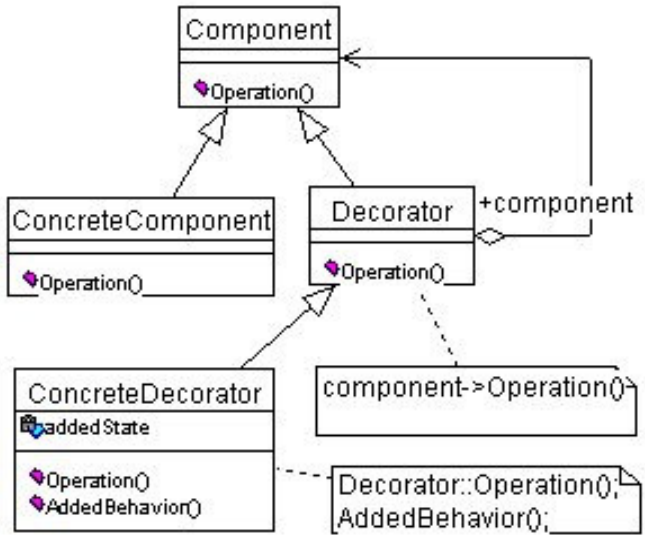
Or

Inheritance: The (Decorator) class has a reference of the class (Component).

Rule 3:Interface: A class (ConcreteDecorator) which extends the (Decorator) class.

Or

Inheritance: A class (ConcreteDecorator) which extends the (Decorator) class.



Figures 4.12: Decorator design pattern structure diagram (Gamma et al., 1995)

```
class Decorator implements Component {
Component component;
      void Operation ()
      {
            component.Operation();
      }
}

Class ConcreteDecorator extends Decorator{
      void Operation ()
      {
      //add behavior
      }
}
```

Figures 4.13: Decorator design pattern code (using interface implementation)

```
class Decorator extends Component {
Component component;
      void Operation ()
      {
            component.Operation();
      }
}

class ConcreteDecorator extends Decorator{
      void Operation ()
      {
      //add behavior
      }

}
```

Figures 4.14: Decorator design pattern code (using class inheritance)

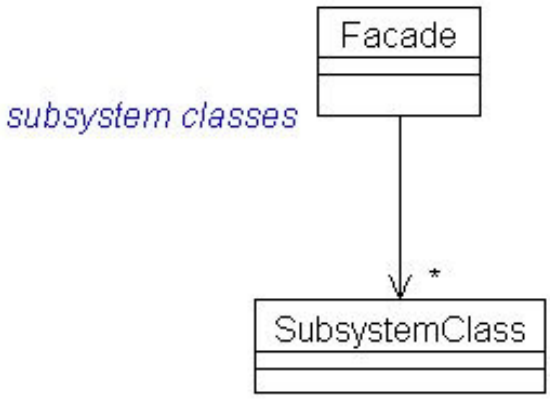
4.2.5 Façade design pattern rules

This section explains the rules for facade design pattern. Figure 4.15 shows the structure for the facade design pattern. The corresponding Java code is shown in Figure 4.16.

The rules of facade design pattern are as follow.

Rule 1: The (Façade) class has references of objects of the (Subsystems) classes.

Rule 2: The (Façade) class calls the methods of the (Subsystems) objects.



Figures 4.15: Facade design pattern structure diagram (Gamma et al., 1995)

```
class Subsystem {
      void MethodA ()
      {
      }
}

class Facade {
      Subsystem s=new Subsystem();
      void MethodB ()
      {
            s. MethodA();
      }
}
```

