Petar Vico (pvico), Jeffrey Gu (j92gu)

Watopoly - Waterloo Monopoly

Introduction

For this project, we decided to recreate the high strategy game Monopoly, but instead called Watopoly. Our version of monopoly had an interesting twist, we decided to make it University of Waterloo themed.
In our game, academic buildings, gyms, and residences replaced the regular properties in Monopoly. We replaced the Chance and Community Cards with Needles Hall and SLC where you either gain/lose money or move forward/backward based on some probabilities. The never ending line at Tims is what we considered Jail, in which you could be sent to from the GO TO TIMS block or in very low probability on a SLC block, replicating a go to jail card.

Overview

Command Line Interpretation

Firstly, we needed to be able to interpret command line arguments, so we decided that it would be beneficial to create a separate file that holds a WatopolyGame class where we can handle loading in game states.

WatopolyGame Class

Since there are loaded games, we had to make two WatopolyGame constructors. One constructor would start a new game, and another would start a game from a saved game file. Likewise, we passed in a testing boolean that determined the type of rolls that the game would expect. Within this class, we had several command interpretation methods as well as a play() method that allowed our games to run and collect user input.

Game Structure

Now, moving away from command interpretation, is the structure of the components of the board. This is where we employed the Observer pattern. A Board object contained a vector of Blocks and a TextDisplay object. The TextDisplay was an observer of each Block which was the subject in this case. Thus, when a Block changed state, it would notify the TextDisplay which would update accordingly. In the Board constructor, we decided it would be a good idea to import a text file and read in the Block information in a clearer fashion.
This is where we introduced a MVC pattern, where Board as the model, TextDisplay was the view, and WatopolyGame was the controller.

Blocks

The Block object contained several subclasses that were based on the type of Block.

Properties

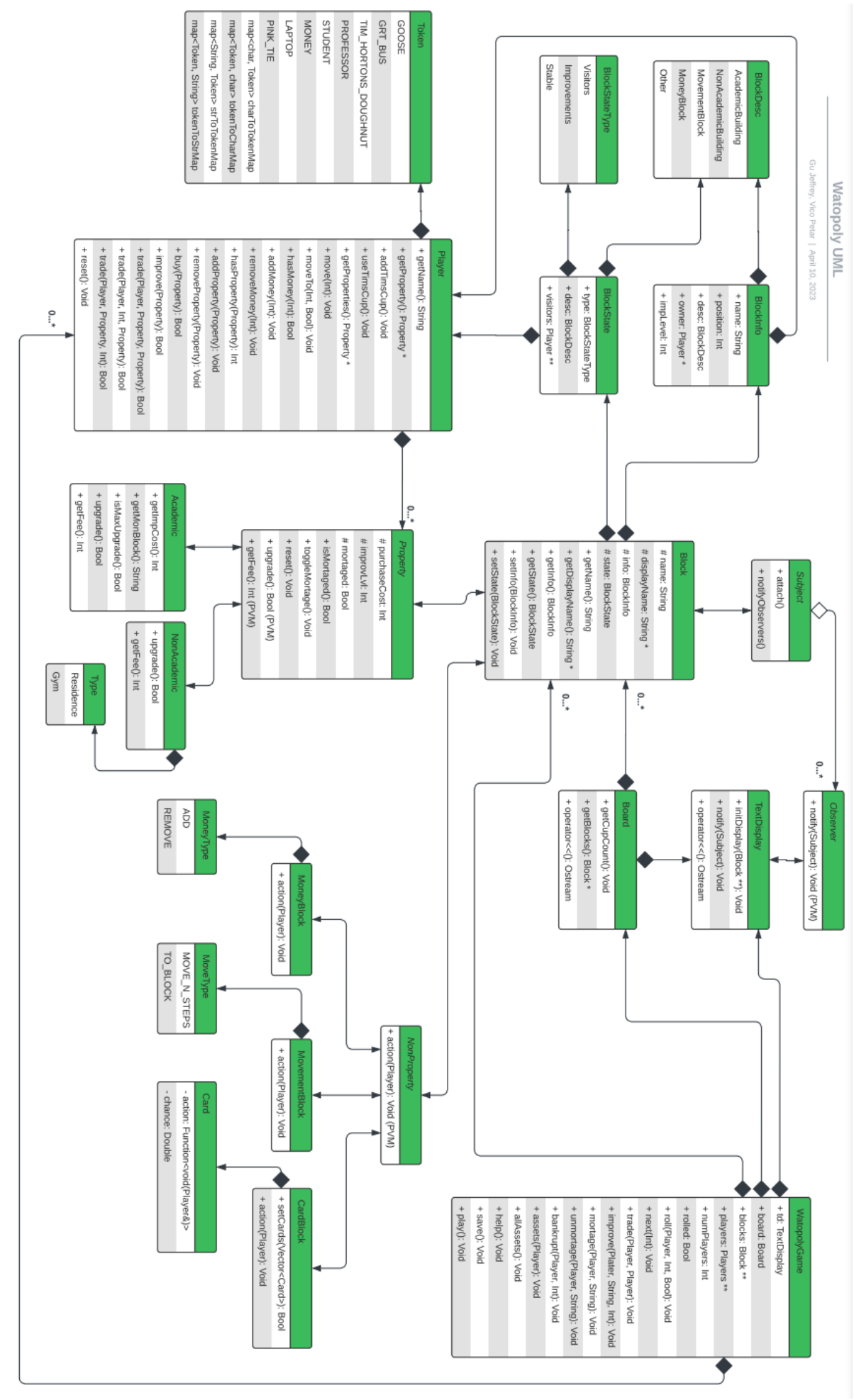
Properties were a subclass of the Block object and were further divided into an Academic and NonAcademic class. Academic classes included AcademicBuildings and their respective

