Watopoly Project Design

Pranav Sachdeva

April 11, 2023

1 Introduction

Our group chose Watopoly as our group Project as we both have had experience playing Monopoly, which we thought would help us save time in understanding the logic of the game. We knew how important our plan needed to be before we started coding, so that we didn't make crucial changes at the end which might lead to us changing the majority of the implementation.

2 Overview

The program revolves around the Controller and Board class, which function as the Model-View-Controller pattern. The Display class acts as the View in this instance, although it is not as integral to the functionality of the program.

Starting from main, the program checks to see if the -testing or -load arguments have been called. If testing is enabled, the testing member of the Controller is simply set to true. If -load is enabled, the load method of the Controller is called with the proceeding argument being taken as the file name. If not, then the loadGame() method is called, and the player is prompted for how many people will be playing, and the names and characters for each player.

The Controller class handles the primary input from the players, which takes place in the play method. The play method is called when the game starts, and the display's render method is called. The controller takes all of the commands in the specification, and also takes a "quit" command in case one wants to terminate the game early. The controller handles any input errors from the player, and skips over any invalid commands. It also checks for whether the command is valid in the current state the game is in, e.g: If a player is insolvent, they can't roll or unmortgage properties. This way, the controller essentially "sanitizes" input so that whatever information that is passed to the Board is valid for processing. The controller utilizes the findPlayer and findLocation methods from Board to retrieve the pointers that correspond to the names given by the player when inputting commands like "trade" or "mortgage". The controller also keeps track of miscellaneous information about players and their turns, like whether they are in the DC Tims line and how many doubles they have rolled in a row. The controller will print some helpful statements about the game's state, like whether a player is in the DC Tims line or if they are insolvent.

The controller also handles changing to the next player for their turn, and keeps track of this with a turnFinished flag. At the end of each turn, if turnFinished is true, and the player isn't insolvent, then it is the next players turn.

Insolvency itself is something not explicitly listed in the specification, but something that we thought made sense to highlight and give a name as a state that a player can be in. When a player is insolvent, they are limited in their options for their turn, and their turn may not finish if they are insolvent. The player may trade and sell improvements and unmortgage properties until they are able to pay off their debt, or will otherwise declare bankruptcy.

The Board class acts as the Model for the game, and handles most of the game's logic. The board contains a pointer to the display. The board has a vector of locations that represent the game board, and a vector of players that are currently playing. There are a variety of methods that cannot all be explained, but, importantly, there are corresponding methods for each command that may be entered by the player. These methods are called by the controller when input has been "sanitized", and takes pointers to whatever objects are necessary. The methods in the board handle various errors, and otherwise perform the functions as required.

The board also contains the auction method, which enters a different auction mode that is separate from the controller. Players take turns increasing bids or dropping out until there is one player left. Players may also enter the "all" command to see their assets.

Bankruptcy is handled by the board, and is separated into two cases. When a player is insolvent, they either owe money to the bank (school), or another player, and this is kept track of in the oweTo pointer and the debt field of the Player class. Note that these are nullptr and 0 respectively if a player is not insolvent. If a player owes money to the bank when they declare bankruptcy, the bankruptcy proceeds as specified.

If a player owes money to another player, then that players' assets are transferred to the creditor. Note that there is a case that is not described in the specification, and that is where a player receives a mortgaged property from a bankrupt player, but cannot afford to pay the immediate 10% required upon reception. We decided to make it so the property instead goes on the open market, and is simply unowned again on the game board. We decided on this so that a player is not punished for bankrupting another player by potentially going bankrupt themselves, simply by playing the game as it is intended.

The Location class is an abstract class that has two subclasses, Ownable and Unownable. The Ownable subclass itself has subclasses AcademicBuilding, Gym, and Residence. The Unownable subclass has subclasses for each of the unownable buildings as listed in the specification.

For Ownable buildings, there is a pure virtual getTuition method. The three types of ownable buildings each have different ways of calculating tuition, so it made sense to separate the retrieval method. There are also various getter/setter methods for things specific to ownable buildings like their mortgage status, their owners, their price, etc.

AcademicBuildings in particular can be a part of monopolies, so they are initialized with a monopoly field that tells the Board which monopoly grouping they are a part of. Tuition is calculated from a vector of 6 different tuition prices, each accessed by the number of improvements on the building.

Gyms and Residences make use of the dice rolls and number of other gyms/residences owned by the player, and thus we have that information as fields of the Player class.

