RAIINET PROJECT DESIGN DOCUMENT

FINAL PROJECT – DD2

Roy Chon, Nicholas Hioe, James Nitsch

CS 246

12/05/2023

Introduction:

Our version of RAIINET was created as the final project for the fall offering of CS246, to showcase our understanding of Object-Oriented Programming and other skills developed throughout this course.

Overview:

Our main classes consist of the game, player, and board classes. The game class is the conductor of our orchestra, connecting our player and board classes and allowing them to work together. Key actions such as initialization and move, go through this class. An instance of a player is created for each player in the game, storing a set of links, abilities, and their scores. We initialize these links by setting their stats, then placing them accordingly in the board class. The board controls all movement logic by storing a grid of cells, in which we can check the contents of a cell for links, server ports, or firewalls, when making moves. The construction of a game class creates both the board and necessary player classes. Any “base” initialization in the game class, branches off to the player and board classes, to correctly construct and initialize their own respective fields, which includes links, abilities, displays, and cells.

The game class stores information on whose turn it currently is, ensuring that the current player can only move links and use abilities that belong to them. Link interactions with other links, or server ports (which are treated as special types of links) are handled in the board class, so this includes any battling, moving into server ports, or moving off the edge. The link objects themselves hold pointers to the player that owns them, to ensure correct identification in these interactions made from the board (such as battling and downloading to mutate player score). Every unique ability inherits from an abstract ability class, allowing us to apply and pass arguments in a uniform matter. The text and graphic display observe every cell, allowing for any

moves or abilities to be quickly accounted for on the board. These smaller classes work within the confines of our three main classes creating, RAIINET.

Changes from DD1:

In the original UML, the abstract class Ability consisted only of one virtual method, but we introduced an additional virtual method, called checkValid, to validate a specific ability applied to a link or cell on the board. This allowed us to check for error handling and make our entire program error safe. We also included a CLParse class, utilized by main. This was responsible for streamlining command line input, for simple parsing, and handle invalid input, such as selecting more than two of the same abilities, or an initial that does not exist. The core structure of our final implementation remained mostly the same, which can be attributed to the extensive time spent planning and scoping out this project as a group.

Object-Oriented Design & Design Patterns:

In our planning phase, we discussed the usage of the observer design pattern, as it has many potential applications in the context of a board-based game. In the end, we decided to implement the observer pattern for cells in the grid (contained by our board) to improve the efficiency of any live updates to the display. Every time a link (virus, data) moves into/out of a cell, observers are notified for those respective cells, where the new state is retrieved, and display is updated to produce the correct output. By using the observer pattern, Board and Cell classes are loosely coupled.

The C++ idiom, RAII, or “Resource Acquisition Is Initialization,” was kept in mind throughout the creation of our project. We utilized smart (unique) pointers to automatically manage or dynamically allocated objects. These were held in our main classes, the game, which owns the players and board, while these held objects such as the displays, links, and abilities, to

express a hierarchy of ownership. This base hierarchy created a strong foundation, allowing us fill in any missing connections with raw pointers by aggregation, and distinguishing memory management from other functionalities.

We emphasized encapsulation, creating all key objects within a game instance. That way, our game class is self-contained and manages all resources allocated. We separated our concerns into our main classes, as well as smaller classes, to ensure modularity, and that certain objects were being altered in a controlled fashion. We also implemented getters and setters to ensure all external objects attempting to access or alter fields, did so uniformly, reducing any risk of invalid values or behaviour.

Resilience to Change:

All existing abilities derive from an abstract ability class. Using this structure, we can override functions of the abstract ability class, but provide necessary changes specifications within the existing functions. For in-game commands, we can easily support new commands by adding additional cases as needed, to the command interpreter. Additional command line arguments can be manipulated via the CLParse class, to avoid cluttering the main function.

We have already implemented the Observer Design Pattern in respect to each cell on the board. This allows us to smoothly integrate new features or modify existing ones, since it allows for lower coupling. Furthermore, we have ensured that our classes have a high degree of cohesion through careful planning and thoughtful consideration. As mentioned previously, the “CLParse” class is a great example of this, keeping our main function tidy. It acts as a cohesive module that parses command line input.

Our organization into modular components and classes made it easier to replace or edit a specific function without affecting the entire system. We have worked hard to ensure that our implementation has minimized coupling. Our modules work together with each other in a simple manner, and we have avoided sharing any unnecessary information between classes. We have accomplished this by not using the friend keyword in our code (with the exception of the output stream and text display), and by keeping functions private wherever possible.

Answers to Questions:

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?

To implement a dual display for both players at the same time, rather than a single flipping display, it would be necessary to introduce dedicated displays for each player. The standard implementation for this project would require a dynamic display, switching the graphics and text display after every turn, showing the perspective of the "current player." To make two displays always present, two text and display classes can be created. The two classes will be responsible for rendering the perspective of the game for their respective player. At every turn, the game must interact with both displays, updating them as more information is revealed to each player. This can make the display more modular, as there is no need to switch between different rendering logic for a single display, as rendering logic is encapsulated within their own respective display classes. The rest of the game logic should remain consistent, as the board and its functions will remain unchanged, and all display logic should be handled in those classes.

While we did not implement a dual display, we did implement board flipping, as an enhancement to our text display, which simulates an actual game being played, where the current player's turn is displayed always at the bottom. We were able to handle this logic completely within the text display class, as we simply iterated through the board in a reverse order. This logic can be simply translated to a dual display by showing both iterations together.

