Introduction

Group: Edward He, Terry Zhou, Jeffrey Wang
Project: RAIInet

Description of RAIInet:
RAIInet is a video game similar in style to Stratego. RAIInet is a game played between two opposing players, who take on the role of computer hackers. Each player controls eight pieces, called links.

There are two different kinds of links: viruses and data. Each player has two goals:

- Download four data.
- Make their opponent download four viruses.

Upon achieving either goal, the player wins the game.

A game of RAIInet consists of an 8 × 8 board on which players initially place links face-down. In this way, only the player who placed the link knows whether it is data or a virus. Play then proceeds in turns. On each player's turn, a player may use a single ability, and then must move one of their links in any cardinal direction. When a player moves a link off their opponent's side of the board, they immediately download it. When a player moves a link onto their opponent's server ports, their opponent downloads it. Additionally, when a player's link loses to an opponent's link in battle, their opponent immediately downloads it.

Overview

Our implementation of RAIInet is based on the Game class, which conducts the main loop of the game after main picks up and interprets command line inputs. This class encapsulates individual games, in case more than one needs to be run. Each Game is composed of a Board object and two Player objects, representing player 1 and 2.

The Board will contain a two-dimensional vector grid of Cell objects acting as the map of the game, with each Cell representing an element of the map. These elements inherit from an abstract Cell class, promoting polymorphism as well as making adding any new kind of element of the game map easier by simply creating a subclass of Cell. In the current version, a Cell can be either a Tile (which holds up to one Link using a pointer to the Link currently on it) or a Server (described in the rules to be associated with a player; which player this is is stored as a field of the Server class). The Decorator Pattern is also used with the Cell to implement firewalls.

Meanwhile, each Player will hold a (pointer to a) Board, a vector of Link objects and a vector of Ability objects, just as a player of a physical board game would have Links, Abilities, and a Board to play the board game. The Player class uses Link objects and Ability objects to interact with the board, and does most of the work when it comes to interactions, such as moving. The Link class plays a more passive role, but still utilizes the Decorator pattern so that the Link Boost ability can be decorated onto a basic Link. On the other hand, each Ability object is an instance of a subclass that grants it a member function implementing that ability according to the rules.

Finally, in order to display the game to the user, the Observer Pattern is used—each Cell is the subject of one or more observers, which include the text-based display and the graphics display (when necessary).

Design Challenges

While not a difficult game to understand, RAIInet posed some design challenges that were somewhat unique.

Starting from the smallest element of the game, the basic Link had to be considered. Each Link came with a significant amount of data—strength, controlling player, link type (data or virus), etc.— that could easily be represented as fields in a Link class. However, two challenges presented themselves from this point.

First, it became apparent that moving the Link would be more challenging than initially assumed, since one had to check which moves were valid and which were not. We decided to put the majority of the logic in Player, since it allowed the Links themselves to be as minimal as possible, reducing the complexity of the Link class. As such, a Link object's move function simply told the Player what Tile they would end up on if they were moved in a certain direction, whereupon the Player would check the validity of moving to said Tile and actually move the piece if it was a legal move.

Secondly, any Link needed to have the capability to become a Link Boost. This by itself could be handled in the move function (ie. by having a boolean to represent whether a Link is Boosted), but we decided to use a Decorator pattern instead, giving us more leeway to create future power-ups of this kind. We gave Link Boosted links all the functionality of a Link Base, but overrode the move function such that it would call the base move twice, only failing if it ended up in an invalid position.

Our implementation for Cells also had to be considered carefully. Initially, we considered using three different subclasses: Tile, which was a Cell that could have a link placed on top of it; Server, which worked as described in the rules, preventing friendly links from being place on top of them, but downloading enemy links that were placed upon it; and Wall, which prevented any movement to them. However, we decided to remove the Wall subclass, since for a rectangular

board, it was far easier to both process and display if the size of the Board was used to determine its bounds (this change, however, would be reconsidered if Walls could be placed by either player, or the shape of the board was changed, as discussed later). We used a subclass system instead of a decorator for Cells because there was no common functionality that needed to be wrapped—Tiles needed to hold links, and Servers needed to download them, and nothing else. However, when it came to Firewalls, we realized that we were once again best served with Decorators (this time on the Tile class), since a Firewall was essentially a basic Tile that downloaded enemy links that were placed onto it.

