Overview

The RAIlnet project follows the Model-View-Controller (MVC) architecture. The primary components include the Controller, Game (Model), and Views.

Controller

The Controller manages the overall application flow, interpreting user-issued commands and relaying these to the Game model. It collects updates from the Game and distributes these updates to the Views, ensuring that the Views always reflect the current state of the game.

Game (Model)

The Game serves as the central model, managing interactions among all game components. It owns instances of Players, each of whom controls game entities such as links and abilities. The Game processes logic, state updates, and ensures game rules compliance.

Views

There are two distinct types of Views:

- **Textual View:** Provides a command-line based visualization of the game.

- **Graphical View:** Offers a visual representation of the game with interactive graphical elements.

Both Views independently receive state updates from the Controller and display their current state upon request.

Design

To bring RAIlnet to life, multiple design patterns were taken into consideration and used. These included:

- **Factory Method:** Used extensively for creating Ability instances. This approach simplifies ability creation and enhances scalability.

- **Decorator Pattern:** Implemented in both Cells and Links, allowing multiple effects to be easily layered. This strategy significantly simplifies extensibility.

- **MVC Architecture:** Clearly separates the responsibilities between the Controller (command management), Model (game logic), and Views (presentation).

- **Observer Pattern:** Views subscribe to the Controller and automatically receive updates. Operator overloading provides a unified yet flexible update mechanism.

- **Dependency Inversion Principle:** Implemented via LinkManager, encapsulating link management logic (adding, removing, decorating links) and isolating complexity from other modules.

Resilience to Change

The project's architecture is intentionally designed to accommodate future enhancements and changes seamlessly. Adding new Abilities is straightforward, facilitated by the AbilityFactory, which avoids extensive modifications to existing structures.

Cell decorators support adding new behaviors dynamically through composition, greatly improving modularity and reducing the impact of changes.

Link decorators similarly allow novel effects to be easily layered, enhancing flexibility for future extensions.

Using enums instead of boolean flags for link types provides an extensible mechanism that simplifies the integration of new link variations.

The LinkManager centralizes operations related to link management, significantly simplifying the addition of new functionalities and interactions involving links.

New Cell types can be easily integrated by wrapping existing Cells, enabling the addition of specific logic without widespread changes.

Furthermore, flexible player turn management and loosely coupled components facilitate the seamless integration of potential network features, supporting broader scalability and extensibility.

Answers to Questions

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?

We chose to maintain a design where the Controller has a vector of vector of views, and it notifies every single view with a player-dependent update. This means that every single view can handle the update independently, which allows us to either create a view class for each player, or create a view class that stores multiple sets of data to display for each player. When display is called, we will call the Game getPlayerIndex() to determine what player's view to display, and call the display functions on all the related views.

This approach allows us to create any number of views; We can create both a text view and a graphics view for each player, or a view that can handle more than one player; In this case, the view pointer would be passed into the vector that maps player indices to views multiple times.

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.

To design a general framework that makes adding abilities easy, we can use the factory method and have an AbilityFactory class to handle adding abilities for players. We have brainstormed three unique abilities to our final game. First, we have the wormhole ability. When a player activates it, they can swap two of their links. This allows for strategic positioning for battles and reaching the goal. Next, we have the quantum entanglement ability, which allows two of the player's links to be entangled together. This means if one link moves in a certain direction, so does the other. This also applies to downloading, so if a player downloads one link, they download both links. Most importantly, we have the Papple ability. This is a strategic move in which a player creates a battle formation to attack and defend at the same time (there must be one link on each corner of the board, all owned by the player). On the next round, when calling the Papple ability, the player wins the game.

One could conceivably extend the game of RAllnet to be a four-player game by making the board a "plus" (+) shape (formed by the union of two 10×8 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 game class will create and maintain a vector of player pointers, switching between them between each turn. Thus the turn-taking mechanisms should be extensible for 4 players.

Interactables that are owned by a player do not depend on anything except their owner and the player they're interacting with for their logic. This means any game piece interactions should not rely on the existence of exactly two players and should be extensible to 4.

Along with these updates, tests would need to be conducted to ensure there are no additional edge cases that were unaccounted for. For example, if one player gets their links downloaded by the three other players, then the one player would have nowhere to move. In two player mode, this is not an issue since if one player loses all their links, the other player must have either 4 viruses or data downloaded to end the game. Thus, another check would need to be made to ensure a player with no links loses in 4 player mode.

Displays for 4 players are also handleable as previously stated.

Extra Credit Features

Quantum Entanglement

The Quantum Entanglement feature introduces a novel mechanic by connecting two links: a primary link and its paired link. When the primary link moves, its paired link automatically moves in the same direction. This ability significantly alters gameplay strategies by enabling more efficient movement across the board or compelling the opponent into disadvantageous positions. Implementing Quantum Entanglement was straightforward, thanks to our existing design patterns and the LinkManager class. By leveraging decorators, we seamlessly added the required functionality without substantial modifications to the existing codebase.

Wormhole

The Wormhole feature enables two links to swap positions instantly on the game board. This introduces an exciting dynamic, providing opportunities to quickly reverse fortunes and requiring players to constantly adjust their strategies. The idea was inspired by our desire to create engaging gameplay scenarios and maintain a fluid, unpredictable game state. Like Quantum Entanglement, integrating Wormhole into our design was smooth and uncomplicated, benefiting significantly from our modular architecture and use of decorators, ensuring minimal disruption to existing components.

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?

Working on this project taught us a lot about software development in teams. We realized how critical careful design is, as even well-thought-out plans can run into unexpected problems. Often, it's better to step back and reconsider the approach rather than forcing a workaround. We found that major technical changes, although daunting, can be worthwhile if properly justified. Additionally, we saw that theoretical correctness doesn't always translate perfectly into practice. Some design patterns, like our initial use of decorators for links, can complicate things if not implemented effectively. We also learned the value of using Git effectively, including continuous integration checks and reviewing each other's pull requests. Adopting an MVP (Minimum Viable Product) strategy ensured we always had a working version to demonstrate, reducing the risk of ending up with nothing at the demo. Finally, improving our testing practices earlier on would have saved us considerable time and effort in the long run.

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

If we had the opportunity to start over, we would have invested more time earlier in the development process. By beginning earlier, we could have spent additional effort thoroughly exploring our design choices and fully understanding the implications of various design patterns, particularly their potential drawbacks. For instance, our use of the decorator pattern initially seemed effective, but later became challenging when storing parameters directly within the decorator classes. Taking more time upfront to anticipate and evaluate these intricacies would have streamlined our implementation and improved overall development efficiency.