n-Player Tambola Game using Multi-threading

Name – Neelabh Sinha

Birla Institute of Technology and Science, Pilani

Problem Statement

The Objective of the Assignment

Problem Statement

- Let there be one Moderator and ‘n’ Players. The moderator displays 10 random numbers (between 0 – 50) on a display screen with one-minute gap between the numbers. Each player is given a card containing 10 random numbers (between 0 - 50). As the number is displayed, the player strikes the number on his card if it matches with the one on the screen. The player who strikes three numbers first will be announced as a winner and the moderator stops generating numbers if a player wins before all the 10 numbers are generated. The numbers generated by the moderator and the player can be redundant. For e.g. if the moderator generates a 20 twice and if a player has one 20. After the player strikes the number, it should not be considered as a match, when 20 appears again on the display screen. The moderator generates 10 numbers and stores it in an array list of integers. When the moderator is generating the number, the players should not read the array list. The moderator should generate the next number only after all the players have read the previously generated number. The array list should not be read by two players at the same time.

Design Structure

Details about the design aspects of the application

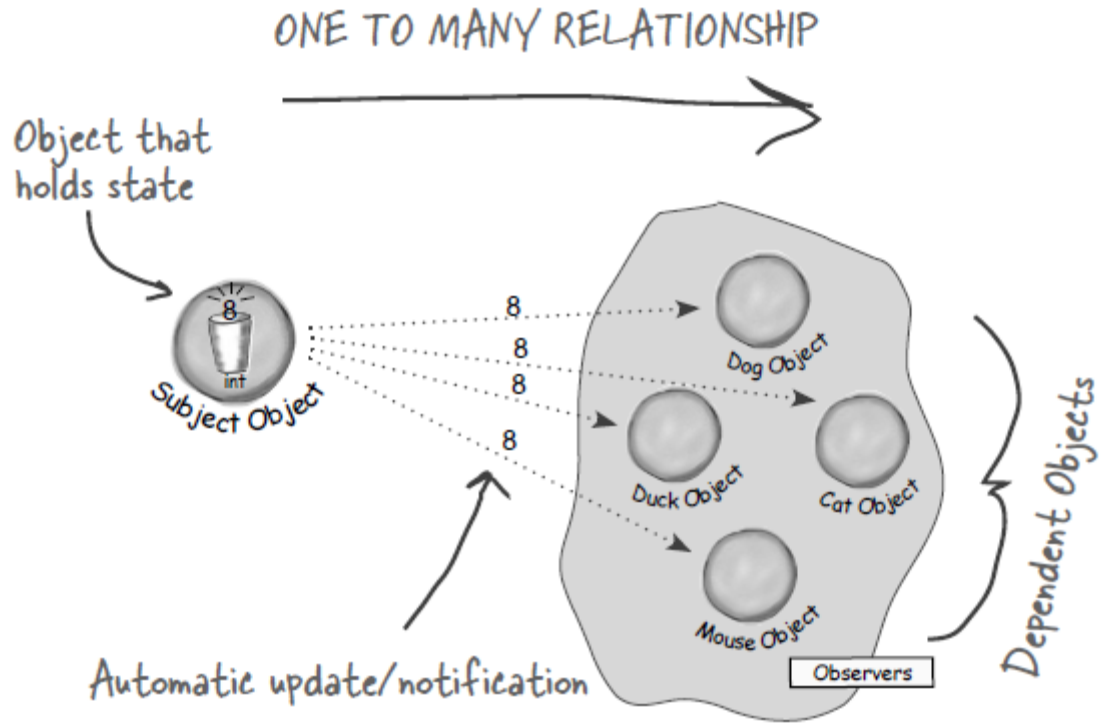
Design Patterns Used

- **Singleton Pattern** – Singleton Pattern is used for the moderator because there is only one moderator that needs to be there to conduct a game, and thus, singleton pattern will restrict creation of multiple moderator objects in the application

