RAIInet Final Design Document

By: Christina Li, Evelina Zheng, Daniel Su
Dec 5th, 2023

1.0 Introduction

RAIInet is a game between 2 players, who each own 8 links that are either classified as a data or virus. In the beginning of the game, the opponents' links' identities are all unknown. Played on an 8x8 board, one can “win” by either 1) downloading 4 of the opponents’ data links or 2) the opponent downloading 4 of your own viruses. A link can be downloaded through movement to server ports, battling, or movement for an opponents’ edge of the board. Ability cards can also be used to affect gameplay. (Please refer to the original project document, RAIInet.pdf, for more game details).

To develop the game of RAIInet, we focused on tackling the **five** main, **core functionalities** that compose RAIInet. The design framework and UML of the project are thus centered around the features listed below:

- Board set up and initialization (randomizing placements if needed, setting the strengths, initiating the ability cards, etc)
- Movement of pieces across the board (checking for illegal moves)
- Downloading of data/viruses through server ports and opponent’s edges
- Battles between players’ pieces
- Ability effects on both board and links

Taking these core functionalities into consideration, the overall structure and techniques used were centered around simplifying the implementation of these features, as well as allowing for flexible adjustments and changes to these key functionalities in the future (if needed).

Furthermore, our team also prioritized our design to reinforce low coupling and high cohesion principles.

2.0 Overview

The key considerations we had when deciding on the overall design framework for the project were:

- Low coupling
- High cohesion
- Resistance to change, maximizing the flexibility of our code to account for the following potential changes:
    - Changes in project specifications (ex: changing the default step size of a link, changing the starting position for links on set up, changing the board size from 8x8 to arbitrary dimensions)
    - Project enhancements/additions (ex: adding additional features, adding another player, etc)

We also adopted an **MVC (Model-View-Controller)** approach; this separated the different classes into 3 general “groups” organized based on high-level purposes. Following MVC, the Model defined all the relevant data structures, the controller manipulated the data, and the view handled the presentation of the data to the user. By following this approach, we were able to have relatively low coupling and high cohesion, thereby maintaining reusability, scalability and flexibility of the code.

M/V/C	Class Name	Important Methods & Members	Purpose/Explanation
Model	Firewall, LinkBoost, Scan [...], AbilityCard	**void activate()** *in abilitycard.cc* - a pure virtual function, handles the activation of that specific ability **Coords coords** *in firewall.cc* - unique to any abilities that act on the **board (ex: Firewall) ;** stores information on where the ability is applied to	These classes define the data structures needed for abilities The concrete classes Firewall, LinkBoost, Scan, etc… all inherit from the abstract class, AbilityCard
Model	Data, Virus, Link	**bool identityRevealed** *in link.cc* - tracks whether the identity of the data/virus has been revealed to the opponent	These classes define the data structure needed for links. The concrete classes Data, Virus both inherit from the abstract class, Link
Model	Player	**int abilityCount** *in player.cc* - tracks the amount of ability cards the player has left. It is decreased when an ability is consumed	This class defines the data structure for a player and holds relevant information such as the number of viruses they downloaded, whether they have won, etc.
Model	Coords, SpecialCoords, ServerPort, EdgeCoords	**Coords coords** *in specialcoords.cc* SpecialCoords own x, y coordinates and a basic getter/setter function for these coords	These classes define data structure for x, y coordinates that are used and manipulated for link movement and tracking locations on the board
Controller	GameBoard	**void movePiece(shared_ptr<Link> link, Direction dir)** *in gameboard.cc* Handles the movement of a link, in the dir direction **void init()** *in gameboard.cc* Initializes the board	This class translates all interactions from view as it handles most of the logic and manipulates the model data when it is needed
Controller	main.cc		Supports translations, calls the relevant functions in GameBoard class
View	TextDisplay	**Vector<Vector<Char>> theDisplay** *in textdisplay.cc*	Handles the program's data representation. In this case, this is done through the 2D array of characters that represents the pieces on the board.
View	GraphicsDisplay	Almost exactly the same as above.	Almost exactly the same as above.

3.0 Design

When planning the development of RAIInet, we used the object-oriented design principles and techniques introduced in class to solve the associated design challenges.