monopolies. NonAcademic classes included Gyms and Residences and their respective fees. Splitting the Property class into Academic and NonAcademic classes made it much easier to handle fees when a player landed on an owned property.

NonProperties

On the other hand, the rest of the blocks were considered NonProperties as the name entains. This is where we had to really think about how we would deal with the actions of some of these NonProperty blocks. We came to the decision that we would make 3 subclasses. The first subclass, named MovementBlock would be for blocks that move a player either a certain amount of steps or to a certain block. Similarly, we created a MoneyBlock subclass which represents a block that gives/receives money from a user. Finally, we create a CardBlock subclass that contained a vector of Cards. This was used for Needles Hall and SLC as we wanted to emulate them as drawing Chance and Community cards.

Updated UML


Watopoly UML
Gu Jeffrey, Vico Petar | April 10, 2023
Token
GOOSE
GRT_BUS
TIM_HORTONS_DOUGHNUT
PROFESSOR
STUDENT
MONEY
LAPTOP
PINK_TIE
map<char, Token> charToTokenMap
map<Token, char> tokenToCharMap
map<String, Token> strToTokenMap
map<Token, String> tokenToStrMap
BlockDesc
AcademicBuilding
NonAcademicBuilding
MovementBlock
MoneyBlock
Other
BlockStateType
Visitors
Improvements
Stable
BlockInfo
+ name: String
+ position: Int
+ desc: BlockDesc
+ owner: Player *
+ impLevel: Int
BlockState
+ type: BlockStateType
+ desc: BlockDesc
+ visitors: Player **
Player
+ getName(): String
+ getProperty(): Property *
+ addTimsCup(): Void
+ useTimsCup(): Void
+ getProperties(): Property *
+ move(Int): Void
+ moveTo(Int, Bool): Void
+ hasMoney(Int): Bool
+ addMoney(Int): Void
+ removeMoney(Int): Void
+ hasProperty(Property): Int
+ addProperty(Property): Void
+ removeProperty(Property): Void
+ buy(Property): Bool
+ improve(Property): Bool
+ trade(Player, Property, Property): Bool
+ trade(Player, Int, Property): Bool
+ trade(Player, Property, Int): Bool
+ reset(): Void
0...*
Subject
+ attach()
+ notifyObservers()
Observer
+ notify(Subject): Void (PVM)
0...*
Block
name: String
displayName: String *
info: BlockInfo
state: BlockState
+ getName(): String
+ getDisplayName(): String *
+ getInfo(): BlockInfo
+ getState(): BlockState
+ setInfo(BlockInfo): Void
+ setState(BlockState): Void
0...*
0...*
Property
purchaseCost: Int
improvLvl: Int
mortaged: Bool
+ isMortaged(): Bool
+ toggleMortage(): Void
+ reset(): Void
+ upgrade(): Bool (PVM)
+ getFee(): Int (PVM)
Academic
+ getImpCost(): Int
+ getMortBlock(): String
+ isMaxUpgrade(): Bool
+ upgrade(): Bool
+ getFee(): Int
NonAcademic
+ upgrade(): Bool
+ getFee(): Int
Type
Residence
Gym
Board
+ getCupCount(): Void
+ getBlocks(): Block *
+ operator<<(): Ostream
TextDisplay
+ initDisplay(Block **): Void
+ notify(Subject): Void
+ operator<<(): Ostream
WatopolyGame
+ td: TextDisplay
+ board: Board
+ blocks: Block **
+ players: Players **
+ numPlayers: Int
+ rolled: Bool
+ roll(Player, Int, Bool): Void
+ next(Int): Void
+ trade(Player, Player): Void
+ improve(Plater, String, Int): Void
+ mortage(Player, String): Void
+ unmortage(Player, String): Void
+ bankrupt(Player, Int): Void
+ assets(Player): Void
+ allAssets(): Void
+ help(): Void
+ save(): Void
+ play(): Void
MoneyType
ADD
REMOVE
MoneyBlock
+ action(Player): Void
MoveType
MOVE_N_STEPS
TO_BLOCK
MovementBlock
+ action(Player): Void
NonProperty
+ action(Player): Void (PVM)
Card
- action: Function<void(Player&)>
- chance: Double
CardBlock
+ setCards(Vector<Card>): Bool
+ action(Player): Void

