Game of Constructor
Final Project Design Document

Yifan Qu, Jiaqi Shang

1 Introduction

In this project, we will design a game called Game of Constructor, which is a variant of the game Settlers of Catan, with the board being based on the University of Waterloo.

This game is a combination of strategy and resource management. In the game Catan, players build their settlement in the land to gain resources from nearby fields. Players can use the resources to build roads, settlements, and cities to gain victory points. The first player who gains 10 or more victory points will be the winner. In our game Game of Constructor, we use similar rules. Instead of building settlements in the land, we build residences in the area of Waterloo. Each time a builder builds a residence or upgrades a residence the builder will receive one building point. The first builder who reaches 10 building points will win the game. To build a residence and road or upgrade a residence, each player needs to gain resources from the field that next to their residences. A residence must be built next to the same player's road. In another word, the road is used to extend each player's available area to build a residence.

2 Overview

We implement this game using c++ with Object-Oriented Programming.

Our game includes 9 classes, which are *Game, Board, Tile, Vertices, Edge, Builder* and *Residence*. The classes can be divided into two categories, one is the branch of Board, the other is the branch of Builder, and both of them is a composition of the class Game. Our idea here is that we let all builders' commands go into the class Game then modifies the Board with the associated commands. If the command deal with the vertices, we trace from Game → Board → Tile → Vertices to get the vertex we want then use the command. After successfully modifies the Board, we go into the other branch to modifies the builder's resources and/or residence/road to the associated command.

We use smart pointers throughout the program to ensure we are implementing the code with no memory leak. In particular, we used `shared_ptr` of Vertices and Edge in Tile, Edge in Vertices, and Vertices in Edge.

More specifically, we will explain each class's functionality and their relationship to each other as below:

Class: `Game`

The class Game is used to control the entire game's process, which includes the beginning of the game, starting a turn, during the turn, and ending one turn. During the game, when it comes to a player's turn this player will be marked as `curPlayer` in

the private field of Game. The class Game also deals with the command that `curPlayer` input, this needs modifications for Board, hence we have a private field of Board called `thisBoard`. At the same time, each player's resource and property would change. Hence we also have a private field of vector list of Builders called `allPlayers`.

At the beginning of a turn, each player needs to roll two dice, which returns two random numbers between 1 and 6. The private field `seed` is used to store this random number (and other random numbers in other class, this will be explained when we introduce to that class).

Key methods:

`initializeGame` : Initialize the game so that players can be ready to start a new game with an empty board and empty resources.

`clearAll` : This function is called when one of the builders wins the game and wants to play the game again.

`play`: This is a method that needs to be called to start a game. It includes beginning a game, beginning a turn, during a turn, end a turn, end a game. Also, this method will process the players' input to call the associated functions to execute the command.

Relationships with other class:

A `Game` **owns a** `Builder` and `Board`.

This makes sense because Builder and Board only exist in the Game and not anywhere else. This means if the Game is destroyed, the Builder and Board should be destroyed as well.

Class: `Board`

The Board class is used to store the board information. The board is divided into tiles, hence there is a private field of vector list of tiles. Sometimes in the game, we have the location of a vector or an edge, but we do not know where is a vertex or the edge in the board, hence we have a map from vertices to tiles and edge to tiles to help to find where in the board is the vertex and the edge through a tile.

Key methods:

`buildRoadAt` : When the player wants to build a road in the Game class, our target is to modify the edge. Hence this function is used to modifies the private field in Edge class to achieve build road.

`buildResAt` : When the player wants to build Residence in the Game class, our target is to modify the vertices. Hence this function is used to modifies the private field in the Vertices class to achieve build Residence.

`whichHasGeese` : While the player rolled a 7, we need to find the geese and deduct half of the resources of the player who has more than 10 resources. Geese is located at tiles, hence we need to use a method in Board to get to tiles to check which tile has the geese so that we can modify the resources of the players in the Game.

`transferGeese` : Similar idea as `whichHasGeese`, we need to go through the board to get to tiles to transfer the geese.

Relationships with other class:

A `Board` **owns** 19 `Tile`, 53 `Vertices`, and 71 `Edge`

This makes sense because Tile, Vertices, and Edge do not exist outside Board, which means when the board destroyed the Tile, Edge, and Vertices destroyed as well.

Class: `Tiles`

Each Tile has a type, a tile number, tile value, and maybe a goose, hence these are private fields of Tile. Also, we need to get to Vertices and Edges from Tile, hence there are two private fields to store the vertices and edges.

Key methods:

`addVertices` : when we called this function, we add Vertices to the Tile.

`addEdge` : When called this function, we add edge to the tile

`updateGeese` : We need to go through the board to get to tiles to update the geese's location between tiles.

Relationships with other class:

A `Tile` **does not own or have anything.**

Class: `Vertices`

A Vertices has a location on the board, and it might have a residence of a builder, hence these are private fields of Vertices. Also, a vertex is a neighbor of two or three edges, hence we have a vector list of locations of edges in vertices.

Key methods:

`addEdgeNeighbour` : Add the neighbour edge to the vertices.

