CS 246 A5 - DD2

Overview

This RAIINET game project is centered around a Model-View-Controller structure where the model(Board) executes changes to the game state, the view(Observer) displays game information, and the controller(Controller) interacts with the players to parse input and use the model and view functionality accordingly. The justification for this and all further significant design choices is explored in the Design section.

In main, the program begins by processing all command line arguments, which are the flags that will inform how the game should be set up. After considering all given flags, the program constructs the foundation of its MVC structure - the controller. Using the Controller class constructor, the program first builds a base constructor with a default Board object (with possible ability specifications from the input flags) and a default TextObserver that are taken by the constructor. Each of the Board and TextObserver are created through their own constructors before being passed to the With this, the minimum Controller, Model, and View are all created and their functionality is all accessible from Controller. Then, before starting the game, the Controller is augmented with links added to Board (either randomly or based on input from files fed on the command line) and/or a Graphical Observer.

The game is started through a method in Controller, it runs a doCommand method on std::cin that takes in an inputstream. This method is at the core of the game usability as it takes in input, parses it, attempts to execute the commands, and indicates any game outcomes / changes / issues that arise from those commands using the Board and Observer interfaces.

Beyond this Controller setup, the remaining key structural components are housed in the Board class. The Board can be divided broadly into two groups. The first are the GameObjects, which encapsulates everything displayed on the actual board as its subclasses. All such GameObjects are stored in one of 3 vectors of GameObject pointers. The ‘grid’ 2D vector holds the physical objects that currently sit on the top of the board: Links, Server Ports, and Board Edges. A graveyard vector holds pointers to all deleted Links, which is referenced by observer modules to display purposes. Finally, the ‘traps’ 2D vector holds all objects created by abilities which sit ‘under’ the previous objects. The other group of Board are Players and the Abilities within them. Board contains pointers to the two players, each of which have a vector of pointers to their five abilities. Each ability also has a pointer to access the Board.

Updated UML

The updated UML contains many changes that reflect what we have implemented in the final product. There were very few changes to the overall structure in terms of relationships between classes, but there are many minor changes like additional ability and obstacle subclasses, methods, and fields.

The board now contains a second 2D vector of GameObject pointers – one for links, serverports, and board edges, and one for the traps/obstacles. As a result, the fire obstacle that is placed by the firewall ability no longer has a GameObject field above. The board also contains a vector of GameObject pointers for the links removed from the board once they are downloaded. The board has methods to retrieve or destroy GameObjects by symbol or by coordinates

The controller now has a vector of Observer pointers to operate more than just one display. Methods were added for retrieving symbols and names for displaying them through the observers, such as those in Ability. The GraphicalObserver class now has fields including a 2D vector of characters representing the drawn current board and integers representing player statistics. While not necessary for its functionality, this allows the Board to identify changes and only update changed areas of the window. There are also new fields to store the margins and square size of the board which are set by default in the constructor and can be modified to adjust the board spacing. Furthermore, all observers have two new functions to display content separate from the board/player view - one for displaying abilities and one for writing exceptions/messages. This was previously done by just standard output in the controller but that reduced cohesion and also did not accommodate graphical displays.

The derived classes of the Link base class, Data and Virus, were removed as the only difference between them is their type, and they were replaced by an enumeration LinkType.

There are 5 extra abilities, with the Portal obstacle corresponding to WarpGate and a few methods for the abilities such as getters and setters added to the GameObject class. The method for using abilities takes in an istream reference instead of a string for the abilities' parameters.

Design

In designing and implementing the RAIINET game, we focused on maintaining a few core objectives of object-oriented programming: specifically maximizing cohesion and minimizing coupling through efficient encapsulation and intuitive interfaces.

RAIINET presented its first design challenge in its described user interface, as there are multiple streams for both input and output. The input must be parsed from the command line,

standard input, and even file inputs. Similarly, the output must be shown through both the terminal as well as a custom graphics display. Given the necessity for input/output customization, we employed a Model-View-Controller design pattern, where our model is the Board, our View is the set of attached Observers, and our controller is simply the Controller class that has the other two in an aggregation relationship. Any input or output task is interpreted through the Controller, which then calls upon the View to display updated visuals and the Board to change the gamestate. This MVC approach allowed us to overcome the obstacle of file-based input through the 'sequence' command. Controller could open and use input streams recursively, while handling any results of those command sequences (i.e. exceptions, game results, etc.). Consequently, cohesion is increased across the input process as command line input and setup occurs in main.cc while commands are interpreted solely in the controller.

