CS246 RAIInet Final Project - Design Document

Cristal Lu | Sabrina Xing | Michael Chen

1. Overview

Cell.cc

Cell.cc holds the Cell class which encapsulates the properties and behaviour of individual cells within the grid. It contains private fields which represent the attributes of the *Cell*. Specifically, *celltype* denotes what occupies the cell, distinguishing between various occupants such as empty cell, link name if occupied by a link or a server port. Meanwhile, *firewall* marks if there is a firewall on the cell indicating ownership by a player (‘m’ if owned by Player 1 and ‘w’ if owned by Player 2). This field is separate from *celltype* because a cell can contain both a firewall and a cell. Additionally, the link field holds a unique pointer to a *Link* object if the cell holds a link and the row and col int fields identify the cell’s position on the grid. The *download()* and *upload(std::unique_ptr<Link> l)* manipulate the presence of links within cells. *download()* detaches a link object from *link* field and *upload* attaches a unique_ptr to the link given representing physically removing and adding a link from and to a cell. This change is then reflected in the *celltype* field. In addition, the *Link* object can be accessed as a reference through *getLink()* which allows for the client to handle the behaviour and property of a link object. All the private fields can be accessed and modified through various accessor and mutator methods to protect encapsulation. Lastly, observers is a vector of *Observers* so that whenever a cell state is called, its data is updated to relevant classes.

Link.cc

Link.cc holds the Link class which manages the behaviour of a specific link piece on the game. If a link is present on the board (hasn’t been downloaded yet) it is always attached to the cell that it is placed on. It’s private fields store the attributes of a link. These fields include *name* denoting the unique identifier of the link (ranging from 'a' to 'h' for Player 1 and 'A' to 'H' for Player 2), *type* specifying whether the link is a Data or a Virus, *strength* representing the link's power, *revealed* indicating if the link is visible during a turn, *permanentRevealed* determining if the visibility persists for both players, '*moveLength* dictating the distance the link can move per turn, and *diagonal* marking if the link can move diagonally or not.

To protect encapsulation we use accessor and mutator methods. Accessor methods include 'getName()', 'getType()', 'getStrength()', 'getMoveL()', 'getRevealed()', and 'getPermanentRevealed()' retrieve the respective values of the link's name, type, strength, movement length, and reveal status. Meanwhile, mutator methods are crucial to implement abilities that change the behaviour of a link. For example, 'giveBoost()' and 'giveDiagonal()' modify the link's attributes by increasing its movement length or indicating its diagonal movement capability which are called from the LinkBoost and Diagonal abilities respectively. Additionally, 'toggleType()' swaps the link between a virus and data type, helping implement behaviour for the Polarize ability. Lastly, 'permanentRevealBool()' and 'revealBool()' manage the link's visibility by altering the reveal status.

Player.cc

Player.cc holds the Player class which represents a participant in the game and holds all the data known to the player.

The Abilities private field represents the player's hand of ability cards. Specifically, it is a vector of *Card* objects, a struct that holds the CardType enum (ability name) and a bool representing if it has been used yet. The *Abilities* field is initialized with the *setAbility(string s)* method and printed using the *printAbilities()* method. Every time an ability "card" is used, the *usedAbility(const char c)* modifies the number of ability "cards" available so that it is reflected for the player and their opponent in the *MyAbil, OppAbil* integer fields using the mutator methods *decrMyAbil()* and *decrOppAbil()*.

MyLinks and *OppLinks* are vectors of strings which hold the strength and type of a link if it is known and a '?' otherwise. The links for *MyLinks* are initialized using *addLink(string s)* and *OppLinks* are initialized with a vector of '?'. The order of this vector corresponds with the alphabetical order of the link names (e.g. data for link a is stored in MyLinks[0]). Thus, whenever a link is revealed to a player, the '?' is replaced with the type and strength via the *revealed(int index, string piece)* method. Since the Polarize ability toggles the type of a link, it needs to be reflected in *MyLinks* or *OppLinks* which is done with *polarizeUpdate(char newtype, int ID, bool mine)*.

Integer fields like *MyD, MyV, OppD, OppV,* store the number of data, and virus that the player holds and that the opponent holds. When links are downloaded these fields are incremented using the mutator methods: *incrMyD(), incrMyV(), incrOppD(), incrOppV()*.

Grid.cc

The Grid.cc file contains the Grid class which defines the structural framework of the board and handles the behaviour between cells and the functionalities of the game. Specifically, the board is built with *theGrid,* a two-dimensional vector of Cell objects. The other private fields include *textDisplay* and *graphicsDisplay* pointers which are used for textual and graphical representations of the game state and *player1* and *player2* objects representing the two players in the game. These players can be accessed through *getPlayer(int n)* which returns a reference to Player n such that any updates to the player's data can be updated. Additionally, *whoseTurn* keeps track of the current player's turn so that the game knows who is executing a specific behaviour.

