RAIlnet report

-Introduction

Our design is simulating real life board game which has a board with many cells and either two or four players participate in the game. In our code, the board is the class *Grid*, and a single cell is the class *Link*. The class link owns all the information a cell should have such as position, data type, name, and strength, which are contained in the class *Info*. One single player is the class *Player*, and each player can know some specific information. As for those information like remaining abilities or the number of Data/Virus they downloaded will be included in the class *PublicInfo*, For computer itself, these parts are enough for playing RAIlnet. But as game players, they have to actually see the board and links through text or graph. Thus, we introduce the class *TextDisplay* and *GraphDisplay* to show the information to real game players. If each class player owns a class TextDisplay, we can successfully simulate eyes of a player. For the graph, if we let the grid to own a class GraphDisplay, we can also simulate the graph of the board to all players. The reason why a link moves to another place, players first see through eyes and then receive information. To simulate it we use the *Observer Pattern*. We want the class link to be the subject and the class Player and PublicInfo to be observer. Plus: For playing model, the first flag you must enter a number 2 or 4 to decide which model you want to play.

-Overview

In our main.cc, we only want to use methods in the class Grid. Besides, we separate the main to two parts, one part is initialization of the game(simply consumed command line argument), and the process of the game(reading in from user command). Therefore, Grid can directly modify everything in this project. Class Grid owns a vector of vector of class Link. Each Link contains a struct Info. And Info contains the name, position, Type(Type::Empty, Type::Data, Type::Port, Type::Virus), strength, and VisibleData(whether can see his type) and VisibleStrength(whether can see his strength). It means Grid contains all the Links with all information. Also, Grid contains a vector of class Player(called player_list), and a vector of class PublicInfo(called public_list). For class Player, each class Player contains a vector of vector of struct Info. For example Player[0][2] means player1 's second link' s information. If visibleData and VisibleStrength are true, it means such player can see

player1’ s second link’ s information. In the Grid, player_list[1][1][0-8] will be eight info, and all set visibility to be true to represent the second player knows all information of the second player. And player_list[1][0][0-8] will be all invisible at the beginning of the game to represent player2 knows nothing about player1’ s links. In struct Info, we do not really put the position in class Player. Since we want the player_list to contain all information like an information system. player enters the name of a link, player_list tell you whether you can see the type or strength. Moreover, each class Player owns a class TextDisplay. Public_list is much easier to understand, for example, public_list[1] represents almost all information of player 2 which is public to all players, such as how many Data downloaded, how many abilities left. PublicInfo has members int Download_V, int Download_D. And a vector of integers to represent the number of remaining ability (called ability_list), also contains a integer called turn to represent which player’ s info is there. Since ability_list is private, it was very easy to set a restriction using an integer turn to prevent any other player from visiting. Back to our Observer pattern, we have our abstract class subject and observer, and the class Link inherits class subject, and Player and PublicInfo(Two information collection) to inherit the class observer. For the class subject it has virtual method like *notify* and *attach*(which can attach observer) with a vector of observer to be private member. And observer to be has a virtual method *getnotified* to receive the message from subject and do the change. We also have a struct called state, state contains a statetype(enum class StateType: FireWall, Add, Leave, Show, Downloaded, Init, Reverse), an int downloader, and an int showsto. The StateType indicates what kind of message we want observers to receive, such as Add(link (I,j) tells all players a link shows up here), and Reverse(link(I,j) tells all players his type has been reversed). If we send message Downloaded to observers, only player which indicated in the state downloader can download this link. Thus, those two integers in the state can indicate certain player you want to tell and other observers will not know it. The Grid contains a lot of independent method like a helper function. For example, we have move and useability, these two parts remain independently, those will be discussed how to be implemented in Design part. Also, we have some very useful helpers like tools of the whole project, such as toDownload, toShow, toAdd, toLeave. These contains notify part and be very friendly for users to add some new features, they just need to know, if you want the link shows up in (i,j) on textdisplay, just use toAdd(I,j). If you want the link disappear(be dowloaded), you use toLeave(i,j) and the link will be set to be empty. If you use toDownload(i,j,downloader), the link will be downloaded by the downloader. And then use

toLeave, it produces a battle. Also, these tools are simply implemented by notify and getnotified between subject and observers. And toShow is important method to ensure someone would see some specific link type. It' s very similar to others, to set member showsto to be the one you want to show, then player will change his visibility to the exact link(we can find it since the link got a name). For textdisplay, when player get some message like toAdd or toLeave, they will modify its textdisplay. And everytime a player enters command board, it will print in grid, with specific player' s publicInfo, and class Player and its textdisplay. As for graph, it simply prints blank links and colour them up by the visibility in specific player, which will be disused in the Design part.

