Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

CS246 - F24 - Project - **RAIInet**

Introduction:

Welcome to the document where we have cumulatively translated our thought process from code into readable english! ~~Hooray!~~ We are Sheehan, Shihan and Ayaan (as you can probably see in the header). Now we chose RAIInet, since we do like cool strategy based games! Let's delve into the intricate details of how we have planned, executed, tested and made this game come to life! We will specifically explaining:

- OOP principles (MVC in particular, SRP, etc) that we have incorporated
- How our UML planning was conducted
- Manner in which we made our code use low coupling and high cohesion to be adaptable to change, yet still robust)
- Various questions highlighted in the RAIInet file
- Extra-credit features that we have implemented (smart pointers, 4-player, multiple display)
- Final Questions highlighted in the Project Guidelines file

Overview:

Timeline of Project:

Below is the true timeline of the project after DD1:

Date	**Plan**
Nov 22	- Complete Game & Player integration with Board & Cell - Start TextObserver & GraphicalObserver
Nov 23	- Complete TextObserver - Start Testing basic abilities and movement
Nov 24	- Complete new Abilities - Start testing movement and abilities
Nov 25	- 4 player functionality - Fixed TextObserver, created Err class to throw exceptions
Nov 26	- Tested game logic - Debugged errors
Nov 27	- Completed GraphicsObserver and multiple displays - Repeated testing on Graphics display
Nov 28	- Decoupled Ability class - Updated graphics UI to look better - Finalized UML and Design document
Nov 29	- Final Checks - Submit
Nov 30 - Dec 1	- Practice Demo - Pray

Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

Key Classes:

Below are the small descriptions of the responsibilities of each of the classes we implemented:

1. Controller • Processes user input and commands. • Manages the game flow by invoking methods on the Game object. • Creates and manages the state of observers • controller.run(): The main game loop that processes commands and updates the game state.	2. Game • Manages the overall game state, including players, the board, and player turns. • Handles the execution of abilities and ensures game rules are enforced. • Notifies observers of state changes.
3. Board • Represents the game board as a grid of Cell objects., providing access and manipulation of individual cells. • Changes size in terms of the number of players playing.	4. Cell • Represents a single cell on the game board. • Stores information about its content, such as links, firewalls, abilities, and special attributes like server ports or warps.
5. Player • Represents a player in the game. • Manages the player's links, abilities, download counts, and elimination status (for four player games).	6. Link • Represents a link (game piece) owned by a player. • Stores its type (Data or Virus), strength, position, and status effects like visibility and trojan.
7. Ability • Represents an ability that a player can use. • Stores the ability type, activation status, and identifier.	8. Err • Provides a struct that has many common error messages to throw any exceptions. • Uses methods as well to create messages custom to ability/link/player
9. TextObserver • Provides a textual representation of the game state, displaying the board, player stats, and messages from the perspective of a specific player (in terms of multiple displays, if not, it is always from the perspective of player 1). • displayAbilities(), displayError(const string &errMsg) and displayGameOver() are called on the current player's observer in multiple-display mode, but always on Player 1's observer in single-display mode.	10. GraphicsObserver • Provides a graphical representation of the game state using the X11 library, displaying the board, links, abilities, and special effects visually. • The methods above for TextObserver for similarly for graphics • Uses std::chrono for time spans to display errors and abilities commands within the graphical interface.
11. Subject • Abstract class of Game • Part of Observer Design Pattern • Attaches & detaches displays	12. Observer • Abstract class of TextObserver & GraphicsObserver • Display itself for the board with virtual method

Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

Structure of the modules:

The project employs Model-View-Controller (MVC) architecture to create a two-player strategy game (with bonus functionality that extends to four players). We have defined clear roles for each module so as to not violate the Single-Responsibility-Principle (SRP):

Model:

