CS 246 Final Report

Zhao Gui, Alexandru Camer, Karem Abdul-Samad
University of Waterloo

Table of Content

Overview:

We tried to model our code as closely as possible to a real life monopoly game. This led to our code revolving around three central classes: the Player, Board and Tile classes. The player class should be self explanatory, it keeps track of all player specific data, which includes cash, tims cups… The Tile class is an abstract base class that represents the different tiles on the board. All academic builds, residences, SLC, DC Tims lines, ... inherit from this class, making it easy to take advantage of Polymorphism when applying tile effects on players. Finally, the Board class is simply just a container of tiles, as well as being a producer of board iterators. In our implementation, the board has 40 tiles arranged in the specified order, but we've designed it in a way to allow expansion of the board (adding new tiles) and / or rearrangement of tiles We used the standard template library of an Array for better efficiency compared to a vector, but we initialized the array using a global constant defined in the utility header. What this allows for is very easy expandability. In order to add a tile all you need to do is increment the constant, and then construct that tile in the board's constructor. To provide a clean API for the interface between the client and the logic between these classes, we decided to add a "Master" class: the Game class. The Game is a container for the Player instances and a Board instance.

For the user User Interface (UI), we decided to follow the MVC design pattern described in the course notes. This required us to define abstract Subject and Observer base classes (more on this later). The model in this case is just the Game class, and the View was an observer on the Game class, as well as its members; the Players and Tiles instances. The View had to observe the Player because the players could change locations on the "game board," and the View had to observe the tiles, because there are certain tiles that have visual effects (e.g. Academic Buildings). The Controller handles user command interpretation; it gets commands from the view, interprets the underlying logic of the command, and then calls the respective Game API functions.

Design:

Board Class: As mentioned in the overview, the Board class is essentially a container class for the Tile instances. Note that we just used the standard library vector to store the tiles in the Board class and we tie the Tile instances to the Board class according to RAII principles. The Board class is relatively simple, but it has one unique component that has allowed us to simplify our Watopoly design immeasurably; the Board Iterator. The Board Iterator follows the typical Iterator Design Pattern, except on construction of such iterator, the caller can choose to make the iterator cyclic, meaning that whenever the iterator reaches the end of the Tile vector, it is automatically set to point to the beginning of the vector, i.e. the iterator will never end in this case. This cyclic feature makes the Board Itertator an ideal method of transportation for the players around the board, as this would allow the players to move around the board in a linear

fashion independently, i.e. without having an external class move the player to a new location on the board. So, on construction of the Player class, we will pass in a Board Iterator pointing to the start of the board (Collect OSAP).

Tile: Tiles are a parent class of all the different tiles. It's fields include the name, its index, and a static int that assigns each tile with a different index. As there are a lot of different types of tiles, most of the function in tile is virtual or pure virtual and inherited by the subclasses to override. Each tile has some sort of effect on a player that just lands on it, so an obvious virtual method for the tile class was a Land method. But, we quickly realised that we wanted to implement a landing message when a player lands on any given, and we noticed we could generalize this. So, we decided to make the Land function a template method, where we output a landing message and we instead made a virtual landEffect function to apply landing effects on players. In terms of the Tile descendants, we initially had two subclasses of tile: non-property and property tiles; however, but we noticed that we only need a property tile and have a boolean private field to set if the tile can be owned or not. Under the property class, there are three more subclasses that can be owned: gyms, academic buildings, and residence. Under academic buildings, we focused on the improvement features that come with academic buildings. For the gym, we focused on solving the calculation of how much the person should pay, based on the number of gyms the owner owns. Residence was a similar implementation to the gym. On the other hand, most of the non-Property descendant tiles' functionalities could just be implemented throughout the landEffect method (with a few exceptions), so their implementation was relatively simple. One difficulty we encountered with the tile implementation was figuring out how to deposit OSAP money into a player when the player passed the OSAP tile. After much brainstorming, we decided to add a virtual pass method to tiles; initially blank. So, while a player is moving around the board, it would call this method on the tiles that it passes, allowing a "pass" effect to be applied to players.

Player Class: The Player class stores all player related data. A key component of the game is the rolling functionality. We made the decision to implement this through the player class, since in the real life monopoly game, it is always the player who rolls. We initially wanted to just have a random function that just summed two random integers in range [1,6], but this method proved difficult as some tiles required more information about the role (i.e. DC Tims line needs to know if a player rolled a double), so we just gave a dice "roll" its own struct (a bit more on this in the Testing Mode explanation). The key members of the Player Class are the player's name, board symbol, tims cups, net worth, bankrupt bool, and position. All but the last should be self explanatory. The position member is just an instance of a Board Iterator (discussed above), pointing to the tile the player is currently sitting on. Most of the Player's data is modified through moving the player. To change the player's position, we have three methods:

rollAndMove and two overloads of a move method. The roll and move method rolls a die, displays its result, then increments position (board iterator) by the sum of both die. The two overload methods of the move function take in either an integer or a tile name. The overloaded integer move method just moves the player by that respective amount; this is useful for tiles like SLC, while the string tile name overload just increments the position board iterator until it reaches a tile of the respective name. Most of the game logic is done through these move methods, i.e. the game function just tells the player to move, and it takes care of the rest.

