RAIInet: Design Document 📝

Jasanjot Gill, Aadil Somani, Tejas Wilkhoo

Introduction

For the CS246 final project, our group decided to implement the game of Raiinet, a stratego-based online board game played between two opposing players, who take on the role of computer hackers.

Overview

Our implementation of RAIInet revolves heavily around our `Board` class, which holds the information of the links and abilities on the game board. Our `Board` class consists of `Cell` objects, specifically a grid-like list of a list of cells, to emulate having 8 rows and 8 columns to work with. Notably, `Board` is a derived class of `Subject`, which is attached to observers for text and graphics, each as their own subclass of the `Observer` class. When the board is constructed, it is initialized with empty spaces on the board, except for the two server ports in the middle of each player's side. Each `Cell` on the board contains information of its coordinates (x and y), the owner of the item on the cell, as well as booleans for knowing if the Cell is a Server Port, Firewall, or (one of our abilities) High Ground. Using set and get methods, we implemented modifying the board to fit the design of the game.

We chose to design a class for a `Player` as well, notably consisting of a count for how many data and virus links have been downloaded and how many abilities remain unused. We considered having the `Player` *own* a `Link`, however since the pieces for each player had a consistent way to identify it (a to h for player 1, A to H for player 2), we decided to implement an unordered map that would help us find the link that corresponds to a given letter. We will see how this was very useful to our implementation.

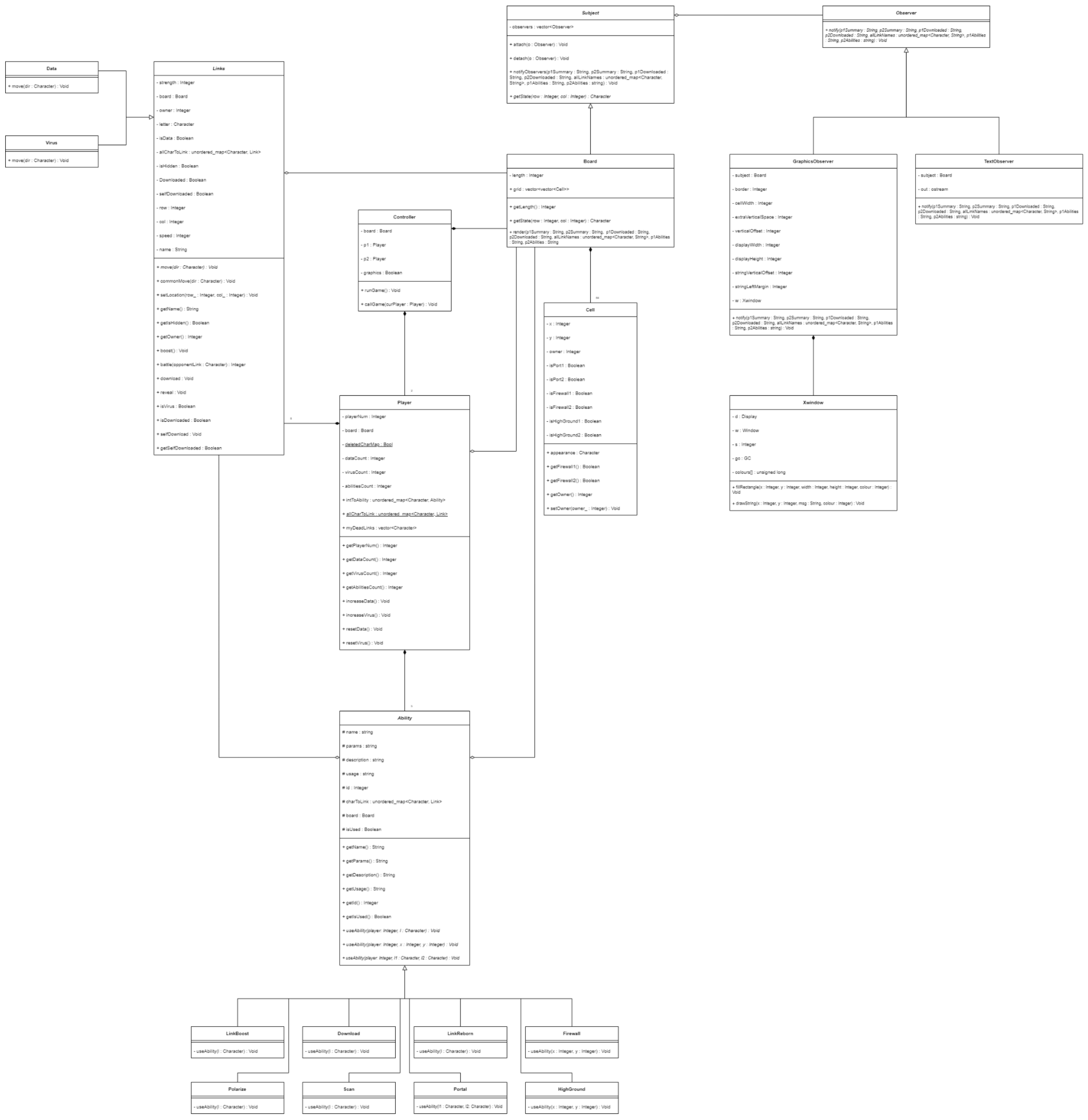
The links in RAIInet are implemented through our `Link` class, a parent class of `Data` and `Virus`. Each link has its own strength, owner and identifier letter, corresponding to the player. Links consist of information regarding if it has been downloaded, if it is hidden and if it has been reborn (see Link Reborn). The most crucial aspect to our RAIInet implementation is the `move(direction)` method on the Link class. This move function had to take in a large amount of conditions, such as moving into an opponent's server port/across their edge, moving into a valid location, interacting with firewalls, high grounds and finally onto an enemy link and battling. One challenge we encountered was ensuring that the spot a link is moving *from* is updated correctly once the link has left. Particularly regarding our firewall and high ground abilities, we had to ensure that these power ups would remain and behave as intended when traversed over. We solved this by adding a method named `movingFromAbility()`, which ensures correct representation of the board. Additional methods include a `download()` method, a `reveal()` method and a `revive()` method.