Ownable buildings have one shared handleEvent method, as the only thing that happens when one lands on an Ownable building is that they are either presented with the option to purchase the building, or they must pay rent (unless the property is mortgaged.)

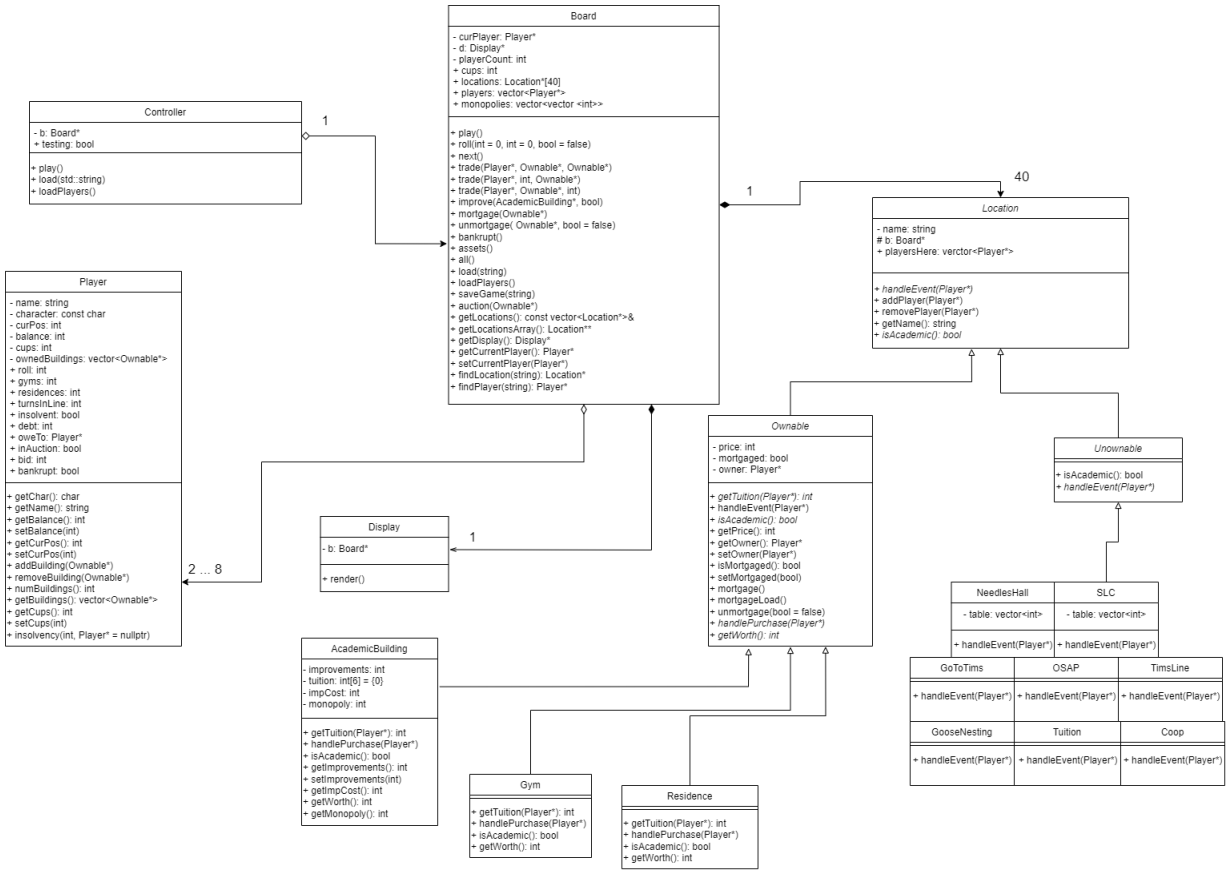
For Unownable buildings, there is a pure virtual handleEvent method, as the different unownable buildings have different events. These range from simply changing a Players balance to randomly generated moves.

We note the SLC and NeedlesHall buildings, which make use of random generation. For SLC, we fill a table of 24 entries, with some entries being repeated based on their probability. For example, moving backwards 3 spaces has a 1/8 probability, which corresponds to a 3/24 probability, and thus it is repeated 3 times in the table. Then we randomly generate a number and take it modulo 24, and then take that position in the table. Thus, we get an entry from the table that corresponds to its probability. We ran tests using this method and found that the distribution of probabilities do indeed match up with the specification. The NeedlesHall handleEvent method works very similar to this.

Note that for both of these methods, there is a 1/100 chance of the player receiving a Tims cup instead, and thus we randomly generate a number and take it modulo 100, and if it is equal to 0 we give the player a Tims cup.

