Watopoly

Project Breakdown:

- Interface for classes: Game; InputManager; Graphics; GameState; Player; Square; Building; Academics; Gym; Residence; NonProperty; CollectOSAP; CoopFee; DCtims; GooseNesting; GoToTims; NeedlesHall; SLC; Tuition; Vec2;
- Game class controls the main game logic and helps managing each game objects (Player, Square, etc) interact with each other
- Player class defines every player's behavior
- Graphics class establishes the text display, it stores every character will be shown on screen and updates every game loop
- InputManager class helps receive inputs from command line
- Square class and all its subclasses are defined to describe each square on the gameboard and they should interact with Player at proper time with the help of Game

- Timeline:
    - Finish implementation for Graphics; InputManager; (h44ding) Aug 8
    - Start implementation for Game; (h44ding) Aug 8
    - Finish implementation for Square; Building; NonProperty; (s43pan) Aug 8
    - Finish implementation for Academics; Gym; Residence; (x732chen) Aug 8
    - Finish implementation for CollectOSAP; CoopFee; DCtims; (h44ding) Aug 9
    - Finish implementation for GooseNesting; GoToTims; (s43pan) Aug 9
    - Finish implementation for NeedlesHall; SLC; Tuition; (x732chen) Aug 9
    - Be able to render the whole gameboard with all squares. Aug 9
    - Finish implementation for Player (h44ding) and establish the interaction between Player and squares. (s43pan, x732chen) Aug 10
    - Save and Load feature; (peter) Aug 10
    - Finish implementation for Game; (h44ding) Aug 11
    - Adjust implementation for all classes, and discuss and restructure new things created during the implementation; (h44ding, s43pan, x732chen) Aug 11
    - Start editing final design document (h44ding, s43pan, x732chen) Aug 11
    - Debug all possible problems and memory leaks (x732chen) Aug 12
    - Add extra features, improve game aesthetics and tidy up. Aug 13
    - Finish editing final design document; updated UML; Finish project. Aug 14

Q&A:

1) After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?

Observer Pattern would not be a fairly good and essential pattern to use when implementing a gameboard. There are three possible schemes. If we consider Player as the Subject and Square as the Observer, then only the Player's owned properties are added into its observers since it is unnecessary to make all squares as the observers of the Player. For instance, we do not need every square to know the position of the Player in each turn since we could just locate the square that the Player lands on and trigger any following actions.

If the Square is the Subject and the Player is the Observer, then similarly only the owner of a square would be the observer and it is unnecessary to add multiple players to one square. Plus in most cases, the Square only needs to apply effect on the Player who lands on it in the current turn, so storing multiple observers for the Square would be unnecessary.

Indeed, in the case of bankruptcy, making the Square as the observer as the Player would allow us to deal with all the assets of the Player (including Auction and transferring ownership) with one call of function. However, we could also loop over every Square to locate the Player owned properties in order to deal with the bankruptcy which is easy to do.

Regardless of game objects, if we use Observer Pattern at Graphics(textDisplay), then the Graphics would be the Observer, and every game objects that requires rendering needs to have a 'if' case in Graphics class. That is, every time we create a new game object class, we need to change Graphics(observer)'s notify, which is not good for cohesion. It is better treating Graphics as a server class that deals with what should be rendered and how to render, and every game objects as client class that uses Graphics as an API to access the output(screen display).

2) Suppose that we wanted to model SLC and NeedlesHall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?

Factory method would work, but it is not necessary to use the factory method. The factory method provides an interface for object creation, and the subclasses SLC and NeedlesHall decide which object (i.e. movement direction) to create, so it is modeled closely to Chance and Community Chest Cards. For example, according to different probability of movements, we can use factory method to create the movement randomly so that it is closer to the Chance and Community Chest cards. However, it is not necessary to use factory methods because all possible movements can be dealt with by simply changing the position of Player and do not have to become objects which occupies more memory resources. Thus we plan not to use it.

3) Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

Decorator Pattern can be a good pattern to use when implementing improvements. The reasons are that there are two different improvements, but they share the same properties which is the cost. It is good that we can have different decorators according to the improvement of different faculty buildings. So that when players need improvements, we can simply call the function in decorator instead of manually modify the state of the current academic building.

However, since the cost and number of improvements is fixed, using a decorator pattern seems too much work. We can simply have two fields in the academic building that stores the cost and the number of improvements since improvements are limited up to 5, which would also work and is easier to understand.

Since the number of the improvements is fixed, we decide not to use the decorator design pattern, but we might change our minds later.