Our implementation of the abilities belonging to each player was executed using an abstract parent class named `Ability` and its corresponding children, those being `Link Boost`, `Firewall`, `Download`, `Polarize`, `Scan`, `Link Reborn`, `High Ground` & `Portal`. Ability consists of three functions following the same name, `useAbility()`, one to accept a character parameter, one for two integer parameters (position), and one with two character parameters. Depending on the ability, the respective function will be called, with each ability's version of the method being different.

Finally, we implemented a class named `Controller`, which serves as the engine for running our game. `Controller` consists of pointers to the `Board` of the game, both players as well as both observers. It also consists of methods `runGame()` and `callBoard()`. When the game is run, the program enters a while loop that terminates when either player wins or “quit” is called. The loop accepts input from the player regarding commands on the board, such as move, ability, sequence and quit. Additionally, on each loop the game will update the board and tally up downloads to check if the game is over. The `callBoard()` function simply tells the controller when the player’s turn is over and switches to the next player.

Updated UML and Changes

Note that a larger, pdf format of our uml is added in our submitted zip file, titled uml.pdf.


Subject
- observers : vector<Observer>
+ attach(o : Observer) : Void
+ detach(o : Observer) : Void
+ notifyObservers(p1Summary : String, p2Summary : String, p1Downloaded : String, p2Downloaded : String, allLinkNames : unordered_map<Character, String>, p1Abilities : String, p2Abilities : string) : Void
+ getState(row : Integer, col : Integer) : Character
Observer
+ notify(p1Summary : String, p2Summary : String, p1Downloaded : String, p2Downloaded : String, allLinkNames : unordered_map<Character, String>, p1Abilities : String, p2Abilities : string) : Void
Data
+ move(dir : Character) : Void
Virus
+ move(dir : Character) : Void
Links
- strength : Integer
- board : Board
- owner : Integer
- letter : Character
- isData : Boolean
- allCharToLink : unordered_map<Character, Link>
- isHidden : Boolean
- Downloaded : Boolean
- selfDownloaded : Boolean
- row : Integer
- col : Integer
- speed : Integer
- name : String
+ move(dir : Character) : Void
+ commonMove(dir : Character) : Void
+ setLocation(row_ : Integer, col_ : Integer) : Void
+ getName() : String
+ getIsHidden() : Boolean
+ getOwner() : Integer
+ boost() : Void
+ battle(opponentLink : Character) : Integer
+ download : Void
+ reveal : Void
+ isVirus : Boolean
+ isDownloaded : Boolean
+ selfDownload : Void
+ getSelfDownloaded : Boolean
Board
- length : Integer
+ grid : vector<vector<Cell>>
+ getLength() : Integer
+ getState(row : Integer, col : Integer) : Character
+ render(p1Summary : String, p2Summary : String, p1Downloaded : String, p2Downloaded : String, allLinkNames : unordered_map<Character, String>, p1Abilities : String, p2Abilities : String
GraphicsObserver
- subject : Board
- border : Integer
- cellWidth : Integer
- extraVerticalSpace : Integer
- verticalOffset : Integer
- displayWidth : Integer
- displayHeight : Integer
- stringVerticalOffset : Integer
- stringLeftMargin : Integer
- w : Xwindow
+ notify(p1Summary : String, p2Summary : String, p1Downloaded : String, p2Downloaded : String, allLinkNames : unordered_map<Character, String>, p1Abilities : String, p2Abilities : string) : Void