Functional methods handle the basic actions of the RAIInet game. *move(char l, string dir), download(Cell& c, int player), battle(Cell& init, Cell& fighter)* and *reveal(Cell& c)* execute the mechanics of moving a link, downloading a link, battling two links, and revealing a link respectively.

The overloaded *operator<<()* is defined as a friend function so that the fields of Grid can be displayed.

Overall, the Grid class serves as the backbone of the game environment, managing the grid, players, cell interactions and gameplay mechanics.

2. Design

Implementing MVC and Observer Pattern

When designing the structure of our game, we struggled to determine how to implement the MVC pattern and what the model of the pattern would be. Initially, we considered that the *Grid* class would be our model and any changes made to Grid would be reflected in our *textDisplay*. However, this design pattern would have had low cohesion and reusability creating code that was difficult to maintain and inefficient. Furthermore, this design violated the Single Responsibility Principle, as the Grid class would have simultaneously managed the board, progressed the game, modelled the board and handled notifying the displays. From there, we switched to the current MVC architecture which inherently has high cohesion and low coupling and creates distinct roles for each class. *Cell* became the model, the displays as the viewer and *Grid* as the controller. *Grid* was then designed with methods to manage the board (*theGrid*) and progress the game. These methods include *battle* (handles the battle gameplay), *move* (handles moving the links) and *nextTurn* (enables the next player to begin). We used an observer pattern to facilitate communication between the controller, model, and viewer. The class *Cell* acts as the subject of the observer pattern while *TextDisplay* and *Graphics*, thus when any changes are made to the model through the controller, the observers of the changed cells are called through *notifyObservers()* to react accordingly.

Furthermore, the viewer class of *Graphics* is simultaneously an observer of *theGrid* and each *Player*. *Player*, as the subject, contains the information of each player through private fields (encapsulation) and get and set methods are used to change and retrieve these fields. When changes are made to *Player* (ex. An ability has been used), *Graphics* is notified through the *notifyPlayer()* method in *Player* and reacts accordingly. This *notifyPlayer()* method takes in a reference to a player rather than a cell, mirroring the way that player information is printed out in the text display, as the information can not be accessed solely through a *Cell* class. A design challenge faced here was the real-time updating of the graphics display which differed from the text display. This meant that player information needed to be updated after every ability used on top of every move. Our solution to this included adding a turn indicator within the *Player* class that could be conditionally checked within *GraphicsDisplay*, similar to how a turn could be checked within *Grid* by utilizing getTurn().

Accommodating Edge Cases

A significant design challenge faced was ensuring that various edge cases and invalid behaviour was accommodated for. This was done through implementing an exception-safe program and writing methods that have basic, strong or no-throw guarantee. For example, in the *move* method in *grid.cc*, before any changes are made to the model, we check that the action requested by the user is valid, ensuring a strong guarantee of the method. Furthermore, within the *main*, before abilities are constructed, we check if the data given by the user to use when executing abilities is valid, also ensuring strong guarantee. We derived our exception classes from the standard exception class provided by <stdexcept> to create clear and specific error messages for the user to understand the game's behaviour and for us when debugging and testing. For errors that could occur within various methods of a class, we used helper

functions to increase cohesion and avoid repetitive code. This differs from the due date 1 submission as we initially considered having all exceptions being caught within the client file. While this implementation abides closer with the Single Responsibility Principle, it would have designed highly repetitive code and disregarded encapsulation.

Using Factory Template Method for Abilities

To implement abilities, we applied the Factory Method Template. Within abilities.h there is an abstract *Ability* superclass comprised of purely virtual methods. *Ability* has an *execute* method. Each specific ability is a subclass of *Ability* with its own unique set of private fields (for the information required to implement the ability) and an overloaded constructor. When an ability needs to be used, we initialize an instance of the subclass such that its fields are set within the constructor using the input given by the player and then when the ability is played, the *execute* method is called and manipulates the behaviour of the fields. This segregates the execution within each class such that there are no direct dependencies between each ability class. In addition, every subclass takes in references instead of holding instances of them.

Low Coupling

We reduced dependencies between classes by reducing dependencies between classes through maintaining encapsulation and minimizing direct interactions between classes. Firstly, we always pass objects between each other through pointers and references rather than holding instances of other classes directly. In addition, encapsulation is maintained by restricting direct access to member fields. One way we did this was by making sure classes provided controlled access to their fields through getter and setter methods. Another way is through our limited use of friend functions which were only used for operator overloading.