- **Core Game Logic:** The `Game` class encapsulates the game state, including the board, players, and the logic for processing moves, abilities, and game progression.
- **Components of the Game class:** Classes like `Player`, `Link`, `Ability`, and `Cell` represent the fundamental elements of the game. Each class is highly cohesive, with methods and attributes that are directly relevant to their functionality.
- **State Management:** The game state is maintained in a shared `Board` instance, which is composed of `Cell` objects representing the grid. Each `Player` instance owns its Link and `Ability` objects

View:

- **TextObserver:** The `TextObserver` class provides a text-based visualization of the game state, including the board and other necessary information for each player.
- **Graphics Display:** The `Graphics` class uses the X11 library for graphical rendering. It visually represents the game board, and other necessary information for each player.
- **Output fully within View:** Each observer (both graphics and text) encapsulates not only displaying the game logic, but also displaying errors and any other possible output.

Controller:

- **Game Flow Control:** The `Controller` class manages the flow of the game by processing player input and connects the Model to the View.
- **Input Handling:** User inputs are parsed and validated by the `Controller` before being passed to the `Game` class for execution.
- **Error Handling:** Comprehensive error messages are defined in the `Err` class and surfaced to the user when invalid inputs or commands are detected.
- **Separate class from `main.cc`:** The `Controller` class handles the game loop while main only incorporates parsing arguments and validating logic for those arguments hence following SRP.
-

Design:

Note: The UML class diagram reflects the actual structure of the project, showing classes, their relationships, and key methods and attributes.

We will explain the structure of the project and how it helped solve much of the design problems we faced, by relating it to the various strategies we employed throughout the entire timeline.

High Cohesion:

In our code, we ensure that each class is dedicated to a single, well-defined responsibility. Specifically:

- Game Class Centralization: The `Game` class handles all changes to the game state. It processes moves, executes abilities, manages turns, and enforcing game rules. By centralizing game state management within the `Game` class, we maintain high cohesion, as all game logic is contained within a single component.
- Player Class Managing Own Links and Abilities: The `Player` class is solely responsible for managing its own links and abilities. It tracks the player's links, the abilities they possess, and their download counts. This ensures that player-specific logic is encapsulated within the `Player` class, maintaining high cohesion by focusing exclusively on player-related data and behaviors.
- Link and Cell Classes with Specific Roles: The `Link` class represents a player's link (game piece), handling attributes like type, strength, position, and status effects. All methods within `Link` pertain to managing

these aspects. Similarly, the `Cell` class represents a single cell on the game board, managing its content and states such as firewalls, warps, and imprisonment. By confining their responsibilities to their specific domains, both `Link` and `Cell` classes exhibit high cohesion.
- **MVC Architecture:** We utilized the Model-View-Controller (MVC) pattern to separate concerns, which inherently promotes cohesion. The Model encapsulates game logic, the View handles presentation, and the Controller manages input. This separation ensures that each component focuses on its specific role without overlapping responsibilities.

High cohesion allowed us to address the challenges of this project effectively. Since each class had only a single purpose that it needed to carry out, we were able to modify code with minimal change to the existing codebase.

Low Coupling:

We designed clear interfaces and minimized dependencies within classes:

- **No Friend Definitions**: We avoided using friend declarations in our classes, hence preventing classes from using internal definitions of other ones.
- **Simple Interfaces for Communication**: Classes communicate through simple parameter/result functions, using basic data types rather than complex structures or arrays. This leads to more simple interfaces.
- **Observer pattern:** Observers are decoupled from the `Game` logic by implementing the `Observer` pattern and subscribing to updates via the `Subject` class, allowing the `Game` to notify these observers without depending on their specific implementations.

Low coupling allowed us to be more flexible when adding or modifying code. The low dependency between modules let us add more features into the game without the necessity to change existing classes. Furthermore, this made debugging and testing code more manageable, as bugs could easily be linked to a single, specific module.

