CS246—A5, Raiinet Design Document (Fall 2022)

Milan Sriananthan Pratham Agrawal Brashan Mohanakumar

Introduction

Overview

The way we designed our program was that we created a board class, which contained a 2D vector of Cells. These cells made up the entire board, meaning there were a total of 64 cells spanning the 8x8 board. These cells' main responsibility was to hold a pointer to the object that was on top of that specific cell. This could be either a player's piece, firewall, or server. From this, we are able to create the basic view of the board, which is done in the board::basicSetup() function. This function places all of the pieces in the original positions as well as the servers.

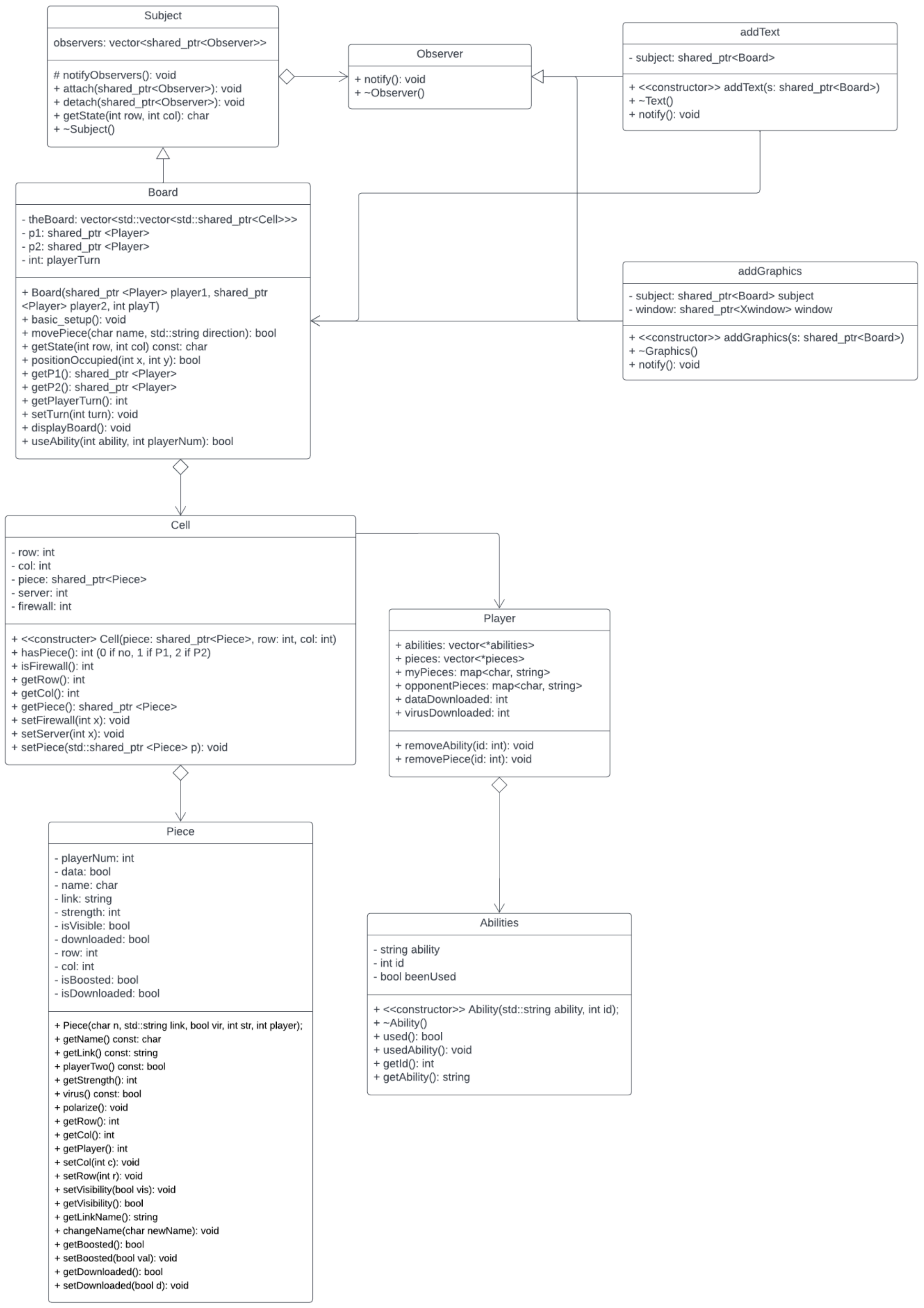
The method board::movePiece(), handles the movement of pieces, by taking in the input of the link to be moved as well as the direction. It then sets the cell the piece was on to null and the new cell location to the piece, it also checks for all invalid cases such as moving on your own server, before making these changes.