-Design

In main.cc, we want to use only methods in the class Grid. Thus, the class Grid will certainly seems more sophisticated than others, including some important helper functions. As in introduction, we mentioned we use the observer pattern. Some things are very significant in the whole program:

1. How to initial a game?

   Since the program optionally need some flag like -links1 D1D2D3D4V1V2V3V4, we first consider the flag is a signal to change specific parts like link order or ability order on the default. Therefore, we begin to implement the default model. First thing is to fill the grid with links(8*8 grid has 64 links). We first setup links as player number you provided to us. And they will be all empty. Then, to set default links of players. That is, by using *set(char name, int owner, int strength, Type t)* to set each and every link with certain Info contains name such as 'a', Type such as Type::Data or Type::port, its strength such as 1, its owner such as 0 (for player1). If we have two players we have to set 20 links, and 40 links if we have 4 players. I know it may seems stupid, but it is a direct, and efficient way, also avoid mistakes when doing loop. For other links who remains empty, we have them to be Type::Empty with name '.' and strength -1, which are really not important except Type::Empty to be set. If there is flag. We just need to modify some info of specific link. Since the flag follows an order, it becomes very easy to change its type and strength, in Grid, we also provide *setlink(int turn, string order)* to change as flag asks to set links of certain player.How to find that certain player? We just need to search the string" links" or "abilities" to know whether we need to change a link order or ability order. Then, we search for numbers and get them in a string which is not hard to find specific player.

Also, there are many restrictions to check before making the change, like whether the string you provided is legal (4 D, 4 V, different strength). Same as abilities. We provided two methods for restriction *linkdefault(string order) and abilitydefault(string order).* If you provided a string which is not fits the game restriction (like you want some ability not exits, or provided two D2, or you want ability more than 5). We just don't need to set links or abilities, and your ability and links become default. Finally, we initial the board. in main.cc, we first use method *init(int num)* in Grid to set a default model grid and then loop the flag to use setlink(int turn, string order) and setplayerability(int turn, string order) to modify the default grid.

2. How to make a link independent from a single chess grid to move, to battle?

We consider that movement and battle will be all achieved by using the method in grid *move(char name, Direction, dir, int turn) .* To make link different from a single chess grid, we just need their info contains different types like Type::Data, Type::Empty, Type::Virus and Type::Port. For convenience, we also add a private member in Link, owner(Empty link's owner is -1). Therefore, we make them independent from a single chess grid. For moving action, in method move, we firstly need tell whether this link is yours and what position it is. We simply look through the all links and find out whether there is such link and get it position using method in Grid *bool locate(char name, int& ii , int &jj, int row, int col),* and let ii and jj be the position of the link(if we find it) and then we tell if it is your link by simply comparing its owner and turn. If you can move it, we will do the moving and battle. And things will be easier to solve. We will use dir and whether the link has an AbilityType like LinkBoost to tell exact position it will be. In this case, we just need to deal with two links. We compare their strength and certainly we know who starts the battle to solve equal strength problem. After battle, we know which link will be leaving the board and which link should keep staying in the board. For the link lost in battle, he just need to leave the board. For the link won in the battle, it may remain in the same place, which is good. But if he changes his place, we will need to pass the old Info to the new one but change the info position, and owner will be correct. Since we have methods toDownload, toLeave, toShow, mentioned before. Such implement is easy to achieve.

3. How to make each player to know exactly what he should know during the whole game?

Since, each class Player contains all the links' real information. We just need to notify them to set their info to be visible when a link is battling or use some ability like scan or Download. That

is what **observer pattern** does. Also, when someone download or using abilities, Observer PublicInfo will receive the information and make the change.

