University of Waterloo

CS246 F2019 Final Project

Game of RAIInet Final Report

Yushuo Han, Yiwen Mei, Yiheng Ding

December 3rd, 2019

An Overview

The game of RAIInet is composed of a main class, *Board,* which represents the board of the game. The *Board* owns 2 *Players*, 64 *Cells* stored in a 2D-vectors, and two display classes *TextDisplay* and *GraphicsDisplay.* Each *Player* owns its 8 *Pieces*, which is provided to the *Board* (so the *Board* has both *Player*'s *Piece*s). Each *Cell* has a *Piece*, and if the *Cell* does not hold a *Piece*, it will then hold a null pointer.

Each time a *Player* calls to move of a *Piece*, the action is handled by the *Board*, which checks the availability of the *Cells* that it owns and are available around. If the *Cell* has the *Player's* own Piece, then the move is forbidden. If the *Cell* has the opponent's *Piece*, then the battle function is called, which is also handled by the *Board*. In the battle, the winner is determined by the *Pieces'* strength, and the loser is downloaded by the *Board* by removing the *Piece* from the *Cell*, set the *Piece's* alive flag to false, and increment the *Player's* fields that counts amount of viruses or data downloaded. The *Board* also checks the coordinates of the potentially destination *Cell*, and if they are out of the *Board*, the move is forbidden. The *Board* also handles the situations where the *Piece* is moved across the boundaries and ports, and downloads the *Piece* according to which port or boundary they passes.

All Abilities are handled by the *Board*. The *Board* takes the ability usage command, and according to which ability the Player uses, it modifies the target *Piece's* attributes.

Our program's display, although not using the Observer Pattern in the strictest term, uses the essence of the pattern, which is notifying the observer classes *TextDisplay* and *GraphicsDisplay.* The display switches concealed and revealed pieces according to each player's turn.

Our program provides users with additional abilities, namely move Diagonally, hint to Attack, and hint to Defend. Also, the program implicitly allocates memory using shared_ptrs. In addition, our program achieves low coupling and considerably high cohesion.

An Updated UML

The updated UML describing the final structure of the program can be accessed via the following link:
https://drive.google.com/file/d/13useQZoqO-3morE4bucqDP-fvWMybK9l/view?usp=sharing

Design

Class *TextDisplay* and *GraphicsDisplay* were implemented with an essence of observer pattern. Although both classes are not subclasses of *Observer*, and we have implemented the *notify()* function for both classes, and these functions are called by our class Board (similar to a subclass of *Subject*) once a change

is made, such as a piece is moved, or an ability is used. We did not use the observer pattern because we decided not to make the *Cell* class as an observer. Each cell, in our program, should only be a holder, or a location, for a piece. The interactions of pieces should be handled by the board according to information of the pieces, but not the cell. Thus, we decided cells do not need to be observers, and hence the necessity of using the Observer Pattern is minimal with only Class *TextDisplay* and *GraphicsDisplay*. Instead of having an observer vector and having the subject to notify each of them after every modification, we explicitly notified each *TextDisplay* and *GraphicsDisplay*, which is similar to the Observer Pattern.

We have deliberately designed our classes so that they reflect the relationship between objects in an actual game. Our *Board* owns class *Player,* and *Cell*, while each *Player* owns his/her multiple *Piece*s. The board, consequently, has the *Piece*s. Also, each player has multiple abilities, and since each ability does not hold any information (their only purpose is to change the piece's properties when they are applied to the piece), they are simply stored as a vector of strings. This is also the reason why we did not implement the *Ability* class. Moreover, in this way we have achieved low coupling and a considerable degree of cohesion in our actual program structure, which will be discussed in the next section.