Exception Safety - Strong Guarantee:

Our program ensures that no game state changes occur in case of errors, providing a strong exception safety guarantee. We achieve this by:

- **Throwing Exceptions Prior to State Modification:** We validate all inputs and conditions before any state changes. These exceptions are caught within the `Controller` class, where appropriate error messages from the central `Err` class are displayed.

This not only keeps the state intact in the state of errors, but also helps a lot in debugging and testing (the error messages were extremely helpful).

Resilience to Changes:

Abilities:

The way our project was designed simplifies the process of adding in new abilities with having only minimal changes to add. Only 2 classes need to be changed, the `Ability` class to contain the definition of the new ability, and the `Game` class to handle the logic of the ability. Input handling and error management are automatically managed, with only occasional adjustments needed if the new ability's error scenarios fall outside the scope of existing generalized error handling.

To generalize the ability input validations, for each ability, we'd define a static helper within the `Ability` class, that returns a vector containing the types of the expected parameters. For example, Firewall requires a coordinate, so the helper function would return {"int", "int"}. This vector would determine what type of input to be read from cin, and also be used to validate input reading.

Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

We chose to store our input parameters for each ability within a flexible vector `std::vector<std::any>`, from the STL `std::any` to generalize input capturing. This allowed us to capture input of every type into one vector, regardless of the parameter set needed by an ability. During execution, the `Game` class's ability logic handler would process the vector, validating any general ability conditions, such as whether the ability has already been activated. The parameters in the vector would then be cast back to their initial types using `std::any_cast` to compatibility the function's parameters before delegating it to the function containing the specific logic of the ability being used.

This design decouples the input capturing, handling, validation, from the definition and implementation of the logic of the ability, allowing for a more scalable program. It ensures that the only changes needed when adding a new ability is simply defining it and its expected parameter within the `Ability` Class, and implementing its logic within the `Game` class. `Error` handling may need to be slightly updated, if the existing generalized errors are not appropriate.

Error handling:

We centralized error handling within a structure called `Err`. This allowed for creating general error messages to be reused across the project. For scenarios where a predefined error message was not appropriate, we implemented reusable error functions that accepted parameters to dynamically generate error messages. This allowed us to create reusable error templates that would be defined within its appropriate use case. By organizing concrete and error templates with the struct `Err`, we made it straightforward to manage and scale.

Game logic:

The `Game` class was designed to operate independently, with its logic fully decoupled from other classes. It is the only class responsible for handling the state of the game and would interact with other classes such as Links/Players using interface files to ensure independence. Moreover, we also generalized everything to be applicable to 4 players, which allowed us to think of unique edge cases to prevent and also made it very easy to incorporate multiplayer modes, variable board sizes, etc. We used vectors to store our objects, making it easy to add more players, more abilities, more links, more cells, all by simply updating the respective constructors to increase the length of the array. Since the logic was decoupled from the creation, this makes it easy to create new objects while ensuring they still follow the same existing logic with no issues.

Answers to Questions:

<u>Question 1</u>: In this project, we ask you to implement a single display that maintains the same point of view as turns switch between player 1 and player 2. How would you change your code to instead have two displays, one of which is from player 1's point of view, and the other of which is player 2's point of view? (NO CHANGES)

Basic Idea: In order to implement two displays, we would have each observer be dedicated to a single player. Hence, for a two player game, we would have two text observers and two graphical observers (if graphics is enabled) defined in main, one for each player. We would then attach all these observers to the concrete subject Game.

We would follow these steps as follows:

- Update the concrete observers `Text` and `Graphics` to have a member field that tracks which player they are observing
- Within each observer, update the code to flip the board (if necessary) and consider changes such as the links that are visible according to the player being observed (with a simple if condition checking which player is being observed by that observer).
- Create dedicated observers for each player in the `Controller` class and attach them to the game subject
- Since the game has these observers attached, every time the state changes, each observer's notify method will be called.

Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

<u>Question 2</u>: 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. You are required to actually implement these. (CHANGES)

We discovered that the framework outlined in our initial plan document was not as easy and straightforward as we expected. It required modifying multiple classes, updating input handling, adding custom error messages and exceptions for the specific parameters required by the new ability if needed, and finally creating a new overloaded useAbility() function within the game class if a new parameter set was needed. This approach was very error prone and we'd very frequently run into bugs by forgetting to update each file to add our first custom ability.

In our redesigned approach, we decoupled abilities from all classes except the game class, which we designated as the sole manager of the game state. To streamline input handling and parameter validation, we generalized these processes so that no modifications were required when adding new abilities. Inputs for abilities were captured and stored in a vector, serving as a uniform parameter list. Each ability defined its expected parameter types within the Ability class, which we used to validate inputs during the input phase. We also generalized errors as much as possible so runtime errors could be reused.

This vector-based parameter storage eliminates the need for multiple overloaded `useAbility()` functions in the game class. Instead, we implemented a single function that accepted the vector as its parameter, delegating the logic to the appropriate ability-specific function. This allowed for minimal changes needed when a new ability was to be added. The framework was talked about more in detail within the Design section of this document.

In summary, to add a new ability in our project, you'd have to follow these steps:
- Define the new ability within the `Ability` class and update the expected parameters if needed.
- Define the ability logic within the `Game` class and update the `useAbility()` function.

<u>Question 3:</u> 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 firewalls 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? (NO CHANGES)

To make the code simple in terms of minimal editing when changing from 2-player to 4-player, we will be editing the cell class to have a boolean field <u>locked</u> specifically to check if we are in bounds, since it is not as trivial as a square. We are planning to use a 10 by 10 square, however each of the corner cells would have the boolean field <u>locked</u> as true, thus conforming to the new plus shape of the board.

Moreover, we would also have a method to validate if any of the players are moving towards an edge. In principle, we would extend our functionality from checking only the row position to download a specific link in the case of 2 players, to also checking the column position; the rest would follow the basic game semantics, and thus would not need much change.

Finally, moving onto the case when the player loses; in this case, our `Player` class has a shared pointer to each of the locations for each of the `Link's` positions, and we would change each of the symbol's to a dot in our `Cell` class. Similarly, each of the `Ability` objects that the eliminated player owns also has the position as a field, and we do the exact same replacement with the dot. Finally, our `Cell` class also has the field to check if the cell is a server port, so we would make these false once the player is eliminated.

Ayaan Nadamal (anadamal)
Shihan Sharar (ssharar)
Sheehan Shams (s2shams)

In summary, we would go about in these steps to generalize our 2-player case to a 4-player case:

- Instead of a fixed size of players, we have a vector of players so that each player can be initialized from the controller with ease using a for loop
- Include a locked feature in `Cell` class to check if we are at corner of 10 by 10
- Utilize a method that does the same checks for rows, but also for the columns, since there will be an addition of player 3 and player 4
- Generalize `text/graphical observer` to print in the point of view for player 3 and player by 4, by retrieving row and column cells as the transpose of the board
- Retrieve location of eliminated `Player's Link` objects and `Ability` objects and replace the char of the corresponding `Cell` class with a dot
- Change the boolean field for the server port eliminated in `Player's Cell` class to false

Extra-Credit Features:

We have implemented many bonuses as we really did like coding this game. As a quick summary, we used smart pointers throughout the code (no destructors), successfully created 4 player functionality, had the possibility of multiple displays in the perspective of each player, and finally even added an extra ability (on top of the 3 extra required) since we really enjoyed strategizing and experimenting with many creative abilities.