Testing Mode: To implement testing mode, we decided to go with a static "preload" method for the Roll struct. By calling this preload function, a new static instance of the class is created and stored. Then, the next time a Roll is created, we will just construct the roll using that preloaded instance.

Game Class: Game class is an API between the controller and the lower level code. It works as follows: as the controller receives a valid input, it tells the program what to do by calling a function in game, then the game will let the board or player's class know. So game class is just a simple transition API.

Logic Exceptions: We struggled on coming up with a way to implement auctions, player debts, tuition payments, being trapped in DC tims lines and purchase options (for properties), as they required communication with the view for a player decision, though there is no direct link between the controller (who interprets user info) and these functions' respective callers (tiles). We came up with a logic exception class to solve this problem. We implemented these as "exceptions", so whenever a tile needs a player to make a decision, it would store the data needed for the operation execution in an exception (a logic exception), and it would throw it, from which the controller would catch it, change the state of the controller based on the type of the exception and then carry on with runtime with a modified state (limiting certain functionality). This proved effective as it allowed players to make trades through the controller, before having to actually execute the logic function. An example of this would be a Debt Logic exception. Whenever a player owes more money than they have, a tile would through a debt exception, where the tile would load two player pointers of the players involved, and the money owed. The controller would catch this, change its state to state X. State X allows for players to make trades, display assets… But a player can only access the end turn command if the controller is in state Y. So, to make sure a player pays a debt before ending their turn, we make the transition X to state Y only after a student has payed its debt. This is essentially how all the logic exceptions work.

Resilience:

During the design process of our project, we asked ourselves “how can we make sure that if a future software developer wants to add 10 new tiles, no 100 new tiles to the board, that the code modification is as minimal as possible.” From this question, the discussion on how tiles can affect players immediately surfaced. We noted that, most likely, a tile can only affect a player if that player either lands on the tiles, or passes the tile. So, to handle these operations, we have the “landEffect” and “pass” virtual functions in tile, so when implementing new basic tiles, these two functions should be enough for most functionality; if a player passes a tile, it would call the pass function of the new tile on itself, and it a player lands on the new tile, it would call the landEffect function of the function of the tile on itself. Note that all this is seemingless, since for the player to move, the game class has to simply instruct it to do so (without parameters), and everything is handled by the player class.

Though, only having the landEffect and pass method does not provide us with all the freedom of functionality we desire. What happens if we want a tile to take user input, as it requires a decision? To solve this, we have logic exceptions. If a tile requires a user decision, it should load all its decision functionality into a logic exception, and the tile would throw, from which the controller would catch the exception and execute the operation (this is described in more detail above). So, with this, there is almost unlimited functionality for the Tile class in the scope of Watopoly. When adding new tiles, its as simple as coding up the new tiles (this includes overloading Tile’s virtual methods, and implementing logic exceptions as need), adding an instance in the desired location in the board class, and handling the new state logic in the controller.

Note also that some functionalities require a player to store data. This is as simple as adding the new fields to the player class and adding accessors / mutators / incrementers / decrementers.

Through a comprehensive analysis of our code, we have deduced that we have a high level cohesion of the majority of our classes. We created the IO class as a separate class because we knew it would require a very high level of access into the fields of many objects we use. Also note that the Player class is almost entirely self-sufficient with its data storage, holding everything from its name to how many turns it has been waiting in DC tims line, except for the properties it owns. Members that are needed to be passed around, like players, were passed using shared pointers. They couldn’t easily be passed by reference because of the highly varying levels of scope the player would be needed in. Coupling wise, we have medium level to low level of coupling, mainly because most of our classes have accessors to their own fields, thus changing a lot of code in one area shouldn't really affect code in another class. However, in the main.cc, game, and controller, a change in code might affect other classes and need recode, thus we think the overall level of coupling is low-medium level.

In our code, we have a considerable amount of accessors, and although this typically isn't the best practice, it allows us to allow for future implementations to have access to all the data they need.

Missing Implementations:

Although we worked tirelessly on this project, we were a bit too picky on some implementations of our code. This resulted in us not completing all features of the project. We are most just missing these bankruptcy functionality. The problem description states that when a player goes bankrupt, if the player owed money to a person, all their assets should be transferred to to player they owe money to, and if the player owes money to the bank, then the player's properties should be auctioned off. We did not implement this, but we will explain how we would have. This problem could be solved relatively simply by the use of logic exceptions. We note that the auction class by itself already works, so we would repeatedly use it if a player declared bankruptcy while owing money to the bank. It would require a bit more logic, but shouldn't be too much more code. On the other hand, to transfer assets between two players would likely require a brand new logic exception. This logic exception should contain pointers to both players involved and a vector of properties. On throw, the players would be loaded into the exception (but not the properties), then when the exception is caught by the controller, the controller would ask the game to load the expectation with the properties to transfer, and then the expectation can complete the transfer.