Specific Design Challenge (in the context of RAIInet)	Technique(s) Used & Why They Were Effective	Example

Updating the View for the user after a piece is moved, or a state is changed regarding player or link information
(ie: How should we update the text display to reflect a change in the game state? What should we do when the user has moved a piece, used an ability, or downloaded a link?)
	Observer Design Pattern
	We used the observer design pattern. By making TextDisplay and GraphicsDisplay observers of the GameBoard, we can easily call notify() whenever there are changes in the link, players or other states and then perform necessary logic to update the displays.
		gameboard.cc:110
		In line 110, inside the movePiece function, after the piece has been moved, as seen here:
		`td->notify(*link)`
		`if (graphicsEnabled)`
		`gd->notify(*link);`
		By passing in the specific link that has been changed, TextDisplay and GraphicsDisplay will be able to change the corresponding x,y coordinates

How can we prevent direct access to internal details? For example, we may need to restrict access to the firewall's x and y coordinates as these should not be allowed to be changed. However, we still want to be able to know what the coordinates are, without exposing the underlying data structure.	**Encapsulation**
	All of the classes use encapsulation such that the private fields cannot be accessed outside of the class.
	We also used **getter and setter** functions for almost all the classes. This allows the private fields to be mutated or accessed, without directly accessing the internal details.
	
	We also did not use friends in this project as that would undermine the encapsulation, and it was not needed in any of our cases
		Firewall.h:5
		`class FireWall: public AbilityCard`
		`{`
		`  Coords coords;`
		`  public:`
		`  Coords getCoords();`
		`[...]`
		This snippet of code shows how the getter function, getCoords() can be used to return the private member, coords, without actually allowing direct access to the internal, private field.

How can we describe specific relationships between various classes? For example, we wanted to express that:
 	
Data is a Link, and a Virus is a Link.
 	
GameBoard has a TextDisplay and a GraphicsDisplay
 	
SpecialCoord owns a Coord
	Inheritance ("is a"):
	We used inheritance relationships to create a hierarchy of classes, allowing code to be reused in many cases (please see polymorphism below).
	Composition ("owns a"):
	Used to handle deletion of the classes that are owned by the base class.
	Aggregate Relationship ("has a"):
		Please see UML for detailed relationships of all classes
		
		An example of Composition:
		GamePiece.h
		`class GamePiece: public Subject {`
		`  Player *owner;`
		`  [...]`
		Here, GamePiece has a Player, which is representation of its owner

How can we maximize reusability of methods and fields within classes?	**Polymorphism & Abstractions** Subclasses can inherit methods of the base class (in this case, our base classes were usually abstract classes), and use those shared methods, thereby increasing code reusability and flexibility. Paired with using override when needed, this allowed for highly polymorphic code	An example of this is in the Link class. Since Data and Virus inherits from Link, and thus are able to use its methods isDownloaded(), getStrength(), etc (in link.h:15)

How do these techniques contribute to high cohesion and low coupling?

- We ensured relatively ***high cohesion*** by using **abstract classes**, and **inheritance** to organize our code. With concrete classes inheriting from abstract classes, we were able to keep related classes and members (functions & fields) together. For example, consider GamePiece.h, which defines the GamePiece class. BoardPiece and AbilityCards both inherit from GamePiece, which therefore keeps the game entities together, therefore guaranteeing high cohesion. Hence, this makes it easy for the controller to access and manipulate game entities, as they are all related to the GamePiece class.
- **Abstraction,** and using **shared interfaces** paired with **encapsulation & accessor/mutator methods** also helped maintain ***low coupling*** as desired because information hiding reduces the dependencies modules have on each other. For example, consider the method found in gameboard.cc:158

```
void GameBoard::battlePieces(shared_ptr<Link> linkp1, shared_ptr<Link> linkp2)
{
   cout << "BATTLE:" << endl;
   cout << "———————" << endl;
   cout << "Initiated by: " << linkp1->getDisplayName() << endl;
   . . .
   if (linkp2->getStrength() > linkp1->getStrength()) {
       downloadIdentity(linkp1, &(linkp2->getOwner()));
       linkp2->setIdentityRevealed(true);
   } else {
    ...
   }
}
```

  - As seen by this code snippet, even though battlePieces is related to the link strengths, the structure of the code means that even if the strengths of a link was altered to be 1-10 instead of 1-4 (as it is now), it would not affect this function at all!