Secondly, through the Observer pattern, we were able to promote loose coupling by defining how classes interact while reducing the direct reliance on specific implementations. Specifically, the abstract classes *Subject* and *Observer* allow classes to communicate through well-defined methods without needing to know the implementation details of each other, and prevent direct coupling.

High Cohesion

We ensured high cohesion by following the Single Responsibility Principle in which we ensured that each class has a clear and specific purpose. For example, the cell class handles all cell properties while Player handles player fields and player-specific functionalities. Additionally for abilities, we created two classes associated with abilities - one to hold the data available to the player such as the name and whether it has been used yet and another to handle the logic and execution of the ability. In addition, we had focused methods where each class only manages the functionalities and operations relevant to its defined role. Overall, these strategies ensure that the methods and attributes within a class are closely related and contribute to the common goal of the game without unnecessary overlap or dispersion of responsibilities.

Code Changes from Due Date 1

Originally, we assumed that the graphics display could be implemented the same way as the text display so we decided to build our program without the graphics display and then implement it after. By doing so, we assumed that *graphicsDisplay* could be an *Observer* of *Cell* similar to the *TextDisplay* class. However, this meant that *GraphicsDisplay* would only be updated when there were changes to a cell and not the player's information. Thus, we made the *Player* a subject and made the *GraphicsDisplay* a special type of observer called a *PlayerObsever* (observing changes in the players) in addition to being an observer of Cell like *TextDisplay*. This allowed us to update the GraphicsDisplay to reflect changes in both the cells and the players.

3. Resilience to Change

Our implementation of RAIInet was designed with scalability in mind, accommodating for a variety of board dimensions, which can be an easily implemented setup command. Within initialization, the field *gridSize* within our *Grid* class is the determining field on the placement of server ports, links and the number of Cells within the grid. This accommodates easily initializing *theGrid* to be size required as components are dynamically placed on top by iterating through *theGrid* using *gridSize* to determine the number of times to iterate. Furthermore, when checking the edge of the board within game progressing methods (*move*, *battle*, etc.) the gridSize as the boundary, dynamically determining if the moves are valid. As well, the number of links a player has is never statically set, rather than iterated through a vector enabling links the number of links to be dynamically set.

4. Questions and Answers

4.1 Switching Displays

Our current solution utilizes Grid as a controller that can notify its viewers including TextDisplay when data has been modified within our model. The Grid contains both players as fields, as well as a pointer to the TextDisplay which allows us to output a display that flips between players 1 and 2 using an operator overload. Through conditional statements using the "whoseTurn" field in "Grid", we print out the grid sandwiched between the two players' information ("whoseTurn" is an integer which indicates the turn). Depending on which player's turn it is, the links are revealed or hidden. The opponent links are hidden and the player's links are revealed. The text display is printed out using a TextDisplay-specific operator overload.

Changing our code to instead have two displays could entail introducing a second text display pointer within our Grid class of type "FlippedTextDisplay", a subclass of "TextDisplay". When the second text display is notified, it will perform the same changes to the text display that the first would, but with flipped movement (if the original link moved north, it would appear as moving south within the flipped text display object)

In this solution, the second player (seeing the flipped grid), will be moving and inputting coordinates with respect to the flipped grid. For simplicity, this solution would invert the coordinates in the client file when it is player 2's turn through a conditional based on the "whoseTurn" field. Furthermore, within the subclass FlippedTextDisplay, the "notify"

method would ensure that the changes are applied to the correct cell. This ensures that our code could produce two separate displays for both players' views without needing to create two separate Grid classes to store all data and changes.

This differs from our previous response, as we changed our pointer

4.2 Adding Abilities

To design a general framework to add abilities easily, it must allow adding abilities without modifying existing code and ensure that each class has a single responsibility related to that specific ability. To ensure our framework follows these two principles, we implemented a Factory Template Method with an abstract virtual class called "Abilities". Within "Abilities" there is a virtual method called execute which handles the logic of each ability and a private field "used" which denotes if the ability has been used or not. Through this, we are using polymorphism to add new abilities, which will not affect existing code. As well, the abilities are initialized and called from the client file, further isolating the subclasses. To accommodate the diverse parameters to execute each ability, the children of "Abilities" have private fields and overwritten constructors.

To demonstrate the credibility of our design we have decided to implement "PlayerSwap", which swaps a player's current standing in the game with their opponent, "MoveServer", which moves one of the server ports to a specified cell, and "MoveDiagonal" which allows a cell to move diagonally.

This solution differs from our initial solution which did not include a private field to indicate if the ability was used or not.

4.3 Four Player Mode

In our implementation, our *Grid* class has a field gridSize which currently stores an integer which then allows us to initialize our board with a square of size gridSize x gridSize. If we were to edit our code to extend the game of RAIInet to account for a four-player mode, our initialization of the game (most likely within the interface file) would take in a game mode (2 or 4 players).