For the setup options on the command line, we have assumed if -link1 or -link2 is passed, the given file will contain valid links. For the -ability1 and -ability2 command, we assume there will be a maximum of 2 of the same ability even though you can add more than 2 of the same ability.

We have assumed the sequence command cannot be used within a sequence command. The purpose of the sequence command is to be more efficient, mainly in testing. Being able to issue the sequence command within a sequence command is redundant.

Abilities were also implemented as subclasses, with each one overriding the same method to take input and activate the ability as specified—we believed this method would be simpler than having an outside loop take input and pass it to the Ability. Some discussion went into the fields contained in each Ability object (what information it needed to hold as fields, such as a pointer to the board or players), but it was eventually decided that the abstract Ability class would contain pointers to all necessary structure, as we believed that the increased efficiency from direct pointer lookup outweighed the slight organizational benefit that less information might provide.

<u>Ability Specifications</u>
To use an ability, you use ability <N>, then it will prompt you to input a link or coordinates according to the ability with ID N.
Link Boost:

- You can link boost any of your own links as long as it is not downloaded
- You can use more than 1 link boost on the same link

Firewall:

- You can only place a firewall on an empty tile, this mean you cannot place on a server, link or another firewall

Download:

- You can download any of your opponents links as long as it is not already downloaded and not encrypted by the Encrypt ability

Polarize:

- You can polarize any link (both your own and your opponents links) as long as it is not downloaded
- You can polarize the same link more than once

Scan:

- You can scan any link (both your own and your opponents links) as long as it is not downloaded and not encrypted
- Scan will only work on hidden links, so you cannot scan the same link twice or scan a revealed link

Strength Boost:

- You can strength boost any of your own links as long as it is not downloaded already
- You can strength boost a link more than once

Encrypt:

- Encrypt protects a link from being downloaded or scanned by Download and Scan ability
- You can encrypt any of your own links that is not already encrypted or downloaded
- Encrypt will last until the end of the game or when the encrypted link is downloaded

Hack:

- Hack can be used to remove any firewall on the board and it can fail
- A creative and fun feature of hack is that if you enter valid coordinates (coordinates in the board) where there is no firewall, the Hack ability will be used

From this point, the implementation of the basic structure of the project was fairly straightforward. The Board holds a two-dimensional vector of Cells and each Player holds a pointer to the Board and to its opponent. Some attention should be drawn to the two Player.move functions—one is a utility function that simply moves a Link with a given ID (character) to the given Tile, without any checks, while the other takes a Link and Direction, calculates where the Link would move to (using the Link's move function, mentioned above), checks if the Link was going off the board, hitting a firewall, or moving into another link, and handles downloading or battling if necessary (and returns false if such a move is illegal so that the player can be prompted for a different move).

To this end, the Game class contains the main loop that works as one might expect—it has a field representing whose turn it is, and based on that, it takes input and calls the necessary functions. As mentioned in the rules, a player's turn only ends once they have made a move, so the Game class only passes the turn to the other player when this does occur. It is also responsible for initialization and setup for most objects we use.

Finally, the Observer pattern we use for display should be mentioned. The two Observers (Text Display and Graphics Display) both work in similar ways. Importantly, they store copies of both the board as it needs to be presented and the player information that should be printed (eg. abilities used), which are updated by different functions (each function is called when something relevant is updated—when an ability is done being activated, the function corresponding to player information is called for both Observers). To this end, both Links and Players are considered Subjects: Links update the copies of the board when moved, and Players update their own information. They also contain a field that tells them who the active player is, also updated when necessary. This allows the Game loop to print only one player's hidden information at a time, as described by the specifications (only publicly available information is shown for the other player.)

