CS246 A5 Part 2: Final Design Document

Watopoly

Clifford Xing, Yiyang (Taku) Gu, Jessica Yin

Overview

The overall structure of our project follows the object-oriented programming paradigm, employing encapsulation, abstraction, inheritance, and polymorphism to create the Waterloo version of Monopoly. All components of the game are represented by classes: WatopolyGame, which contains the gameplay; Board, which contains the visual representation and player move functions; Player, which is the parent class of ActualPlayer and Bank, with each player being an object of ActualPlayer; Square, which is the abstract base class of all squares; BaseBuilding, Gym, and Res, which are the property subclasses of Square; NPS, which is a subclass of Square and the abstract base class of subclasses representing each individual non-property square; SLC and NH, which are subclasses of Square running the SLC and Needles Hall probability events; and Monopoly, which keeps track of the academic building monopolies.

Design

Board

We created a canvas of the board with empty characters, made 2 draw board functions which can override the board, and also a draw logo function that draws the logo at the upper left corner. We used ordered_map to map each index (1 to 40 inclusive) to the appropriate position they should be drawn on canvas. The 2 draw board functions take in different parameters. One takes in the position as well as the information needed to be drawn, the other one takes in an extra parameter of the player vector so player symbols can be drawn on the square, and calls the first draw square function to draw the basic square.

Squares – Factory Method Design Pattern

In order to create a vector containing all the squares on the board as objects, we created an abstract base class, Square, and subclasses for every type of square: BaseBuilding (academic buildings), Gym, Res, SLC, NH, and NPS (Non-Property Squares), where NPS has subclasses for every type of NPS. When initializing the vector of Square objects, the subclasses decide which object to create based on the name of the square, which implements the Factory Method design pattern. A switch statement is used, creating a new object of one of Square's subclasses according to the name passed in as a parameter. Using the Factory Method eliminates the issue of creating objects without specifying their concrete classes, which is why we chose to use it for the initialization of the squares vector.

Academic Buildings

The Academic Buildings are implemented as objects of the BaseBuilding class, which is a subclass of the Square abstract base class. BaseBuilding, along with all other subclasses of

Square, inherit all members from Square so that all of its public methods, such as setOwner, can be used by Square and all types of property squares. All data for the buildings, including its name, monopoly block, purchase price, improvement price, improvement level, and tuition prices, are stored as private members, with getter and setter functions as public methods, which override the pure virtual methods contained in Square. When a player lands on an academic building square, the "event" method is called, which then prompts the player to either pay rent if the square is owned, or choose to purchase the property if unowned.

Improvements
The tuition with improvements is stored as a vector in BaseBuildings, building_tuition, while the improvement level is stored as an integer variable, which is incremented or decremented when a player purchases an improvement. When a player needs to pay tuition, the tuition price is set by obtaining the integer at the index given by the improvement level in the building_tuition vector.

Mortgaging
The choice to mortgage a property is given to a player as one of the possible commands on each turn, as well as anytime the player needs to pay money, in case they do not have enough funds. The mortgage and unmortgage functions are public methods of the ActualPlayer class, which performs all monetary and property transactions.

Residences and Gyms
Residences and Gyms are implemented as subclasses of Square, with the event method allowing a player to purchase the residence or gym if unowned, or making the player pay tuition (rent) if owned.

Non-Property Squares
NPS (Non-property squares) is a subclass of Square, the abstract base class for all NPS subclasses, and uses inheritance and function overriding. Each NPS is an individual subclass of the NPS base class, with the event method overridden for the specific event that occurs on that square.

DC Tims Line
The DC Tims Line square is one of the subclasses of NPS. When a player lands on the DCT square, the event method checks if they are a) 'passing by' or b) 'in line' and proceeds with either a) nothing or b) telling the player how they can leave the line. Event checks if doubles were rolled, which would automatically allow them to leave the line, and if not, the option to pay $50 or use a Roll Up the Rim Cup is given.

