RAIlnet Design Document

Gavin Chen (g225chen)
Fangda Dai (fdai)
Kelvin Yeung (k3yeung)

University of Waterloo
Fall 2023 CS 246 Final Project
December 5, 2023

Introduction

RAIInet is a strategy game project that utilizes the object-oriented programming paradigm to create movement and combat across an 8x8 board between 2 players. The steps of formulating this project design include creating a UML diagram, implementing classes and modules while adhering to best standards, and thoroughly testing common and edge cases. This project demonstrates the importance of creating solid object-oriented design with a focus on the value of writing scalable code through various patterns and principles.

Overview

Our project consists of 7 classes that focus on either displaying the Board's visuals or power the game's logic. To begin, our DisplayBoard class creates the text display of our game. This class has a vector of vectors of chars to represent an 8x8 board of links, unoccupied space, server ports, and a public method to print the board to the CLI.

Our DisplayBoard class also dynamically displays graphics. This class is responsible for rendering the game board in a visually appealing format – updating the display in real-time after any commands or actions taken in the game. It uses a class `Window` and graphics libraries, such as `X11/Xlib.h`, to provide an interactive UI along with the text display.

For the logic of our game, a Board class and a Player class are implemented, along with other smaller essential classes that facilitate these implementations, such as a Link class and a Square class. The Board class holds a vector of vectors of Squares, and every Square represents each cell on the board, and the program tracks whether a Square is occupied by a Link, a server port, or nothing. To note, unlike displaying text to the CLI, the Board class can set game states, such as toggling players' win or lose state; moving and downloading links, which updates players' data/virus count; and interacts with the concept of combat in the game, like changing the state of the Board via battling and abilities.

The Player class is an integral component of our game structure. The class manages the collection of Links. Links are the primary way through which players interact within the game board. This Player class tracks each player's virus/data counts and controls the Links' actions on the Board; so, this class essentially encapsulates the player's decision-making processes by having the player execute their best game actions given the public methods.

To uplift creativity, our Ability class holds many implementations of various abilities that affect Links and Board, enhancing the strategic depth of the game. Each ability is uniquely designed to create different scenarios in RAIInet, offering a wide range of tactical options for players. The state of the abilities is also tracked and you can only use one ability per turn.

In summary, our project is a complex interplay of visual and logical components, all working in tandem, to create an engaging gaming experience. The `DisplayBoard` class brings the game's visual elements to players, while the Board, Square, Player, Link, and Ability classes drive the core mechanics and strategy of the game, offering players a dynamic experience.

Design

We planned out our classes and their interactions based on OOP concepts and CS 246 patterns to help structure our program.

Expanding from two clear recognizable design routes: model and view

Our design distinctly separates two worlds in our program: a display functionality via text and graphics using the DisplayBoard class, and a logic functionality of our game using the Board class. By having these two major classes, we can recognize which class you're expanding upon: either expanding on the display of the game or the logic. For example, since our Player class influences the logic of the game, we know that Player will expand from the Board or similar logic modules. Likewise, our Square class influences the UI of the game, which means the DisplayBoard or similar display-like modules are involved here. This contrasting design between model and view made it easier to **scale** our program by promoting easier decisions on organizing the modularity in our program.

Utilizing enumerations

We used 2 enumeration classes called AbilityName and Direction to facilitate our class implementations. The enum AbilityName contains 8 enumerators which represent all of the names of our Abilities. Further, the enum Direction contains 4 enumerators that represent 4 directions ("UP", "DOWN", "LEFT", "RIGHT"). Incorporating enums has improved the readability and maintainability of our implementations throughout the entire time.

Utilizing encapsulation

After acknowledging the concept of player info in RAIInet, we focused on the concept of encapsulation by needing to obscure sensitive information from clients. From this, we've bundled up code as much as possible within classes while minimizing the number of public data members and member functions. For instance, encapsulating the Player class by hiding the class' states, like the number of data and viruses each player has, is imperative to avoid unwanted tampering of data from clients. As a result, being aware of information hiding by limiting scope throughout class implementations raises security and peace of mind.

Resilience to Change

A low coupling, high cohesion environment mitigates the inconvenience of new problems

This is a standard practice that our group did our best to ensure that further changes in our modules wouldn't lead to new inconveniences. Throughout our project, we made sure that our Board and DisplayBoard classes had little reliance on other modules to decrease the amount of which our code has to change in the case of any further module change.

In terms of cohesion, we have multiple classes working together during the program's lifespan that keep track of the program's state and interactions between the players. As mentioned, the logic of our program – the Board class – worked in sync with our text and graphics display class – the DisplayBoard class – while having smaller classes that supported the functionality of larger programs. Following along with the low coupling and high cohesion practice welcomes further development by welcoming new code to **interdependent modules that aren't fragile.**

Adhering to the single responsibility principle promotes the separation of concerns
We aimed to create classes that only have one simple functionality while avoiding overloading each class with irrelevant fields. For instance, our Link class would represent the logic of data or virus in the game, but this wouldn't be a class that would involve the logic of combat. We would only implement methods to the Link class that relate purely to the essence of the class, such as a method that increments the strength of a Link. As a result, when concerns are well-separated, there are opportunities to **maintain and reuse modules for future development** while promoting a simpler program environment for developers.