TextObserver
- subject : Board
- out : ostream
+ notify(p1Summary : String, p2Summary : String, p1Downloaded : String, p2Downloaded : String, allLinkNames : unordered_map<Character, String>, p1Abilities : String, p2Abilities : string) : Void
Controller
- board : Board
- p1 : Player
- p2 : Player
- graphics : Boolean
+ runGame() : Void
+ callGame(curPlayer : Player) : Void
Cell
- x : Integer
- y : Integer
- owner : Integer
- isPort1 : Boolean
- isPort2 : Boolean
- isFirewall1 : Boolean
- isFirewall2 : Boolean
- isHighGround1 : Boolean
- isHighGround2 : Boolean
+ appearance : Character
+ getFirewall1() : Boolean
+ getFirewall2() : Boolean
+ getOwner() : Integer
+ setOwner(owner_ : Integer) : Void
Xwindow
- d : Display
- w : Window
- s : Integer
- gc : GC
- colours[] : unsigned long
+ fillRectangle(x : Integer, y : Integer, width : Integer, height : Integer, colour : Integer) : Void
+ drawString(x : Integer, y : Integer, msg : String, colour : Integer) : Void
Player
- playerNum : Integer
- board : Board
- deletedCharMap : Bool
- dataCount : Integer
- virusCount : Integer
- abilitiesCount : Integer
+ intToAbility : unordered_map<Character, Ability>
+ allCharToLink : unordered_map<Character, Link>
+ myDeadLinks : vector<Character>
+ getPlayerNum() : Integer
+ getDataCount() : Integer
+ getVirusCount() : Integer
+ getAbilitiesCount() : Integer
+ increaseData() : Void
+ increaseVirus() : Void
+ resetData() : Void
+ resetVirus() : Void
Ability
# name : string
# params : string
# description : string
# usage : string
# id : Integer
# charToLink : unordered_map<Character, Link>
# board : Board
# isUsed : Boolean
+ getName() : String
+ getParams() : String
+ getDescription() : String
+ getUsage() : String
+ getId() : Integer
+ getIsUsed() : Boolean
+ useAbility(player: Integer, l : Character) : Void
+ useAbility(player: Integer, x : Integer, y : Integer) : Void
+ useAbility(player: Integer, l1 : Character, l2 : Character) : Void
LinkBoost
- useAbility(l : Character) : Void
Download
- useAbility(l : Character) : Void
LinkReborn
- useAbility(l : Character) : Void
Firewall
- useAbility(x : Integer, y : Integer) : Void
Polarize
- useAbility(l : Character) : Void
Scan
- useAbility(l : Character) : Void
Portal
- useAbility(l1 : Character, l2: Character) : Void
HighGround
- useAbility(x : Integer, y : Integer) : Void

Changes from due date 1:
Comparing the UMLs from due date 1 and our current UML, the most important general change is that we were able to clearly understand and define every piece of functionality that each class needed, which is why every class has more methods and attributes.

Another major change is that after due date 1, we decided to employ the MVC architecture along with the observer pattern that we already defined. This was a change we considered while implementing the player and board classes separately, which is when we realized that we would greatly benefit from having a central controller with which the player could communicate with, and also organize the interactions with the program.

Additionally, there is a change with what associations each class has with other classes. As mentioned below in design details, one of our goals was to minimize coupling between classes, which was a difficult task to do considering the interdependence of many of our classes i.e. board and player classes. However, we eliminated any unnecessary associations which were present in our first due date UML, such as the abstract subject class being a parent of a Cell.

