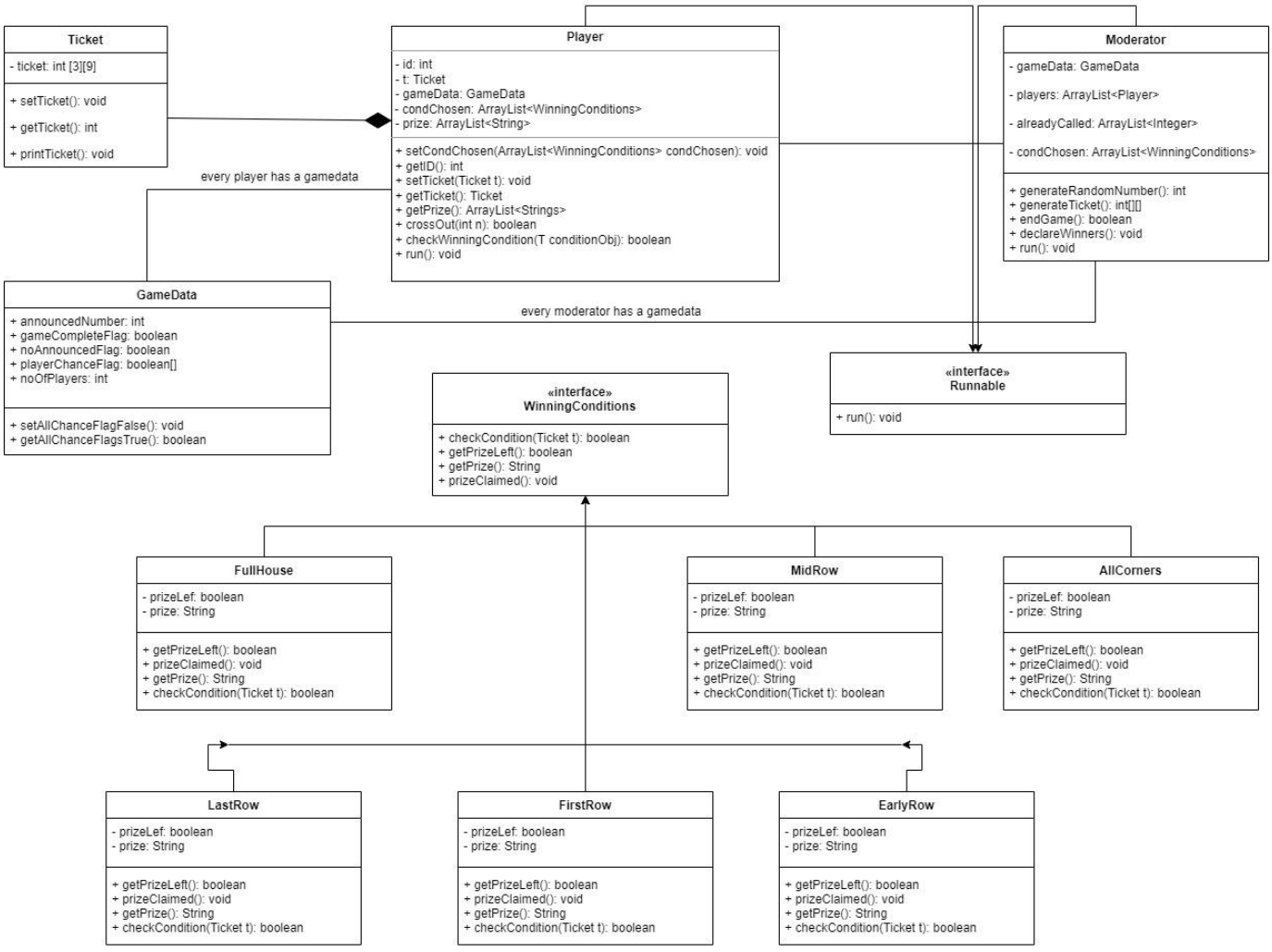
DOCUMENTATION

The code is well commented on all of its main aspects. However, this document explains all the classes in detail.

Video Explanation Link:- https://www.loom.com/share/0f49da6cf30a404d81060fa6f127cc76
OnlineGDB Code Link:- https://www.onlinegdb.com/ZsfIII3Nf

UML:-



The program has an interface named WinningConditions that has the following methods:

1. checkCondition - It takes the object of ticket class as a parameter.
2. getPrizeLeft - It is used to get the value of prizeLeft attribute of the classes that implements the WinningConditions interface.