SLC, Needles Hall, and Roll Up the Rim Cups

The SLC and Needles Hall (NH) classes are subclasses of Square and the randomized event is implemented using the random number generator given in shuffle.cc. The 1% chance of receiving a Roll Up the Rim Cup is run first, followed by rolling for the specific event using the given probabilities if the cup is not obtained. To keep track of the number of active Roll Up the Rim Cups there are, a static integer variable is defined in Square (timCupsActive) and the event of winning a Roll Up the Rim Cup is only available if the variable is less than 4.

Players

Each player is an object of the ActualPlayer class, which is constructed with the user-inputted name and symbol once the game begins. The ActualPlayer and Bank class inherits all members from the Player class so that public methods, such as pay_money and addProperties, can be used by Player or ActualPlayer objects. All player data is stored as private member variables, such as the current square the player is on, the number of Tims Cups they have, and the most recent numbers rolled, with getter and setter functions as public methods. Actions that players can take are implemented as public methods in the ActualPlayer class as well, including mortgaging and unmortgaging properties, paying money (forced and optional), and calculating total or mortgage worth (how much money a player would have if they mortgaged all properties).

Bankruptcy

The option to declare bankruptcy is only given at any point in which the player owes money. A boolean variable in ActualPlayer indicates whether or not a player has declared bankruptcy, and before every turn in the game, the checkBankruptcy method in Board checks if any player has declared bankruptcy, calling the declareBankruptcy method in Board and passing the player as a parameter if one has.

Auctions

The event of an auction occurring only takes place when a player chooses not to purchase a property. This can be found if the event function for a BaseBuilding, Gym, or Res return '0' instead of '1', signifying that the player has chosen not to purchase the property. When the return type for the event is 0, the board function begins an auction, asking each subsequent player one at a time if they are willing to purchase the property. If all other players reject buying the property, the property returns to the bank.

Game Ending

At the end of every player turn, the number of players who are bankrupt are checked by looping through the player vector and checking if the player's bankrupt variable is set to true. If the number is equal to the total number of players minus one, then the game ends.

Cohesion and Coupling

The implementation of our modules reflects low coupling and high cohesion, which makes for maintainable code that will support the possibility of future changes. Low coupling is achieved by storing a majority of the variables as private members and using getter and setter functions, which are public methods. High cohesion is achieved as all elements within a module are directly related to the functionality that module is supposed to serve. For example, in the ActualPlayer class, all private members are related to player data while all public methods are related to retrieving or changing the player's data, or performing a player function, such as mortgaging a property.

Resilience to Change

Our chosen design supports the possibility of various changes to the program specification by preserving maintainability in our code. To begin, almost all of our functions involving transactions and values take parameter values instead of hard-coded numbers; this means if the prices of the buildings were to change, we would simply just construct the buildings with a different value. With this, all of the functionality of our program would remain the same thanks to the fact that we use getter/setter variables for the values of the buildings. Moreover, our board output is also incredibly adaptable; due to the fact that we print every tile separately, we can actually modify the order of the board squares by simply changing the order we input the squares in one constructor. Moreover, we can actually randomize the board using code inspired from shuffle.cc. With these features in our code, if we wanted to change the price of anything (for example, if we wanted a Needles card to deduct money, we could simply just change the value it passes into our pay function and the program will run fine) we would be able to. Moreover, if we wanted to board order to change, this could be easily done due to our adaptable board output functions.

Questions

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

Although we had initially decided that the Observer Pattern would be a suitable choice for implementing this board game, we eventually chose not to use it in our final design because we found an alternate solution that was more effective and efficient for us. Since the subsequent changes that must occur after an event are generally very different for each type of event, for example a player moving to a non-property square versus a player making a trade, we chose to code all the 'ripple effects' of an object changing state in the "event" method itself. For example, when a player lands on a property square, the player is either given the option to buy the property or must pay rent, both of which lead to prompts asking for input (yes or no to purchasing), the options of mortgage and declaring bankruptcy given, and transactions involving money and/or properties. This process is implemented in the overridden “event” method for each type of building subclass in order to ensure the sequence of actions that follow landing on the square are specific to that subclass. On the other hand, the process that follows landing on an NPS, such as Go To Tims, is distinctly different from that of landing on a property square, thus implementing the specific process in the overridden virtual method of Square, “event,” was the most effective for us.