4. How to use ability? and some important point about firewall.

We use the method in Grid, *void useability(int turn, bool& b, std::string s, int ld)* to achieve this. The Boolean b it consumed will be set to be false if the player uses the ability successfully, and it also ensures that player will not use ability again in his turn. And when each turn switches, Boolean b will be true again(in main.cc). This method is the second important method in this project (First one is move). It certainly contains a lot of restrictions for whether the player can use it correctly. Such as if you want to use Download, you cannot download your own link. For different abilities, there will be different restriction, for firewall, we decide that you can set your firewall to recover others(small chance but still a situation, and there is no direct words like you cannot set firewall upon others firewall in A5). In main.cc we pass parameter like "a" or a position like "12" (only position is required for firewall) to this method (parameter s). And we then for certain ability we can use some certain method to achieve what effect you want(such as ability 8 bomb will be mentioned later). For firewall, it is very easy to add in the textdisplay even if you want certain player to see it simply passing a state with StateType::FireWall and state member showsto = 1, the player 2 will receive the message and print firewall on textdisplay. Furthermore, the firewallowner in Link is essential when the link leaves or moves to the firewall. Calling the method toLeave(int i, int j) can determine the ownership of a firewall at i, j based on the firewallowner. In consequence, the textdisplay will be notified so that the correct character (in this case, '#' represents firewall) will be shown at the right spot.

5. When there are four players, winning and losing are different.

In the two-player case, there must be one player wins and the other one loses, which is simply checking the number of Data and Virus they downloaded after each move. However, when there are 4 players, if a player loses his links and his ports will be gone by checking the owner and setting links to be Type::Empty(including Port). However, the game will not end. Hence, in main.cc, we will check the member stillin in class Player to ensure the turn is correct(it will not let a loser keep moving). Also, if three men lose, the last player is winner. They are all well discussed in methods in Grid:

Bool out(int turn) will check whether the player is out, it will check the Downloaded number and

make the judge. Void lose() will clear all links of a player and set the stillin to be false. Bool won(int & winner) will give us a winner if it returns true.

6. How does graph works?

   Since the Grid owns a class GraphDisplay. If you provide a flag -graphics, we will call constructor of GraphDisplay. And use method in Grid *gprint(int turn),* it first print out all links in the current board with black colour, and then fill the colour by checking a specific class Player to find out what current player should know and what shouldn't know by simply checking whether its Type and strength visible or not. And then we have a list of names he should know the Type and Strength(It is impossible to know only Data but not knowing the strength in Graph, because if you only know the Type, it must be downloaded by other players and it is gone). Then we have a list of names he should know, we go through all the links and colour them up depending on his Type and show his strength.

7. The difference between DD1 and DD2 design:

Both designs are very much similar. We keep the thought to make the class Player to contains all true information of all links, and Textdisplay will be the eyes of the class Player, it is owned by Player, and Player receives message from links and make change to themselves and his own textdisplay. The difference is that we want the graph to be very similar to Textdisplay, and it originally designed to be owned by Player. However, we find it really unnecessary because the Grid contains all the information, we can simply print them out and colour them up by checking information of a certain player (As for textdisplay, when we find it not necessary to let each player carries one textdisplay, we have already finished this project, but such design will still have many possibilities when adding some new abilities, we decide not to change it).However, each player cannot own a class Graphdisplay. Since the graph is always showed. If there are 4 GraphDisplay, we will have to have 4 windows. And that is prohibitive.

-Resilience to Change

1. to change the board size

When we finish doing the project, we realized that we made an 8*10 grid instead of a 10*10 grid. I'd like to mention it since this is a good example to illustrate the flexibility of the program. We used only half an hour to change the board size. There only need to be change when the game

initializing. We just need to push two more vector of links into class Grid. And change its position when you first set them one by one(remember that direct but not smart way?). Also, we need to change the restriction of edge to avoid segment fault. Last, extend the row in the graph. Actually, after this accident, We think we can change to whatever shape we want, just three step above. And also, if you want to change the initial link place, it is very easy to complete by just set a different position.

2. to add ability

We separate abilities to be two parts, one part is instant like a link just bombed or downloaded, the other part is like a buff along with the link like boost a link. So for some instant ability, we don't even to name a type in enum class AbilityType. We can use our basic tools methods to create a new instant ability(using toLeave, toDownload, and so on). Like the new ability we introduced, you just need to add some restriction of using abilities in the method useability. Or you can make a new tool like toSetFireWall using our basic tool methods. Such instant abilities, we put them in the method useability and it is very easy to implement. However, for those ability taken by the moving link. We first introduce it in the enum class AbilityType(Empty, LinkBoost, Vengeance). If you want to add something like this, I believe you have to implement it in the method move in Grid. Because such ability mostly has their effect when they are moving. So, all you need to do is give a buff(private member of link, AbilityType buff) to link in the useability method and add some using restriction before use it to prevent you fail to use it but your ability minus one(like instant ability). Then, another thing to do is to set effect in the move method. Such as when the link is already boosted, just make the distance he move plus one. Or like vengeance(will mention below), you just need to add some restrictions when links are battling. In a word, to add an ability you just need to give restriction in useability method and add the effect in whether useability or move methods depending on the ability type(instant or along with the link).