The effects of documentation on long-term ease of understanding
We strongly believe in writing concise and detailed comments in the places of convoluted code. Adding changes to an existing program requires that existing implementations are fully understood by the programmer, especially when there are many dependencies in a module. In our Square class, we explicitly commented that the class' purpose is to represent each "pixel" on the board, and each pixel comes with many data members that facilitate the implementation of larger classes like the Board class. All of this may not be apparent at first to any programmer, so providing documentation can **improve code readability** during program changes – and this is especially helpful after many months from release.

Answers to Questions

Q1) Display: 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: We write a Board class that takes all information about the game. There is a field "vector<shared_ptr<Player>> players" which stores two shared pointers of two players so that we can access all information of them. We use an integer field "int currentPlayer" to

indicate which player's turn it is, so the DisplayBoard class we wrote can efficiently display the correct view of the current player in texts and graphics based on the information from the Board class.

Q2) Abilities: 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: We can create a class (call it Ability) designated for making adding abilities easy. Within that class, we can define an enumeration class (call it AbilityName) with each ability's name. The Ability class should have the following fields: name(type: AbilityName) and id(type: integer). Outside of the class, we can implement a function (call it useAbility) that will decide which ability to use based on the command given (i.e. the ID and the name of the ability). If we want to add in new abilities, we can simply append the AbilityName enum class with the new ability's name, and add its corresponding functionality inside the useAbility function.

Q3) Winning & Losing: 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, all 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?

Answer: First, we need to randomly assign 8 links to each player if links are not given. Next, we need to add a boolean field to each player to indicate whether the player has lost or not. This way, the program can correctly skip the turn of the lost player. Besides, we need to add extra boolean fields for each link to indicate whether the information of the link is visible to an opponent, since each player has 3 opponents in this case. Moreover, we need to change the winning condition, like one player has downloaded 4 data or 3 players have downloaded 4 viruses. For the display, I will change the size of the board and the display structure to show the board and information of each player properly in the view of the current player. We think it is possible to have this mode in addition to the two-player mode with minimal additional work, but it is hard to achieve.

Extra Credit Features

The entire program has been successfully developed to function without any memory leaks, and this has been achieved without the necessity for explicit memory management. This was accomplished through the utilization of smart pointers, specifically std::shared_ptr, and Standard Template Library (STL) containers, such as std::vector.

Final Questions

Q1) What lessons did this project teach about developing software in teams?

Answer: This project taught us the importance of delegating the right tasks to maximize time efficiency and the quality of each member's work. For instance, we had a double degree member who loved and had experience in report writing while another member was decently experienced in working with low-level graphics. Had we all focused on the same task and progressed to the next same task, each member's value wouldn't have been maximized during this 2-3 week project period. To note, working on the same task was our initial plan, but we realized midway that allocating the right and equal amount of tasks allowed us to develop intrinsic motivation to complete our tasks quickly along with working around our particular strengths that would maximize our group output.

Another lesson we learned was the importance of communicating our conflicts immediately. Though it felt like an obvious group trait, it was easy to passively struggle with a problem independently at first. We realized quickly that when someone didn't voice their concerns about the project, someone else would likely encounter the same problem which led to two members working alone on the same problem. Throughout our code development, we heavily valued speaking up and fostering any communication of ideas to all be on the same page and recognize each other's flaws in thinking.

Q2) What would be done differently if there was a chance to start over?

Answer: There are different work approaches that we would do differently that would have increased our time efficiency while developing RAIInet. We should've done work horizontally over the week instead of vertically; that is, it would've been better to work less on fewer days instead of working a lot on fewer days. For instance, we would commit our entire Mondays, Wednesdays, and Fridays for this project, but meeting up more frequently while not overburdening each day would've allowed our ideas to marinate and avoid diminishing returns on our productivity during previous hectic days.

Another aspect we would've done differently is spending more time on design than being overly focused on implementation. Our approach was to design a workable solution, and implement it immediately since we were eager to see group results fast, then fix up solutions we would encounter. However, time into the planning stage, such as thinking more

about different scenarios during the stage, would have allowed each member to grasp a clearer understanding of the interdependency of our RAIInet modules which would've lessened the number of problems that had arisen from several necessary data members and member functions we missed as we were implementing. Focusing too much on implementation was counterproductive, and having a better design foundation would have saved more time.

Conclusion

Using standard principles and patterns in the object-oriented programming world has facilitated the writing of scalable code in a collaborative setting, such as using the Decorator pattern to implement several more abilities in RAIInet in a time-efficient manner. Developing this complex project not only taught relevant object-oriented standards in C++ but also taught ways to delegate work and communicate our issues and ideas as a team for the first time. This project developed experiences of best practices in designing and implementing large software that is imperative in any development environment.