Final Project Report for ICOM 6089

Object Oriented Software Design

Game_library

Submitted By: Pardeep Kumar

INDEX

Description of System:

Project Name: game_library

Introduction & purpose: This project is based on my real life requirement, while working as Java TA for advanced programming course (ICOM 4015). During the course I have developed and received many projects (generally Game projects in java) for and by the students respectively. As I have faced many problems while checking the quality of code of submitted projects as all of the students have used their own names for classes and methods so it was very difficult to deeply test or validate the code. So I develop an idea in my mind to solve this problem by giving them a pattern to follow while coding.

As a result I created this small game_library, in which you can easily integrate any game project which either have followed the pattern of game flow you have provided (i.e. they must have extended template of your design) or you have the option by providing them the adapter which they can implement in their project to integrate with the library.

The purpose of this project could be seen as a platform to integrate all the projects developed by the community of developer who has the knowledge of specific template which your game_library provide to integrate. And for those who do not follows the given template can integrate their game project to the library using adapter pattern technique provided by the game_library.

In this project, I have used three games, where two of these i.e. Rowfongo game (which has also implemented the Observer Pattern) and Country game are developed by myself, are integrated into library using game_library's provided template and the third game i.e. Proximity is integrated using adapter technique provided by the game_library project.

In this project I have came across many situations where I found a need for pattern and following is the list of patterns I have used in this project.

- Chain of Responsibility pattern.
- Decorator Pattern
- Template Pattern
- Adapter pattern
- Observer pattern

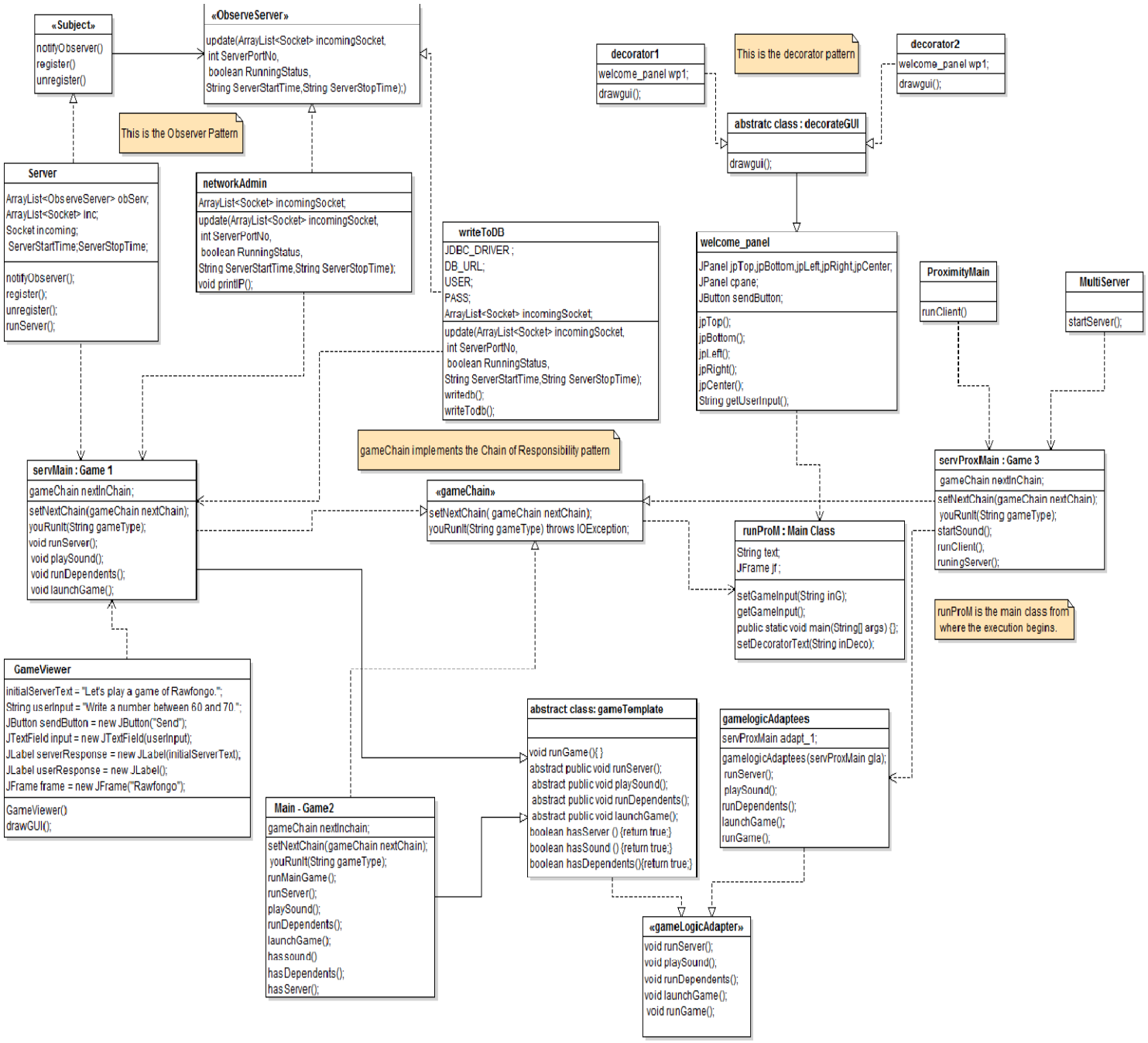
All of these patterns played very key role in overall project and will be discussed shortly.