Figures 4.16: Façade design pattern code

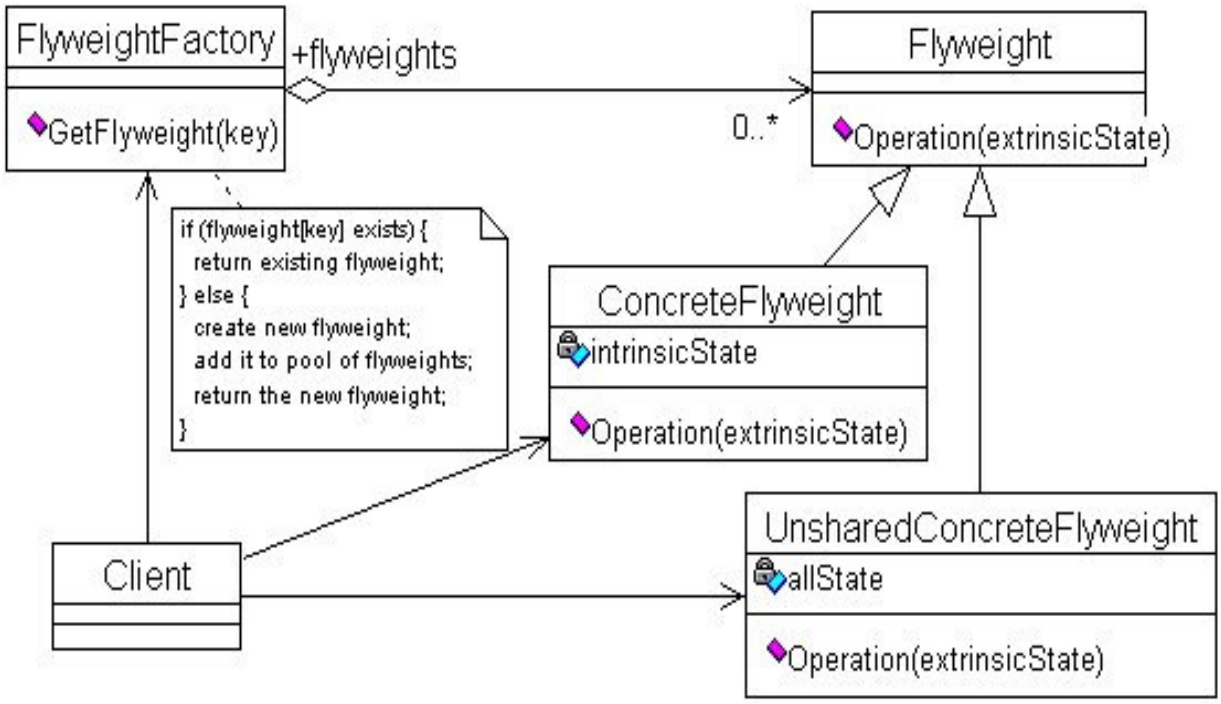
4.2.6 Flyweight design pattern rules

This section explains the rules for flyweight design pattern. Figure 4.17 shows the structure for the flyweight design pattern. The corresponding Java code is shown in Figure 4.18.

The rules of flyweight design pattern are as follow.

Rule 1: The (FlyweightFactory) class has a reference to the (Flyweight) class.

Rule 2: The (FlyweightFactory) class has a method (getFlyweight) which its return type is the (Flyweight) class.



Figures 4.17: Flyweight design pattern structure diagram (Gamma et al., 1995)

```
class FlyweightFactory {
Flyweight f;

      Flyweight getFlyweight()
      {
      // return existing flyweight or create new one and // return
      it
      }
}
```

Figures 4.18: Flyweight design pattern code

4.2.7 Proxy design pattern rules

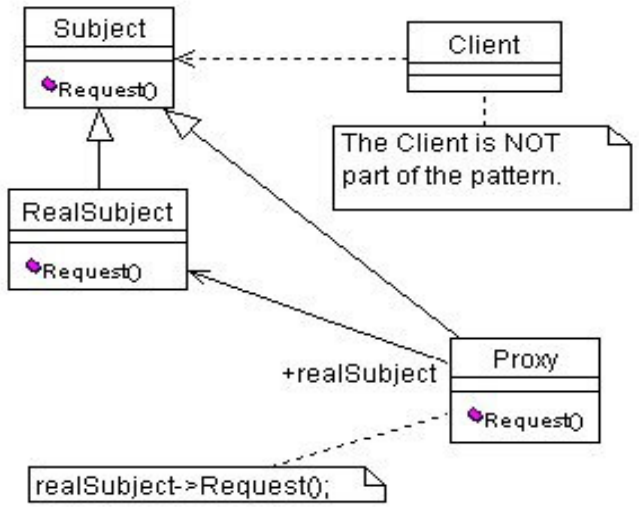
This section explains the rules for proxy design pattern. Figure 4.19 shows the structure for the proxy design pattern. The corresponding Java code is shown in figure 4.20. The rules of proxy design pattern are as follow.

Rule 1: A class (RealSubject) that implements an interface (Subject).

Rule 2: A class (Proxy) that implements an interface (Subject).

Rule 3: The (Proxy) class has a reference to class (RealSubject).

Rule 4: When (Proxy) class implements (Request) method, it uses the reference to the (RealSubject) class to call (Request) method of the (RealSubject) class.



Figures 4.19: Proxy design pattern structure diagram (Gamma et al., 1995)

```
class RealSubject implements Subject{
      void Request()
      {
      }
}
class Proxy implements Subject{
      RealSubject rs;

      void Request()
      {
            rs. Request();
      }
}
```

Figures 4.20: Proxy design pattern code

4.3 DPDV Detection and Visualization Techniques

DPDV consists of two parts: detection and visualization. The detection part is discussed first.

