ARYAN PUWAR (PROJECT 11 - SIMULATE A GAME OF HOUSIE)
DOCUMENTATION

1. Interface - WinningCondition

Methods - public boolean checkAvailability();
public void setAvailability(boolean b);
String verifyCondition(Ticket t);
String getReward();

2. Class - FirstRow

Instance Variables - private boolean availability;
private final String reward;

Methods - a) public boolean checkAvailability()
-checks whether the winning condition is available.
b) public void setAvailability(boolean b)
-sets the Availability equal to boolean argument b.
c) public String verifyCondition(Ticket t)
-takes a ticket as input and verifies whether it has satisfied the first row condition.
d) public String getReward()
-returns the reward as a string to be shown.

3. Class - SecondRow

Instance Variables - private boolean availability;
private final String reward;

Methods - a) public boolean checkAvailability()
-checks whether the winning condition is available.
b) public void setAvailability(boolean b)
-sets the Availability equal to boolean argument b.
c) public String verifyCondition(Ticket t)
-takes a ticket as input and verifies whether it has satisfied the first row condition.
d) public String getReward()
-returns the reward as a string to be shown.

3. Class - ThirdRow

Instance Variables - private boolean availability;
private final String reward;

Methods - a) public boolean checkAvailability()
-checks whether the winning condition is available.
b) public void setAvailability(boolean b)
-sets the Availability equal to boolean argument b.
c) public String verifyCondition(Ticket t)
-takes a ticket as input and verifies whether it has satisfied the first row condition.
d) public String getReward()
-returns the reward as a string to be shown.

4. Class - FullHouse

Instance Variables - private boolean availability;
private final String reward;

Methods - a) public boolean checkAvailability()
-checks whether the winning condition is available.
b) public void setAvailability(boolean b)
-sets the Availability equal to boolean argument b.
c) public String verifyCondition(Ticket t)
-takes a ticket as input and verifies whether it has satisfied the first row condition.
d) public String getReward()
-returns the reward as a string to be shown.

5. Class - Ticket

Instance Variables - public int[][] playerTick = new int[3][9];

Methods - a) public int[][] getPlayerTicket()
-returns the player ticket as an array.
b) public void printTicket()
-prints the players' ticket when called.

6. Class - Player

Instance Variables - public String playerName;
public int playerId;
Ticket playerTicket;

Methods -a) public boolean hasCurrentNumber(int n)
-checks whether the mentioned player's ticket contains the announced number.
b) public String verifyWin(ArrayList<winningCondition> wincond)
-verifies whether the player has satisfied any winning condition and has won a prize

7. Class - Moderator

Instance Variables - ArrayList<Integer> rand = new ArrayList<Integer>();
int numberOfPlayers;
ArrayList<Player> playerList = new ArrayList<Player>();
ArrayList<String> winners = new ArrayList<String>();
public ArrayList<winningCondition> wincond;

Methods - a) public void setWincond(ArrayList<Integer> set)
-sets the winning conditions chosen by the user and makes them available.
b) public int drawNumber()
-draws a random number between 1 and 90 to be announced by the moderator.
c) public int[][] generateTicket()
-generates a random housie ticket for each player.
d) public void assignTicket()
-assigns the randomly generated ticket to each player created in the player class.
e) public boolean checkEndGame()
-checks whether the game as ended, meaning all the winning conditions have been completed and no more reward remains.

8. Class - GameData

Instance Variables - public int announcedNumber = 0;
public boolean gameCompleteFlag = false;
public boolean noAnnouncedFlag = false;
public ArrayList<Boolean> playerChanceFlag;

Methods - a) void setPlayersChanceFlagFalse()
-sets the chance flag for all players as false when moderator has not announced a Number.
b) boolean everyPlayerChanceOver()
-checks if all the players have had their chance at reading the announced number.

