CHAPTER 6

SYSTEM IMPLEMENTATION

6.1 Introduction

This chapter contains explanations of some crucial codes of DPDV system and shows the use of some design patterns in the implementation of DPDV system. The use of Reflection technique to extract information from DPDV code is illustrated in this chapter. Moreover, the use of object-oriented programming (OOP) is explained in this chapter since the implementation of DPDV adopts object-oriented programming. DPDV consists of seven java classes that are Dialog1, FilesLoader, ClassDetails, Detector, ReadMethods, Structure and Visualizer and each of them will be explained here. The system requirement to deploy DPDV is mentioned in Chapter 1.

6.2 Usage of design patterns in the implementation of DPDV

Since DPDV system is a system to detect structural design patterns from Java code, it is natural for DPDV system to use some of the design patterns in its implementation. Façade design pattern is clearly used in the DPDV. Façade design pattern has many benefits as stated below (Jalasutram, 2007):

- Reduce complexities of a system.
- Decouple subsystems, reduce its dependency, and improve portability.
- Make an entry point to subsystems.
- Minimize the communication and dependency between subsystems.
- Security and performance consideration.
- Simplify generosity to specification.

Figure 6.1 shows the usage of Façade design pattern in DPDV. The Java code for DPDV system in a separate file is loaded into the DPDV system and façade design pattern is detected as shown in Figure 6.1.