As all java projects has main method we have one too, i.e. runProM – which initializes the Graphic User Interface. The user has been given three option to decorate its GUI, i.e. Advanced, GameAddictBoy or GameAddictGirl. If user enters nothing it is redirected to simple GUI or default otherwise its GUI is re-directed or repainted using Decorators provided for the GUI class.

After selecting the GUI user has to enter the name of the game he want to play from the given list of options. Once user clicks the send button the GUI executes the Action listener for the button where it further take the input text and send it to the `setGameInput() inside` runProM class.

The setGameInput method is responsible for creating gameChain objects for each integrated game inside library using Chain of Responsibility pattern and propagates the received input text to this chain.
The resulting game which gets executed is decided based on the input text and the responsible class is then called to handle that request.

Class Diagram:



Classes, attributes and methods:

```
public class runProM  {
    private static String text = "";
    public static JFrame jf ;
    public static void setGameInput(String inG){
    public static String getGameInput(){return text;}
    public static void main(String[] args) throws IOException {
    public static void setDecoratorText(String inDeco){

}
```

```
public class welcome_panel extends JPanel{
public  JFrame jf = new JFrame();
public  JPanel jpLeft;
public JPanel jpRight;
public JPanel jpTop;
public JPanel jpBottom;
public JPanel jpCenter;
public JPanel cpane = new JPanel();
public JButton button ;
public JButton sendButton;
public JTextArea jtf = new JTextArea(20,20);
public static String userInput = "Enter the name of game you want to play";
private static JTextField input = new JTextField(userInput);
private static JLabel jl1 = new JLabel("Enter Country Game or Server Game or Proximity");

welcome_panel(){

    public void drawgui(){
    public void jpLeft(){
    public static String getUserInput(){
    public void setUserLabel(String inJ){
    public void jpRight(){
    public void jpCenter(){
    public void jpTop(){
    public void jpBottom(){
    public static void main(String args[]){
}
```

```
abstract public class decorateGUI extends welcome_panel{
      public abstract void drawgui();
}
```

```
public class decorator1 extends decorateGUI{
    welcome_panel wp1;
    decorator1(welcome_panel wp1){
    public void drawgui() {

}
```

```
public interface gameChain {
```

```
        public void setNextChain( gameChain nextChain);
        public void youRunIt(String gameType) throws IOException;
}
```

