CS 246 Final Project Design Document

Boya Zhang, Jason Zhang, Helen Huang

Introduction

All three members of the group have a soft spot for the classic game Monopoly, having played it many times over and over again. As immigrants, playing the Canadian version of Monopoly introduced members to the diverse cultural offerings and vast geography of Canada, and has become a fond childhood memory for us. Therefore, for the final project, we were more than ecstatic to be able to implement the Watopoly game. Reading through the game specifications, we had many discussions in order to start planning out an initial implementation approach. The group separated the game into its main aspects, and thought about the base classes required and the design patterns that utilized them.

Overview

Our program consists of 4 main classes: the Player, the Tile, the Display, and the Game.

The Player class represents a player in a game and contains information such as the player's current position on the board, net worth, properties owned (especially the number of Gyms and Residences), their game piece, and relevant data for the DC Tims Line. Each Player object is also responsible for rolling the dice, much as it is in real life, and notifying Display to “move” the appropriate number of tiles.

The Tile Class represents a tile on the game board. It includes the name of the tile, as well as an index to represent its position on the board. It acts as a parent class, allowing each of its children to override its methods to determine what specialized action the players should take when they land on the tile.

Specifically, its children are split into Ownable and Unownable Classes:

- The Ownable class represents all ownable properties, so stores information such as its owner, purchase cost, rent, value, and whether or not it has been mortgaged.
    - It has children such as AcademicBuildings, Residences, and Gyms.
- The Unownable class represents special tiles that modify the game, and cannot be owned by a player.
    - It has children such as SLC, Goose Nesting, Collect OSAP, DC Tims Line, Go to Tims, Tuition, Coop Fee, and Needles Hall.

The Game represents Watopoly and handles gameplay, such as adding players, rolling dice, trading or mortgaging properties, or buying and selling improvements. The class stores all the players, the game board, keeps track of whose turn it is, and initiates property auctions.

Our project also has an additional Display class, which displays a visual representation of the state of the board as a 2D array of characters. Similar to the example provided in the

assignment specifications, it has 40 tiles that are traversed in clockwise order. Each tile is constructed with a width of 8 characters and height of 5. All players are represented by a character on the printed board, and this is updated every time they move, as the Player is a subject that notifies the observing Display.

Interactions with the players are handled via a user interface in a Controller class. It receives input from the player and calls the appropriate functions in the Game class based on the input received. The Controller class acts as an intermediary between the user interface and the game logic (Game class), ensuring that the user input is processed correctly and the game state is updated accordingly.

For specific game functionalities, we also implemented an Auction class and an OweMoney class to handle bankruptcy.

Auction is responsible for auctioning off a property to all the players. It keeps track of the highest bid, the current highest bidder, and the participants. It also provides methods to add participants, place bids, withdraw participants, and end the auction.

OweMoney is used to represent debts owed by one player to another player (or the Bank) in the game. If a player is unable to pay off their debts with cash, they must first traverse through their properties and attempt to auction them off. If this fails, then they are bankrupt and removed from the game.

Design

In our program, there is high cohesion within each module. For example, in the Player class, all the member functions and variables relate to the state and behavior of a single player in the game. Similarly, in the AcademicBuilding class, all the member functions and variables relate to the state and behavior of a single academic building in the game. This high cohesion within each module makes it easier to reason about the behavior of each module and to modify it if necessary.

Moreover, there is generally low coupling between the modules. For example, the Game class acts as the central coordinator for the game logic, but it does not depend heavily on other modules such as Player or Tile. Similarly, the Display class acts as the visual for the game, but it only relies on function calls to communicate with other modules. This loose coupling between the modules makes it easier to modify or extend the program without affecting other parts of the code.

To implement the game user interface, we decided to follow a Model-View-Controller (MVC) design pattern to separate the application into three main components: the Model, the View, and the Controller. In this project, the Game class serves as the model, as it manages the game state and implements the game rules. The Display class serves as the view, as it handles the rendering of the game state and displays it to the user. The Controller class acts as the controller, as it receives commands from the user and updates the Game and Display classes accordingly. By separating the concerns of the application into these three distinct components, the MVC pattern promotes modular and maintainable code. Additionally, it allows for greater flexibility in the development process, as changes to one component can be made without affecting the others.

Additionally, our code uses encapsulation to hide the internal details of the classes from the outside world. For example, the Game class encapsulates the state of the board and provides methods for updating and accessing that information, and the Controller provides a public API to start and play the game. We also used iterators to traverse the board, allowing us to visit each tile without exposing the underlying implementation details. This helps to maintain the integrity of the code and prevent unintended modifications from external sources.