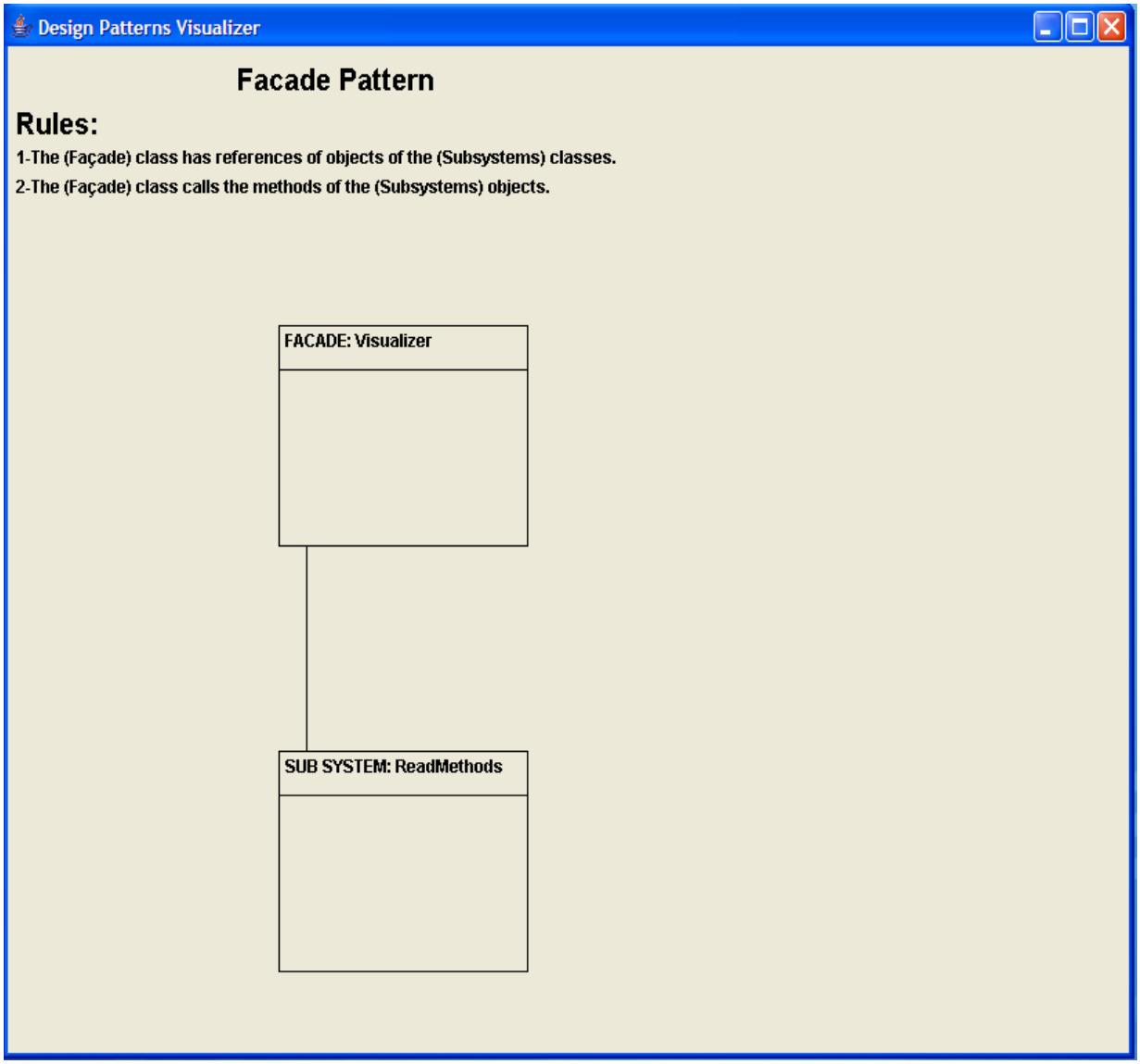

Figure 6.1: Use of Façade design pattern in DPDV's implementation

6.3 Use of Object-Oriented Programming in the implementation of DPDV

DPDV uses Object-Oriented Programming (OOP) in its implementation. OOP supports reusability in terms of inheritance or composition. Reusing code saves time in program development. Some uses of OOP in DPDV are shown as followed:

6.3.1 Use of member access controls

DPDV uses member access controls which are private, public and protected. Member access control supports abstraction and information hiding. Figure 6.2 shows partial code of the 'ReadMethods' class which explains the usage of member access control. The code shows that instance variables have private access control and instance methods have public access control.

```
public class ReadMethods {

  private Method Method_Name[];
  private Class Field_Name[];
  private RandomAccessFile rf;
  private static String Mthd;

   public boolean CheckField(Class SubClass,Type cin[]){

   }

}
```

Figure 6.2: Use of member access control in 'ReadMethods' class

6.3.2 Use of composition

The composition relationship is a good example for reusability. Composition supports loose coupling between classes. Composition relations are available in few classes in DPDV. For example, class 'Detector' has composition relationship with class 'Structure'. Figure 6.3 shows the use of composition relation in class 'Detector'.

```
public class Detector extends JFrame {

Structure st2;
// some code ……

    public void jButton1_actionPerformed(ActionEvent e) {

      // some code ……

      st2=new Structure
      (nFiles,ExtCounter,ImpCounter,c,SuperClass,SubClass,ImplementerClass,t,mSup
      erClass,mSubClass,mImplementerClass,mInterfaces);
      // some code ……
    }
}
```

Figure 6.3: Use of composition relation in class 'Detector'

6.4 DPDV code explanations

This section explains some of the crucial parts of DPDV codes. DPDV consists of seven java classes which are Dialog1, FilesLoader, ClassDetails, Detector, ReadMethods, Structure and Visualizer. Some codes of each class will be explained in the following subsections.

6.4.1 'Dialog1' class

An object of Dialog1 Class is created to load the first screen that disappeared after few seconds. Dialog1 is only for showing the DPDV main screen interface.

6.4.2 'FilesLaoder' class

FilesLoader class is for loading java files from any directory chosen by a user and then copying those java files into a local directory called "files". DPDV extracts the information from java files which are saved in the "files" directory. Figure 6.4 shows partial code in 'FilesLoader' class which loads the java files. It allows multiple files to be loaded.

```
public void jButton1_actionPerformed(ActionEvent e) {
int nFiles=0,result=0;


     jFileChooser1.setMultiSelectionEnabled(true);

     result=jFileChooser1.showOpenDialog(this);
     nFiles=jFileChooser1.getSelectedFiles().length;

     filesName=new String[nFiles];
     filesPath=new String[nFiles];

     files=(jFileChooser1.getSelectedFiles());


         for(int i=0;i<nFiles;i++){
         filesName[i]=files[i].getName();

         filesPath[i]=files[i].getPath();
            // JOptionPane.showMessageDialog(null, "file name "+filesPath[i]);
         }

    if(result==JFileChooser.CANCEL_OPTION)
    return;

    if(files==null)
    JOptionPane.showMessageDialog(null, "Invalid File Name");

    setFiles(filesName,filesPath);
    jList1.setListData(filesName);
}
```

Figure 6.4: 'FilesLoader' class (code to load files)