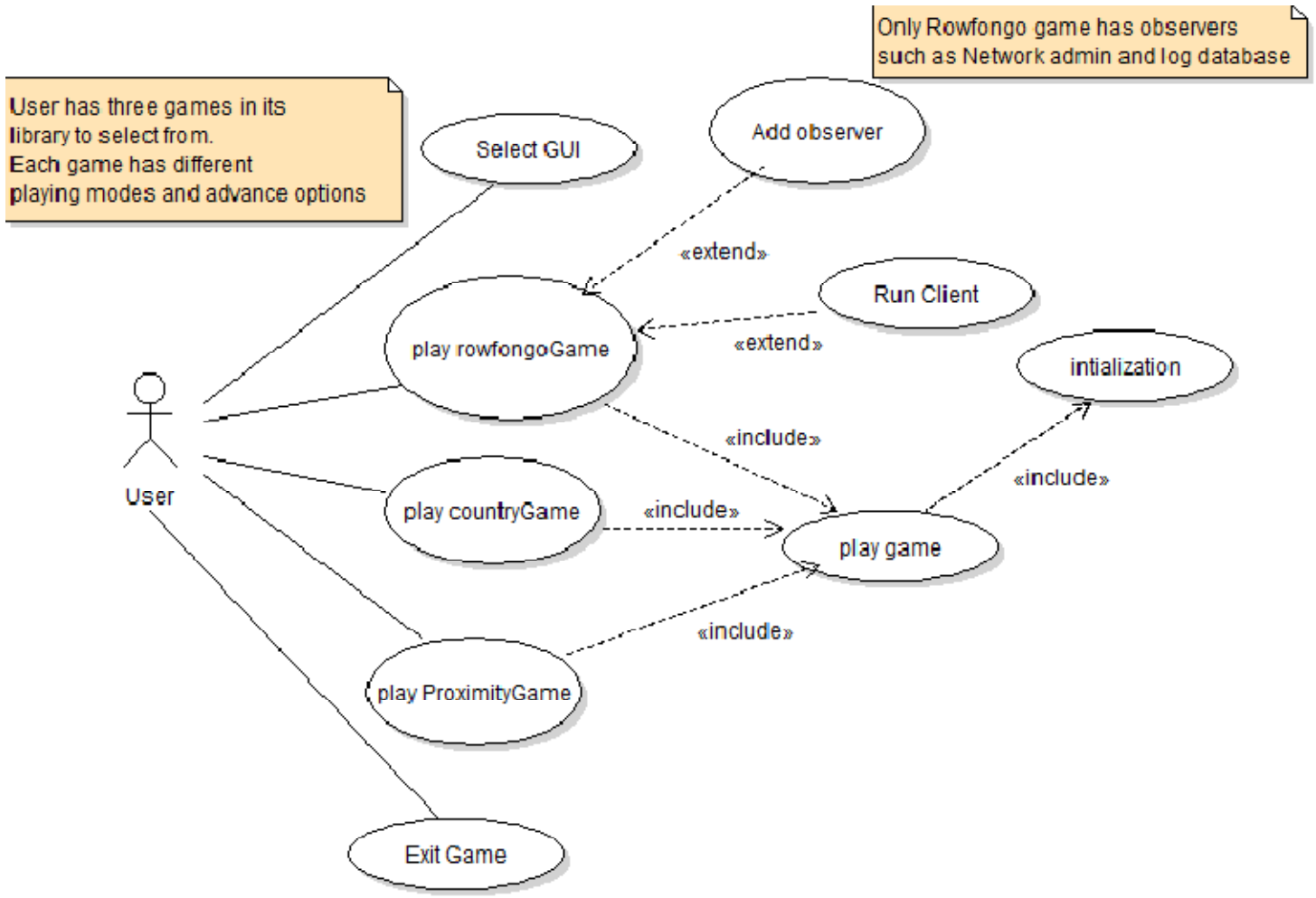
```
public class gamelogicAdaptees implements gameLogicAdapter {
    servProxMain adapt_1;
    public gamelogicAdaptees(servProxMain gla){
    public void runServer() {
    public void playSound() {
    public void runDependents() {
    public void launchGame() {
    public void runGame() {

}
```

```
public interface gameLogicAdapter {
public void runServer();
public void playSound();
public void runDependents();
public void launchGame();
public void runGame();
}

abstract public class gameTemplate implements gameLogicAdapter {
 public final void runGame(){
        if(hasServer() == true){runServer();}
        if(hasSound() == true){playSound();}
        if(hasDependents() == true){runDependents();}
        launchGame();
 }
 abstract public void runServer();
 abstract public void playSound();
 abstract public void runDependents();
 abstract public void launchGame();
 boolean hasServer () {return true;}
 boolean hasSound () {return true;}
 boolean hasDependents(){return true;}
}
```

```
public class servProxMain   implements gameChain{
    private gameChain nextInChain;
    public void runingServer(){
    public void setNextChain(gameChain nextChain) {
    public void youRunIt(String gameType) throws IOException {
    public void startSound(){
    public void runClient(){
    public static void main (String args[]){

}
```