Resilience to Change

Since we employed many useful design patterns, our code is very resilient to change. One example of this is our implementation of the board dimensions. We made lots of global constants in textdisplay.cc that controlled the dimensions of the board. If we wanted to, updating board dimensions, as well as block dimensions would be easy due to the way that the board and textdisplay were handled.
Additionally, since we employed a MVC (Model View Controller) model, adding additional commands would also be not so challenging to do as they could be added as methods to the WatopolyGame class and handled in the play() method. Furthermore, our handling of Block classification with several layers of inheritance also makes it easy to change some blocks around if need be.
Finally, our implementation of the Community and Chance cards that correlated to Needles Hall and SLC also allows for easy addition of features, as well as change to the respective probabilities. Making different cards is also possible since we have dedicated the CardBlock and Card class to do so.

Answers to Questions

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 to a block (a player stepped on it, 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, Block is the subject and Display is the observer. Whenever an event occurs to a Block, such as a player movement, Block 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 Block 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?**

   Ultimately, we chose the strategy pattern for implementing SLC and Needles Hall. We created a CardBlock class that inherits from NonProperty and we also defined a struct named Card containing a function and a double. CardBlock has a vector of Cards as one of its fields and also stores their cumulative probabilities. Each card contains a function that can be executed on a player and its respective probability.

   The std::function<void(Player&)> action in the Card struct enables different actions to be associated with each card. Consequently, SLC and Needles Hall, which are instances of the CardBlock class, can be closely modeled after Chance and Community Chest cards by assigning various actions to the cards in the CardBlock instance. In our case, action is typically one of four functions: move() and moveTo() for moving the players; add() and remove() for adding and removing money from the players. This implementation allows for more flexibility and modularity when defining card actions, and can be easily changed in the future.

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 Improvements. Every Improvement has the same structure, with a set series of purchase costs, improvement costs, and tuition/rent at each improvement level. This can be easily implemented using a dynamic array (vector) instead of the Decorator Pattern.

Extra Credit Features

As mentionned above, we implemented the Community and Chest cards into our game with the help of the Card and CardBlock classes. Creating cards is very easy as well as assigning their respective probabilities.

Final Questions

1. **What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs.**

   We learned a lot of valuable lessons throughout this project. The most important thing we learned is the importance of clear communication amongst team members and how lack of communication or miscommunication can really slow a team down. Luckily, we kept a frequent communication and had regular meetings/discussions set up so that our project goals and designs are mostly aligned.

   Another key takeaway is the value of version control and code review in facilitating teamwork. We used Git to help us to track changes, manage different branches, and prevent conflicts when merging our work. Code reviews ensured that our code remained consistent in terms of quality and style; if case there are some errors in one person's code, it'll most likely be caught by another team member before it's merged to main.

   Time management was also very important for a large project like this. We broke down the tasks into smaller ones and were eventually able to conquer most of them. The task deadlines really helped us stay on track and progress towards finishing this project without having to cram last minute.

   Overall, we learned a lot from doing this project and hopefully we get to apply what we've learned in our future positions.

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

   If we had the chance to start over, we would have focused on completing the backend first before moving on to the frontend. The frontend is more or less dependent on the backend, and we found that we had to make frequent changes due to the simultaneous development of both parts.

   By completing the backend first, we could have provided a more stable foundation for the frontend development, reducing the need for constant adjustments and ultimately improving the overall efficiency of our work process.

Conclusion

In conclusion, this project has been an enriching and insightful experience for our team. Throughout the development process, we encountered various challenges and learned numerous valuable lessons in both technical skills and teamwork. Our decision to implement certain design patterns allowed us to create a codebase with a good amount of flexibility and maintainability, while tackling the backend and frontend simultaneously highlighted the importance of a well-thought-out development plan.

Despite the obstacles faced, we worked collaboratively to overcome these issues and was able to deliver a functional and well-designed application. Hopefully, as we move forward, we will be taking the lessons learned from this project and applying them in the future. Overall, this project has been a rewarding journey that allowed us to grow both individually and collectively as a team.