4.0 Resilience to Change

Resilience to change was another key criteria we kept in mind as we were determining how to structure our classes. Specifically, apart from the low coupling and high cohesion designed to improve our code's flexibility, we also used the specific techniques below to accommodate changes in rules, input syntax, game specifications, new features, etc. In particular, we wanted to be able to accommodate change with minimal changes to our original code.

Enumeration for AbilityCard and LinkType

As seen in link.h:5 and abilityCard.h:8, we took advantage of enumeration; this is because enumeration centralizes the definition of all possible values; for example, the enumeration for a linkType is data and virus. Hence, in the future, if a change is required (such as adding a link type, removing a link type, changing the name of a link type), all that would need to be changed is the enumeration, and any code that checks for a specific enum. This makes the code easily modifiable for changing ability types or links, which are the most likely to vary in the future.

The ease of accommodating for changes in our abilityCards as demonstrated through our addition of 3 new abilities, as listed below (along with the changes we made to accommodate each new feature). As can be seen below, the changes were fairly minimal.

Board Boundaries, EdgeCoords and ServerPorts Are Dynamically Calculated Based on Board Size

Another technique we used was having the classes EdgeCoords, ServerPorts and keeping track of coordinates for Board Boundaries. Based on the size of the given board, and using the default step size for links, we were able to write nested for-loops to calculate the coordinates of boundaries, edge coords, and server ports upon initialization of the board. (in this case, it was an 8x8 board). The figure below shows a representation of what coordinates would be stored in the 8x8 case.

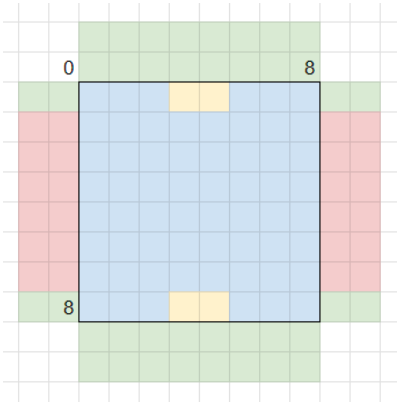

Figure 4.0 [Left]: Here, the green pieces represent the "edge pieces" — showing where pieces need to be in order to be downloaded. The yellow pieces represent the server ports, and the red pieces represent board boundaries. Obviously, depending on the maximum step size for the pieces and the dimensions of the board, these coordinates would be different, and adjust accordingly. We kept this information in the GameBoard class:

```
vector<EdgeCoord> edgeCoords;
vector<Coords> boardBoundaries;
vector<ServerPort> serverPorts;
```

The advantage of this approach is that it accommodates boards of **any size**, and **any step size** for link movements. Hence, if the board size were to be changed, all that would need to be modified would be the `const int BOARD_SIZE = 8;` value, as all the calculations for the above information are based upon this constant only, and as a result, all of the move checking functions for links also wouldn't need to be changed at all.

Handling Next Turn to Accommodate for 2+ Players

Instead of simply having our gameboard own only 2 players (as is in the original game), we instead decided to have:

```
vector<shared_ptr<Player>> players;
```

This allowed us to easily accommodate 2+ players in the future, as all we would need to do is initialize additional players with unique names, and emplace_back into our players vector. Furthermore, we also implemented the following player functions:

```
int getCurrPlayerIndex();
int getNextPlayerIndex();
int getWinnerIndex();
```

These functions all iterate through the players array, and thus would cycle through each player's turn seamlessly, even if there are 2+ players in the game. Again, this made it easy to accommodate for more players — such as having a 4 player game.

Eliminating "Magic Numbers"

We found that having magic numbers scattered throughout the code can lead to maintenance challenges, especially when these numbers need to be changed in the future. Therefore, to accommodate for this, we defined all our necessary values as public consts in the gameboard.h file. This not only made our code more readable, but it also ensured we would only need to change 1 line (the associated const definition),

instead of searching for each instance of the magic number. Our code actually does not have a single magic number (or magic "string"). All values are constants and defined in our .h file.

Dynamic Link Movement Based on Step Size

Whenever a link is moved by a player, instead of simply incrementing the coordinates in the appropriate direction by 1, we actually access the link's step size, which is a private field. This can be seen in boardPiece.h:8, int stepSize. When the function to move a link is called, the x or y coordinate is actually just increased by the stepSize. This not only allows for LinkBoost to be easily applied (and also changed; for example, if we increased LinkBoost to 3 steps instead of 2), but it also allows for the default step size to be easily modified. All we would need to do is change the step size in the ctor, which is only a 1 line change. No other files would need any changes.