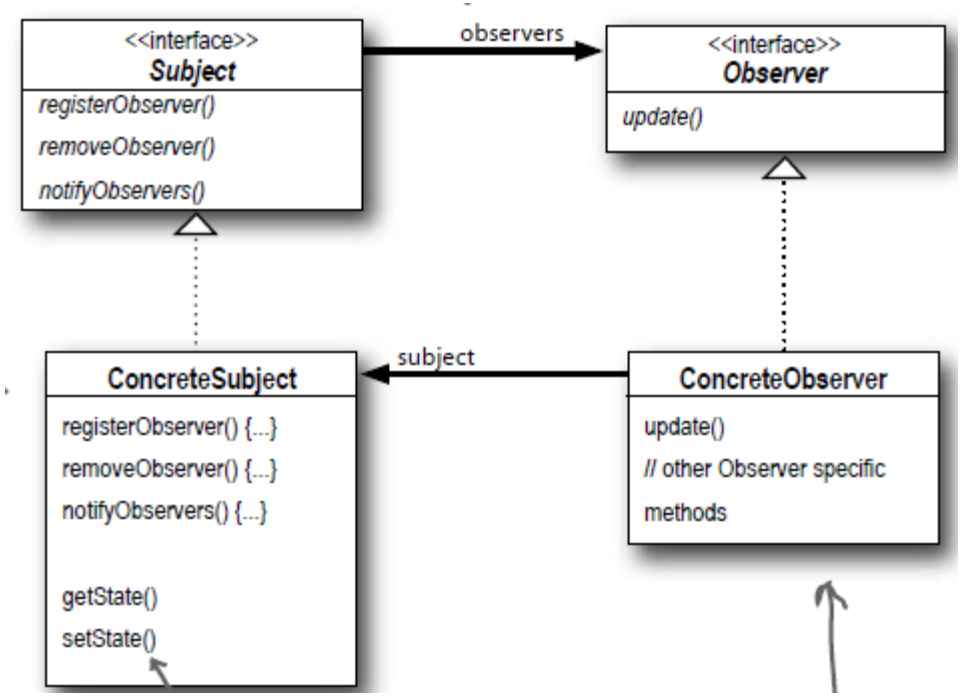
- **Observer Pattern** – Observer Pattern is used to establish interaction between the observer and the players, because moderator is the subject that establishes a change, and then notifies the players to respond to it, which the players do. Hence, the players are the observers.

Observer Pattern

- **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



Observer Pattern



Singleton Pattern

- **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

- We need only one moderator. Singleton Pattern will ensure no more ConcreteModerator objects are created

How is Singleton Pattern ensured?