Figure 6.5 shows another section of code in class 'FilesLoader'. The code is meant for opening connection with the input files from where the java files will be loaded. Moreover, the code is for creating output files where the files should be saved.

```
     int i;
     FileReader fin[];
     FileWriter fout[];
     fin =new FileReader[nFiles];
     fout =new FileWriter[nFiles];

     for(int n=0;n<nFiles;n++) {
     try
      {
      //open input file
            try
            {
                fin[n] = new FileReader(filesPath[n]);
            }
            catch (FileNotFoundException fnf)
            {
                System.out.println("Input File Not Found");
                return;
            }

                              //open output file
             try
             {
                 fout[n] = new FileWriter("C:/Documents and
Settings/Administrator/Desktop/DPDV/src/dpdv/files/"+filesName[n]);
             }
             catch (FileNotFoundException fnf)
             {
                 System.out.println("Error opening Output File");
                 return;
             }
             catch (IOException io)
             {
                 System.out.println("Error opening Output File");
                 return;
             }

     }
     catch (ArrayIndexOutOfBoundsException aiob)
     {
        return;
     }
}
```

Figure 6.5: 'FilesLoader' class (code for opening output and input files)

6.4.3 'Detector' class

Figure 6.6 shows partial code in class 'Detector' that uses reflection technique to extract sub classes, super classes and classes implementing interfaces. Figure 6.7 shows the use of reflection technique in class 'Detector' to extract methods of interfaces as well as the

methods of the classes implementing those interfaces. Figure 6.8 shows the use of reflection technique in class 'Detector' to extract methods of sub classes as well as super classes.

```
for(int i=0;i<nFiles;i++){

 int u=filesReciever[i].toString().lastIndexOf(".");
 Name[i]=filesReciever[i].substring(0,u);

 c[i] = Class.forName("dpdv.files."+Name[i]);


      if((c[i].getSuperclass()!=null)&&(c[i].getSuperclass()!=Object.class)&&(c
      [i].isInterface()==false)&&(!(c[i].getSuperclass().getSimpleName().equals
      ("JFrame")))&&(!(c[i].getSuperclass().getSimpleName().equals("JDialog")))
      )
      {

         ExtCounter++;

         SuperClassTemp[i]=c[i].getSuperclass();
         SubClassTemp[i]=  c[i].asSubclass(SuperClassTemp[i]);
     }


      if((c[i].getGenericInterfaces().length!=0)&&(c[i].isInterface()==false))
      {
           ImpCounter++;
           ImplementerClassTemp[i]=  c[i];
     }

   f[i] = c[i].getDeclaredFields();
   m[i] = c[i].getDeclaredMethods();

}
```

Figure 6.6: 'Detector' class (code to extract classes using reflection)

```
for(int ic=0;ic<ImpCounter;ic++){

              ImplementerClass[ic]=Class.forName("dpdv.files."+Tem[ic]);

      t[ic] = ImplementerClass[ic].getGenericInterfaces();

      mImplementerClass[ic]=ImplementerClass[ic].getDeclaredMethods();
     mInterfaces=new Method[ImpCounter][ t[ic].length][];

}

for(int ic=0;ic<ImpCounter;ic++){
      for(int tic=0;tic<t[ic].length;tic++){

            mInterfaces[ic][tic]=((Class)t[ic][tic]).getDeclaredMethods()
            ;

       }

}
```

Figure 6.7: 'Detector' class (code to extract interfaces methods using reflection)

```
for(int ec=0;ec<ExtCounter;ec++){

       SuperClass[ec]=Class.forName("dpdv.files."+Tem2[ec]);
       SubClass[ec]=Class.forName("dpdv.files."+Tem3[ec]);

       mSuperClass[ec]=SuperClass[ec].getDeclaredMethods();
       mSubClass[ec]=SubClass[ec].getDeclaredMethods();
}
```

Figure 6.8: 'Detector' class (code to extract classes' methods using reflection)

6.4.4 'Structure' class

'Structure' class is used as the data structure to store the extracted information from the java code. It also matches this information against each structural pattern rules to detect the possible design pattern. Since 'Structure' class is a big class, only the code to detect Bridge pattern rules is discussed here. Figure 6.9 shows 'CheckBridge' method of class

'Structure'. The code checks the extracted information against rules of bridge design pattern using interface implementation. The 'CheckBridge' method will return 'true' value if the extracted information from the loaded java files matches with the Bridge design pattern rules. The rules of bridge design pattern were discussed in Section 4.2.2.