As aforementioned, due to the distinct output streams required for this program, we employed an observer pattern where the Controller held a vector of Observers, which we made abstract with text and graphical observers as its two concrete subclasses. A text observer is created alongside the controller by default while a graphics observer attached to the controller afterwards is specified by the -graphics flag. As these observers were meant to display all facets of the current game state, we decided to make them each observe the entire Board object that the game was running on, rather than sections of the grid. Encapsulating the game display in these observers simplified implementations and increased cohesion within the overlying controller. Each observer independently collects, interprets, and displays the gamestate using their board pointer, so the controller only notifies them of what is being requested to be displayed. This is divided into two functions (that each call a different notify function within all of the observers) - one updates the game display while the other shows ability cards. Although these could have been combined, the contents being displayed are mutually exclusive, so we decided to maximize cohesion within each function and separate them. Given the slow Xwindow graphics display, this choice also avoided the tedious and unnecessary redrawing of the game board.

In designing the observers, we considered holding the observers in both Controller and Board. Ultimately, we designed the program with observers contained/utilized in the controller class to continue our approach of regulating all input/output within the controller and focusing the model / Board class on appropriately changing game states, maintaining higher cohesion. Since the Board class links to several other classes, this decision to avoid further coupling within Board appears justifiable. However, if Board had included the observers as may be more intuitive in the Observer Pattern, the observers could have been notified more precisely while making specific changes to the game state. However, we found that this would only benefit the graphical observer marginally and not benefit the text observer at all. We will comment further on these choices through this document.

The final and most significant component of our MVC implementation is the model: the Board class. We designed this class to contain all relevant components of the game, have the ability to to change all of those components privately, and also provide a large public interface for other objects to access or manipulate Board in a controlled manner. We decided that the best way to do this would be through the further modularization of the primary components of the game, which we identified as the links, server ports, board edges, players, and abilities. However, we quickly realized that several of these should be further encapsulated for their shared attributes, clear relationships, and most importantly ease of implementation.

Our first critical choice was to embrace polymorphism and make all objects shown on the board as separate subclasses that inherit from a common GameObject superclass. This superclass provides fields indicating ownership, display symbol, object type, amongst other fields that are relevant to the behaviour of all of the board's objects. These fields are defined uniquely in each subclass's constructor but are returned/manipulated by the same non-virtual get/set functions, avoiding the need for similar redefinitions across classes. We also had the GameObject class provide many virtual functions to allow other modules, specifically Board, to abstractly manipulate GameObjects without further specification of their type.

Our design differentiates GameObject subclasses by the virtual GameObject functionality and private fields that are specific to each one. For instance, the Link subclasses have a unique enumerated LinkType field that is used to determine its 'type'. Portal, on the other hand, is the only object to override the virtual get/setDest methods as they are currently the only object that links to another (other end of the portal).

This polymorphic GameObject implementation critically allows Board to own its primary playing field as a grid (2D vector) of GameObject pointers within the Board. When manipulating the grid through Board methods, we could call the same GameObject methods onto elements at any non-empty position without specifying what type of GameObject they are. In fact, we implemented the below layer of 'traps'/ability objects as a 2D vector of GameObjects as well because the immense abstraction of that single class allowed methods to manipulate both 'boards' similarly. Furthermore, it simplified the issue of having to find and return pointers to grid positions such that external modules (e.g. observers) could access the board. If traps and obstacles had been a new class, many methods would have to reimplement/repeat similar logic just to simply handle a trap if a corresponding GameObject could be found.