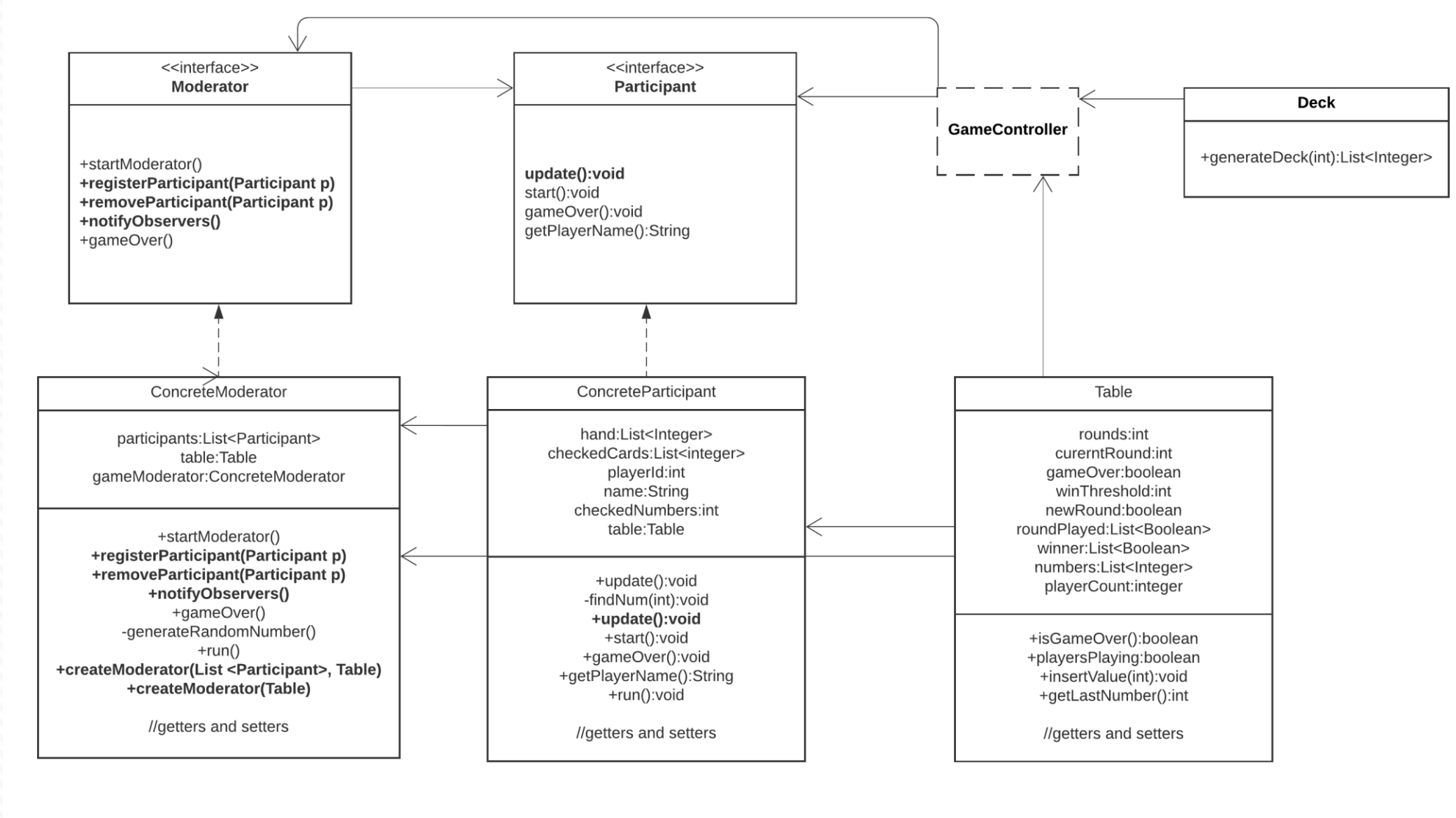
```
private volatile static ConcreteModerator gameModerator;

private ConcreteModerator(List <Participant> participants, Table table) {
                super("Moderator");
                this.table=table;
                this.participants = participants;
}

public static ConcreteModerator createModerator (Table table) { //method to create singleton object without parameters
        if(gameModerator == null) {
                synchronized(ConcreteModerator.class) {
                        if(gameModerator == null) {
                                gameModerator = new ConcreteModerator(table);
                        }
                }
        }
        return this.gameModerator;
}
```

UML Diagram

UML Diagram of the Project


<<interface>>
Moderator
+startModerator()
+registerParticipant(Participant p)
+removeParticipant(Participant p)
+notifyObservers()
+gameOver()
<<interface>>
Participant
update():void
start():void
gameOver():void
getPlayerName():String
GameController
Deck
+generateDeck(int):List<Integer>
ConcreteModerator
participants:List<Participant>
table:Table
gameModerator:ConcreteModerator
+startModerator()
+registerParticipant(Participant p)
+removeParticipant(Participant p)
+notifyObservers()
+gameOver()
-generateRandomNumber()
+run()
+createModerator(List <Participant>, Table)
+createModerator(Table)
//getters and setters
ConcreteParticipant
hand:List<Integer>
checkedCards:List<integer>
playerId:int
name:String
checkedNumbers:int
table:Table
+update():void
-findNum(int):void
+update():void
+start():void
+gameOver():void
+getPlayerName():String
+run():void
//getters and setters
Table
rounds:int
curerntRound:int
gameOver:boolean
winThreshold:int
newRound:boolean
roundPlayed:List<Boolean>
winner:List<Boolean>
numbers:List<Integer>
playerCount:integer
+isGameOver():boolean
+playersPlaying:boolean
+insertValue(int):void
+getLastNumber():int
//getters and setters