Smart Pointers:

- **Shared Pointers:** For all "owns a" relationships such as between `Game` & `Board` and `Board` & `Cell`, we always use shared pointers to manage memory; we do not have a single destructor across our entire implementation.
- **Weak Pointers:** For observers (`TextObserver` and `GraphicsObserver`), we research and use the CPP reference to use a special smart pointer that demonstrates a non-owning relationship. This helped us still access useful information from `Game` to display, while still maintaining the `Game` class in the event that the observer destructed (corresponding to a `Player` getting eliminated).

4 Player Functionality:

Game Generalization: We took a firm approach in NOT hard coding the game functionality, by always keeping in mind the general functionality (since we were trying to achieve four player functionality). This included, thinking about elimination, finding the next active player for the next turn, implementing `Game::move()` and `Game::battle()` in a manner that accommodates Links from the new player 3 and player 4, and even generalized calculating a valid out of bounds move (`Game::validOB()`)

- **Board Change**: Instead of focusing on the new plus shape of the Board, we implemented it as a 10x10 matrix, (contrary to 8x8 matrix for 2 players), but created a boolean field locked in the Cell class that was set to true in each corner. This made it very simple to generalize the shape of the board, without much of a difference from the 2 players.
- **Player Constructor:** We had to add new Links and also had to make sure the Cell class was now adjusted to set the initial board at new coordinates than that of the 2 player mode. For example, player 1's link would normally be (0,0) for 2 players, but it would be at (1,0). This same principle follows the other players as well, but a vertical shift for player 3 & 4.
- **Text Display:** One of the biggest differences in the 4 player mode, is the fact that there are players that can now play from the left and right in the 2D perspective of the board that we see. To have the same player stats displayed above and below the board in the 2 player mode, we created specific print methods (`TextDisplay::player_line_1`, `TextDisplay::player_line_2`, etc) along with the `<iomanip>` library to center a move the board to accommodate for the print of players on all 4 sides of the board.

- **Graphical Display:** We also had to change the GraphicalDisplay, but this followed pretty easily from the TextDisplay; we only had to change the cell size (when printing on X11), rotate the board similarly for each player perspective, and print the stats for Player 3 and Player 4 as well.
- **Main:** The initial change was also to the `Main`, to make sure it can also handle initial command line arguments for links and abilities. This was an easy generalization as we had to focus on taking the input for `-ability3`, `-ability4`, `-link3` and `-link4`. It is pretty safe to say, this was the "easy" part of making 4 player mode functional.

Multiple Displays:

- **Observer:** Multiple displays involved a key fix to how the Board would be printed. We had a function that tweaks the two (nested) for loops, that change the manner in which the board (8x8 or 10x10) is iterated over, to make sure each of the displays is in the perspective of the player observing. We did this by iterating in the various manners such as first by row, then column (general way) or by column, then row (transpose way). In essence, it simulates the board rotating 90 degrees for each of the 4 players or 180 degrees for 2 players.
- **Main:** We had to adjust the `Main`, to make sure it can also handle the command line argument `-multiDisplay` to attach n observers to the Subject (Game) where n is the number of players playing.

Extra Abilities:

- **Omit:** Omits a cell from the board. Can only be used on cells that don't contain anything. Introduces a new mechanic where players would have to move their links either over the cell (using LinkBoost) or moving around the omitted cell.
- **Block:** Blocks a server port from being entered for 1 instance, if an opponent attempts to enter a server port with a block, the round gets used up and it becomes the next player's turn.

Final Questions:

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?
I think the main outcome that we all realized was DO NOT START CODING! Well, at least not immediately…Our initial plan of attack was to make the UML, discuss a bit of how we want to approach the code and finally start coding. This might seem fine, but we soon realized that each of us interpreted the ideas slightly differently to the others and this ended up causing IRL (in real life) merge conflict: what I mean by that is we each approached the problems differently, and this caused misunderstanding. Luckily, we came to understanding and fixed this very soon; we got to planning immediately, by meeting up and thoroughly discussing each aspect of the game, drawing various game states, and properly delegating each part.