4.3.1 DPDV Detection Technique

DPDV detects the structural design patterns from the Java code in three steps. Firstly, DPDV uses reflection technique to extract the attributes of each entity and the relationships among them from the code. Secondly, DPDV arranges this information in a data structure defined as a class. Finally, the tool refers to this data structure to detect the possible design patterns used in the code by mapping between the design patterns instances captured in the data structure and the predefined patterns rules which have been discussed in Section 4.2. Each structural pattern has a unique structure that differentiates it from other structural patterns structures.

For example, if the code contains the Flyweight design pattern and consists of three classes A, B and C. DPDV uses reflection technique to extract the attributes of each entity and their relationships from the code such as sub classes, super classes, implemented interfaces, declared fields and declared methods. Those extracted information will be arranged in a class data structure called ‘Structure’. After that, DPDV will refer to the ‘Structure’ class to do mapping between the extracted information from the three classes and the rules of each design pattern. The Flyweight pattern will be detected since there is a match between the extracted instances and the flyweight pattern rules.

4.3.2 DPDV Visualization technique

Design pattern visualization is best represented using UML class diagram. UML is widely accepted and understood by all users. DPDV draws a simple representation in the form of UML class diagram to show the design pattern found in the code. Moreover, DPDV can show the full details about the classes, interfaces and relationships in textual description.

4.4 System Analysis

This part involves identifying the functional and non-functional requirements for DPDV. Functional requirements define what the system shall do. The system must be able to do the work for which it was intended. Non-functional requirements define the qualities of the system such as usability, modifiability, performance, availability, testability, security and so on. The results of the analysis done on existing design pattern detection and visualization tools (Section 2.6.3, Table 2.1) also served as an input for consideration in this stage.