Our final program's design has some differences to our initial design. Besides adding many flags, getter and setter functions for the information-holding class *Cell, Piece,* and *Player,* one of the major modifications made is that we have abandoned using the Decorator Pattern for our *Player* class. There are two reasons. First, in our initial design, we attempted to use the Decorator Pattern to overload functions in class *Player*. Later into the implementation, we have switched to use the *Player* class to store player's information, and all functions involving piece interactions were implemented in the *Board* class. In this case, in order to overload functions for piece interactions, we would need to have the abilities as concrete decorators of the class *Board* instead of class *Player*. However, the abilities are properties of each player instead of the *Board.* Thus, using Decorator Pattern is not logically reasonable. Also, the function that we need to use for each ability varies. Instead of only using and overloading the *movePiece()* function that we have imagined in our initial design, we would need to change fields of various objects in different classes. Some of the abilities (such as our Hint to Attack and Hint to Defend) need to directly move a piece. In this case, the program using the Decorator Pattern and overloading functions will be less clear than our current implementation, which directly handles each ability in class *Board.*

Resilience to Change

1. Support to various changes in program specifications
   Since our program does not utilize the Decorator Pattern, adding a new ability option in the game would require the programmer to modify both the *Board* class and the *main* function. This would be rather simple since the *movePiece()* function, although lengthy and complex, already handles all the possible moves. Thus, all new abilities that will be applied to a piece would only need to change the *useAbility()* function in *Board*, in which the programmer would need to modify the attributes of the piece indicated. The not-so-convenient part would be the *main* function, in which the programmer would need to add new exemptions in the exception-handling to allow the ability

to be recognized and used. Compared to the changes required if the Decorator Pattern is used, although our design would not give the programmer the convenience to simply add a new ability class that inherits from the superclass, the programmer would also only need to make minimal changes in our *Board* class and *main* function, which makes our program considerably resilient to changes.

If we need to add players to the game, a great change of the program would be required. Not only two more player objects would be initialized and owned by the *Board*, but we also need to modify the *Player* constructor so that the *Piece*s it initializes has the correct position according to the player name. For example, if player3 is initialized, we would need to initialize all new *Piece*s to the left most cells of the Board. Additionally, the middle left and middle right cells should be initialized as ports as well.

2. Coupling and Cohesion
   The structure of our program is of low coupling and considerably high cohesion.

   Class *Cell* and *Piece* are classes that hold basic information, and *Cell* has a *Piece* if the *Piece* is on the cell. Class *Player* stores information of each of the two players and their abilities, and keeps track of the parameters to determine the winner. Also, each *Player* owns its *Pieces*. Lastly, our *Board* class owns only the objects that it absolutely needs: two *Players*, some *Cells*, and the *TextDisplay* and *GraphicsDisplay*. In this case, all of our classes own other classes only if they absolutely have to, and other than the ownership, they pass information only through functions and access the info through return values. Thus, the structure of the program is of low coupling.

   Most of the classes qualify the criterion for high cohesion. Class *Cell, Piece,* and *Player* only hold information, and each of their attributes is unrelated to others, so elements in the objects do not pass data to each other. They do not handle any player moves or abilities, so each of the classes only perform exactly one task: to hold information. Similarly, class *TextDisplay* and *GraphicsDisplay* are only responsible for display. *GraphicsDisplay* class thus does not have any fields to store information or handle player interactions, and *TextDisplay* only has minimal fields that need to store temporary information for *Piece*s. In general, our display classes are only performing the display task. The only class that has lower cohesion than others is our *Board* class, which not only handles player interactions, but also handles their decisions of using abilities. We are aware of this disadvantage, but since the abilities are closely related to the *movePiece()* functions (player interactions), and each ability does not need to contain any extra information, we thus did not make another class for ability to increase our design's cohesiveness. In conclusion, our program is of considerably high cohesion.

Answers to Questions from Project

Note: parts of our answers are different to what we have submitted on Due Date 1. We have indicated the deletions using strikethrough, and bolded the additions.