The separation of the two types of 'boards', despite having identical structures, was a highly beneficial decision. This removed the need to layer objects (similar to a Decorator pattern) as we had originally planned. The explicit differentiation of the two layers allowed methods to treat objects in the two layers differently, all RAIINET expects us to despite all being GameObjects. For example, the behaviour of moving a link onto another link or server port is

inherently distinct from moving it to a new position with a firewall. In the former, there are restrictions to the move actually occurring while in the later we need only to consider the firewall behaviour after the move is complete. However, the one drawback of this implementation is that it restricts the board to 2 layers. This will be touched upon later.

While GameObjects encapsulate the majority of objects/functionality within Board, Board separately owns two Player objects, representing the two players of the game and encapsulating their personal statistics as fields such as data/virus downloads. This enables other modules to access/manipulate players through Board and the public methods that are shared by Player.

Abilities are a cornerstone of the game functionality, but as they uniquely belong to individual players, we gave Player ownership over a vector of its 5 Ability objects rather than giving the Board ownership. This was also a difficult choice as one must access Player through Board already, but our approach leads to higher cohesion within the Board and Player classes. We also employed a high degree of polymorphism within Ability for similar reasons to GameObject. All of the ability classes share a few central fields and share similarly implemented methods and virtual methods. Thus, Player and clients using player can use all of the abilities in its vector similarly without having to identify each individual type. This polymorphism was also employed to facilitate the creation of new abilities, which will also be elaborated upon later in this report. Finally, we made the decision to use forward declaration and give each ability a pointer to the board because the point of using abilities is to ultimately change the game state, and that functionality is deliberately unique to Board as aforementioned.

Resilience to Change

Much of our program's ability to adapt stems from the polymorphism and cohesive design choices explored prior. However, we will now consider how those principles benefit us when trying to expand specific program parameters.

We implemented the MVC design pattern, which allows us to separately and readily change each of the game behavior, visual output, and input streams. For instance, additional observers as subclasses of the Observer abstract base class that can be attached to the controller like TextObservers and GraphicalObservers in order to augment how the game is being displayed. Similarly, the Controller allows for input to be modified through the command interpreter which can take in any input stream. Currently, this allows us to change from standard to file input without adding any new implementation. Due to the high cohesion and low coupling of these classes and their methods, input/output changes effectively do not impact the remainder of game functionality and only require small or modularized additions. Even a more significant

The modularization of our code also means that the classes interact through interface functions, which makes it easier to adapt if rules are changed. In most cases, we can simply change the implementation without changing the interface. This has already been seen in our Board class where the addition of new types of objects has rarely changed our Board interface but only required straight-forward additions to existing implementations. This is also because the GameObject abstract base class enables inheritance and polymorphism that allow current functionality to remain unaffected after adding more subclasses. An example of this is if we created a new ability that creates a single obstacle/trap, the clear ability would work on that obstacle without any changes as it deletes the GameObject on a specified coordinate of the traps 2D vector. Furthermore, the essential get/set functions do not need to be changed at all. Even a more significant change such as adding a third type of link would not be difficult. After making this new enumerated LinkType and adding checks for its object type, we would only need to add the new behaviour of the link under standard methods such as Board's move() and battle() and observers' notify(). Due to the polymorphic nature of GameObjects, many of these functional changes would also only require adjustments to methods/fields in the GameObject Link.

As will be explored in the question responses, we also made it easy to add abilities by adding an Ability abstract base class that new abilities can inherit from. When adding a new ability, we simply create a new class that extends ability and add a line in the Player constructor to allow them to pick the ability by passing the corresponding character to the command line argument. If necessary for the ability, methods like getters and setters can be added to the GameObject class and new GameObject derived classes can be created easily as we have demonstrated with our WarpGate ability and corresponding Portal obstacle.

Answers to Questions

1. How would you change your code to instead have two displays, one of which is player 1's view, and the other of which is player 2's?

While our plan was to invert the display using an iterator, this has changed with our new implementation and understanding of the game. Firstly, our observers have already been created to display player perspective based on the inputted turn. Therefore, we would change this so that rather than considering a single turn, the display occurs for both perspectives. Furthermore, we would have to slightly reorganize the observers to format the output for a second board. In the text observer, this would mean running the implementation for each line of the display consecutively, such that each contains the corresponding output string for both players with padding in between. For graphical observers, this is even simpler as the side margins must be changed for the second display such that its coordinates are all to the right of display 1. This is not a big change as we calculate coordinates already using predefined margins.