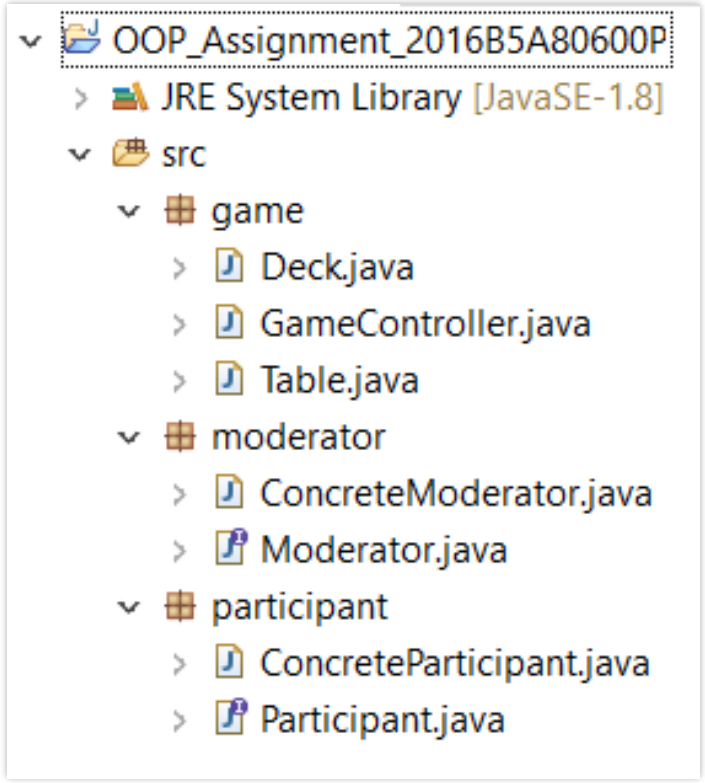
Code Explanation

Complete Walkthrough of the implementation with Explanation of different modules

Coding Ethics Followed

- Proper Packaging of different modules
- Coding to the interface, and not the implementation
- Properly Commented Code (so that it is easy to understand while going through it)

Project Structure



- 3 packages for the game, participant (Player) and moderator part
- **Deck** – handles generation of cards for players
- **GameController** – has responsibility of controlling the game (main thread)
- **Table** – shared data where game is played
- **Moderator** – interface that defines skeleton of Moderator (Subject as per observer pattern)
- **ConcreteModerator** – Implementation of Moderator (with Singleton pattern associated)
- **Participant** – interface that defines skeleton of participant (Observer as per observer pattern)
- **ConcreteParticipant** – Implementation of Participant

A Note before we begin

- At certain places, there are delays made in I/O by making the thread sleep, so that output can be visualized properly.
- However, since this happens in between and other threads are waiting for that particular thread to notify(), it does not affect the overall functionality. It will still behave the same way if the delays are removed.
- Actual communication happens using wait() and notifyAll(), so, this interaction will be consistent with any value of delay
- Since this is forced, code's running time can be made remarkable fast by removing these, with no other drawbacks

Code Walkthrough

Explanation of the code through the java files in the project.

Conceptual Understanding

Concepts involved in designing the code

Design Patterns

- Already dealt earlier

Conforming to SOLID Principles

- Single Responsibility – We can easily see that there is one class for each responsibility. Like, table, Game Controller, Deck, Participant, Moderator, everything.
- Open/Closed Principle – Interfaces are open for extension, and closed for modification. We don't need to modify the interfaces, just implement them. We have everything that is needed
- Liskov Substitution – In moderator and participant, we can either use the reference of moderator/participant, or ConcreteModerator/ConcreteParticipant. It will behave the same
- Interface Segregation – Classes don't depend on interface they don't use
- Dependency Inversion – Not much of dependency already.

Exception Handling

- Interrupted Exceptions have been properly handled with accurate description where it occurs and the stack trace

- No other checked exception occurs in the code, and by thorough review, no unchecked exception is inspected to occur either, at almost all places.

Use of Generics and Collections

- ArrayList collection of list interface has been used at various places wherever required instead of arrays

- This is because it provides several out-of-the-box functions to do operations that are needed, and allows us to write cleaner and compact code by using those

I/O

- I/O has been used using System.out.println() for normal output and System.err.println() for exception displaying wherever needed for friendly output of the code that makes user to understand the process properly

Multithreading

- One of the main concepts involved in the entire application
- There are separate threads for moderator and each object
- Main thread starts these threads, and moderator and players communicate through inter-thread communication methods wait() and notifyAll(). This allows proper communication regardless of which processor executes the code as sleeping duration way of controlling execution of threads is very hard to control for universal execution
- At the end, since it is a good practice that main thread concludes at last, join() method is used in main to wait while all threads join it. It then announces the results and terminates at last.

Thank you!

Neelabh Sinha

2016B5A80600P