Petar Vico (pvico), Jeffrey Gu (j92gu)

Watopoly - Waterloo Monopoly

Major Components

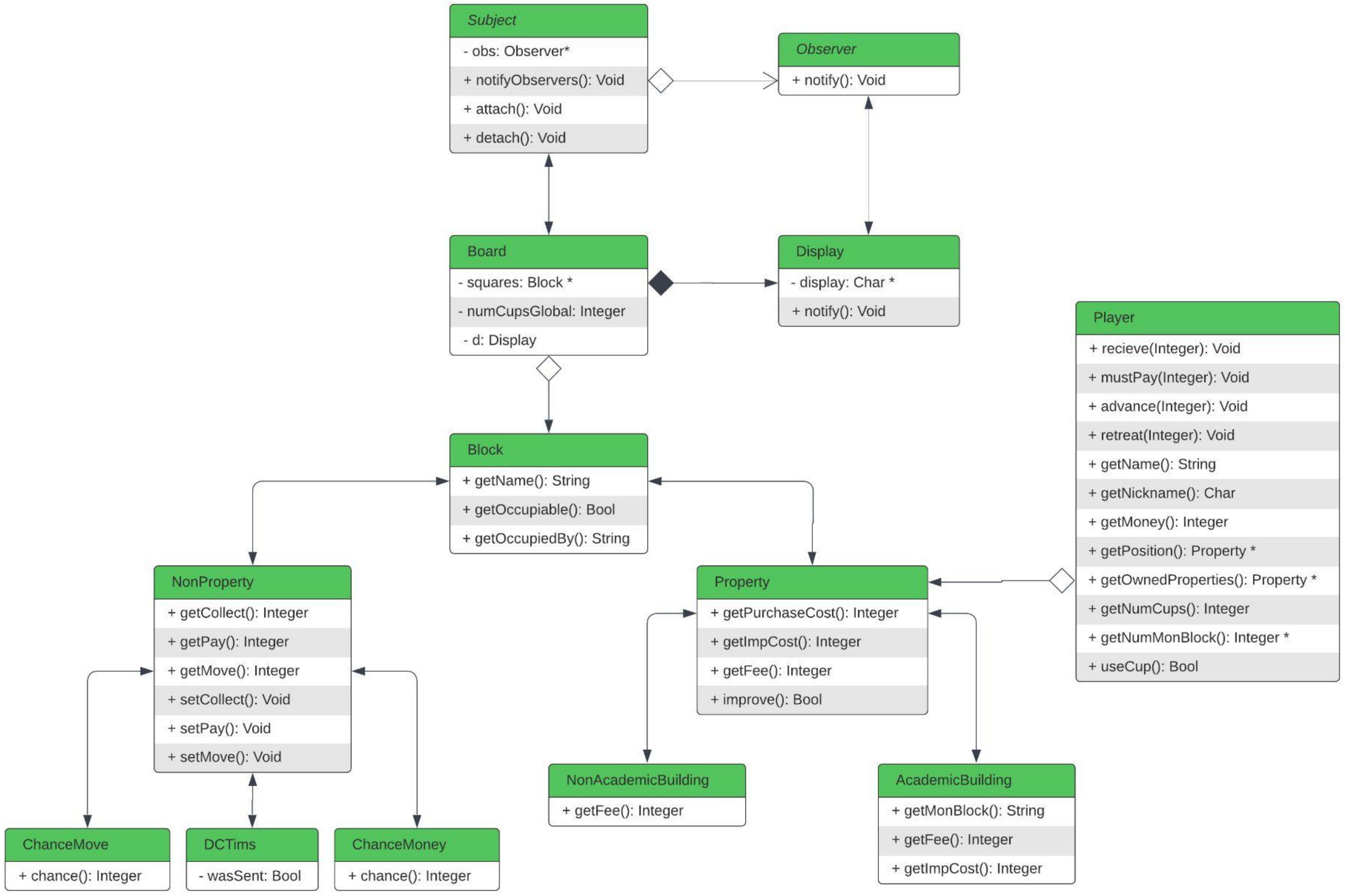
1. **Board:** We assign the Board class to be observed by the Display class which will deal with displaying the board on stdout once notified of a play/action. The Board class holds a vector of Blocks which have their own respected class.
2. **Buildings:** This will be the subclass of the Block class that we named Property. These buildings are referred to as the Property class in our UML.
    a. **Academic Buildings:** a subclass of Property class and it will deal with tuitions that are needed to be paid as well as improvements and monopolies.
    b. **Residences/Gyms:** Also a subclass of Property class where they will be handled by the Player class which will keep track of the number of residences owned.
3. **Non-Property Squares:** Represented as a subclass of Block class. In our implementation, we think this subclass to either collect money, retrieve a players owed money, or move the player.
    a. **Collect OSAP:** Non-Property with a collection of $200, and pay = move = 0.
    b. **DC Tims Line:** Dealt with in main.cc.
    c. **Go To Tims:** Non-Property that moves Player to DC Tims. Sets wasSent to True.
    d. **Goose Nesting:** Non-Property where collect = pay = move = 0. Does nothing.
    e. **Tuition:** Non-Property where pay = 300 or 10% of savings.
    f. **Coop Fee:** Non-Property where pay = 150.
    g. **SLC:** Uses a subclass of Non-Property that was named ChanceMove that moves the player based on chance.
    h. **Needles Hall:** Like SLC but with ChanceMoney class instead, dependent on probabilities.
    i. **Roll Up The Rim Cup:** 1/100 probability dealt with in SLC and Needles hall. Player object fields are updated as well as the global number of cups (cannot exceed 4).
    j. **Players:** Player class containing many fields, including a vector of Properties owned, a Property pointer for the Property that the Player's currently on, as well as a monopoly tracker.
4. **Command Interpreter:** Dealt with in main.cc. We will consider adding a command interpreter class, possibly multiple classes.