The Template method is used when implementing tiles. All tiles have a fixed structure, with methods that are overridden depending what specialized action the players should take when they land on the tile.

Moreover, to ensure that no invalid input is accepted, we use exceptions. For example, in our AcademicBuilding class, if a player tries to mortgage a property that has improvements or tries to buy an improvement without first possessing a monopoly, it will throw an exception. This makes it clear to the caller that an error occurred and allows for more sophisticated error handling.

Another technique used in the project is to ensure that game requirements are met before allowing certain actions to occur. For example, in our Player::rollDice() method, the method first checks if the player is in the DCTimsLine. If the player is, it checks if they have already rolled the dice three times. If they have, the player must pay a fine before being allowed to roll again. This ensures that the game rules are followed.

Resilience to Change

Our design approach prioritizes modularity, which allows us to be resilient to changes in the program specification. Specifically, we have designed our system with separations of concerns between different modules, with functions to allow for communication between them. This means that we can modify or replace individual modules as needed without having to redesign the entire system. For example, we used the Model-View-Controller (MVC) design pattern, which separates the application logic (Game) and the user interface and control flow (Controller).

Additionally, we have implemented the Observer design pattern, which allows different parts of the system to observe and react to changes in other parts. For example, our Display class observes Player and AcademicBuilding classes and updates the printed board accordingly. It separates the display logic from the actual gameplay logic. This means that if the game logic changes, such as if new rules are added, the display code does not need to be modified. Moreover, the Display class constructor also takes in two parameters (width and height) so that the printed board can be resized and automatically reindexes each tile with minimal changes to the source code. The code maps tile indices to a specific location on the printed board. This allows for the possibility of rearranging tiles by changing the values in the map, without having to modify anything else. For example, if we want to swap the positions of two tiles on the board, we only need to swap their indices in the map and update their corresponding locations in the map, without having to change the printing logic in the Display class.

Moreover, our Game class maintains a vector of Tile objects to represent the game board. Adding a new Tile to the board only requires creating a new Tile object and adding it to the vector. This is because vectors are dynamically resizable containers that can easily add or remove elements without having to move other elements around in memory, which simplifies the implementation and makes it more efficient. Our game logic is also equipped to handle any number of players, as long as the board is large enough.

Answers to Questions

1. Would the Observer Pattern be a good pattern to use when implementing a game board? Why or why not?

The observer pattern is intended to create a dependency between subjects and observers, such that observers are notified when the state of the subject object changes.

Thus, the observer pattern can be used for the game display, where the display observes the status of all the players and tiles. The rationale behind this is that the display represents the overall state and should be notified to update when events occur within the game, such as players moving forward, going bankrupt, or property improvements.

As subjects, the players and properties send notifications to the display when relevant changes occur, and so that it can update its state and execute actions accordingly. For instance, when a player lands on a property, the player (subject) can notify the display (observer) about the event, so that the display then knows to output the player's character within the tile for the property.

The Observer pattern is not needed between players or tiles, as actions are only executed between one player and one tile. Thus, other players and other tiles do not need to be notified.

DD2 Revised Reflection:

Our stance on using the Observer Pattern remains the same.
In our final project, our Display class prints out a visual display of the board, containing the positions of each player and any improvements on the academic buildings.
This is done by having Player and Academic Building both be subjects, and having them notify the Display whenever a player moves or buys an improvement for an Academic Building.
However, there is a drawback to the method in that one observer must be able to receive notifications from two types of subjects, which must be handled by overloading notify.

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 original behavior of SLC and Needles Hall is enforced by the idea of constant probabilities for the deck. However, modeling them closer to a deck of cards (similar to real-life plays of Monopoly) suggests that we can use the Iterator Pattern instead. The Iterator Pattern provides a way to access the elements of an aggregate object (like a deck of cards) sequentially without exposing its underlying representation. We can add a number of each card according to their probabilities given to a vector or linked list, then traverse with a cardIterator and using methods such as next() every time a player lands on the squares.

Once the iterator reaches the bottom of the stack (the last element in the list), the cards are shuffled (with a separate algorithm) and the iterator is reset to the new first card in the list.