Design Details

MVC Architecture

At its core, this program employs the MVC architecture for organization and user interactions. The user is able to communicate directly but only with the controller, which handles and dispatches all the necessary commands based on a user's input, and throws exceptions as required. The model is our `Board` class, which is a class that other classes have a reference to in order to determine the current state of the game i.e. which links are placed where, which abilities are activated, etc. Finally, the view is implemented using the observer design pattern, in order to communicate changes back to the user in a visual manner, both terminal-based and through a graphical window.

Strategy Design Pattern

The eight abilities are methods that exist on the `Player` class. Each ability changes the `Player` class that owns it in some way, as some abilities affect their five links, while others affect the board that they play on. However, leaving all eight of these methods in the `Player` class violates the open-closed principle, as any changes or additions to these abilities would imply opening the `Player` class and changing it. Instead, we employed the strategy design pattern, creating an abstract `Ability` class with which we could implement three overloaded functions of `useAbility()`. Now, the player simply needs to call one of these three `useAbility()` functions, and the rest of the ability implementation is handled apart from the `player` class itself.

Simple Factory Idiom

To facilitate the creation of both `Virus` and `Data` links while also keeping them as separate classes for organization, we used the simple factory idiom. We did not want the construction of viruses and data to be combined, since changing functionality for one but not the other would be difficult to implement without affecting the other class. The simple factory idiom allows us to separate these classes and keep track of their differences in a more organized manner, so we would not have to change a link class if only one of the two classes must change. We decided the full Factory method design pattern was not necessary for this purpose, since we only ever create 4 data and 4 links per player.

Observer Design Pattern

The MVC architecture is coupled with the observer design pattern to effectively reflect visual changes on the board to the user. This was useful for us since we have two different observers, one for X11 graphics, and one based in the terminal. The two observers are subscribed to notifications which the `Board` class sends, triggering a re-render of the two observers. This pub-sub model was effective to show the changes in the board after every move/ability.

Low Coupling and High Cohesion

It is natural that our classes involve a lot of interdependence, such as a `Player` that owns abilities, owns links, and has a reference to the board, while the links have a reference to abilities and the board, etc. Despite this, it was important that we try our best to minimize coupling and maximize cohesion. One way we were able to minimize coupling was to use `unordered_maps` and static maps in different places, such as a map reflecting every link's name and owner, and a map for a player's abilities. By doing so, we were able to only pass in a reference to this map instead of having to expose the entire class to other classes and reduce coupling, while also supporting constant-time lookup (which was a major optimization considering how often players would use abilities and move links). We also maximized cohesion at the same time, such as having a `Cell` class which the `Board` owns 64 of (to create the grid), and not using `Cells` anywhere else.

RAII Principles

After learning about RAII, we felt that it was necessary to adhere to RAII standards wherever possible in an effort to write effective modern C++. By using vectors and smart pointers instead of traditional arrays and `new/delete`, we were able to employ RAII principles to automatically clean up whatever was not in use, such as when we switch back to `std::cin` from reading a file, or when we destroy all components at the end of the game.

Polymorphism

Based on our design architecture and design patterns, incorporating polymorphism in different settings was useful for us to further organize our code, and make some classes abstract. With the observer, we created abstract `Subject` and `Observer` classes which the concrete subject (`Board`) and concrete observer (`TextObserver`, `GraphicsObserver`) would implement. With the strategy design pattern, our eight distinct abilities implement their own version of the `useAbility()` function, which is found in the abstract parent class, `Ability`. Finally, we also used polymorphism with links, as we wanted to separate the differences in functionality between data and viruses.

Single Responsibility and Open-Closed Principles

Adhering to the single responsibility and open-closed principles was at the forefront of our design, as we wanted to be able to create a game which was resilient to change in different aspects (described below). The usage of the various design patterns outlined above allowed us to do this, making sure that classes only have one reason to change and other reasons are abstracted away into other classes, i.e. the `Ability` class or `Link` class.

Exception Handling

With effective exception handling, we were able to account for any errors that might arise when the user enters an invalid command, instead of terminating the entire program. This can include if a player tries to move an opponent's link, uses an ability out of bounds, etc.

Resilience to Change

Our choice of employed design patterns is at the core of how our code is resilient to change, and how we avoid violating the single responsibility and open-closed principles in many different situations.

Extra Abilities
As programmers, if we wanted to include extra possible abilities in the code, we would simply need to add another class which is a child of the abstract `Ability` class. This is due to us employing the Strategy design pattern with the abilities, implying that we would not need to open any other classes to add or remove an ability, i.e. the player class will remain as is.