3. getPrize - It is used to get the value of the prize attribute of the classes that implements the WinningConditions interface.
4. prizeClaimed - It is used to change the prizeLeft attribute of the classes that implements the WinningConditions interface to false once a player wins the prize.

The program contains the following classes:

1. Ticket.java - This class has a private attribute named ticket which is a 2-D Array of dimensions 3*9. It also has getter and setter methods for the ticket. It also contains a printTicket method that prints the ticket. Internally the indexes that are displayed as blank spaces store '-1' in the ticket and the indexes that are crossed out on calling are stored as '-2' in the ticket.

2. FullHouse.java, FirstRow.java, MidRow.java, LastRow.java, EarlyFive.java, AllCorners.java: All these classes implement the WinningConditions interface. They have a boolean prizeLeft and String prize as their attributes. The constructor initialises the prizeLeft with "true" and prize with the corresponding winning condition'. prizeLeft tells if the prize is already claimed by a player. prizeClaimed method sets prizeLeft to false once the prize is claimed by any player. Finally, they have a checkCondition method that takes an object of type Ticket as a parameter and returns true or false depending on whether the ticket satisfies the corresponding winning condition.

3. Moderator.java - The moderator has the responsibility of assigning a random ticket to all the players once its constructor is called. At the same time, it also stores all the winning conditions that are chosen by the user. It implements the runnable interface and overrides its run method for multithreading. It generates a random number and assigns it to gamedata. Then its thread waits until all the players have read the announced number, after which it repeats the process from generating a random number.

4. Player.java - Each player has a unique randomly generated ticket and an ArrayList storing all the prizes that the player has won. Player thread waits while the moderator has not announced a new number. When a number is generated and stored in a GameData object, the player thread accesses it and crosses out the number from its ticket if applicable. It then checks all the chosen winning conditions and if it is satisfied and the prize is left, it claims the prize.

5. GameData.java - This helps the moderator and player threads to know when to get active and when to wait so as to avoid conflict. The boolean array stores the chances of reading the announced number of all of the players. Until each player's chance is not over, the moderator thread waits to announce a new number. The noAnnounced boolean stores if a new number has been announced by the moderator until which all the player threads wait for the number to be announced.

6. Main.java - It is the driver class that contains the main function. In this method the user has to enter the number of players who are playing the game and select the winning conditions. It also creates separate threads for the moderator and each of the players and starts them.

Design Patterns

The Housie program resembles the observer design pattern.

1. In Strategy Design pattern, a class behaviour or its algorithms can be changed at run time. This type of design pattern comes under the behaviour pattern. It is not used in our program.
2. The Observer Design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   In our program the Moderator is the subject i.e. the one who notifies and updates all its dependencies and all the players playing the game are the observers i.e. the one who waits for the subject to notify them the changes and update them.
   In the game, the Moderator randomly announces a number and notifies all the players through the GameData object about the current number and if the ticket of any of the players contains the announced number, then that number is crossed out in that player's ticket and the ticket is updated.
3. Decorator Design pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. It is not used in our program.

OOP PRINCIPLES

1. Encapsulate what varies - The code successfully applies this principle. The attributes of all the classes are defined private and we have public getter and setter methods to access them and so that the user cannot modify them directly.
2. Favour composition over Inheritance - Classes should achieve polymorphic behaviour and code reuse by their composition rather than inheritance from a base or parent class. Composition offers better test-ability of a class than Inheritance. If one class

consists of another class, you can easily construct a Mock Object representing a composed class for the sake of testing. This privilege is not given by inheritance. This code is a good example of composition as Player class has an attribute ticket of the type Ticket and can easily do modifications of it. Also player and moderator class have an attribute of type GameData.

3. Program to an interface not implementation - Coding to interfaces is a technique to write classes based on an interface; interface that defines what the behaviour of the object should be. This code is a program to interface as it has an interface named WinningConditions. Classes like FullHouse, FirstRow, MidRow, LastRow, AllCorners, EarlyFive implements WinningConditions interface as objects of all these classes have similar functioning.
4. Strive for loose coupling between objects that interact - The code does achieve this mostly. It can be evaluated to be partially following this principle. Most member variables in class are private and can be changed only via setter methods. This is loose coupling.
5. Depend on abstraction, do not depend on concrete classes - The code fails to achieve this as all the classes are concrete and no class is abstract.