3. add more players

Since we have parameter for playernum to decide 2 or 4 players, why certainly able to have more players like 5, 7, 9, 13. How to implement it? Quiet easy. Since we can change the board size, we first change it to a larger size board. Then, use that direct set link way to set other players links with owner 5,7 , 10…… Last change the main's playernum to be what we want! And when initial the grid, we pass the playernum before, and it just resize the board, and resize the player_list size. Not

hard to do it. But the edge restriction will be a problem if you want the shape be very strange.

4. the cohesion and coupling of the design

Our project mainly devided to four part: class Grid for moving and useability, Player to handle the separate of links and information hiding(such as player 1 knows player2’ s links type but player3 know nothing). And PublicInfo to contain abilities. And link becomes a subject to connect Player and PublicInfo. And textdisplay is a real easy part since it is controlled by player and only contains some char. We don’ t mention graph to really be a part of the project since it just prints links and use player’ information. If the graph happens to be a mistake, it must be the problem of Player and Link, since the graph actually contain nothing, it is more like to be a tool to print. By the way, we put useability and move in the Grid but actually they are different part. We will not go tracing the move method when there is a mistake happened when you use an ability. And these part will be independent to have a low coupling. When moving a link, the grid takes care of the links, and links send messages to Player. Player just shows the information he has. If happens a mistake, you check the link and the print of Player and you know what part is wrong. When using abilities, if such thing like ability did not minus one, you know it must be the mistake of PublicInfo. These parts works independently like group members working. Sometimes it was the message it sent goes wrong, but if everyone else work perfectly, you know it not about message, that is about yourself. Same for these parts. Also, we want our code to be high cohesion as well. That means though we have a lot of helper functions in each class, we seldomly use it outside the class except for some main method like move and useability in grid. Setlink in the link. It was like each class makes an important method using helpers inside the class and send it to other classes to use. Like judging win or lose, you can let PublicInfo to do the job and return a value back to grid. Also, we find it very efficient when fixing mistakes(but we do need to open some files to trace the problem), we can usually trace the problem to an independent class and tracing the method to find the problem of some helper functions or logic mistake. For main.cc, we only use the method in Grid to ensure Grid is the main class of this project and lower down the risk to find mistake other than the class Grid, we believe it would cost much time and being very hard to correct. That is what we learn in A4q4. Also, we use main to take care of switching turns. And these parts start working with the beginning of receiving an integer turn, that makes what we want to separate players.

-Answer to Questions form project

1. four-player model: we discussed even more players above in Resilience to Change

2. have two displays : each class player has a textdisplay, then as turn switches, textdisplay actually switches.(Except firewall, there actually not nothing different in textdisplay). And we have four textdisplay by the way.

3. adding abilities easily : we discussed above in Resilience to Change

-Extra Credit Features

1. 4-player model: If we decide we should contain two model(2-player, 4player), we have to make each class like Grid, PublicInfo, Player to contain the playernum. It is very important deciding how many link info a player carries, how many class PublicInfo a Grid should own, and of course the shape of graph and textdisplay. For method when initializing a grid(and initialize the class Player), we must need a playernum to put links and decide whether to put links of Player3 and Player4.

To 4-player model, we still have some important thing to solve:

~1. Winning and losing: Discussed in design part.

~2. Visibility of information: What a player should know is all contained in the class Player. It might occur such situation, Player 1 downloads A: D2 of Player 2, player 3 knows Player 1 downloads D but without strength. He knows A is D, thus in the text for Player 3, it shows A: D?. It is because when we send message from link when being downloaded, it send message like everyone sees my type but not my strength, and use method toShow to show the type and strength to the battle player.(Simply changing the Boolean in Info of class Player, there are members in Info: visibleData, visibleStrength)

~3. When a player loses, his links and port must be gone:

If a player lose, we go through all the links and set the links with owner of him to be Type::Empty, and name to be '.' (simply using method toLeave). But we still prevent others to know the type and strength of them.