Different Link Types
Currently, the game and code only supports data and viruses as the two types of links. In the future, if we wanted to add more, or change the functionality of a data or virus separately, we would not be required to change the abstract `Link` class at all. Instead, employing the simple factory idiom with links means that we would only have to modify the lone class we need to, and add/remove different types of links if needed without opening other classes.

Other Observers
The game features both a text and graphical observer. Using the observer design pattern helps us adhere to the open-closed principle, as making changes would involve only changing the single observer class we require. Additionally, adding additional types of observers would be easy and would comply with the single-responsibility principle of other classes, since the `Board`'s notify method will automatically notify all observers of changes, whether we add or remove observers.

Board size
We did not want to fix the size of the board in the code, although the default is the specified 8x8 grid. Instead, we have the length (8) as a private attribute in the Board class, allowing it to be changed if needed.

Answers to Questions

Question: In this project, we ask you to implement a single display that flips between player 1 and player 2 as turns are switched. How would you change your code to instead have two displays, one of which is player 1's view, and the other of which is player 2's?

Our current implementation is to use an observer, from which we derive both the TextDisplay and GraphicsDisplay. This, however, switches through the two different displays and prints different boards for both players. In order to keep track of both displays at the same time without having to implement any extra classes, we would firstly make a change to our TextDisplay and GraphicsDisplay classes to print both boards (player1 and player2) side by side. This would not require any changes to our observer, as a single notification would suffice to reflect a change for board 1 and board 2. Alternatively, we could consider wanting two separate windows for a

GraphicsDisplay, one for each player, in order to also let players know of their own secret pieces without displaying them to the opponent. To do this, we would need to instantiate two different GraphicsDisplay objects for each player, where the only difference for the player 2 window would be to print the board backwards (emulating sitting on the other side of a gameboard). With this implementation, the single notification to update the displays can still update both windows without defining extra classes.

Question: How can you design a general framework that makes adding abilities easy? To demonstrate your thought process for this question, you are required to add in three new abilities to your final game. One of these may be similar to one of the already present abilities, and the other two should introduce some novel, challenging mechanic.

We would define an abstract abilities class (interface) and create derived classes for each ability. By defining each ability as a derived class, it would make it easier for each new ability to already have the required methods and functionality as it is already defined in the parent class. We will be adding three new abilities to our final game.

<u>New Ability 1: Link Reborn</u>
Our first ability is Link Reborn, which allows players to have one of their downloaded links return to the game. The link returns to its corresponding starting position, but if the space is occupied, the player cannot use this ability on the specified link. This ability does not affect how many links the opposing player has downloaded, i.e. if player 1 loses link V3 because player 2 downloaded V3 through a server port, then player 2 still has a virus V3 downloaded even after V3 has been reborn. Any given link can only be revived once.

<u>New Ability 2: High Ground</u>
Our second ability is High Ground. This allows the caster to place a block in the middle 4 rows, and the other player can no longer move onto this block. However, if the player who casted High Ground is on the block they have chosen, then the opposing player may challenge that player on the high ground. When on the high ground, the caster's link gains a +1 strength boost, and then the battle commences. The normal battle rules apply - however if the opposing player wins, the High Ground is removed from the game, if the defending player wins then the High Ground remains.

<u>New Ability 3: Portal</u>
The final ability we will add is called Portal. This allows the user to swap the position of any two of their links on the board with each other.

Question: One could conceivably extend the game of RAIInet to be a four player game by making the board a plus shape (formed by the union of two 10x8 rectangles) and allowing links to escape off the edge belonging to the opponent directly adjacent to them. Upon being eliminated by downloading four viruses, each links and firewall controlled by that player would be removed, and their server ports would become normal squares. What changes could you make in your code to handle this change to the game? Would it be possible to have this mode in addition to the two-player mode with minimal additional work?

Firstly, we would have to implement a change to the board class in use. Ideally, we would create two derived classes for a Board, one containing the setup for a game with two players, and one for four players, which could be toggled with command line arguments. Accordingly, we would change our text and graphics observers to look over the correct board space for both scenarios. In regards to the two additional players, we would have to implement a method to remove a player from the board once they have downloaded four viruses, which would remove their server ports, links, firewalls and high grounds. This method to delete a player would exist on the Board class, since the board class would encapsulate the players, and we would account for the case where only two players are left (since we do not want to delete a player if two players are left, we simply end the game). Finally, we would have to adjust the recording of downloads to allow for scoring in the edges of the opponents adjacent to them. Overall, our most significant changes would be directly to the Board class (creating two derivations of it, rather than our singular two player mode), and its move functions to account for new boundaries and the logistics regarding initializing players in the command line and delegating the inputs.