- *Question 1*

  To display both player1's board and player2's board after each player's move, we would need to have two pointers to the TextDisplay class and two pointers to GraphicDisplay classes.

```
class Board{
        ...
        TextDisplayPtr td1; // responsible for the text display for player1's board
        ~~TextDisplayPtr td2; // responsible for the text display for player2's board~~
        GraphicDisplayPtr gd1; // responsible for the graphic display for player1's board
        GraphicDisplayPtr gd2; // responsible for the graphic display for player2's board
};

class TextDisplay{
        ~~Board* theBoard;~~
        ~~PlayerPtr player1;~~
        ~~PlayerPtr player2;~~
        bool player1First;
        public:
        void playerFlip();
};
```

  After each move, the player's class will invoke the move function in the Piece class, which, through its "board" field, will update each of td1, ~~td2*~~, gd1, gd2 of the board. During the update, the char vector in TextDisplay is modified accordingly. For GraphicsDisplay, rectangles representing the modified cell will be re-drawn.

  Note that we implemented our overloaded <<operator for TextDisplay that depends on the flag *player1First*. For TextDisplay, we changed our answer from using two TextDisplayPtr to a single one. After each move / usage of ability, we first display the view from player1's view by hiding player2's info. Then we would call the playerFlip() function from *main* function to change the flag *player1First* to false, then print the board again. The board will subsequently hide player1's info and display player2's info, and in this way it will display the board from player2's view.

  GraphicsDisplay class can be implemented and programmed in a similar fashion. **Since GraphicsDisplay needs to initialize an Xwindow, the Board must own two GraphicsDisplayPtr objects to display the view entirely different. In GraphicsDisplay, we have implemented a flag and a *notify()* function similar to TextDisplay so that the views will flip.**

  Note we deleted PlayerPtr and Board pointer because all the information should be accessible from the board pointer that is passed into the *notify()* function.

- *Question 2*

  Decorator pattern would be utilized to make adding new features convenient. The parent, abstract class would be our TopPlayer class, in which we have all of our function's declaration set to pure virtual. The BasicPlayer class is a derived class of TopPlayer class, which has concrete implementation of a basic player without any abilities used, and contains the standard implementation of a player's move. The abstract decorator is our Ability class, in which we have all our ability function's set to pure virtual. Each of the five required basic abilities, and the new 3 abilities we would provide below, would be implemented as concrete decorators, which are derived classes of the Ability abstract decorator, and will contain concrete implementation of each ability's overloaded function. After each ability card is used, we will continue to call the normal movePiece() and other functions that is only defined in BasicPlayer class, and will not call any decorator functions. In the following, we will provide a brief description of each concrete ability's implementation. The first three abilities are our new abilities:

  - *Hint to attack*: we will move one of the player's link in attempt to attack the opponent's nearest link. To implement this, for example if we want to help player1 to attack, then we would calculate the distance from each of player1's link to the closest opponent's link to that player1's link, and move the link with the shortest distance to the opponent's link one cell closer to opponent's link. We implement this feature by, instead of overriding the movePiece() function of the decorator, adding a new function called hintAttackMovePiece() function in the concrete decorator.
  - *Hint to defend*: we will move one of the player's link in attempt to avoid attack from the opponent's nearest link. This can be implemented similar to that of our "hint to attack" ability, also by adding a new function called hintDefendMovePiece() function to the concrete decorator.
  - *Walk diagonal*: after this feature is activated on a link, that link can only move diagonally (i.e 4 possible directions are up-left, up-right, down-left, down-right). To implement this, we would add the diagonalMovePiece() function of the decorator so that it consumes two strings: the link's name and one of the the four directions, and move the link accordingly.
  - *Scan*: we would add a scan() function to loop through the Piece vector in the player's field, find the link to be revealed, and set its isRevealed field to be true. In the TextDisplay and GraphicDisplay class, we would display the link's type if it is revealed.
  - *Polarize*: we would add a polarize() function in the Polarize concrete decorator class to modify the link's type to the opposite type (virus to data, data to virus), using the available setter function setType().
  - *Link Boost*: to implement this, we would add a linkBoostMovePiece() function to the decorator so that the link is moved two cells/units of the indicated direction.
  - *Firewall*: For player1, we would implement this by setting the "firewallOne" field of the cell to be true. For player2, we would implement this by setting the "firewallSecond" field of the cell to be true. This will be implemented in the new addFirewall() function in the Firewall concrete decorator class, which will search through the board pointer in the player's pieces, find the cell, and add the firewall.