The following statement is from the code shown in Figure 6.9. The statement calls the class 'ReadMethods' which is responsible for searching through each method of a class to check whether any method of a class implements a method of the referenced interface (Rule 2 of bridge design pattern rules discussed in Section 4.2.2). 'ReadMethods' class will be discussed later in the next subsection.

```
d=rm.CheckMethod(C[g],((Class)T[ic][tic]).getDeclaredMethods();
```

The following code is a part of the code shown in Figure 6.9. The array 'Bridge' keeps some relevant information to be passed to and used by 'Visualizer' class. "Bridge[0]" stores the class that has a reference of an interface and implements one of its methods. "Bridge[1]" stores the implementer class. "Bridge[2]" stores the implemented interface. The Visualizer' class is to create visualization for the detected patterns. It will be discussed later as well.

```
Bridge[0]=C[g].getSimpleName();
Bridge[1]=((Class)implementerClass[ic].getGenericInterfaces()[0]).getSimpleName();
Bridge[2]=implementerClass[ic].getSimpleName();
```

```
public boolean CheckBridge(){
 ReadMethods rm=new ReadMethods();
 boolean b=false,c=false,d=false;

  for(int ic=0;ic<impCounter;ic++){

    for(int tic=0;tic<T[ic].length;tic++){

      for(int tt=0;tt<minterfaces[ic][tic].length;tt++){
        for(int im=0;im<mimplementerClass[ic].length;im++){
            // Rule 3 of Bridge design pattern rules
            if(minterfaces[ic][tic][tt].getName()==mimplementerClass[ic][im].getN
            ame()){
          c=true;

           for(int g=0;g<nfiles;g++){

                  if((C[g].isInterface()==false)&&(C[g].getGenericInterfaces().le
                  ngth==0)){

                    if(C[g].getDeclaredFields()!=null){
                 for(int =0;v<C[g].getDeclaredFields().length;v++){
                        // Rule 1 of Bridge design pattern rules
                        if(C[g].getDeclaredFields()[v].getType()==((Class)T[ic][t
                        ic])){
                              // Rule 2 of Bridge design pattern rules
                              d=rm.CheckMethod(C[g],((Class)T[ic][tic]).getDeclar
                              edMethods());
                          if(d==true){
                             Bridge[0]=C[g].getSimpleName();
                             Bridge[1]=((Class)implementerClass[ic].
                   getGenericInterfaces()[0]).getSimpleName();

                              Bridge[2]=implementerClass[ic]
                              .getSimpleName();
                          }
                      }
                    }
                    }
                 }
             }
          }
        }
      }
    }
   }
}

          if((c==true)&&(d==true))
          b=true;

       return b;
 }
```

Figure 6.9: Matching extracted information against Bridge pattern rules in 'Structure' class

6.4.5 'ReadMethods' class

'ReadMethods' class is responsible for searching through each method of a class to find a specific value or string. Figure 6.10 shows the method 'CheckMethod' in class 'ReadMethods'. The method is for searching through methods of a class to find whether any of those methods have a reference to another method of an interface. The method 'CheckMethod' will return 'true' value if an interface method found in any of the methods of a class. This search is to check 'Rule 2' of the bridge patterns rules discussed in Section 4.2.2.

```
    public boolean CheckMethod(Class SubClass,Method in[]){
      boolean b=false;
       Method_Name=in;
       String rs;
       try{
           rf = new RandomAccessFile("C:/Documents and
Settings/Administrator/Desktop/DPDV/src/dpdv/files/"+SubClass.getSimpleName().
toString()+".java","r");

           if(rf!=null){
               for (int i = 0; i < rf.length(); i++) {

               rf.seek(rf.getFilePointer());
               rs = rf.readLine();
                 if (rs != null) {
                    if(Method_Name!=null){
                        for(int v=0;v<Method_Name.length;v++){
                              if (rs.contains(Method_Name[v].getName())){
                              b=true;
                              Mthd=Method_Name[v].getName();

            }}}}}}
           rf.close();
       }
       catch(Exception ex){
       JOptionPane.showMessageDialog(null,ex);
       }

 return b;
     }
```