`buildRes` : When this function is called, we need to build a residence at this vertex.

`clearVertex` : This function is called from `clearBoard` when we start a new game.

Relationships with other class:

A `Vertices` **does not own or have anything.**

Class: `Edge`

An Edge has a location on the board, and it might have a road of a builder, hence these are private fields of Edge. Also, an Edge is a neighbor of two vertices, hence we have a vector list of the location of neighbor Vertices in Edge.

Key methods:

`addVerticeNeighbour` : Add the neighbour vertices to the edge.

`clearEdge` : This function is called from `clearBoard` when we start a new game.

Relationships with other class:

An `Edge` **does not own or have anything.**

Class: `Builder`

Each Builder is each player in the Game, hence it has its color to represent which player. During the builder's turn, the builder can choose to roll fair dice or loaded dice, hence there is a private field called fair dice to decide which dice they want to roll. For each random number, they rolled we store this number in the seed. Also, each builder has their resources, residence, and road, hence we have three private fields to store these three properties that the builder has.

Key methods:

`rollDice` : When it comes to this builder's turn, this builder will roll dice by calling this function.

`switchFairDice` : The builder can choose to use a fair dice or a loaded dice by calling this function.

`upgradeResidence` : When the builder wants to upgrade their residence this function is called.

`printStatus` : This function prints the status of this builder.

`printResidence`: This function prints the residence of the builder.

`buildRoad` : This function is called when the builder wants to build a road.

`buildResidence` : This function is called when the builder wants to build a residence.

`calculatePoints` : This function is used to determine when the builder will win the game.

`haveEnoughRssForResidennce`, `haveEnoughForResidence`, `haveRssForImprove` : These functions check if the builder have enough resources to build a residence/road/upgrade residence.

`haveResidence` : This function checks if the builder has a residence at a location.

`highestLevel` : This function checks if the residence at the location is at its highest level.

`trade` : Trade the resource from one builder to another.

`clearBuilder` : This function is called from game when we start a new game.

`modifies resource` : Modifies the resource when needed.

Relationships with other class:

A `Builder` **owns** some `Residence`

This makes sense because the Residence exists dependently on the builder, which means if we destroy the builder, the residence is destroyed as well.

Class: `Residence`

The residence is built at a location, and it has a level of a maximum of 2, building points is the level plus 1, these are the private fields of Residence.

Key methods:

`clearRes`: This function is called from Builder when we start a new game.

Relationships with other class:

A `Residence` **does not own or have anything.**

3 Design

Read information from file

When loading the board, we choose to read from a board template, because the frame of the board will not change throughout the game, and the only thing we need to modifies is the letters in the template. We tried to hard code the template at first

but found that it was not as easy as just read from a file when we need to modifies the vertices during the game.

In addition, we also read from a file when deciding what are the neighbors of edges and vertices. We choose to read from a file instead of hard code for this because when we hard code, there are too many irregular cases to consider, such as the corner of the board and the top/bottom little triangles on the board. This makes the location of the vertices and edges irregular. Hence, we choose to read from files, this makes the problem easy to solve.

Another part we choose to read from the file is when creating the map between tiles and vertices/edges. A similar, reason as above, the irregular board shape makes it hard to code to find the tile that has the vertex/edge.

Mapping

When we want to find the tile that has the vertices/edge, we choose to use mapping functions to find the corresponding tile by reading the matched tile to vertices and tile to edge to create a dictionary. Hence when we want to find the tile that contains the vertex or edge, we only need to pass the vertex number or edge number to this dictionary, then we will get the corresponding tile.

Since there is more than one tile that has the vertex or edge, but we only need to choose one of them, hence we decided to choose the first tile that is associated with the vertex or edge.

Random Numbers and Seed

When rolling a dice, the result of the dice is randomly selected between 1 and 6. Here we use a random engine in the standard library to set a seed that stored the random numbers, hence if we have the same seed then the random numbers will be the same as well. Random Numbers generating is also applied when we want to randomly select stealing resources and losing resources.

Maximize Cohesion and Minimize Coupling

We try to maximize cohesion by defining multiply functions where each function gives a specific objective, and we made sure that each class only fix on their functionality. To minimize coupling, we use composition between each class, this makes the UML diagram clear and neat to see the relationship between each class, and from the UML we can see the program is in the minimized coupling.

4 Resilience to Change

Avoid Potential Memories Errors

We originally takes the initiative of applying observers on to vertices (since when a residence is built, the neighbour residences are not available for building anymore). We were fascinated by the idea of mutually accessible and extended the idea of connecting graphs so that information about neighbours (vertices or edges) can be observed and

detected. However, when we applied smarted pointers and the STL, we found that the issue of **circular referencing** often led to memory related issued discovered in valgrind. To resolve this issue and still keep the use of smart pointers, we extended the idea of observer and combined the essence of "observing" surroundings into the function of class Vertices. In our final implementation, we keeps only a little information for each class so that it reduced the difficulties in implementation and reduces the possibilities of causing memory errors.