Resilience to Change

Our design is meant to be resilient to change in a number of ways.

While there are some rules changes that will always be difficult to account for, our design does allow for a number of quick modifications by using constants when necessary. For example, changing the number of Links players start with is a task that relies only on initialization, as each player can operate with any number of links. Similarly, the number of links needed to win can be accomplished by simply changing a threshold in Game. Changing the game to involve more players invites slightly more work (as described later in the report), but changing it to involve more rounds can be easily accomplished because more Game instances can be created. The size of the board can be changed in one line of code, and reshaping it is possible as well, if slightly more involved—the Board data structure is set up to be reasonably flexible by nature of its vector setup. Rules changes are always difficult to make efficient completely, but by using constants and flexibly-lengthed containers when we are able, we seek to cover all but the most complex of modifications to the specifications.

Input is a less open-ended issue. If the syntax of a command changes, we can change the input needed to activate the given command easily. This is thanks to the compartmentalization of our project—we try to keep most of our input parsing in the main Game loop, which means that changes to things like the names of commands need not be hunted down across the entire codebase. For this reason, changes to input that do not affect the function of the code means that only the code to accept it needs changes, and most of the code can be preserved. In addition, the code for sequence is kept different from the main input code, which means that if the sequence command was altered to parse a file with different syntax than the main input, no structural changes would need to be made.

The most interesting changes to account for come in the form of new features. Working from the bottom of our project upwards, we designed Links as being fairly easy to change—the abstract Link class can contain any number of fields, meaning that any changes to the class (eg. each Link gets a color) are easy to add. Meanwhile, the decorator pattern means that additional enhancements to Link can also be added and even combined.

This logic applies for Cells as well. For example, imagine a Tile that sets the strength of Links that pass through it to 5. Not only would creating such a Tile be reasonably easy (make a new subclass), the Decorator pattern would mean a Firewall could be placed on such a Tile with no additional work. This means that new features relating to Cells are easy to add.

For this reason, the Board, as a composition of Cells, would face relatively little modification from any new features. The only such change it might have to contend with is a new shape, which is easily handled because it contains a two-dimensional vector full of Tiles—a different board size would just change the dimensions of this vector, and maybe (if necessary) reimplement Walls to ensure that Links could not go out of bounds.

The Players and Game are also something that could receive modification as necessary. As discussed previously, the Player handles most of the logic when it comes to moving pieces around. If a change needed to be made, the correct code to change would be readily available, and since it relies on functions in other classes, a change in a different spot would not mean the Players' behavior changed. Similarly, the Game relies mostly on functions from other classes, and besides changes in input and turn structure (which it is well-equipped to handle, since it has a field for player turn that is passed as a parameter into most functions it calls), would not be altered much.

Finally, the Observers' basic functionality would not need to be altered much, but if the program was to be printed out differently, all the necessary code to be changed is in one of the Observers. In addition, the observers hold all data they might at some point need to display, so if the amount of hidden information was reduced, they would have the necessary data (or pointers to it) to prevent major structural change (increasing hidden information would be a much easier change to implement, since it only involves showing a smaller portion of the data already known).

All in all, we believe we have implemented RAIInet in a way that is resilient to changes in the specifications, and most logical changes can be implemented reasonably quickly using the structure we have decided on.

Answers to Questions

1. **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?**

In our current project, we have a Display class that inherits from an Observer, which itself is attached to the board, thus forming the Observer pattern. There is a vector of vectors within the Display class that acts as the grid for the display. In order to display two displays that differentiate player 1's view from player 2's, we could add a second vector of vectors to the field of the Display class, one for player 1, one for player 2. Alternatively, we could add a separate concrete observer for the second player, so there's an observer for each player. We would then adjust our operator<< that draws the game board accordingly so that it can output 2 views. As for graphics display, we could have a second window for the other player.

2. **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.