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.

To make adding abilities to the game easy, the general framework must be extensible and modular. An abstract base class for abilities provides a common interface, allowing all abilities to have a relatively consistent design. Existing concrete abilities will be inherited from this base ability class, and they will all have to implement the common methods declared in the interface. At the start of every game, a player has the option to select five ability cards and up to two of each kind of card. We found that it sufficed to have the players store an array/vector of these abilities and utilize the functions of the ability abstract base class, which resulted in minimal effort in adding abilities, and a uniform process. Everything from generating new, modifying existing, and removing old abilities, can be easily manipulated through the ability framework without impacting the rest of the game.

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?

The more modular and flexible the existing code is for a 2-player mode, the easier it will be to adapt to a four-player mode. Some aspects of the game, however, may require more significant changes to adapt to any rule or behaviour changes introduced. Fundamentally, features such as the board dimensions, the number of views, game initialization, and the turn system will need to be adjusted for the number of players. With respect to game logic, link movement may be adjusted to only allow escape off the edge of an adjacent rectangle (to the link's player) and eliminated players must be accounted for by removing their assets. If the code is modular, we can reuse functions and core-logic, which should remain relatively constant despite the game mode change. A good way to handle these additional players (if not already implemented), is attaching an indicator onto certain assets that belong to these players. It is possible that a two-player implementation takes advantage of the binary behaviour of Boolean logic (suppose true represented player 1, false represented player 2), but in the case of four-players, it is not possible. Links or other player-specific assets may require more specific indicators, such as a pointer to the player that owns the asset. This may streamline processes, such as rendering an individual player's view (and what pieces should be revealed). Additionally, leveraging data structures like arrays (where possible) to dynamically accommodate a different number of players, can minimize altering player-related logic, such as the turn system. Having this modular design avoids the limits of a hardcoded approach and increases the flexibility and

scalability of the game. Maximizing this modularity ensures minimal changes to the core game logic when shifting from a 2-player to a 4-player game.

Extra Credit Features:

Board Flipping:

With enhancements enabled, our board flips with each move. It displays such that on a player's turn, their pieces are displayed at the bottom of the screen, and their side of the board is also on the bottom. It appears in the same manner as an online game, or physical game board in real life. In the context of one device, it allows for a pass and play style of game.

Extra Ability:

We implemented four custom abilities instead of the required three abilities. Here are brief descriptions of our custom abilities:

- **Takedown ('T')** – Disables an enemy link so that it can no longer move.
- **Heal ('H')** – Reduces the virus count of the player who uses the ability by 1.
- **Power Augment ('A') -** Increases the strength of a link that the player who calls the ability owns to 4.
- **Conceal ('C') –** Hides the type and strength of all revealed living links of the player who calls the ability.

Conceal and Takedown are abilities with the most game-changing mechanics. Disabling a particular link's movement prevents them from moving for the rest of the game! Using this in tandem with an ability like scan can stop some of the strongest of links in strength, or positionally. Information is your most powerful enemy! Conceal tests the memory of your enemy. Catch them off guard and regroup your links, leaving your opponent second-guessing every next move.

No Dynamic Memory Management:

We don't use any "new" or "delete" statements anywhere in our code. Instead, we used smart pointers. The only times where we use raw pointers in our code are when we need to obtain some information or mutate something, never to express ownership.

Final Questions:

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?

Communication is extremely important. Without proper communication, a project can quickly diverge in different directions which will lead to problems in the future. We chose to communicate with each other every day about what we were working on, and to have regular meetings to ensure that we were on track. We constantly gave updates to minimize overwriting work and repeated effort.

Additionally, using version control software like GitHub properly is essential. It allowed us to work on different parts of our code simultaneously. However, using GitHub was not without its issues. We had to deal with merge conflicts and other problems, which helped us get more proficient with Git.

Planning and documenting a project before starting the implementation is very useful and it will save the group an enormous amount of time later. Since we created a UML diagram and agreed upon an overall design for our classes and how we wanted them to work together, all members of our group were aligned, and our implementation went smoothly as planned.

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

Implementing an “AbilityManager” class would make it easier to create new abilities and apply abilities to specific links and cells on the board. Handling custom functions and arguments will be easier in a separate class. While our implementation was able to effectively add new abilities with minimal effort, from a scalability standpoint, having this additional class can further improve separation. Ultimately, it will manage our list of abilities and activate all the abilities that it contains. We can handle and manage different abilities in a modular manner. Time restrictions prevented us from creating this, but our solution was the best decision for the scope of the project, given the constraints.

Additionally, we would consider the Factory Method Design Pattern regarding when links are randomized. The Factory Method alleviates the randomness when producing links of different types. We would also consider applying the Decorator Design to both cells and links. With respect to cells, other abilities similar to firewall can easily be added without cluttering the fields and functions of a cell. The same logic applies to that of links, which may have more revolutionary ability changes if this project had a greater scale.

As a team, we collaborated very well. In the context of a bigger project, we could potentially simulate a real workplace, utilizing agile methodologies, 2-day sprints and maybe even daily morning stand-ups!

Conclusion:

Overall, we all had a great time with this project. This strengthened our understanding of object-oriented programming, as we were able to apply concepts into a real functional game. It was also a valuable experience for us to work on a group-based coding project, as we had limited experience with multiple programmers on the same codebase. We will all take the things that we learned with us as we move to the workplace. Thank you to all the instructors of CS246! 😊