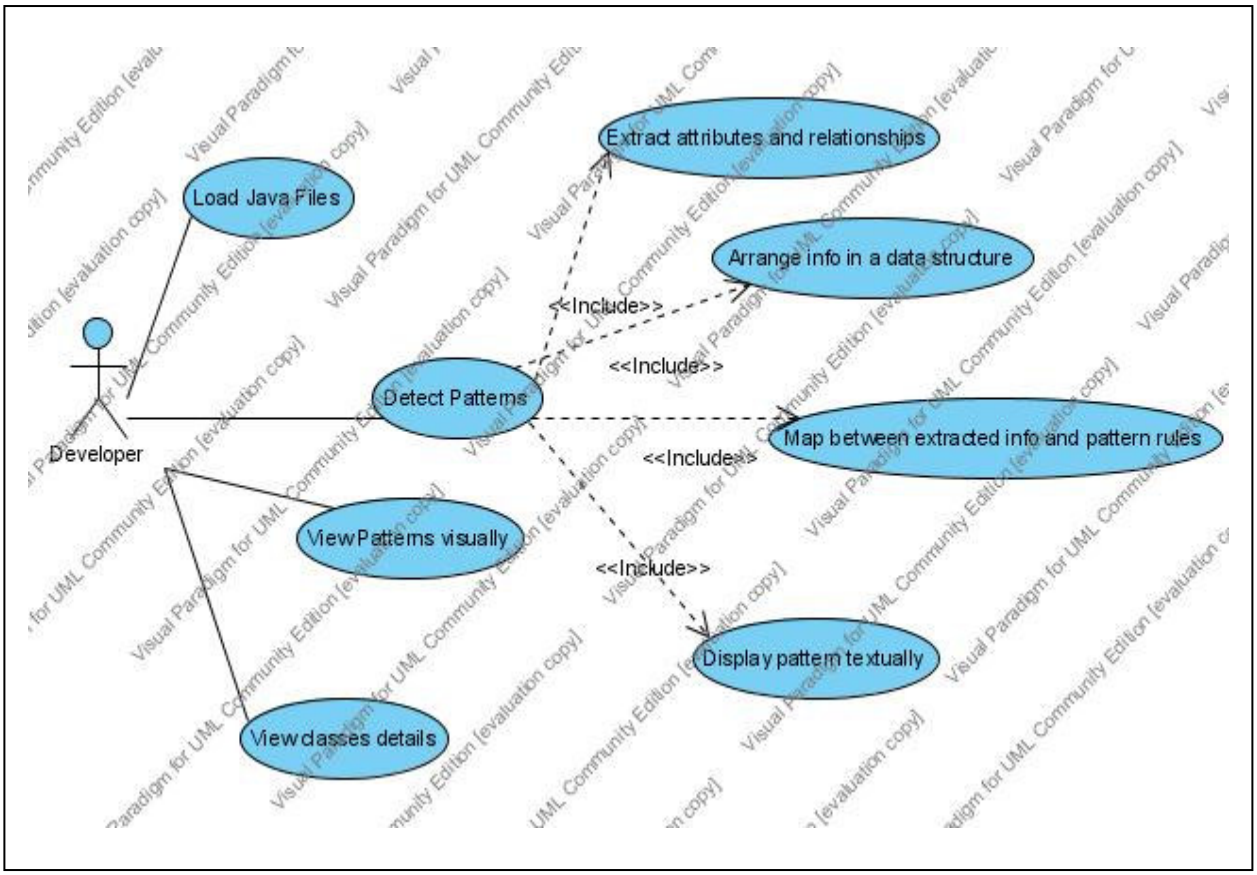
4.4.1 Functional requirements

The following are the main functional requirements of DPDV system:

- Load Java files
- Detect design patterns
- View

4.4.1.1 DPDV use case diagram

Figure 4.10 shows the use case diagram for DPDV.



Figures 4.21: DPDV use case diagram

4.4.1.2 Use case specification

This part shows the use cases specification for the system. Use case specification includes brief description, pre-condition, post-condition, flow of events, and alternative flow. Specifications of Load Java Files, Detect patterns, Extract attributes and relationships, Arrange info in data structure, Map between extracted info and pattern rules, View patterns textually, View patterns Visually, and View classes details use cases are shown in Figure 4.22, Figure 4.23, Figure 4.24, Figure 4.25, Figure 4.26, Figure 4.27, Figure 4.28, and Figure 4.29 respectively.

Table 4.1

Load java files Use case

Use case name	UC1- Load java files
Brief Description	Loading java files from relevant directory.
Pre-condition	1- The files must be java files 2- The files should not have any package name specified
Post-condition	The loaded files must be displayed on "Loaded Files" panel.
Flow of events	1- System displays the main screen 2- User clicks on "Load files" button to load code files 3- System displays "open file" dialogue box 4- User browses and selects the relevant files and clicks "OK" button 5- System loads the files and displays the loaded files names on the "Loaded Files" Panel.
Alternative flow	Not applicable
Include	Not applicable
Extend	Not applicable

Table 4.2

Detect patterns Use case

Use case name	UC2- Detect patterns
Brief	Detect design patterns from loaded java files.

Description	
Pre-condition	The loaded files should be displayed on the “Loaded Files” Panel.
Post-condition	Detected pattern should be displayed on the “Detected Patterns” panel.
Flow of events	1- User clicks the “Next” button. 2- System displays “Design Pattern Detector” screen. 3- User clicks the “Detect Patterns” button. 4- System displays the detected pattern on the screen.
Alternative flow	System did not detect any pattern existed in the code and display empty message
Include	UC2.1- Extract attributes and relationships, UC2.2- Arrange info in data structure, UC2.3- Map between extracted info and pattern rules, UC2.4- View pattern textually
Extend	Not applicable

Table 4.3

Extract attributes and relationships Use case

Use case name	UC2.1- Extract attributes and relationships
Brief Description	Extract attributes of classes and relationships among classes
Pre-condition	The developer clicks the “Detect Patterns” button.
Post-	Extracted information is captured in the data structure class called

condition	"Structure"
Flow of events	1- The systems searches for the structural properties such the classes, interfaces, and *implements* and *extends* relations from the classes or interfaces. 2- The system extracts those found information. 3- The system saves those extracted information in arrays in class "Detector"
Alternative flow	Not applicable
Include	Not applicable
Extend	Not applicable

Table 4.4

Arrange info in data structure

Use case name	UC2.2- Arrange info in data structure
Brief Description	Arranging the extracted attributes and relationships in data structure of type class.
Pre-condition	There are extracted information
Post-condition	Information being stored in the data structure
Flow of events	1- The system reads the arrays that contain the extracted information from the "Detector" class. 2- The system transfers those arrays from "Detector" class to "Structure" class 3- The system arranges the arrays in the "Structure" class

Alternative flow	Not applicable
Include	Not applicable
Extend	Not applicable

Table 4.5

Map between extracted info and pattern rules

Use case name	UC2.3- Map between extracted info and pattern rules
Brief Description	Mapping between extracted information in data structure and pattern rules
Pre-condition	The extracted information is arranged in the data structure called "Structure" class.
Post-condition	Pattern detected or not detected
Flow of events	1- System reads the extracted information from the data structure called "Structure" class. 2- System maps between the extracted information and the rules. 3- If the extracted information matches with design pattern rules, then the system displays a message stating the type of pattern detected.
Alternative flow	If the extracted information does not match with design pattern rules, the system displays an empty message.
Include	Not applicable
Extend	Not applicable