- *Download*: for example, we want to download player2's link. If it is a data, then we would add1 to player1's downloaded data field. Conversely, if it is a virus, then we would add1 to player1's downloaded virus field. Also, we would reduce player2's aliveNum parameter by 1, and set that link's isAlive field to be false, so the link will not be displayed. This will be implemented in the download() function, which is specifically implemented in the Download concrete decorator class.

- *Question 3*

  First of all, we would change the size of the gameboard. Although it is a plus shape, we will treat it as a 10 x 10 square board, but we will not print the four 1x1 squares on the corner. We will add 2 more fields (portX, portY) to indicate server ports' position in the Player class. We will add two more fields into the cell: firewallOne (for player1) and firewalTwo (for player2), so that we know which player setup the firewall. Then in the move function, we will add if conditions to see if any link is trying to escape off the edge belonging to the opponent directly adjacent to them. If yes, we will let the player download that link. Then, the winning condition will be unchanged: whoever downloaded 4 data win the game, so there is nothing need to change. The losing condition (if we have more than 2 players) will result in a quite big change: we have to remove the links, firewall, and the server ports. Let's call the loser Player4. We will loop through each Player4's piece and set all the corresponding cells to be unoccupied, and its firewall(s) to be unset. Then we will reset the port position on the board and update the TextDisplay and GraphicDisplay. Then we will unlink Player4 to the TextDisplay and GraphicDisplay, and finally we will delete the Player4 object.

Extra Credit Features

1. Smart pointers

   Throughout the entire project, we used *std::share_ptr* from the <*memory*> module to achieve the goal of completing the program without leaks with the premise of not explicitly allocating any memory. The challenge we encountered when using shared pointers was simply because of our unfamiliarity to the concept and the lack of practical experience of using it. We managed to use it by first implementing our program using solely raw pointers to ensure our programming efficiency to achieve a correct and complete game. After our program was bug-free and not leaking memory, we introduced the *share_ptr* by defining them using *typedef*, and simply replacing most raw pointers in our original program to the new type and removing all explicit memory allocation.
2. Extra Features (3 self-invented special abilities)

   In addition to the five required abilities, we designed and implemented 3 new abilities, namely diagonal, hint to attack, and hint to defend.

   a. Diagonal

      Players can include this ability to their machinery by including *Z* in the command line argument (i.e. *-ability1 ZZDDP*). Players can use this ability by indicating which piece

(belongs to the player, not the opponent) they want to apply this ability to (i.e. *ability 1 a*). When this ability was applied to a piece, the piece can move diagonally or cardinally (8 directions). If ability LinkBoost and Diagonal are both applied to the piece, then the piece can either move diagonally ($\sqrt{2}$ units each move), or move cardinally (2 units each move). This extra ability is relatively the easiest extra abilities out of the 3, since we only need to implement our *movePiece()* function in class *Board* to move in 4 new directions. The implementation was similar to the implementation we had for the 4 cardinal directions, with simple modifications on checking diagonally-spaced cells instead of the cells that are cardinally next to the Piece.

b. Hint to attack
Players can include this ability by adding *A* in the command line argument (i.e. *-ability1 AAZDP*). Players can use this ability directly (i.e. *ability 1*). After applying this ability, the program will select among the Player's available links and choose the link that requires the least number of moves to download one of the opponent's data (the link chosen must have higher or equal strength to the opponent's data). It was a great challenge to implement the function (*Step()* in class *Board*) that calculates the number of moves required for a piece to download the opponent's data. We applied the Markov Chain Monte Carlo method for the step calculation. Note that it was of great difficulty to implement this calculation with the premise that the algorithm halts when the potential moves require the piece to move out of the boundary of the board. After calculating the steps required for each piece, the program will automatically move the piece that requires the lowest number of moves to attack.