Another very important lesson was to compartmentalize code to make logical sense, i.e. the modularity of our code. Many times while coding, we realized that there were many parts of the same code that were being repeated. We quickly adhered to the principles of good code organization and created helpers to generalize the block (for example: to handle errors, to use ability, to validate bounds, to apply abilities, etc). This generalisation method of thinking was also extremely helpful when we started generalizing for 4 players. This greatly aided us in debugging as we were able to target and single out the root cause and deal with it much better than having to worry about stepping through every line.

We also found it extremely beneficial to see each other's code using version control. While working together and meeting up, we would often have merge conflicts on GitHub, until we started communicating better with who is editing which files. However, sometimes with certain classes such as Game or the derived Observers, we really needed a combined effort to work on them since they were very long and tedious. Therefore, we use VisualStudio's LiveSharing feature and this greatly aided us in working real-time, viewing each other's changes and fixing potential bugs quickly.

Next thing is drawing on boards! We found it extremely beneficial to go through scenarios on the whiteboard and this really helped us catch errors in our code, think of edge cases to test and even simplify logic too. When designing a game, a thorough process to plan, test, code, implement, debug are all very important cycles present in every game designing team, and we think visually drawing & identifying this was a key aspect to our successfully built project.

Finally, communication is key! Working/developing software in teams requires a level of collaboration, ideation and discussion among teammates that helps them use the team to get work done efficiently. This was one of the key aspects, we believe helped us progress forward as a team. The constant interaction and exchange of ideas & changes gave everyone a certain responsibility and helped us understand & respect that. Moreover, this communication was not only emphasized verbally, but also in terms of code documentation. Initially, we did not really document much, but as soon as we started, it helped speed things up and also aided in comprehension of the code better.

Question 2: What would you have done differently if you had the chance to start over?
If we had to start over, we would definitely do many things to better improve our process of creating this game. First off, we would create a timely plan that not only creates goals/personal deadlines like we did in DD1, but also attainable and specific. I think it's quite a common problem to often overestimate capabilities before actually starting the programming, and this could be the reason that one of the issues we had to frequently deal with was time management. Often, we were all free at different times and sometimes worked alone, however these changes were only properly communicated when we met up every 2 days; therefore, within that short timeframe, if someone else also edited the code, it resulted in unnecessary merge conflicts and issues. Hence, the first thing we would change would be having daily rundown sessions, just so that everyone knows what they are doing each day and aren't exactly trying to fix the same problem or worse creating more!

Secondly, we would definitely try to do more code documentation as we progress. Now although, we did realize this a few days after coding like crazy, we understood that documentation not only helps you, but in a team situation also helps others understand your code, and therefore fix any problems or even use it for other purposes in the future. Hence, we would most definitely make sure that we start off with documentation to ensure we end off quickly and without any problems!

Finally, despite not having too many bugs to deal with, in a much larger scaled version of working with a software development team, continuous debugging and testing is very important (especially before git pushing code!). We often just pushed code and assumed it worked, and initially did not perform many unit tests on the smaller classes. However, this proved to be slightly problematic when dealing with the code, as we only properly started testing when we had a somewhat functional game running. Luckily, projects like these do not have much depth, however for huge codebases in companies, it would be much more beneficial to practice unit testing. This ensures that the code progresses in a logical manner of dependence and does not jeopardize the entire project.

Conclusion:

To sum everything we have mentioned above, RAIInet was truly a fun game to code, but a more fun experience to have with a team. It not only taught us code collaboration from a zoomed out perspective, but also the intricacies behind software development for a game. We truly had a rigorous, yet fruitful two weeks of creating a robust set of steps, ideating a method to tackle the possible problems, implementing OOP principles, programming RAIInet itself, and finally translating this entire process into this document you see above! Every step helped us move from stepping stone to stepping stone, and although we did falter sometimes in between, we did cross the finish line. We are truly proud of our final project (despite many others doing the same…) and hope you love our take on Railnet!