Table 4.6

View patterns textually Use case

Use case name	UC2.4- View patterns textually
Brief Description	Display the name of the detected design patterns for viewing purpose
Pre-condition	There must be some pattern detected.
Post-condition	The name of the detected design pattern is displayed.
Flow of events	1- System displays the name of the detected pattern on the screen.
Alternative flow	System displays an empty message if pattern is not being detected.
Include	Not applicable
Extend	Not applicable

Table 4.7

View patterns Visually Use case

Use case name	UC3- View patterns Visually
Brief Description	Display the detected design patterns in UML diagram for viewing purpose
Pre-condition	There must be some pattern detected
Post-condition	The detected design pattern is displayed in UML class diagram.
Flow of events	1- User clicks on the "Next" button in "Design Patterns Detector Screen". 2- System displays "Design Patterns Visualizer" screen that shows the detected pattern as a diagram as well as the rules of the detected pattern.

Alternative flow	The system does not display any diagram if there is no pattern detected
Include	Not applicable
Extend	Not applicable

Table 4.8

View classes details Use case

Use case name	UC4- View classes details
Brief Description	Display full details about classes, interfaces, relationships, attributes, and methods.
Pre-condition	The loaded files should be displayed on the "Loaded Files" Panel
Post-condition	The full details about classes and interfaces are shown.
Flow of events	1-User clicks on the "Classes Details" button 2-System displays "Classes Details" screen and displays all the classes, interfaces and relationships among them 3-User clicks on any class or interface to show the relevant information about the selected class or interface such as the attributes and methods.
Alternative flow	Not applicable
Include	Not applicable
Extend	Not applicable

4.4.2 Non-Functional requirements

The most important non-functional requirement or quality for the DPDV system is accuracy. DPDV needs to be accurate; especially it needs to be able to detect design pattern accurately. If there is a code that contains a specific design pattern then the DPDV should be able to detect it correctly and create the visualization of the recommended pattern.

A concrete quality scenario can be used to specify the non-functional requirement/quality of a system (Bass et al., 2003). It consists of six parts that are source of stimulus, stimulus, environment, artifact, response, and response measure. Figure 4.25 shows the concrete quality scenario for the accuracy quality of the system. A user wishes to detect design patterns from the code at run time and the system displays the possible design pattern existed in the code with a detection success rate more than 80 %.

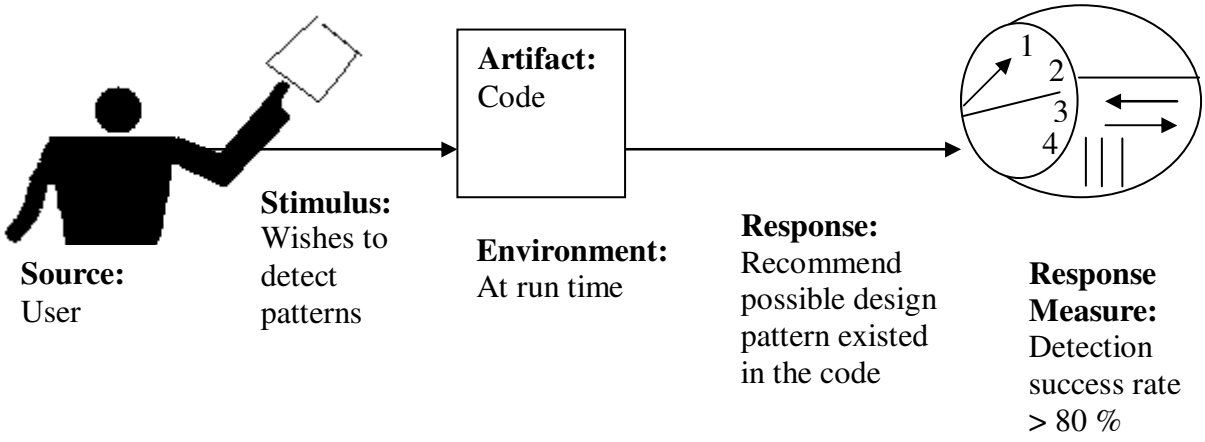


Figure 4.22

The concrete quality scenario for the accuracy quality of the DPDV system

4.5 Summary

This chapter presented the unique rules of each of the structural design patterns. Analysis on the detection and visualization techniques was also discussed here. Finally, this chapter explained the system analysis that entails the functional and non-functional requirements of the DPDV system.