Roll / Next / Trade / Mortgage / Unmortage / Bankrupt/ Assets / All / Save: In main.cc.
Improve: Handled in Player and Property class. Command is handled in main.cc.

5. **Gameplay:** Mostly handled in main.cc. Some more implementation in respective classes.
6. **Ending The Game:** Handled in main.cc. May create an additional class to determine the end of the game by analyzing player data.
7. **Command Line Options:** Handled in main.cc.
8. **Random Number Generation:** Use a deviation of the provided shuffle.cc.

Watopoly UML

Petar Vico, Vico Petar | March 30, 2023



Questions/Answers

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

   We believe that the Observer Pattern would be a good pattern to use when implementing a gameboard, because the text display needs to be updated whenever something happens on the board (a player moved, a trade has happened, an improvement is purchased, etc). Since in a game of Monopoly, the state of the game changes frequently as players make plays, we want a simple and efficient method for updating the display of the board.

   In our case, Board is the subject and Display is the observer. Whenever an event occurs on Board, such as a trade or a playermovement, Board notifies Display of the change; Display would proceed to update itself and output the current game state.

   Implementing the Observer Pattern here allows for a very decoupled design. Both Board and Display can be modified without needing to change the other. This is good because in the case we need to enhance our program by adding new features onto the board, we do not need to worry about changing Display at the same time.

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

   We can use the Observer Pattern to model SLC and Needles Hall more closely to Chance and Community Chest cards. Since Player is only dependent on Property and not actually dependent on NonProperty (since they cannot own them), we can make Player an observer of a Card object.

   The idea is that whenever a Player lands on Chance and Community Chest cards, the Card object is changed and the Player will be notified of the change and move/pay. In our implementation of Watopoly, we can create a Card field that starts with a blank card and will be updated randomly every time a player steps on SLC or Needles Hall; Card would then notify the Player of its change, causing the Player to act accordingly in response.

   With this implementation, the actions caused by landing on SLC and Needles Hall is separated from them and are instead managed by Board, making the code cleaner and easier to maintain.

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

   We believe that Decorator Pattern may not be the best pattern to use when implementing improvements. Though Decorator Pattern is very useful for adding behaviour to objects of the same class dynamically, that feature is not particularly useful when implementing Imporvements. Every Improvement have the same structure, with a set series of purchase cost, improvement cost, and tuition/rent at each improvement level. This can be easily implemented using a dynamic array (vector) instead of Decorator Pattern.

However, we do think that Decorator Pattern is a good pattern when implementing the non-property squares, because each non-property block has a unique behaviour that’s different from others, whether it’s move you back 3 squares, make you pay $200, or give you $50. The Decorator Pattern allows us to add (and potentially remove) some of the behaviours as we need, making the implementation of the non-property blocks easier to implement.

Schedule

*** All the dates below are 11:59 p.m. of that day. ***

1. **April 1st**:
   Jeff: Finish writing block.cc/.h
2. **April 2nd**:
   Jeff: Finish writing property.cc/.h
3. **April 3rd**:
   Petar: Finish writing board.cc/.h
   Jeff: Finish writing nonproperty.cc/.h
4. **April 4th**:
   Petar: Finish writing display.cc/.h
   Jeff: Finish writing player.cc/.h
5. **April 5th**:
   Together: main.cc & Makefile
   We will work on a prototype, then edit/test/debug.
6. **April 6th**:
   Together: Testing, getting the program to compile properly → function properly.
   Revisit and make refinements to the all the files.
7. **April 7th**:
   Petar: Update UML.pdf as needed, add all the changes that we've made to the original plan, including the trouble we've faced, our thought process in making the edits, etc.
   Jeff: Write README.md, include program structure and user instructions (maybe updates).
8. *Consider enhancements…*