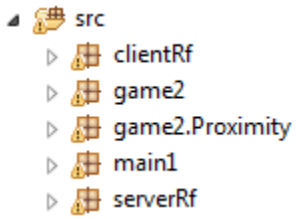
```
public class Main extends gameTemplate implements gameChain{
    public static boolean answer = false;
    public static boolean timeOut = false;
    static String country1;
    static ActionListener listener= new Time();
    final static int DELAY = 15000;
    public static Timer timer = new Timer(DELAY,listener);
    private gameChain nextInchain;
    public void setNextChain(gameChain nextChain) {

    public void youRunIt(String gameType) throws IOException {

    public static void main(String[] args) throws IOException {

    public Main(){
    public void runMainGame()  throws IOException{

    public void runServer() {

    public void playSound() {

    public void runDependents() {

    public void launchGame() {
    boolean hassound(){return false;}
    boolean hasDependents(){return false;}
    boolean hasServer () {return false;}
    }


 * Handles and builds the GUI of our Rawfongo game.
public class GameViewer extends JPanel {

    private static String initialServerText = "Let's play a game of Rawfongo.";
    private static String userInput = "Write a number between 60 and 70.";
    private static JButton sendButton = new JButton("Send");
    private static JTextField input = new JTextField(userInput);
    private static JLabel serverResponse = new JLabel(initialServerText);
    private static JLabel userResponse = new JLabel();
    private static JFrame frame = new JFrame("Rawfongo");
    private GameListener listener = new GameListener();
    public GameViewer(){
     * Obtains the user input.
    public static String getUserInput(){
     * Updates the text shown as the user response.
    public static void updateUserResponse(String s){

     * Updates the text shown as the server response.
    public static void updateServerResponse(String s){

     * Takes care of drawing the GUI.
    private void drawGUI() {

    public static void main (String[] args){
}
```

Use Case Diagram :

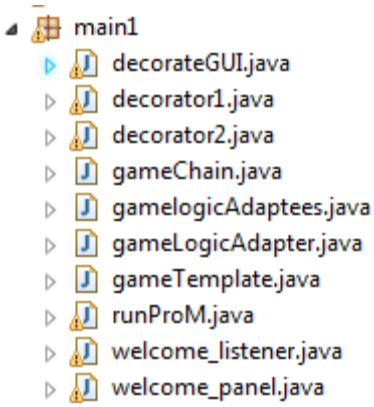


Brief description of important classes and methods:

Package Heirarchy:



Inside Package Main1:



This is the main method from where execution begins. This class basically depends on welcome_panel class and uses the object of gamechain interface to initialize all the games added in the library.

```
public class runProM {}
```

void setGameInput(String inG) {..}// this method is used for providing the input from GUI to set of chain.

This class is responsible for drawing the GUI for the user to interact with.

```
public class welcome_panel extends JPanel{..}

       drawgui(){..}

              public void jpLeft(){ ..}

              public void jpRight(){ ..}

              public void jpTop(){ ..}

              public void jpBottom(){ ..}

              public void jpCenter(){ ..}
```

Abstract class for implementing various decorator for our welcome_panel

```
abstract public class decorateGUI extends welcome_panel{
	public abstract void drawgui();
}
```

Below are the two decorators which uses the welcome_panel class object to add the behavior at run time to the already existed GUI.

```
	public class decorator1 extends decorateGUI{..} drawgui();

	public class decorator2 extends decorateGUI{..} drawgui();
```

Interface for gameChain:

```
/*
 * This is the gameChain interface which itself is a inclusive part of Chain of Responsibility Pattern.
 * It provides the chain of different games available in library
 * It help in rotating the turn to each of its implementing class according to input form client
 */
public interface gameChain {
	//this method provides the functionality to add or set next chain.
	public void setNextChain( gameChain nextChain);

	//this method decided either the class will take the responsibility or pass it to the next class in chain.
	public void youRunIt(String gameType) throws IOException;
}
```

Template for game integration using Template method:

```
/*This is the gameTemplate for the developer so that they can simply develop there game algorithm simply using this template.
 * Which would further ease the process of adding the game in a chain to let the user play it.
 * It defines the set of rules or procedure which should be followed while developing game for our library
 *
 */
abstract public class gameTemplate implements gameLogicAdapter {
```

```
//this method provides fixed algorithm to implement by its child classes so no rules can be
negotiated.
 public final void runGame(){
       if(hasServer() == true){
              runServer();
        }
       if(hasSound() == true){
             playSound();
       }
       if(hasDependents() == true){
             runDependents();
       }
       launchGame();
 }
abstract public void runServer();
 abstract public void playSound();
 abstract public void runDependents();
 abstract public void launchGame();

// these are the hooks to use with template method pattern to avoid conflict when someone just don't
want to over-ride any method in the template thus makes it more flexible.
boolean hasServer () {return true;}
boolean hasSound () {return true;}
boolean hasDependents(){return true;}

}
```