DD2 Revised Reflection:

We still believe that an Iterator design pattern would be appropriate. To expand on this idea, we could have a Card class that stores the information for each card, and a Deck class that represents the collection of cards. The Deck class would define a method that returns an iterator object over the deck, which would implement the Iterator interface with methods like hasNext() and next() that allow the game to iterate over the collection. It could be implemented to return the Card objects in a random order, since in real-life Chance and Community Chest cards are usually drawn randomly.
The advantage of using the Iterator pattern is that it would allow the cards to be traversed in a uniform way, regardless of how they are implemented or stored. It would also allow the implementation of the deck to be hidden, which would simplify the code and make it more maintainable.

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

No, the decorator pattern is not the most optimal pattern to use for implementing Improvements.

The decorator pattern allows the addition of new functionality to an object and extends their behavior, without affecting the underlying structure and thereby objects of the same class. This is most ideal when scaling an object class with a series of new functionalities, represented as subclasses of the original component.

However, since the only additional behavior that improvements add are changes in the tuition price, this can be done in the original class by mapping the integer amount of improvements to the updated costs. Since there's a limit of 5 on the number of improvements, this approach is more efficient to calculate the tuition cost, and easily enforces the limit when updating the integer value. Implementing a decorator pattern just to generate a subclass for the increased costs will lead to unnecessary complexity that we would like to avoid.

In addition, there are many different buildings, each with variations in tuition fees between improvements. There's no apparent pattern to the increased cost behavior, so rather than implementing 5 decorators based on the improvement level, we would likely need to implement a decorator for each building type and hardcode the behavior the values in order to return the changes in tuition fees.

Hence, we recommend storing data used in the improvements feature in the original Tile class.

DD2 Revised Reflection:

In our implementation, we found it quite simple to implement improvements within our Academic Building class. This is because academic buildings are the only class for which improvements can be purchased. Moreover, there was no need for decorators, as we used an integer variable to store the number of improvements a specific building had. This approach is much more straightforward than creating a whole other class. To calculate the tuition with improvements, we created a vector with hardcoded values, such that the index represents the number of improvements.

Extra Credit Features

In our implementation, we use shared Player and Tile pointers to access and mutate the objects. This greatly helped us avoid memory leaks and dangling pointers, as it automatically managed these dynamically-allocated objects for us. It also simplifies our code by eliminating the need for manual memory management, such as explicitly calling delete on pointers.

Another additional implementation that was completed was our board colour theme. We know how boring black and white games can be, so we wanted to add some “pop” to our game. Players are given the option to select the colour that they want out of 9 total colours which will then be implemented as part of the board game.

This was achieved through the use of ANSI escape codes, which are a series of characters that are used to control the formatting, color, and other output options in text terminals. As such, we were able to apply color to the output text, enhancing the user experience and making the game more visually appealing. Of course, users are still given the option to play the default colour option if they so choose.

Final Questions

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

Through this project, we learned several important lessons about developing software in teams. Firstly, effective communication between team members ensures that we are all on the same page regarding project goals, timelines, and progress. It also helps to avoid misunderstandings and ensures that everyone's contributions are valued. Also, clear organization and delegation of tasks, as well as monitoring progress and adapting plans as needed, help ensure that the project stays on track and meets its goals.

Finally, we also utilized Git version control to manage our code and ensure that everyone was working on the most up-to-date version. We also made use of UML diagrams to visualize the structure and design of our software system. These tools were crucial in helping us work together efficiently and effectively, and allowed us to make sure that our project was well-organized.

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

If given the chance to start over, there are several things that we would have done differently. One major change we would have made is to put more effort into planning and organizing our development process, to ensure that we were working efficiently and effectively. More time than necessary was spent on resolving merge conflicts, linking conflicts, as well as making sure that all of our parameter types matched up.

Another area for improvement would be communication within the team. We would have worked to establish more regular check-ins and status updates to ensure that everyone was on the same page regarding project progress and any issues or roadblocks that arose. Moreover, we would have placed greater importance on making sure our code is well-documented, as there were instances where team members were confused on how to use some functions created by others.

Moreover, we would have put more emphasis on testing throughout the development process. This would have helped us catch issues earlier and avoid re-coding later on. Overall, if given the chance to start over, we would have taken a more structured approach to project planning and management, and placed a greater emphasis on collaboration and testing.

Conclusion

Our team had a fun time implementing this project and learned a lot, despite the long hours and several sleepless nights. We believe that our solid initial plan allowed us to incorporate the essential OOP topics that we learned in class, and ensured high cohesion and low coupling in our implementation.