2. 3 more abilities: we introduce those 3 abilities which can only use on certain link with certain type and strength. Because, we want to make every link be independent and unique form other links and raise the connection between players, making them start to look after or more focusing on others depending on his position.

~1. Ability 6: Vengeance: such ability can only use on the link V1 or D1 of his own. And the link must have no boosted. Then, V1 and D1 will beat V4 or D4.

We introduce this since V1 and D1 are so weak. D1 may only stay at home and V1 just keep moving ahead, which is not interesting. We add this ability by adding AbilityType::Vengeance to AbilityType. And the link will carry such ability. We also need to add some restrictions like the AbilityType must be AbilityType::Empty, and adding some logical stuff in method *move* when judging who should win the battle. And such logic will not be simple a>b or b<=a, since 1 < 4 but V1 can still beat D4. It needs to judge if the link carries the AbilityType::Vengeance and make sure it step in to the right zone(zone for battle win and battle lose).

~2. Ability 7: Hero: such ability can only use on the link D2 of his own. And the next player will download D2 and the number of Virus we download minus 1. Such ability raises the relation between the player and his next player, there begin to raise friendly action between players but not simply hurting each other, and raise the complex of strategy. And D2 is like a hero who might save the service port. This ability is easy to implement, but some restrictions also need to take care of, such as if we have 0 Virus downloaded, we can' t minus to -1. And the turn for next player will not cause a segment fault. Others are simple download and writing like a helper in PublicInfo to decrease the Virus downloaded.

~3. Ability 8: Bomb: such ability can only use one the link V3 of his own. When using it, you download V3 and the rectangle around V3 all get bombed. If A virus get bombed, the owner will download the virus. Others like Data or Port will not be affected. We introduce this ability because it raises the risk and player will have to think of it more deeply by not just sending it V3 to beat other Data. Even in the worst circumstances, player will not give up hope. Such ability concerns more segment fault and will also have restrictions on whether it can be used or not. We will add restrictions when using this ability by judging if it is V3 and if it is of his own link. For V3, it is simple download, but for others around him, we need to write some helper like *getbombed(int i, int j, int row, int col)* to first judge whether there is a link (out of the edge of the board) and whether it is a Virus, and download them by his owner(A link contains the who he belongs to). Also, make sure they all leave the board.

3. The graph will show certain player what he knows: it mainly contributed by our design, since a player carries Info of all real links (except empty link), and the Info contains whether this player can see the type and strength, thus, we do not need a board like a: D1 b:V2 c: ? on the graph, because

if a player should know the type and strength of other players' links, they will show on the graph with a specific colour and strength. As, for what he does not know, the colour of the link will be balck. It makes the game more friendly to players, and very easy to implement by simply searching through a certain class Player.

-Final Question

I learned how to work as a team.

1. A Due date is important to be set. It normally occurs some situation like one person finishes his job and the other did not. And that makes the first one has nothing to do and wastes time. If there is a Due Date and is reasonable, no one will complain anything, and make the work more efficient. Also, more Due Dates will certainly help teammates set up different goals to complete, which makes team more focusing on details.
2. Tests are super important. For assignment, I usually think tests not matter and boring, that is because assignment is not as complex as this project. When I first run this project and print them out, I think things are mostly done. But when tester test it with more and more complex tests, the project shows different kinds of problem, some are small details, others are big segment problem. And they are all fixed among three days testing and fixing. And we are confident to this project right now.
3. The cohesion and coupling and good first design are important during the completion of the program. Since if the design won't work, we might have to do it again and think of it again. If such things happened, that's all because we did not think through all process a method in a high level. But fortunately, we think every step, and which function in which class will be called to implement a complex method like moving a link. Though there are things being corrected after the first design. But the project runs as we think. Also, we need the project design to be low coupling so that we can separate work correctly, and makes the work for each person easy to solve. As for high cohesion, such project is designed to be not low cohesion, since moving could be really hard to connected to use an ability. And we also become really easy to fix bugs. If moving shows a problem, we fix move method, if the information of player becomes a problem, we fix the Player.

-Some Improvement we will do if we can restart it.

1. textdisplay does not need to be owned by each player, it can simply go through the Grid, and owned by Grid.

2. we will prepare an ability list to contain name but not use Ability::Type. Then we can avoid having a link can only take one ability. (We did not think of it because there is actually one ability along with the link)

-Conlusion

We are happy we did it.