We would need a new field to house this game mode, which would then be a parameter within the initialization method within the *Grid* class. Depending on the game mode, the initialization method would either create a two-dimensional vector of cells as expected (in a square shape) or would "cut the corners", where squares of size 1 x 1 would be initialized to *celltype* 'X', rather than *celltype* 'n' signalling that links cannot be moved onto these specific cells. This would essentially simulate a board created with two overlapping 10 by 8 rectangles and also allow for customizability in future implementations of a variety of board shapes.

To ensure that players are only able to download from the edges opposite to their starting positions, the player class would have an additional vector storing the coordinates of the edges they can walk off of, and another vector to store the coordinates of the server ports that they own.

When a player is eliminated, removing firewalls would require searching through the two-dimensional vector for the firewall character pertaining specifically to the eliminated player. Since a vector of ports per player would be implemented in the player fields,

removing the server ports would require switching the *celltype* of the cell at the coordinates from those vectors into ‘n’. Finally, removing links would be similar to downloading links, however, we may need to introduce a new boolean field and method isLost() per player. The download method would first check if *isLost()* produces true, and then not update the *MyD* or *MyV* within the player fields if it is. This would make a four-player mode possible in addition to a two-player mode with minimal code additions.

5. Extra Credit Features

In our implementation across the project, we exclusively utilized stack-allocated objects, vectors, and unique pointers aligning with the guideline of leveraging automatic memory management through STL containers and smart pointers. By using vectors, we dynamically managed collections of objects, allowing for dynamic resizing and handling of memory. Additionally, by only using unique pointers, we maintained ownership and handled memory deallocation automatically when the unique pointers went out of scope, preventing any chance of a memory leak. Through these strategies, we eliminated explicit memory management through delete statements and solely relied on the stack for automatic memory deallocation of local objects and smart pointers like unique pointers to ensure dynamic memory cleanup without leaks.

6. Final Questions

6.1 What lessons did this project teach you about developing software in teams?

One of the main lessons learned while developing software as a team is the importance of effective communication. Clear and constant communication among team members was essential to ensuring everyone was on the same page regarding our project goals, timelines, and individual responsibilities. This level of communication allowed for us to understand the strengths of one another and leverage our collective skills effectively. For example, one group member may have been more familiar with creating a graphics display while another may have a better understanding of writing exceptions. This capitalization of diversity, both in terms of technical expertise and problem-solving fostered an environment where innovative solutions emerged.

Additionally, this project emphasized the significance of version control and documentation in a group getting. Maintaining well-organized and documented code facilitated smoother collaboration between our group members, making it much easier for us to pick up where another group member left off.

Finally, this project highlighted the iterative nature of software development. Being able to embrace feedback and adapt to changing requirements was something that we had to work on individually to become good team players, and was what inevitably allowed us to continuously improve our code and deliver a high-quality final project. Overall, this project underscored the importance of teamwork, effective communication, adaptability, and a shared commitment while developing software in a team setting.

6.2 What would you have done differently if you had the chance to start over?

Upon reflecting on the coding process, there were several key areas of our process that, if revisited, could have enhanced both the group's efficiency as well as the final outcome of our project. First, we allocated an entire week's worth of time to brainstorm and plan how our code would work and interact with one another. Although this gave us grounds to begin coding and implementation at a faster rate, it also seriously disadvantaged us in terms of discovering our shortcomings as we began implementation. An earlier initiation of our coding process would have allowed us to more deeply understand the intricacies of our program and how different classes worked together with one another.Beginning earlier would have additionally afforded a greater margin for trial-and-error iterations, mitigating the strain imposed by the deadlines we had set for ourselves and ultimately resulting in a more refined end product.

Secondly, an earlier integration and thorough analysis of the graphical display would have been beneficial. We made the incorrect assumption that the graphics display would run in tandem with the text display without needing to make any changes to our code, however, due to the issue of conditional updating of player headers which was resolved through our *Grid* class for the text display, we needed to update our *Player* class to become a subject that the graphical display could be notified through. This meant that we needed to make some substantial changes to our code later on in the process, which could have been avoided had we spent more time discussing how our graphical display would work.

Lastly, establishing and adhering to a more structured approach within our Git repositories would have been advantageous in a group setting. Within enhanced branch management and adherence to more professional Git practices, we could have fostered a more organized collaborative environment, mitigating potential conflicts and ensuring a smoother development process.

Reflecting on these aspects of our coding process, a revised approach encompassing a greater amount of time spent coding, an earlier integration of the graphics display, and a more structured approach to version control would have likely yielded a more streamlined and optimized coding process, facilitating improved collaboration and code quality.