CSF213: Object Oriented Programming Project

Topic
Simulate a game of Monopoly(Topic 1)

Group Members
Nidhish Parekh : 2020A7PS0986P
Debjit Kar : 2020A7PS0970P

Teaching Assistant
Prarabdh Nilesh Garg

https://drive.google.com/drive/folders/1E6Dv89vq2mBAcO23j02yt7pj2hquCmZh?usp=sharing

Description of Classes & Methods used in the code

Square Class : Square class represents all the squares which will be present on the board

Bank Class: Bank class contains 2 static ArrayLists, one for holding the squares called “tickets” and one for the Community Chest/ Chance cards called “commCards”.
It has 2 methods, setProp() to add Squares to the tickets ArrayList and setCards() to add Cards to the commCards ArrayList

Player Class : Player class has a constructor which takes the name of the player as the only argument. The class assigns 1500 to the wallet of every player and initializes his position to 0. An ArrayList boughtProp holds the names of all the properties that the player owns.

Card Class : Card class has a constructor which takes the name of the card as the only argument. Objects of this card class will be stored in the commCards ArrayList created in the Bank class.

State Class : State class has a static integer called game_turn which represents the current turn for each round of the game.

Dice Class: Dice class has a roll() method which initializes an object of the Random class since we want to generate a random integer from 2 to 12 which will serve as output for the roll() method.

Board Class : Board is the class which implements multi-threading. So it implements the Runnable interface. It’s constructor takes information(player_No, player himself and his turn no) about the player whose thread is being initialized. Since, this is the class which implements Runnable, we need to provide a definition to the run() method which is an abstract method of Runnable.
We put the entire content of the method in a while loop which will run infinitely.
We create a sleep condition for the thread which will keep the threads of other players asleep when one player’s thread is running.
An ArrayList has been created to store the information of each player.
An object of the Dice method is created to call the roll() method.
The output of the roll() method will be assigned to an integer called ‘move’ which represents the number of steps moved by a player in a turn.
To ensure the player's position comes back to 0 after the 28th block, we make a if statement which will keep the position less than 28 and will also add 200 to the player's wallet for passing GO.

Now,
Players can land on a ticket square or a special square, which we have differentiated using a boolean.
If he steps on a ticket square, there are 3 possibilities

1) Square is available to buy and he has enough money, then he buys it
2) Square is available to buy but he does not have enough money
3) Square is already owned , so he pays rent

There are 6 special squares

1) Community Chest : Cards are stored in a stack which pops off the first element when a player picks a card and adds it to a random position in the stack afterwards
2) Chance : same functioning as Community Chest
3) GO
4) Free Parking

5) Jail
6) Go To Jail

Once every player has played once in 1 round, the current Game State will be displayed such that the Player name will be followed by current position,wallet balance and properties owned.

Main Class: Main class takes the number of players from 2 to 4 as input from the user.
A For loop has been used to initiate threads for each player. The threads contain the logic for each player and hence game will start simulating

CRITICAL ANALYSIS OF OOP PRINCIPLES

1) Encapsulate what varies

This is one of the most important OOP principles. It helps make the code expressible and maintainable. The idea is to use expressive methods and concrete classes.

In my code, I could have used getter and setter function to return values rather than directly use the value This will als

The value of the price varies for different tickets. I could have achieved encapsulation by declaring all the variables in the class as private and writing a public method in the class to SET and GET the values of the variables.

As the complexity increases and the code base grows, without encapsulation the code will become very messy.

2) Favour composition over inheritance

We use composition to establish a Has-A relationship rather than a Is-A that is achieved by inheritance.

In my code, I have certain properties which are ticket_square and available, ticket_square but not available and not ticket_square.

I could have used composition to implement this functionality.

3) Strive for loose coupling between objects that interact

   Ticket_square and Special_square have similar implementations. So, they have tight coupling between them. If we want to change anything for ticket_square by changing the Square class, it would change things for special_square too

4) Program to an interface not implementation

   The Square class is a concrete class which has different implementations like ticket_square and special_square.

   The square will be bought and sold depending on if the boolean ticket is TRUE or FALSE

5) Classes should be open for extension and closed for modification

   Keeping classes open for extension but closed for modification is important.

   For example, when we run tests on our code, we don't want to modify what is already working but rather extend it by adding code depending on what test we want to run.

   Since the static variables implemented in the State class and even the ArrayList containing instances of the Player class are declared static which means they can be accessed from anywhere and modified from any class.

   If implemented this principle, we would not have this issue.

6) Depend on abstraction, do not depend on concrete classes

   In my code I have used concrete classes which limits our ability to re-use it.

   Using abstract classes, it is possible to adapt your code to work with any number of classes; as long as they implement that common interface.

OBSERVATION OF CODE WRT DECORATOR OR OBSERVER PATTERN

1) Decorator Pattern Implementation

We know that inheritance is static while composition is dynamic.

Has A relation is an Aggregation relationship
Eg: Pizza interface has toppings so Toppings class will be an aggregation to pizza

I have used a Square class which represents all types of squares on the board.
In my code, I have added variables and booleans to represent different squares and it gets rather messy.
Instead what I could have done was use a Decorator Design Pattern where I have a Square interface which Has-A abstract decorator class(an Aggregation Relationship) which helps assign different properties to different squares.
So, first I would create a default square which would have some basic initial values to the variables and booleans.
I would pass an object of this class to the Decorator class which would have getMethods which return the modified values of any new square that extends the Decorator class.
For example, I have 2 types of squares, ticket_square and special_square. To initialize a ticket_square I made the boolean called 'ticket' = TRUE while I made it FALSE for special_square.

2) Observer Pattern Implementation

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

In the code, we could have used the Observer pattern. Even though we have multiple players running their own independent threads, I should have implemented a Moderator thread on which the player threads run and where the Game State output would be coded but for the current project it was sufficient to

have the Game State within the try block of the overriding run() method of the Board class.
However, there is data sharing between the classes as they are not independent.