2. How can you design a general framework that makes adding abilities easy?

Each individual ability is encapsulated with an abstract Ability parent class. There is a pure virtual method use() which is overridden by each concrete ability subclass. This void use() method modifies the game state given the player number and an istream which is taken in by the controller class. The Ability class has a Board object, where we can use the board methods to directly manipulate the game state. When you want to add a new ability, you would make a new ability subclass, name it, and write a symbol in the default values in its constructor. Afterwards, you would have to write the overridden use method, and use the board objects methods. If there are any additional inputs/arguments, you can check for them in the istream parameter. In the case your new ability requires you to place an obstacle on the board, you would have to create a new Obstacle subclass and begin checking for that obstacle when moving links in the move method. While there are multiple steps to this process, each of these steps is well encapsulated, does not differ greatly from existing code, and can largely be implemented through functionality provided by board. Thus, the changes are easy to make and do not reduce the overall cohesion nor increase the coupling of any modules.

The major difference from this response to the one from the plan is that instead of the controller processing input into a string, we instead just passed down the istream. This is a better implementation since we would not have to update the controller when implementing new abilities, and we can just error check for the specific inputs necessary for abilities in the ability class itself. More information about the new, unique abilities we have added is available below in the extra credit features section.

3. What changes could you make in your code to handle this change to the game? Would it be possible to have this mode in addition to the two-player mode with minimal additional work?

Firstly, we would have to change the size of the board; change the size of the 2d vectors containing the game objects. We would also need to create two new player objects, with new sets of characters to represent their links on the board during setup. Also, it is important to delete all of the new players' game objects (e.g. server ports belonging to them) if they lose but the game carries on, in order to prevent memory leakage. The initial setup would be modified including setting new initial squares for board edges / server ports / links, creating more players within Board, and expanding the dictionary of symbols that are used for players' game objects. All observers would have to reflect these changes as well. We would also modify a few lines of code in the Controller class to implement turns to support 4 players' turns rather than oscillating between 2.

Beyond these setup modifications, there are very few conceptual changes that would have to be made to the implementation of Board. Methods that execute input commands and impact game state generally rely upon a few properties of the GameObjects they are attempting to change: player ownership, visual symbol, and object type. All necessary checks for player ownership are done using comparisons of objects owner integers and the turn number, and this would not change for a 4 player game.

We could alternatively implement the 4 player game above the 2 player game. We would have to create a new controller class with certain code modified to accommodate for 4 players, such as the turn logic specified above. The only other thing necessary to do is to create a subclass of board, such that we add a new gameobject vector field so that we have two boards on top of each other that represent the entire board and two new player pointers. We would modify the methods that access and change the game objects on the board to search for and look at both boards to accommodate the new squares on the plus shaped board. Similarly, board bounds would change.

Extra Credit Features

Abilities:

WarpGate allows the player to place a two-way portal by specifying two coordinates to place them. Moving a link onto a portal will teleport it to the other portal immediately. We created the WarpGate ability by making an Ability subclass, an Obstacle subclass, and adding a getter and a setter so each Portal knows where its corresponding Portal's coordinates are. A challenge that we faced was determining how to differentiate the portals so that the players know which portal leads to which since the WarpGate can be placed up to 4 times, twice by each player. We overcame this challenge by creating a static int field for WarpGate that counts the uses of WarpGate, and adds that to the char 'O' to make the portal symbol 'O', 'P', 'Q', or 'R'.

Exchange swaps two links that the player specifies the symbol of. There were many cases to take care of, which caused us trouble. We had to make sure that the symbol represented a link, the link was still on the board and not deleted, and that the link cannot be exchanged with itself. Two methods of the board helped us with solving these cases – checkLinkSymbol(char) which checks if the symbol is a valid link, and getCoords(char) that retrieves the coordinates of the link or the coordinate (-1, -1) if it was not found on the board.