c. Hint to defend
Players can include this ability by adding *G* in the command line argument (i.e. *-ability1 GGAZD*). Players can use this ability by indicating which piece (belongs to the player, not the opponent) they want to apply this ability to (i.e. *ability 1 a*). When this ability is applied to a piece, the program will try to move the piece away from the opponent's pieces that have higher or equal strength than the indicated piece. Note that it is only sensible for a player to use this ability to a data, but the player can also apply this ability to a virus. Note that the program needs to calculate the number of moves it will take each opponent's link to attack the piece that this ability is applied to. To implement this calculation, we used the Greedy Algorithm and the Markov Chain Monte Carlo method. Note we also faced the difficulty to calculate the number of moves with the premise that the algorithm halts when the potential moves require the piece to move out of the boundary of the board. Lots of if-statements were added to handle this restriction.

Answer to Final Questions:

1. Important lessons for team programming.
    a. The importance of code review

It took us a considerable amount of time to debug mainly because of the fact that we lack code review for each team member's code before compiling them together. For example, the getter function *hasFirewallTwo()* and setter function *setPort2()* were carelessly false implemented to return field *firewallOne* and set *port1*. These simple mistakes can be easily discovered and quickly fixed during code review, but it took us exponential time to discover these simple errors when debugging the entire program.

b. The importance of a correct allocation of work
   During implementation, we discovered that there were some flaws in our initial distribution of work (classes to program) in terms of the classes' relatedness. For instance, the class *Board* and function *main* should be programmed by the same person as there are lots of exception-handling details that are closely related to the communication between the thrower and the catcher. This was one of the reasons that we re-distributed of our tasks.

c. Teamwork revision
   The team should debug together so that the person programmed the class to be debugged is aware of the potential effect of each modification on the entire program. For example, when one of our team members is debugging the *movePiece()* function in the *Board* class, he was not responsible for programming the class and hence was unaware of the fact that the *firewallDetect()* function already downloaded the piece if the piece is a virus and steps on the firewall. Consequently, he programmed the function so that the function re-downloads the piece that was already downloaded by the *firewallDetect()* function. This mistake could be easily avoided if our team debugs together.

d. The importance of convenient communication between team members
   Our team was very cohesive and our members are very close together, which played a great role in our high programming efficiency as a team.

2. Possible areas for improvement in future:
   a. Set up version control (i.e. git) for remote working and version difference comparison.
      We decided not to set up and utilize version control mainly because our team members are very close and are mostly available during the project weeks. However, we soon realized the difficulties of merging modifications from various team members to a single file. At times, some of our team members continued working on old-versioned files and consequently encountered great difficulties when merging their changes to others' modifications. In future courses, we would definitely setup git for version control.
   b. Deliberately design the entire program before implementation
      Although we have already spent a considerable amount of time designing the structure of our classes and their relationships before we started programming, we would be more deliberate on initial design of the entire program if we had a second chance. During the design, although we carefully considered how the implementation should be carried out, we lacked considerations for our code's resilience to changes and did not envision the challenges that we would face if the program is programmed in our current design.
   c. More structured exception-handling.

We did not design a well-structured exception-handling. Each exception should be thrown in class *Board*, and caught by the *main* function. Although most of our code's exception-handling was done in this way, our *main* function also handles invalid inputs and prints to cerr stream. This consequently leads to some of the edge cases of input not being handled by either *main* or functions in class *Board*, and so we had to handle lots of edges cases separately. On the contrary, some invalid inputs were handled by both *main* function and *Board* class. This could be avoided if our exception-handling mechanism was well-structured.