When players go bankrupt, they are removed from the vector of players, their properties and assets are redistributed, and then their memory is deleted. When there are only 2 players left and one goes bankrupt, the other is declared the winner, and the game terminates.

3 Updated UML



4 Design

The program makes use of a modified Observer pattern. We didn't see the need for an abstract Observer or Subject class, as the display was the only observer, and the board was the only subject. So, when there are changes made to the model of the game, which was represented in the Board class, the display was notified by use of the render method.

From a software engineering perspective, we analyzed the coupling and cohesion of our program multiple times throughout the development process. We found that at some points the cohesion of our board class was not as high as it could have been, as some methods like player insolvency and event handling were better left to their respective classes. In its final state, we feel that our program has relatively low coupling, and high enough cohesion that it can be reasoned with.

The methods for our board are strictly for game logic, and dont operate only on specific parts of the game board like separate players and locations. Whenever a board method is called, it normally manipulates the data of multiple objects on the board, or retrieves data that is important in the context of the entire game itself, not just specific instances within the game logic. The same can be said for the other classes in our program, like the display and controller classes, which operate purely for their intended purposes.

We feel that the coupling of our program, while satisfactory, is not at the same level as our cohesion. In particular, the issue with secondary input remained on our minds throughout the development of the program. While ultimately we feel that handling secondary input in the board methods made more sense for our requirements, we recognize that this leads to higher coupling than desired. We had thought of workarounds like a "bus" system where requests for input were taken, and then delivered to the controller class for the controller to handle, no matter what the state of the game was, but we felt that this introduced much more abstraction and complexity to a game that we felt should be relatively simple. All in all, we are proud of the design of our game, and we feel that we made good choices when balancing simplicity vs. complexity in the context of the requirements of the specification and

the time constraints of the assignment.

5 Resilience to Change

We designed our program with its resilience to change in mind. We adapted the Model-View-Controller pattern so that we could separate our program into different modules, with the game logic, the "Model", being the Board class in our case. Using this pattern allows us to change different aspects of the program without other parts being affect. As an example, if some of the rules of Watopoly were to change, it would suffice to change methods in our Board class, so long as the changes to the rules don't involve the direct input from the player(s).

One may also consider the possibility of introducing a graphical interface. Our Controller class handles all input, and is designed for text based input, however this could be substituted (or supplemented) with a graphicalController class, for example. Because our Controller class only ever interacts with our Board class directly, we would simply need graphical equivalents for the text based Controller input for our program to work, e.g: instead of typing in "roll", one could click a button to roll.

This also allows for error handling in input to remain in the controller class. As one may note, there are respective methods in the Board class for each of the commands that a player may call. What is notable is that the parameters for these methods are not strings, but actual Player/Location types when needed. This is because the controller class handles inputs, and checks to see whether the inputs are valid. If they are valid, the controller class handles converting the inputted strings into pointers to the relevant Players or Locations, and then passes this information to the Board, which can then handle errors to do with ownership (i.e, a player didn't own a location it wanted to trade). This separates the game's logic entirely into the Board class.

This is not to say that the splitting of our program into the Model-View-Controller pattern is entirely consistent. Early on in the development of our plan for the program, we realized that there would be "secondary" input. An example of secondary input is where a user would input that they would like to roll the dice, they would be moved to an Ownable property, and then be prompted as to whether or not they would like to purchase the property. This secondary input is predicated on the fact that they landed on the Ownable property, and is thus tied to the logic – the "Model" – of the program. We thought of some alternatives, but in the end we decided that letting some input be handled in the Board class of our program was necessary if we didn't want to introduce too much abstraction.

We recognize that this means if there was an overhaul of the game's Controller, it would not be as simple as redesigning the Controller class only – aspects of the Model would also need to be changed. This was an instance where we were in favour of simplicity in the design and programming of our game rather than abstraction in favour of modularity.

Nonetheless, for the "View" part of our program there is ample room for future improvements or changes. The possibility of a graphical display is easily implementable, as simply writing a graphicalDisplay class with its own render method would suffice. The display of our program has relatively low coupling with the controller and model, as the display's render method is the only part of the display class that is invoked by any other part of the program.

While the Model-View-Controller pattern looks at the "bigger picture" of the program, there are more subtle, smaller features in our design that also imply better resilience to change.

Within our controller class, we look for individual commands entered by the user, and so the introduction of another command would simply need its own method to handle its logic within the model, and simply another "if" block to catch said command in the controller.

