CS 246 - RAIlnet Plan of Attack

Ryan Tonucci (rtonucci)
Yuto Chau (ychau)
David Hu-Liu (dhuliu)

In addition, your plan of attack must include

- a breakdown of the project, indicating what you plan to do first, what will come next, and so on.
- Include estimated completion dates, and which partner will be responsible for which parts of the project.
- You should try to stick to your plan, but you will not be graded by the degree to which you stick to it.
- Your initial plan should be realistic, and you will be expected to explain why you had to deviate from your plan (if you did).
- Finally, your initial plan of attack must include answers to all questions listed within the project specification itself.
- You should answer in terms of how you would anticipate solving these problems in your project, even though you are not strictly required to do so.
- If your answers turn out to be inconsistent with your final design, you will have an opportunity to submit

Responsibility Breakdown

Yuto	Ryan	David
• By Nov. 23: Observer Pattern (TextObserver and GraphicsObserver classes) and proper integration with Game class • Beyond: Contributions to Game logic and abilities (will be the meat of the work) • Throughout: Testing as needed, as well as collaborating with teammates ensure that all components	• By Nov. 21, I will have implemented and tested the Link and Cell classes. Testing will by done by by manually testing in the main function. • By Nov. 22, I will have implemented and tested Board class again using the main function to manually test. • Beyond that I will be working towards implementing the	• Ability class, using template design pattern, virtual method to activate any subsequent ability. • Ability identification, get ability, toggle use ability, used ability, • Completion / implementation of various individual abilities for example linkboost, diagram, extrastrength, etc. • Guessing also

fit together seamlessly ● After core functionality is fully tested and completed, will try working on extra credit features	Game class and main function with the help of my teammates. ● For this project I will focus on testing each class as I go to limit the number of mistakes when all the classes are finally combined.	creation of MakeFile, XDisplay, etc and helping with other various files ● Writing test cases/trying to break the game

Progress Tracker

Date	Progress	Contributions
Nov. 9	● Discussed which game we wanted to do for the project and decided on RAIInet. ● Read the instructions carefully, coming to an understanding of the game, and strategies. ● Began designing the UML for the assignment. ● Began writing a plan. ● Brainstormed ideas for additional abilities: *extra strength* (+1 strength to your link of choice), *black hole* (lose link if you land on it), *barrier* (neither player can move a piece there), *diagonal* (assign a link diagonal movement only)	All members
Nov. 16	● Continued designing the UML ● Discussed which design patterns were most appropriate ● Decided on using an observer pattern similar to the implementation in A4Q3 for displaying text and graphics ● Completed an initial draft for the UML	All members
Nov. 19	● Started implementing Link and Cell classes	Ryan
Nov. 20	● Worked on plan and finalized UML	Yuto and Ryan
Nov. 21	● Completed the plan and divided the responsibilities for coding up	All members
Nov. 22-23	● Will work on core functionality (Board, TextObserver, Cell, Link) and test suites for each	All members

	class	
Nov. 24	• Work on implementing Game, abilities and the graphics observer	All members
Nov. 25-26	• Will work on combining all the classes and testing the program as one entity.	All members
Nov. 27	• Will work on the final UML, design document, and thinking of edge cases.	All members
Nov. 28	• Will work on bonus features (time permitting)	All members

Project Breakdown

RAIInet is a two-player game consisting of a board, and pieces, called links. There are two types of links: viruses and data. Each player starts the game with four of each. The goal of each player is to either:

- Download four data (could be player's or opponent's)
- Make the opponent download four viruses

Achieving either of these goals wins the game.

Overview

We are planning to incorporate several design patterns in our program. For starters, we will use the observer pattern to display changes in data both as text (through stdout) and as graphics (using the XWindow class). We will also use the template method pattern to target specific behaviour to customize in certain ability subclasses while maintaining behaviour that is universal to all abilities (as defined in the Ability superclass). Additionally, we considered using the decorator pattern, but found that it would not fit as well in our program as we previously thought. To avoid unnecessary complexity, we have decided to forego the use of this pattern until we find the need for it arises again.

From a structural standpoint, we are planning to use the MVC (Model, View, Controller) Architecture. The relationship between the Model and View will exist as an observer pattern. The Controller (the Game class in our implementation) will govern user input and pass commands to the Player and Board classes. This will allow our program to adhere to the Single Responsibility Principle.

In terms of memory management, we plan to avoid explicitly managing memory in any of our classes. Instead, all memory management will be done through vectors and smart pointers.