Our program also includes a Player class, which is created for both player 1 and player 2. This player class contains a vector of the player's pieces, a vector of their abilities, and the data/virus download count. This allows us to easily access all of a player's pieces and abilities as well as increment both data and virus count when needed.

Our program also includes an ability class that stores a boolean value if the ability has been used or not and the name of the ability and the id. This way, we can assign a vector of ability pointers incorporated in each of our player objects. Thus, when a player wants to use an ability, we cycle through this vector, check if it has been used or not, and use said ability. Interestingly enough, our useAbility() function is incorporated into the board class. The reason being is that in many of our abilities, the ability is able to manipulate the board. Thus, since our board also has pointers to both our player objects (and our player objects have vectors of pointers to their pieces), we are able to reference all things that the abilities need to manipulate. For abilities that must manipulate the board, like Firewall() and Hurricane(), we can tap into the board and manipulate the individual cells. For abilities that must manipulate the pieces, like all the other abilities, we are able to tap into the individual players whose piece the character belongs to and manipulate it, to the ability's desire. When an ability is used, we can easily tap into the pointer to said ability and toggle that it has been used. This will then alter the amount of "usable" abilities which is printed on the board.

Finally, we have both a text and graphical observer. When the board is entered into the program it notifies the observers. For the text display it takes in the board as the subject and retrieves all the information from both player 1 and player 2 and displays them as needed, then it goes ahead and display's the board, by using the getState method, which is responsible for outputting a char, for the different values that are on the cells. The graphical observer works in a similar fashion, but instead uses different color boxes to represent the different piece types.

Updated UML


Subject
observers: vector<shared_ptr<Observer>>
# notifyObservers(): void
+ attach(shared_ptr<Observer>): void
+ detach(shared_ptr<Observer>): void
+ getState(int row, int col): char
+ ~Subject()
Observer
+ notify(): void
+ ~Observer()
addText
- subject: shared_ptr<Board>
+ <<constructor>> addText(s: shared_ptr<Board>)
+ ~Text()
+ notify(): void
Board
- theBoard: vector<std::vector<std::shared_ptr<Cell>>>
- p1: shared_ptr <Player>
- p2: shared_ptr <Player>
- int: playerTurn
+ Board(shared_ptr <Player> player1, shared_ptr <Player> player2, int playT)
+ basic_setup(): void
+ movePiece(char name, std::string direction): bool
+ getState(int row, int col) const: char
+ positionOccupied(int x, int y): bool
+ getP1(): shared_ptr <Player>
+ getP2(): shared_ptr <Player>
+ getPlayerTurn(): int
+ setTurn(int turn): void
+ displayBoard(): void
+ useAbility(int ability, int playerNum): bool
addGraphics
- subject: shared_ptr<Board> subject
- window: shared_ptr<Xwindow> window
+ <<constructor>> addGraphics(s: shared_ptr<Board>)
+ ~Graphics()
+ notify(): void
Cell
- row: int
- col: int
- piece: shared_ptr<Piece>
- server: int
- firewall: int
+ <<constructer> Cell(piece: shared_ptr<Piece>, row: int, col: int)
+ hasPiece(): int (0 if no, 1 if P1, 2 if P2)
+ isFirewall(): int
+ getRow(): int
+ getCol(): int
+ getPiece(): shared_ptr <Piece>
+ setFirewall(int x): void
+ setServer(int x): void
+ setPiece(std::shared_ptr <Piece> p): void
Player
+ abilities: vector<*abilities>
+ pieces: vector<*pieces>
+ myPieces: map<char, string>
+ opponentPieces: map<char, string>
+ dataDownloaded: int
+ virusDownloaded: int
+ removeAbility(id: int): void
+ removePiece(id: int): void
Piece
- playerNum: int
- data: bool
- name: char
- link: string
- strength: int
- isVisible: bool
- downloaded: bool
- row: int
- col: int
- isBoosted: bool
- isDownloaded: bool
+ Piece(char n, std::string link, bool vir, int str, int player);
+ getName() const: char
+ getLink() const: string
+ playerTwo() const: bool
+ getStrength(): int
+ virus() const: bool
+ polarize(): void
+ getRow(): int
+ getCol(): int
+ getPlayer(): int
+ setCol(int c): void
+ setRow(int r): void
+ setVisibility(bool vis): void
+ getVisibility(): bool
+ getLinkName(): string
+ changeName(char newName): void
+ getBoosted(): bool
+ setBoosted(bool val): void
+ getDownloaded(): bool
+ setDownloaded(bool d): void
Abilities
- string ability
- int id
- bool beenUsed
+ <<constructor>> Ability(std::string ability, int id);
+ ~Ability()
+ used(): bool
+ usedAbility(): void
+ getId(): int
+ getAbility(): string