5.0 Answers to Project Specification Questions

Our answers to the project specification questions did not change.

6.0 Extra Credit Features

There were 3 main extra-credit features our group attempted, as follows:

1. **"Complete the entire project, without leaks, and without explicitly managing your own memory."**

This bonus enhancement was accomplished by using smart pointers. There were no delete statements in our code because the smart pointers/shared pointers were able to automatically deallocate the memory they manage when they go out of scope. The usage of smart pointers can be seen in many parts of this project's codebase, specifically in the gameboard.h file which shows that many fields are actually arrays of smart pointers. Some challenges we ran into were:

- Deciding between std::unique_ptr, std::shared_ptr, and raw pointers. We had to think deeply about the ownership between objects, and in the end, we kept some of our raw pointers because they represented a "has a" relationship while the unique pointers were used to show "owns a" relationship.
- Ensuring no memory leaks: this required a lot of testing and debugging, especially as we were double-freeing in some instances

2. **Graphics enhancements; seen by using -graphics during compilation**

Some challenges we ran into:

- We spent a lot of time ensuring our graphics were clean and our UI was not only functional, but appealing. This required a lot of calculations to be done regarding grid sizes, piece sizes, and strings to be used to "draw" the text display regarding player information.

3. **Exception safety:basic guarantee**. We ensured that if an exception occurs, the program will be in some valid state — nothing is leaked, no corrupted data structures, all class invariants maintained

   Challenges:

   - We had to spend some time making our custom exceptions and working out the ctors and methods for such exceptions.

7.0 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?

Working in a team of 3, this project taught us the importance of having version control, and also the value in maintaining best code review practices while we were working on the project together. We found it extremely helpful to have a structured system for code-review and division of work, and found Gitlab repository to be a great asset in helping us with merging our changes together and keeping our code-base up-to-date as we made progress in different areas.

Using Gitlab to review & merge code: We leveraged Gitlab's pull request system to streamline our process for reviewing and merging code; after we have made changes or progress in a specific area of the code, we would test locally and then subsequently make the PR for the code to be reviewed by another team member. Not only did this keep us accountable for the work we did, but the extra pair of eyes also helped us catch any mistakes and/or give constructive feedback before changes were pushed into our main branch.

Using Gitlab for version control: Githlab also allowed us to easily track changes and revert any changes if necessary, which contributed towards maintaining a stable codebase. This was especially necessary in our team, as multiple changes would be made to the codebase throughout the day, and it was sometimes necessary to revert changes to previous versions when errors arose.

Clear communication & coordination: This project also taught us the importance of having an initial plan of attack, as was seen in our plan.pdf in DD1. This plan allowed us to efficiently work on different parts of the project at once, and minimize merge conflicts as well.

Hence, this project taught us the **importance of code review/best practices, version control, and communication & clear coordination** — especially when developing software in teams.

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

Although we were satisfied with the final product, this was also our first time working on a large-scale project; therefore, if given the chance to start over, we would make the following improvements:

1. Improve priorities alignment

While working on the project, we found that our group members had different priorities when it came to code — for example, one of our group members prioritized code documentation and having the best quality code, while other members were more focused on just producing "code that works". This led to some conflicts where members did not agree on the standard of code, which took time to resolve. Therefore, if we had the chance to start over, it would be beneficial to agree on a general criteria of code documentation and quality before we started coding.

2. Refine plan to account for blockers

While working on this project, we often found certain tasks were "blocked" as we were waiting on other items to be completed; for example, before we were able to start progress on ability cards, we had to first

complete the basic movement functionalities. Therefore, if we were to start over, we would've created a timeline that showed task dependencies, thereby showing us what tasks to first focus on.

8.0 Conclusion

In conclusion, the design choices outlined in this document represent our thoughtful and deliberate approach to developing raiinet in a way that is flexible towards change, while also maintaining high cohesion and low coupling. We have used many of the OOP concepts taught in lecture to create robust and maintainable code while following best practices and strong coding principles. By leveraging abstraction, encapsulation, inheritance, Observer design patterns, and MVC, we are confident that our codebase not only accomplishes the task at hand, but does it in an elegant and efficient way that showcases everything we have learned this semester! ☺