Extra Credit Features

All of our extra credit features are implemented directly within the main driver code, no need to pass in the `-enablebonus` argument.

Default States: Upon construction of the game, it would be far from ideal to terminate the game if a command line argument is misspelled or is of improper form. For example, if `-ability1` is passed in with an improper ability order i.e. `-ability1 abcdefghijk`, or an invalid link placement file is provided for a player, the game should not abort but instead continue with default values. We have ensured that the game will use default states for abilities and placements if invalid arguments are passed in, so that players can still play the game, while also being notified through the `TextObserver` (terminal) that one or more of their command line arguments failed. Additionally, these arguments are case insensitive, so passing in `-ABILITY1 PDFSL` is the same as `-ability1 pdfsl`, for flexibility.

Help: We added another default command interaction, `help`. As the name suggests, help presents the user with a list of all possible commands and their detailed usages, making the game more accessible and easier to start playing.

RAII: We were able to avoid any instances of new or delete, operating only with `unique_ptr` and `shared_ptr`, with which we were able to adhere to RAII principles.

Final Questions

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

There were many lessons we learned through the process of developing this game. One very important thing we took note of quickly was how useful Git was when working in teams. It made sharing code incredibly efficient with the use of the pull and push commands. The use of branches to hone in on a single feature was incredibly useful as it allowed us to work on code separately without conflicting with each other. It also made it easier to revert back to earlier versions of our program. This way if we merged incorrectly or we made mistakes that did not exist before, we could return to a version that was successful and work from there again.

When working on code individually, it is easy to miss mistakes and inefficiencies in your own code. There are times when you move on too quickly from a piece of code you have written, you may think you have accounted for all possibilities but sometimes it takes a different perspective to spot something you may have overlooked. When we were working on this software there were several instances where one of us looked over a program someone else had written, and we made suggestions and thought about different conditionals that we may have missed. We were constantly making adjustments and small changes along the way because we always had a different viewpoint to bring to the table. It also helped that we were open and honest with each other, and no one shied away from giving or taking criticism.

It can be an incredibly strenuous and troubling process if at any point in time, especially when you come across a problem that you are unfamiliar with and are struggling to take on. We noticed that through the large majority of this process, we were not often stressed or worried. This was because we knew that we had others to go to if we faced any complication or came across a roadblock. It also eases the amount of responsibility everyone has, so it didn't feel like a big project that would take a long time to complete because it was split up evenly amongst each other.

Undergoing a task with a known weakness that is going to hinder you throughout the process can be extremely concerning. An important lesson we learned was the benefits of having individuals with different expertise and strengths. If one of us had a weakness in any area, it was no longer a liability because we had others who were competent in that field who could educate and assist with the task at hand. In most situations, by the end of the process it was no longer a weakness because the rest of us had greatly enhanced their skill in that area for them.

Many of us create a schedule for ourselves around our courses and commitments in order to create a path for us to achieve our goals at a speed which we are accustomed to. Another lesson we learned was that we couldn't always work at the pace that was best for us. A lot of us have different schedules that are not always in line with each other. This led to a lot of compromises in terms of meetings and dedicated time allocated towards working on the project. As differing responsibilities arose, especially nearing finals and the end of the year, it became increasingly difficult to find a time where we were all available.

What would you have done differently if you had the chance to start over?
There were several things regarding the design of the program and the overall process of the completion of this project which we would do differently given the opportunity.

Something we would definitely have done differently is spending more time on the planning process. As can be seen from the changes done to our UML diagram, we implemented far more methods and attributes than we previously did. This was because during the process of coding we achieved a better understanding of our program because writing a piece of code challenges you to think of the most efficient way to complete a task. We could have avoided a lot of these moments with a more rigorous and thorough planning process.

We would have also benefited from a more careful consideration of certain issues that arose while we were coding, in order to alleviate the hassle of the debugging process. For example, we understood the benefits of having smart pointers in our program and so we decided to incorporate them into our program. However, as time went on we recognized our little understanding of where to implement `unique_ptr` and `shared_ptr`. This led to an arduous procedure of figuring out where we wanted multiple pointers to matching resources, and where we wanted just one pointer to an object.

Conclusion

Thanks for playing! -Aadil, Jasanjot, Tejas.