An Ability class can be created that acts as an abstract class for the abilities we want to implement. The abstract Ability class will have the virtual method ability() to indicate that each subclass must implement an ability. In order to add a new ability, a new subclass can just be created to inherit from the abstract Ability class. The Player class will hold a vector list of these Ability objects in its field (aggregation - “has an ability”) which can be used with addAbility() to add any newly created abilities, and useAbility() to call ability(). Some possible ideas for new abilities include strength boost, encrypt, and hack.

The Strength Boost ability increases the strength of a link by 1 for the rest of the game. The Encrypt ability encrypts (shields) a link so that it cannot be downloaded or scanned by the Download and Scan abilities. The encryption will last until the link is downloaded or the game ends. The Hack ability will allow the player to remove any firewall placed on the board.

3. **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?**

To extend the game of RAIInet to be a four-player game, we would have to adjust a number of classes in the project.

First, the board would have to be initialized as a “plus” (+) shape for four players, as described in the question, and some method of showing this board would have to be implemented— this would likely involve adding a Wall subclass of Cell back to the project, from which point a 10x10 board would be created with Walls in the corners. Other concerns include making sure that the empty corners of the board could be properly printed, adjusting the move function so that pieces could move off the board in the right places, and finding new characters that could be printed, since the original game’s system of capital and lowercase letters works only for two players.

The Player class would also have to be adjusted slightly. The current version of RAIInet does not require the Player class to access their own servers or firewalls—the former are automatically placed upon initialization and the latter are saved by the board, not the Player. This would have to change in a four-player mode, to support the implementation of a function called upon a player losing the game that would handle its remaining links, firewalls, and

servers (the first two could be removed from the board, while the servers could be cast to normal squares).

Finally, the Game class would have to own two more Player classes (for players 3 and 4), which would need to be initialized and stored. For the game to be played, the code present to take turns would need to be adjusted to give each player a turn, and to only end the game when one player wins, or three players lose.

While all these features are unnecessary for the two-player game, and require a decent amount of work to implement, they do not prevent the two-player game from running correctly (assuming that a choice of boards was allowed). In other words, it is possible to have both modes within the same code with minimal work, allowing the user to choose the number of players when starting the game.

Extra Credit Features

We have added features for better user experience, such as colours and prompts. We have also used smart pointers throughout (unique_ptr and shared_ptr), meaning that we did not have to manage memory. This allowed us to focus on adding the features that we needed instead of what happened to them after we finished using them.

Final Questions

1. **What lessons did this project teach you about developing software in teams?**

The most important lesson we learned was that communication is very important when working in teams. A poorly communicated system means that features are left uncompleted, or two people work on the same chunk of code at the same time, meaning that merge conflicts occur frequently. In addition, a change to the structure of the program, no matter how minor, might cause other people's code to no longer work, so such a change needs to be communicated. However, just as poor communication creates problems, good cooperation can solve them—one person might be able to solve someone else's problem much faster, or have a good idea that speeds up the process. We found that meeting in person was very helpful to this end, since it made asking for help much easier. Overall, we learned a significant amount about working in teams and hope to be able to apply such knowledge in the future.

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

When it comes to code, we think that our original structure was for the most part, sound. However, we believe that we should have put slightly more attention into convenience over

compartmentalization. For example, when it came to abilities, we spent a long time trying to pass each ability as little as possible in order for it to do its job, but we ultimately decided that there were relatively few downsides to letting the abilities access the Boards, Links, or Players. Realizing this earlier might have saved us some time on this and other Objects throughout the project. In general, we think we may have prioritized form (code that looked nice) to function (code that ran efficiently and was easy to change) to some degree, and would have put more emphasis on the latter if we had a chance to start over.

Conclusion

We believe we have constructed this project to the best of our ability, and hope that some of our hard work is able to show through. We would also like to thank the TAs for reading this report, as well as everyone who helped in CS 246 this year, especially the professors, without whom this project would not have been possible for us to make.