Interface for GameLogicAdapter:

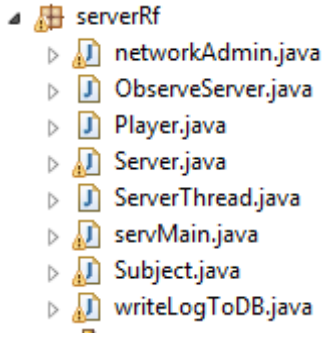
```
/*this is the basic interface for both our Template and adapter are implementing this *common
interface and play with it as required.
*/
public interface gameLogicAdapter {
public void runServer();
public void playSound();
public void runDependents();
public void launchGame();
public void runGame();
}
```

Implemented adapter for one of the game proximity:

This gamelogicAdaptees provide way to those classes which do not implement template class and in this project we have servProxMain class which used this Adapter.

```
public class gamelogicAdaptees implements gameLogicAdapter {..}
```

Inside Package serverRf:



We have interface subject a key part in Observer pattern: It provides common set of methods to implement by the Server class in order to let Observer register, unregister or get notified when changes occurs in Server.

```
public interface Subject {
	public void register(ObserveServer o);
	public void unregister(ObserveServer o);
	public void notifyObserver();

}
```

Another interface as a key part of observer pattern: It provides the changes coming from the Subject to its child classes.

```
public interface ObserveServer {

public void update(ArrayList<Socket> incomingSocket, int ServerPortNo, boolean RunningStatus, String
ServerStartTime,String ServerStopTime);

}
```

The server class for one of our game(serverRF) which has implemented observer pattern and has some observers too:

```
public class Server implements Subject

	public void runServer() throws IOException{ .. }
```

The Observers of our server has implemented ObserveServer interface to get updates from server when some changes occurs:

```
public class networkAdmin implements ObserveServer {

	public void printIP() { .. }// this method provided the information about the incoming socket

public class writeLogToDB implements ObserveServer { .. }

	public void writedb() { .. }// this method writes the log of server to the database
```

Inside package game2:

For proximity game

In game2 servProxMain implements the gameChain interface and acts as an adaptee to our adaptor pattern

```
public class servProxMain implements gameChain{}
```

for country game which inherits the template class and also participates in chain of responsibility pattern.

```
public class Main extends gameTemplate implements gameChain{}
```

Description of how design patterns were used:

Lets talk about all the design pattern used in this project so far one by one.

1 **Observer Pattern:** This design pattern is used inside package serverRf which includes the server code for one of our game name “Rowfongo”. This game cannot be played without server is running and one can run server either locally of any other host making sure that client which is in package clientRf has the correct IP address of server to connect to.

The observer pattern is added to the server for various tasks , as our server is capable of receiving requests from game client and to play game with server itself using java Sockets.

Every time a client connects to server on specific port number and IP address to play game, server accepts the connection and opens a thread for the same to play game. As well as we have added some observer here which keep any eye on performance and security of server promising full logging of the activities of server.

Hence I have added

-Network Admin

-Log database

Where network admin is a team which receives updates from the server whenever a new connection is made or whenever an attempt to change status of server is made by program itself or any external factor.

Further, we have a live log collection database which no matters what happens collect the stats of server that when it was up who connected and other information.

The Subject interface is used as one of the component as described in Observer pattern which help in Observer to register & unregister the observer and also to get update notifications from server.

```
public interface Subject {

	public void register(ObserveServer o);
	public void unregister(ObserveServer o);
	public void notifyObserver();

}
```

```
The ObserveServer is used as an another component describe in Observer Pattern which helps in
linking different observers to get updates from server.

public interface ObserveServer {

	public void update(ArrayList<Socket> incomingSocket, int ServerPortNo, boolean RunningStatus,
String ServerStartTime,String ServerStopTime);
}
```

2. **Chain of Responsibility Pattern**:

```
/*
 * This is the gameChain interface which itself is a inclusive part of Chain of Responsibility
Pattern.It provides the chain of different games available in library
 * It help in rotating the turn to each of its implementing class according to input form client
 */
public interface gameChain {
	public void setNextChain( gameChain nextChain);
	public void youRunIt(String gameType) throws IOException;
}
```

This pattern has played a very important role in my project by binding all of the game classes together and propagating the responsibility to the class which is able to handle the user action.