Here is a table of our classes and their descriptions:

Class Name	**Purpose**
Game	Hold the most fundamental pieces of the game, both players and the board. Performs any functions that may be done by the user (for example moving a link or using an ability).
Player	Is the class responsible for holding and modifying user information such as user links and abilities.
Subject	Holds the observers and the methods to attach, detach and notify the observers as part of the observer pattern.
Board	Holds the game board and allows access to the board through a public getter and setter for each cell on the board.
Observer	This is an abstract class which holds a pure virtual method notify as part of the observer pattern.
TextObserver	Is a concrete observer responsible for producing output shown to a player on any given turn; the board and their abilities along with any visible opponent's pieces. It displays the this in an ascii art display in the CLI
GraphicsObservers	Is a concrete observer similar to the text observer except using a window as a GUI instead of the CLI.
XWindow	Is the class responsible for managing the resources of the window(s). It contains methods that allow the GraphicsObserver class to use the XWindow for output.
Ability	Is an abstract class as part of the template method pattern which provides methods to check basic information about an ability such as whether it has been used and its name. Also has a pure virtual method "activate".
Firewall, Linkboost, Download…	Override the "activate" method and allow the player to activate their ability with the correct implementation.
Cell	Is responsible for managing the state of a giv en cell including any special abilities used on a cell and any links in a cell.
Link	Is responsible for managing the state of any given link, including whether the link is data or a virus and whether the link has any special abilities.

Design Patterns, Coupling, Cohesion, and MVC

Design Patterns

Observer Pattern - The observer pattern is used to display the state of the game to each player. It does this through an ascii display in the CLI or through a window. The observers have a pointer to the board as a member and all observers are notified every time a link moves on the board.

Template Method Pattern - The template method pattern is used in the Ability class and its subclasses. We use this pattern since we want the subclasses to override just some of the aspects of the Ability superclass. In particular, we want to override the way the activate method is implemented depending on the specific ability used. We want to use the superclass functions to check if an ability has been used or to get the name of the ability since those methods are the same regardless of subclass.

Coupling - In order to reduce coupling as much as possible we have designed the program to have as few subclasses, friendships and protected or public data members as possible. The subclasses in our design are all part of a design pattern and we have no friendships. Additionally, we have divided the program so that each class serves one purpose outlined below.

To best minimize coupling, we make use of design patterns that best create baseline abstract classes in which we can build ontop of later. For example, abstracting virtual methods such that the subsequent class makes use of their own implementation of said method, thus improving maintainability and scalability of code solution.

Cohesion - To maximize cohesion we have separated the program into classes which each have one specific job as described in the chart above. In our initial UML, we planned on writing one main studio class that would handle the game, the board, the players, the pieces, etc. Which from a cohesion perspective, is generally bad practice. To thus maximize cohesion, we are trying to work toward having each class dial in on a very specific task to improve testing, and understanding of the code. For example, an entire class just for the link, just for the cell, just for the board, and have the board class interact with the cell which interacts with the links. A separate player class under the game class that then needs to interact with the board class.

Model View Controller (MVC) - To best structure a program to incorporate the desirability of both low coupling and high cohesion, it's best practice to separate into a model for data and logic, view for UI and user experience, and controller for handling input and and giving updates to the model/view.

The Controller receives input from the user, passes commands to the Model, and receives results from it. The Controller also communicates with the View to provide information to be displayed, and the View passes control back to the Controller. In our program the model is the Game class, the view are the concrete observers, TextDisplay or GraphicsDisplay. Finally, the controller is the the Subject class since it is the class which holds the model, the Game class, together with the view (the concrete observers).

Questions from Project

Question: 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?

Answer: To allow 2 separate point of views of the board, we would change the notify method in TextObserver or GraphicsObserver to call one of two render functions based on what player's turn it is. Both concrete observers have this info since they have a pointer to the game object being used. From there one function would render normally as player one's point of view and the other would render in the opposite point of view.

Question: 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.

Answer: To add a new ability to the game a new subclass of Ability must be created which overrides the pure virtual function activate in the Ability class. This method gives the instructions for using a specific ability, so a new ability must have a unique implementation of this method depending on what the ability actually does. Additionally, a flag should be added to all cells or links depending on if the ability is applied to a cell or a link. This will keep track of whether a particular cell or link has this ability activated on them.

Question: 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: To allow a four-player game mode, we would need to modify the Game class to hold four Player pointers instead of two. We would need a method that is able to remove all the assets on the board of a player that has lost. This could entail having a scraper to iterator across the entire board and delete any assets owned by a given player. Alternatively, there could be some method of deleting all of the links owned by a given player, but still that said player's physical abilities placed on the board needs to be removed. In addition, we'd need to change the representation of the board (or at least avoid the "corner" squares) in the game logic (as well as when printing in the concrete observer classes). A new class to set up the newly displayed "plus" sized board is likely needed with new piece positions. Moreover, currently we define turn = {1, 2} being player 1 and player 2 turn respectively, for a 4 player game we would try to assign logic for turn = {1, 2, 3, 4}. Considerable new amounts of logic is likely needed to extend RAIInet to incorporate both 2 player and 4 player functionality. Adding and switching between the game modes should be trivial with booleans, however, the way the game is played in 4 player will be different.