Design

Abstraction: One thing that our group tried our best to prioritize is abstraction. We wanted to hide the details of how our code and how we implemented certain designs. We did this by adding helper files and several class files that both organize our file but also do not reveal too much as to how our code does things. For example, despite our main.cc file being the main file for our codebase, we use a helper.cc file to organize exterior functions that are needed in our main.cc and include them in our main.cc file. This keeps the codebase organized while not revealing the implementations for our designs.
Inheritance: We used inheritance in our design patterns. Specifically our observer pattern included and subject and an observer. Our concrete observers are able to inherit the properties for notify and using the board as a subject, and are able to quickly update. This is very important because when in our text observer and graphic observer (our two concrete observers), we are able to easily print the display using a single .notify() call. This also makes it incredibly easy to add more displays in the future.
Polymorphism: Similar to how we used inheritance in our design patterns, our program also uses polymorphism. We are able to provide a test and graphic display using the observer pattern and notify() function. Despite having completely different types, they are able to be handled under the same interface.
Encapsulation: One thing that is easily noticed about the design of our program and something we really tried to practice is encapsulation. For many of the properties of the different classes used in said program, we tried to keep most if not all of the private. The reason for this is to avoid direct access to these properties from an object. Instead, it was the better practice to use getters and setters to allow for greater encapsulation and more private/protected data.

One of the major challenges in this project was managing memory, we decided to attempt to manage memory without using the two keywords used to manage memory, “new” and “delete”. This means that we had to use smart pointers to dynamically allocate and manage the memory. The main challenge that we faced were converting systems that we were very used to coding and implementing in a new way. An example of this would be the observer pattern which we were used to implementing by allocating memory and passing in pointers. We had to pivot many practices that were ingrained in our programming styles and come up with new innovative solutions with the use of smart pointers. However, by ensuring all our memory was managed by smart pointers we were able to have no memory leaks which is a reward worth the effort.

Resilience to Change

Since abilities are such a big and important part of the game, we wanted to make them easy to add so that in the future, if need be, it would be simple to add new abilities. Thus, we’ve made it so that given the new ability starts with a new character (something different the one that we have so far), then all that’s needed to develop that ability is to add the name of the ability and the corresponding string to the CharToAbility(char c) function and modify the useAbility(int ability, int playerNum) in the board.cc file to how the ability should be implemented.

Low Coupling:
In our design we tried to aim for as low of coupling as possible, meaning there is little or no dependency between the different modules/classes of our program. This can be seen between our different classes. This can be clearly seen in our UML diagram as there is a clear linear flow between the different classes with barely any looping. Our basic reliance structure is outlined as Board → Cell → Piece & Player → Abilities. This allows us to easily create multiple instances of these classes without much change in the rest of our code. For example, if we wanted a larger board of 10x10, then we would just create more instances of the cell class when creating our initial board.