This interface must be implemented by all such classes which are able to act as per the user input. In our project we have three chain classes such as:

```
gameChain g1 = new Main();
gameChain g2 = new servMain();
gameChain g3 = new servProxMain();
```

After initializing the chains we have to set the order in which they should pass request to one another until it gets handled. We implemented that as:

```
g1.setNextChain(g2);
```

```
try{
	g2.setNextChain(g3);
}catch(Exception e){System.out.println("No game with this name");}
```

The final part is to set the chain on fire means initiating the chain and passing the incoming text from user and it is done as :

```
try {
	g1.youRunIt(getGameInput());
} catch (IOException e) {
	// TODO Auto-generated catch block
	e.printStackTrace();
}
```

Where getgameInput() is responsible for getting input from the gui and passing it along the chain.

3. **Template method pattern:**

As we know from the starting that game developers should have a prototype in front of them to develop any game for our library, this is achieved by using the template pattern.

Developers are free to extend this template and its logic and could write their code in anyway as they like but keeping the template of their class as it is mentioned in the class by over-riding all the methods applicable.

Hooks are also provided for all the methods as it could be the case that some may want to over-ride all the methods or some of them.

```
abstract public class gameTemplate implements gameLogicAdapter {

 public final void runGame(){
       if(hasServer() == true){
              runServer(); }
       if(hasSound() == true){
             playSound(); }
       if(hasDependents() == true){
             runDependents();}
       launchGame();
 }
abstract public void runServer();
 abstract public void playSound();
 abstract public void runDependents();
 abstract public void launchGame();

//Hooks for the extending class
boolean hasServer () {return true;}
boolean hasSound () {return true;}
boolean hasDependents(){return true;}

}
```

Those extending classes which do not have implementation for one of method defined abstract in above class should use a hook given for that method and override its value to false. Thus that method would get ignored at runtime.

4. **Decorator Pattern:**

As the name suggest “decorator” its duty is to decorate and in my project I have used it for decorating my welcome GUI for three different types of users i.e. Advanced, boys or girls.

```
abstract public class decorateGUI extends welcome_panel{
       public abstract void drawgui();
}
```

All it has is this simple abstract class which extends the another class on which decoration has to be done. Here welcome_panel class is responsible for creating frame adding panels labels etc and the decorateGUi provides the way of modifying or enhancing effect of welcome_panel without changing the class itself.

All those classes which will extend this abstract class would be able to add behavious to the exitisng class.

```
public class decorator1 extends decorateGUI{
```

```
	welcome_panel wp1;
	decorator1(welcome_panel wp1){
		this.wp1 = wp1;
	}
	public void drawgui() {..}
```

This is the given example of one of the decorator.

5**. Adapter Pattern:**

Again, as the name suggest Adapter – it works very similar to adapter we use at our home. As we know our template pattern helped us in providing developer to develop game according to our library's needs but what about those guys who developed their games as a freelance and still want to integrate the game in our library. Adapter patters provides the solution.

```
public interface gameLogicAdapter {

public void runServer();
public void playSound();
public void runDependents();
public void launchGame();
public void runGame();


}
```

Above is the Adapter interface which is must be implemented by all the clients who do not want to extend our template bt still want to be part of our library.

```
public class gamelogicAdaptees implements gameLogicAdapter {
servProxMain adapt_1;
	public gamelogicAdaptees(servProxMain gla){
		adapt_1 = gla;}
	public void runServer() {
		// TODO Auto-generated method stub
		System.out.println("Initializing Server.."); }
	public void playSound() {
		// TODO Auto-generated method stub
		adapt_1.startSound();}
	public void runDependents() {
		// TODO Auto-generated method stub
		adapt_1.runClient();
		System.out.println("client are running fine");}
	public void launchGame() {
		// TODO Auto-generated method stub
		System.out.println("Server is up");
		adapt_1.runingServer();}
	public void runGame() {
		// TODO Auto-generated method stub
		System.out.println("Game is running");
	}}
```

Above is the live example of our proximity game which implemented the adapter and its adaptee class provides the interaction between the methods of proximity class and our logic adapter.