9. Class - Main

Takes input from the user (number of players, winning conditions, player name etc.)
Prints the UI of the program.
Prints the ticket, the drawn number, the player from whose ticket a drawn number has been cut, the reward and the winners.

Our code resembles a lot of properties with the `Observer Design Pattern`. The Moderator Class is the subject which observes/calls a new random number, it then notifies each Player that a

new number has been called and to check the called number on their respective tickets. All the players who have subscribed to the game from the moderator are the observers in this Design Pattern. These observers check their respective tickets for the presence of the called number.

`Strategy Design Patten` wasn't used in my code, as here the behaviour/ properties of Players and Moderators are limited and this restricts the stategies that these Players can follow. So this pattern wouldn't help us here.

`Decorator Design Pattern` wasn't used in my code, as we don't want the functionality of repeatedly encapsulating the objects at dynamic memory time. This method would turn out to be redundant for our problem in hand.

DESIGN PATTERNS

I haven’t implemented Strategy and Decorator Design Patterns in my code as they won’t be of much help because the concept of housie and my code structure is not good for these types of design patterns.

The Observer Design Pattern is really useful in my code. The observer pattern is a software design pattern in which a subject object keeps track of its dependents, called observers, and notifies them of any state changes automatically, usually by calling one of their methods.

So, as we can clearly compare, the players are the observers who depend on the host (the moderator) and the functions called by the host. The whole code revolves around the methods the moderator calls which keep the game of housie going on. So, the subject is the moderator. For Example, the moderator calls the generate ticket, assign ticket, draw number, end game and set winning conditions methods and the players are given tickets, or cut the announced numbers from their tickets using the hasCurrentNumber method.

CRITICAL ANALYSIS OF 4 OOP PRINCIPLES -

1. Program to an Interface not Implementation -

I have used the winning condition interface to define the methods for the 4 winning conditions - First Row, Second Row, Third Row, Full House.
This made it easier for me to implement the similar logics in these classes and it will be easier in the future to make changes in the classes, as the interface can be changed to give clarity about what needs to be changed in each class.

I could have made a better use of the Interface concept by using interfaces to define classes like moderator and player instead of implementing most of the code in moderator class. This would have made the code more elastic, or you could say, flexible instead of the currently comparatively crowded moderator class.

2. Encapsulate what varies -

Encapsulating what varies is an approach for dealing with details that change regularly. When code is constantly updated to accommodate new features or requirements, it tends to become messy. We reduce the part that will be affected by a change in requirements by separating the sections that are prone to change.

For Example, to add a new winning condition, we have an interface WinningCondition and any new winning condition can easily be implemented. This is similar to point 1, as interfaces help us too make the code flexible and easily changeable for new details.

3. Classes should be open for extension and closed for modification

This is one of the most important principle of Object Oriented Programming.
This principle's overall concept instructs us to write your code in such a way that you may add new features without having to change the present code. This avoids instances where a change to one of your classes necessitates the adaptation of all dependent classes.
I tried making my code as adaptable as possible. But, I still feel that it is not enough because inheritance can make a code even more adaptable, which I haven't used. Using inheritance would have made it a lot better, which I think I should try in all future codes I write.

4. Depend on abstraction, do not depend on concrete classes

When you rely on a specific instance of an object, you're introducing a dependency into your code, which limits its reusability. You will always have the requirement for a specific concrete class if you code to it. If you code to an interface/ abstraction, however, you may alter your code to operate with any number of classes that implement that common interface.
In Java, this indicates that you should code to an Interface whenever possible, rather than having your function reliant on specific things.
I feel like in some places I have tried to make concrete classes, especially the moderator class, which has been concretely coded for the specific features used in the code. The interface has been used for Winning Conditions, which makes my code a little abstract, but still, more abstraction would have made it easier for me to make future changes to the program if required.

ACKNOWLEDGEMENTS -

https://stackoverflow.com
https://stackify.com

https://en.wikipedia.org
https://github.com