High Cohesion:
We also planned out our classes so that they would all contribute to a common goal of making the game, and that all methods and fields of each class would support a central purpose.
For our board class, all methods were responsible for causing change to the board of the game, this was either by moving a piece or the basic setup methods.
Our cell class supported the common goal of handling everything for their specific cell, which is why we created the methods setPiece(), setServer() and setFirewall() to handle it.
The Piece class contained all responsibilities for the Player's individual pieces. This meant having the fields of where it was a virus, data, the strength, the name of the link, the position it had on the board, the player it belonged to, as well as if it was visible to the opponent. Each of the fields also had their respective getter and setter functions to make them protected from the user.
Our player class is responsible for all information regarding the player, which means their pieces, their abilities and data/virus count. This enables us to quickly receive information about the player without going through other classes.

Answers to Questions

Question 1:
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?

Answer 1:
The way we implemented the display's currently, is that we have an observer, either text or graphics-based which takes in a player object as its subject. Then using the subject's information

on the board's current situation (link locations, abilities, etc) it displays the board for that player, whether this be player one or player two. In this scenario where we are asked to have two displays, we would have two observers open at the same time, either text or graphics-based. And each observer would be fed one of the two players into it. Then, every time one of the players makes a move, the notify method would be called and both observers would be updated with the relative information.

Question 2:

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.

Answer:

Concrete: Concrete is an ability that makes one of your opponents' pieces become immobile for 5 rounds. During said state, they cannot move and the user is forced to move another piece of their choosing.
Umbrella: Umbrella is an ability that acts as a shield against opponent abilities. When you add an umbrella to one of your pieces, they are able to block being downloaded, scanned, concreted, or polarized by the enemy.
Hurricane: This ability acts as a natural disaster that covers the opponent’s half of the board. This becomes problematic because the opponent may not know where they are running into (i.e a firewall) making traversing through the map even more difficult. This will stay on the board for approximately 5 rounds and then go away.

Question 3:

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 link 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?

Answer 3:

It is definitely possible to add a four-player mode in addition to the two-player mode with minimal additional work. In order to implement a four-player game mode to the code, we’d first have to manipulate our player class to add a boolean variable, alive, and a method, isAlive(), to determine if said player is alive or not and if their abilities/pieces/server ports should be on the board. In addition, we’d have to alter the vector of the vector’s size to accommodate the extra board space (increasing the vector size). Finally, we'd have to keep track of the player that is directly adjacent to them in the player class. The reason we have to do this is because a player can only allow links to escape off the edge belonging to the opponent directly adjacent to them. Meaning, if a player were to go off the edge of an opponent, that is not adjacent to them, it would cause undefined behavior.

Extra Credit Features

- Implicit Memory Management: The entire project was done through the use of smart pointers and vectors to manage all memory. We also ensure there were no memory leaks.

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?

Lesson 1: Communication
By Far the most important lesson our team has learned from this group project is the importance of communication. When working individually, communication is never an issue as every single line of code is handled and written by you. However, when working as a team, communication becomes one of the key factors for success. In the initial part of the project, our communication plan wasn't established and there were many issues that were introduced from that such as duplicate work, different implementation style and multiple merge conflicts. Later, we improved on this by having a set communication plan. This consisted of whenever a team member would work on the part of the project, they would inform the others on the work that was going to be done, the files changes/added, how it was going to be implemented and how it changed the rest of the program. This plan helped improve our communication overall, and resulted in a more smooth workflow for the remainder of the project.

Lesson 2: Planning
There was a strong emphasis from DD1 on the thorough planning of the project, from both the UML diagram and the project breakdown document. Both of these were well appreciated in the later portion of this project as they helped extremely in streamlining the workflow. As every group member was aware of the classes and features that needed to be implemented and how it would be done.

2. What would you have done differently if you had the chance to start over?
One thing we would've liked to incorporate is more design patterns. While we did use the observer pattern, we devised a system to use decorators for a more flexible addition of abilities as well as changes in specification. The plan of attack that we had devised was a three classification process of abilities. Class one: an ability that changes how a piece behaves, class two: an ability that changes how a cell behaves, class three: an ability that is a one time use that

affects a game component. Class one abilities would have been implemented with a decorator to the piece class. This would allow us to modify any piece actions conveniently, without having to actually go into the piece class and make any changes. An example of this would be the Boost ability, in which we would override a method called vector<int> move(string direction) which could return the x and y coordinates depending on the fact that the piece is boosted. Similarly, class two would override a method that acts on a piece moving onto it. Class three would just perform the action directly.