Within our model and game logic, we made sure to separate the different types of locations into subclasses, with those subclasses then having subclasses of their own (e.g: Location -¿ Unownable -¿ SLC). Thus, the introduction of a different type of location is as simple as creating a new subclass for the new location type, and populating it with its relevant methods. Moreover, if there is the need to modify the behaviour of a certain subclass, (say we want SLC squares to give the player a Chance card), it would only require the modification of the handleEvent method for each subclass.

The handleEvent method itself is another instance of future-proofing. The handleEvent method of a location is called whenever a player lands on said location. Then, the appropriate action is taken depending on what kind of location calls the method. While this same behaviour could have been

achieved with a series of "if"/"else" statements, separating the different actions into callable methods allows for much lower coupling.

6 Answers to Questions

6.1 Observer Pattern for Game Board

The observer pattern makes sense for implementing the game board, although we will modify the standard observer pattern, as there isn't necessarily a need for an abstract observer class. The display will work as an observer of the model, but we anticipate that this is the only instance where we will need one class to be an observer of another. We will notify the display of any changes that the model makes, as these changes will likely result in a visual change that the display needs to account for.

Note that we did not deviate from our previous answer to this question in terms of design. We modified the observer pattern so that there weren't abstract observers or subjects, as it made sense for the display to simply observe the game board. While developing the program, we noticed that updating the display happened at different "layers" so to speak. As the board contained a vector of Locations, which were each divided into subclasses of subclasses, changes to some types of locations warranted the notification of our display, whereas others didn't.

6.2 Chance and Community Chest Cards

A suitable design pattern would be the template method pattern. As the different chance methods and community chest cards will have different events, we need different methods that need to be overriden. We would have a placeholder abstract HandleEvent() method for a Card class, and override it for different types of cards. Some cards will have very similar functions (e.g: a player gains/loses a certain amount of money), which could be separated into its own class. Others may be things that give players free improvements or Tims cups, and would be handled accordingly with the overriden HandleEvent() methods. Ultimately, we did not end up implemented any improvements or extra features to the specification, like making SLC and Needles Hall function more like the chance and community chest cards, so our answer to this question remains similar.

6.3 Decorator Pattern for Improvements

While we previously answered that the decorator pattern would be a good idea for the improvements in Watopoly, we did not end up implementing them with the decorator pattern. This was an instance in the development of the game where we had to think about the tradeoff between simplicity of code and the level of abstraction that we wanted to have. While in the future one may want additional behaviour to occur from the improvements to academic buildings, we felt that there needed to be more existing functionality to the improvements beyond the increase in tuition that would warrant the decorator pattern. The decorator pattern, while very versatile and useful, can be hard to reason with or comprehend at first glance, and we did not want to complicate our program for a feature that felt largely trivial. Implementing improvements simply as fields of the AcademicBuilding class, a subclass of a subclass, felt like the right approach for us.

7 Final Questions

1. When developing a software in a team, we had to split tasks while also sometimes working together on one. While the majority of the time we could split up roles and work independently, we realized that we may overlap on some new functions and members. We worked in a single git repository to make sure each change is easily pulled or pushed by one of us. While setting it up took some time but in the long run, it made it much easier and time-saving to communicate. After each change was pushed we saw how those changes affected our task and whether there was a contradiction. If one occurred we usually just discussed it and came to a conclusion. We also realized how important it was to have a plan of attack and a basic idea of members and methods that we wanted to use. We also made sure that we know what we need to implement first so that we can make our testing easier. Display and movement across the board were the first things we worked on as we knew these were necessary before

moving further. Even when working on a single task we tried to divide it as much as we could and make sure we communicate changes properly. We did meet sometimes to discuss the project when needed, otherwise pushing and pulling with comments was effective. It was also important to take each other's perspectives and ideas as one might come up with a more effective and efficient solution than the other. Keeping backup was also important in case of any unsaved issues.

2. If we had to start over, I think we would try to make a checklist of important implementation requirements before working on the project. While the project guidelines and the watopoly pdf did help us in providing the requirements, we believe keeping our own checklist would have been of big help. This is because many requirements were part of big paragraphs which are easy to miss. When we were working on our code towards the final stages of the project, we would realize some small implementations were missed, which led us to make changes, recompile and submit our code again. This could have been avoided if we had scanned through the documentation thoroughly and made a checklist. We also believe we would have made our program such that more errors could be printed easily so that a wrong input doesn't break our code.