Clear allows the player to specify a coordinate of the board to clear a trap, such as a Fire(wall) or a WarpGate (Portal). Creating clear was difficult because clearing a portal must clear its corresponding portal as well. However, our implementation of WarpGate allowed us to get the coordinates of the corresponding portal easily, which we deleted, and set the position to nullptr. Additionally, had we kept the original implementation of the board and firewall, it would be

difficult to delete a fire(wall) object from under a link as the fire(wall) had the link. Making the grid and traps layers of the board separate helped prevent this issue from occurring.

Invincible allows the player to make one of their links stronger than any other unaugmented link with a strength of 9. You cannot make one of your opponent's links invincible. This link should be able to defeat all other links in a battle, assuming they have not been made invincible. If your link is invincible and your opponent's link is invincible, and you move onto their link, you would win and vice versa. This ability is symbolized with the letter "I". The implementation was similar to linkboost but instead of modifying the link's speed, it modified its strength.

Arrest is an ability that allows a player to arrest/freeze one of their opponent's links. You cannot freeze your own link. This link now cannot move in any direction for the rest of the game. Your opponent may not know their link is arrested, and if they try to move it, they will waste a turn. This ability is symbolized with the letter "A". The implementation was similar to linkboost.

Exception Handling:

In addition to detecting exceptions, this program is able to neatly display customized messages to indicate what caused an issue with any invalid input from the user. By throwing messages that relay relevant information and printing these messages through customized notify-like functions in all observers, we are able to display these i0ssue-specific messages to all displays and guide the user towards inputting valid commands. While this was not technically challenging, we were able to further enhance error messages by having consecutive error messages appear together similar to an ongoing 'chat' on the right side of the graphics display, only to be cleared whenever a valid action is finally executed.

Final Questions

1. What lessons did this project teach you about developing software in a team?

One major lesson we learned was to communicate and be clear with our teammates with changes in the plan. If any part of the UML or one of the implementations with the classes and their methods was changed, this may affect the other person implementing a subclass or classes aggregated or composed of the class. It was also important to ask someone working on a certain class to create new methods to ensure good design and encapsulation when the other person needs to manipulate this class. After early delays due to necessary modules being modified, we began working on developing complete interfaces for every module before implementation. This provided a constant reference for other collaborators to work around.

When we were first starting the project, when we were designing the implementation, we all had different approaches to the design of the project and even which game to choose. It is crucial to weigh in all of our opinions to come to a conclusion while giving everyone autonomy.

Another major portion we learned about was the use of Github and working on code together. We learned about branches, and the various features of git that were necessary for us to put together the project. It was also important for us to review each other's pull requests/code changes to ensure everyone is doing their job right, and so there wouldn't be any miscommunication of information. A fresh pair of eyes on new code is important to verify logic and implementation as well. Another person may also be able to think of a more efficient implementation which would improve the code.

It is also important to get everyone testing each other's code. There is bias when you are testing your own code, and we found that others can quickly think of edge cases that the original coder had overlooked. It is important to find a balance promoting independence through modularization while also re-evaluating seemingly functional code.

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

In terms of our coding process, we would focus on setting up tools to validate our code earlier. For example, we should have transitioned onto the linux work environment earlier because we spent a long time trying and failing to get X11 graphics working locally towards the end of the project. So, we had less time to actually test that component of the project. Similarly, rather than writing test cases, we initially used manual inputs only which consumed more time in the long run and allowed small errors to be unnoticed until reviewed by another member. This was also due to a lack of forethought about the additional functionality we wanted our game to provide. By continuously changing the game specifications, even if only marginally, we were unable to develop a concrete idea of what we had left to complete.

In terms of our code itself, the main improvement we would hope to make is regarding the coupling with the Board class. While we were able to make the Board highly cohesive in that all game actions were completed through that class, we also made the sacrifice of coupling many different classes together with the Board. GameObjects, Players, even the Abilities are all interdependent with Board. The main reason we chose not to remedy this now (other than time constraints) was that we believed this dependency would not truly be reduced but rather moved onto another class. However, consider an entire second attempt, reducing coupling may very well be possible through better initial design choices. We also would have liked to make some less significant improvements such as optimizing graphics to transition more smoothly, implementing smart pointers to manage memory, and adding more commands to improve the player experience with our specific interface.