Figure 6.10: 'CheckMethod' method in 'ReadMethods' class

6.4.6 'Visualizer' class

The 'Visualizer' class is for creating the visualization of the detected design patterns using UML class diagram. Figure 6.11 shows how to draw the bridge design pattern and display some relevant information on the diagram such as a class and an interface name.

```
if((Pattern=="Bridge Pattern")||(Pattern=="Bridge Pattern-
abstraction")){
        //draw class
         g.drawRect(200,220,xx,yy);
         g.drawRect(600,220,xx,yy);
         g.drawRect(600,510,xx,yy);
            //draw line inside class
         g.drawLine(200,250,200+xx,250);
         g.drawLine(600,250,600+xx,250);
         g.drawLine(600,540,600+xx,540);
            //info
         g.drawString("ABSTRACTION: "+Bridge[0],205,235);
         g.drawString("INTERFACE: "+Bridge[1],605,235);
         g.drawString("IMPLEMENTATION: "+Bridge[2],605,525);
         g.drawString(Mthd+"()",605,270);
         g.drawString(Bridge[1],210+xx,230);
         g.drawString(Bridge[1]+"-->"+Mthd+"()",205,290);
            //relationships
         g.drawLine(200+xx,235,600,235);
         g.drawLine(615,220+yy,615,510);

    }
```

Figure 6.11: 'Visualizer' class (code to draw bridge design pattern)

6.4.7 'ClassDetails' class

'ClassDetails' class displays general details about all the loaded java files. The general details are such as names of all classes and interfaces, their methods and fields and the relationships among them. Figure 6.12 shows partial code of 'ClassDetails' class. The code is for displaying names of classes and their relationships. On the other hand, the code in Figure 6.13 is executed when a mouse is clicked. The code is to display a class's fields and methods when the class is clicked.

```
EX=new Object[extCounter];
        IM=new Object[impCounter];

        for(int i=0;i<extCounter;i++)
        EX[i]=subClass[i].getSimpleName()+" extends
"+superClass[i].getSimpleName();

        for(int i=0;i<impCounter;i++)
        IM[i]=implementerClass[i].getSimpleName()+" implements
"+((Class)implementerClass[i].getGenericInterfaces()[0]).getSimpleName();


        jList1.setListData(C);
        jList2.setListData(IM);
        jList6.setListData(EX);

        jScrollPane1.getViewport().setView(jList1);
        jScrollPane2.getViewport().setView(jList3);
        jScrollPane3.getViewport().setView(jList4);
        jScrollPane4.getViewport().setView(jList2);
        jScrollPane5.getViewport().setView(jList5);
        jScrollPane6.getViewport().setView(jList6);
```

Figure 6.12: 'Visualizer' class (code to display class details when loading the class)

```
public void jList1_mouseClicked(MouseEvent e) {
F=new Object[((Class)jList1.getSelectedValue()).getDeclaredFields().length];
M=new Object[((Class)jList1.getSelectedValue()).getDeclaredMethods().length];
Object empty[]={};
 jList3.setListData(empty);
jList4.setListData(empty);

 for(int
i=0;i<((Class)jList1.getSelectedValue()).getDeclaredFields().length;i++){

F[i]=((Class)jList1.getSelectedValue()).getDeclaredFields()[i].getType().getSim
pleName()+"
"+((Class)jList1.getSelectedValue()).getDeclaredFields()[i].getName();
            jList3.setListData(F );
         }

 for(int
i=0;i<((Class)jList1.getSelectedValue()).getDeclaredMethods().length;i++){
           M[i]=((Class)jList1.getSelectedValue()).getDeclaredMethods()[i];
           jList4.setListData(M );
        }
        try{
        Object[] rs;
        rf = new RandomAccessFile("C:/Documents and
Settings/Administrator/Desktop/DPDV/src/dpdv/files/"+((Class)jList1.getSelected
Value()).getSimpleName().toString()+".java","r");
        Long  ii = new Long(rf.length());

        rs=new Object[ii.intValue()];
        if(rf!=null){
                    for (int i = 0; i < rf.length(); i++) {
                    rf.seek(rf.getFilePointer());
                    rs[i] = rf.readLine();
                      if (rs != null)
                      jList5.setListData(rs );
                    }}
        rf.close();}
        catch(Exception ex){
        JOptionPane.showMessageDialog(null,ex);}}
```

Figure 6.13: 'Visualizer' class (code to display classes' details when mouse is clicked)

6.5 Summary

This chapter presented the use of object-oriented programming and design patterns in the implementation of DPDV system. Moreover, some crucial parts of DPDV codes have been shown and discussed.