Answers to Questions:

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

Answer: The Observer Pattern would indeed be a good pattern to use when implementing the gameboard. Redrawing the gameboard at every step would be computationally inefficient as it would require redrawing all 40 tiles and all players at each step in the game. As a result, by making the gameboard an observer on the players and tiles (subjects), when a player is moved or a building is improved or any other visual updates..., these subjects can just notify the gameboard that a change occurred and the gameboard would simply redraw itself (or perhaps only redraw the changed parts). This offers a much more efficient solution to keeping the display up to date as we only update the display when a display-related change occurs. Therefore, the observer pattern is a good fit for the implementation of a gameboard.

DD2 Reflection:

We used the Observer Design Pattern and it indeed proved very useful for the view implementation.

Question*: Is the Decorator Pattern a good pattern to use when implementing improvements? Why or why not?*

Answer: Decorator Pattern is not a good pattern to use when implementing improvements. The Decorator Pattern is intended to let you add functionality or features to an object at run-time rather than to the class as a whole, however, in this case, improvement costs don't necessarily scale as a well-defined function of the number of improvements, so utilizing the decorator pattern would require a lot of hard coding to determine relationships between the current improvement and the next. Furthermore, since we know there are at most five improvements in total, it would be easier to just provide improvement details at object construction and not worry about it later. In conclusion, it would not make a lot of sense to use a decorator pattern.

DD2 Reflection: As we stated above, we still do not think it was a good pattern, thus we did not use a decorator pattern. Instead, we wrote them as methods in the building class.

Question*: 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?*

Answer: Template method will come in handy when trying to implement needles halls and SLC more closely to Chance and Community Chest cards. A template method defines the steps of an algorithm in an operation, but lets the subclasses redefine certain steps, though not the overall algorithm's structure. At the Template parent class, we would have a template function with potential operations that the chance and community cards could have on a player (move player or withdraw/deposit cash into player). These potential operations would be virtual and the subclasses for each card would implement these potential operations with respect to the card's

effect. Another benefit to the template method is that we can customize the string the player reads upon card pickup without changing how the card works or is implemented.
DD2 Reflection: We did not end up using the template method, instead, we built it inside needles hall and slc as subclasses of non-property class. It is actually much simpler, as we decide the probabilities thru random and apply the outcome that way.

Extra Credit Features
Final Question
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?
Each one of us learnt a lot about developing software in teams.
Zhao: One of the most important lessons I learned is about merging and editing code when three people are working on three different parts of the game. It is important to stick to your own work, as it might cause merge errors when you change others. Furthermore, the functionality of github is new to me. Before I felt like github was just a place to post your code, but it is much more complex and powerful. I realized how an UML can help to make a large program so much easier to write, we spent a good minute on editing and refining our UML graph and it really did help me a lot when writing the code.

Alex: Well, I learned that git is god. Up until now, its been very difficult for me to work on software in teams. This is likely why I never enjoyed Hackathons. Throughout this project, not only have a learned git, but I have become sufficient with it. I also realized the importance of planning ahead. Without a good plan for a team-software project, it is just a complete mess. I also found myself trying to think 4-5 steps ahead when coding, so that when trying to implement new features, most of the infrastructure would already be laid out, paving the way for a simple implementation.

Karem: I learned that teamwork is the biggest factor in success. I learned that spending dozens of hours working on a niche solution that nobody can understand is worse than not solving the problem but writing a good codebase that others can add to to make it work. I learned that knowing what you are trying to solve and searching for prepackaged and elegant solutions before attempting a difficult problem is much better time spent. Finally, I learned how to work on a git tree with many contributors using branches, merge requests, and assigning files to contributors to avoid as many merge conflicts as possible

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

Even though we feel like we spent two full days coding on August the 6th and 7th were sufficient; however, if we started coding a day earlier, we could have implemented more extra credit features. Coding wise, we could have used another way to implement the tile classes, such as template or in another part of the program. The layout of the data structures used could have been better planned out ahead of time. We had to rewrite a lot of code multiple times due to changing the data structures around in order to solve a particular problem.

Conclusion

In conclusion, this project was the first time trying to code and work as a team. We learned a lot throughout this project, such as the importance of building a UML graph before coding, communicating between teammates, and how to build a program that requires a lot of different classes and design patterns and much more. There are definitely ways to improve the results such as spent more time on graphics or find a lower run time way of doing time; however, all of our code worked and we did work as a team and did our best.