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?

The additional feature of modeling SLC and Needles Hall more closely to Chance and Community Chest cards was not implemented in our project, thus the proposed design pattern, Iterator, was not employed. However, if we were to implement this feature in the future, we would still elect to use the Iterator Design Pattern to replicate the actual deck of Chance and Community Chest Cards (no duplicated cards until all cards have been drawn and the deck of cards resets when all cards have been played). To simulate the actual deck where a player draws from the top of the stack, we would implement a class representing a stack of cards, which will contain a vector storing all of the possible Needles/SLC card objects. When players draw cards, ie. land on a SLC or Needles Hall square, an Iterator will iterate through the stack of cards to produce a card on each turn. When the end of the deck is reached, ie. the end of the vector, the iterator will be set to the start again to simulate drawing cards in the same order once the deck has been fully played.

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

Though we originally believed that the Decorator Pattern would be a good pattern to use when implementing Improvements, we chose not to use this strategy as we found that it would be unnecessary and defeat the overarching purpose of using the design pattern. Since the new improvement simply overrides the old improvement instead of cumulatively incrementing the cost of tuition, there is no need to continuously ‘wrap’ the object with improvement decorators. Alternatively, we chose to implement the improvements by creating a vector of integers, building_tuition, for each building, that stores the levels of tuition, and an integer variable, improvement_level, which will be used as the index.

4. What lessons did this project teach you about developing software in teams?

Throughout the development process of our Watopoly project, we learned countless significant lessons and new development concepts. To begin, we learned about how utterly crucial it is to develop a strong plan before beginning any development. Throughout the course of crafting the actual program, we encountered countless flaws and holes within our initial plan, resulting in countless hours of debugging and replanning the approach to our game. Although creating a strong plan is important in both individual and collaborative programming, we learned that it is even more important during development in teams; all members of the team must be on the same page and continually working towards the same goal, meaning that the plan is incredibly significant. Moreover, we also learned about how important communication is in collaborative programming; to enhance our teamwork, we regularly did in person meetings so that explaining concepts and linking our code would be much more effective. Without continual reminders and status updates, we believe that our project would have taken much longer to do; through tenets such as teamwork, leadership, and continual communication and updates, we were able to stay on the same page at all times whilst developing this software in teams. Another important lesson we learned whilst developing in teams is how to utilize tools such as Github and VSCode to our advantage when developing alongside each other. By sharing our code in one Github repo as well as utilizing VsCode's liveshare feature during our in person meetings, we were able to swiftly program collaboratively. Finally, the most important lesson that we learned was to ask for help when we faced an obstacle; instead of troubleshooting or debugging alone, we found it much more helpful and efficient to seek others for help, guidance, or simply just another perspective. Throughout the development process, we found that we often were able to troubleshoot or navigate our dead ends much more effectively when we consulted each other and collaboratively discussed our intentions and issues.

5. What would you have done differently if you had the chance to start over?

As discussed above, one of the first things that we would do differently if we were to start over was to utilize more time and effort into creating our plan. Whilst crafting our project, we found that our plan had several flaws and issues that we did not think about; a clear example of this scenario was that we required more classes than we initially planned, which meant that we had to allocate more time, ultimately delaying our development schedule. If we were to thoroughly plan out our project and delegate our tasks effectively from the start, we believe that our project would have run much more smoothly, allowing for more time in the end to debug and add new features. Moreover, if we were to start over, we would also allocate more time in the end for debugging and testing; our project was finished the day before the due date, however, we still had to test for memory leaks and bugs in the end with less time than we wanted. If we prioritized testing and debugging from the start, we may have been able to have more confidence in our program.