Create options to provide capabilities for adaptations

For stages of the game, we provide possibilities for user to random a random seed based on the system timing. The use of standard default random engine throughout the project provides the possibilities that we can allow users to randomize a seed in a more convenient and appropriate approach. For future generator use of implementation, we noticed that since we have different types of resources, we may randomize the resources by the rarity of the resources in real life to make it more realistic and fun.

Graph variety Support In our implementation, we tried to use variables and constants to represent a specific data, such as total number of vertices. And we use file to read the board template, format and connections between edges and vertices, in this way it's very convenient and easy to support other variabilities within different shapes of the graph (possible suggestions are circle, square, triangle or uneven shapes).

Variety of Buildings and resources associated In our implementations, we provided large flexibilities and capabilities in increasing the number of new buildings and resources by easily extending the control statement for levels and types. And if we extend this idea to a real city simulations, when the number of buildings is large, we can easily read information of buildings and apply them in Residence class.

5 Answer to Questions

Question1:

We can use the **Strategy**design pattern to implement the feature of randomly setting up the resources of the board and reading the resources used from a file at runtime. The randomness of setting up the resources of the board is controlled by the user inputting the flag '-random', based on this flag, we can apply different algorithm to control the input reading method. In addition, since this is done at runtime, then we could apply the Strategy design pattern.

In our implementation, we will not apply the exact Strategy design pattern introduced in the lecture; instead, we will extend the core idea of the Strategy design pattern and simplify it. Since we only have two ways of setting up the resources of the board

and reading the resources used from a file at runtime, it's a binary control, and we can simply use a boolean variable to control with two different algorithms implemented in two different functions. If in the future, we have extra features that allow us to have a third, a fourth, or more algorithms to implement this feature, we will certainly consider using the Strategy design pattern to allow us to isolate the codes, the internal data, and the dependencies of different algorithms from the rest of the implementation.

Question2:

We can use the **Strategy** design pattern to implement the feature of switching dices. The use of the dice is controlled by the user inputting the command 'load' and 'fair', based on this command, we can apply different algorithms to operate dice rolling when using input command 'roll'. In addition, since this is done at runtime, then we could apply the Strategy design pattern.

In our implementation, we will not apply the exact Strategy design pattern introduced in the lecture; instead, we will extend the core idea of the Strategy design pattern and simplify it. Since we only have two dices when rolling, it can be considered a binary control, and we can simply use a boolean variable to control with two different algorithms implemented in two different functions. In this way, the implementation will be much easier since we only store boolean and two-member functions for dice rolling. If in the future, we have extra features that allow us to have a third, a fourth, or more algorithms to implement the dice we want to roll, we will certainly consider using the Strategy design pattern to allow us to isolate the codes, the internal data, and the dependencies of different algorithms from the rest of the implementation.

Question3:

We can use the **Template** design pattern to implement the feature of settling board layout and size. The display of the board can be control by defined flags (e.g. '-hexa', '-graphical') we can apply different algorithm to control the essential display of the game. In addition, since the commands are input at compile-time, hence the decision of game modes is also made using at compile-time, then we could apply Template design pattern.

If we want to implement a board shape feature using this **Template** design pattern, a 'Shape' class can be helpful with child classes being 'hexagonal', 'regular', etc. We pass instantiated child class of class 'Shape' as parameters to a class 'File' to control the algorithm of layout and size of the board. Similarly, if we want to implement a graphical display feature, we apply a similar idea using class to control whether we use graphical display tools such as GUI to visualize the game.

Question6:

We can use the **Decorator** design pattern to implement the feature of changing the tile production. **Decorator** design pattern lets us add this feature to the tile at run-time rather than to the class as a whole. In this way, if we want multiple types of resources from a tile, we can have two child classes of the **decorator** class namely **addition** and **remove** to be layered on top of the tile for adding one more resource and removing one

resource from the existing tile. Based on the change of the tile's production, we add or remove decorator classes (or "layer") from the current tile (either with or without layers covered).

Question7:

We used exceptions on users' input and input files in our projects. We use exceptions in those two cases because once we encountered invalid input or invalid files we can catch these errors and throw exception and error messages in the main function to notify the user that the input is invalid. In this way, we can make sure that all input and input files that read into the actual game are valid.

If we consider invalid command, including the command user use in the game round and turns, for early stage, we will assume the inputs are valid as described. However, in future implementations (such as in the extra features), we can add blocks to ensure the exceptions can not be arisen by prompting invalid command messages and allow users to re-input commands. Moreover, such an idea of handling exceptions and potentially some errors can also be extended in checking the validness of parameter values.

6 Extra Credit Features

(1) Deal with memory management by using `shared_ptr` and vectors.

We use `shared_ptr` and STL containers throughout the program, hence we do not have any memory leak. In particular, we use `shared_ptr` in the class Game to point to the class builder in a vector list of shared pointers of builders. We use a smart pointer here because, in this way, we can make sure the pointer to the builder class is the same thing as we intended to dereference to when we want to modify the private field in each builder in the vector list. In this way, when we want to modifies the builder's private field we can just use the shared pointer to the builder then call their public functions. In addition, we want to modify the vertices, edge in the Board. In the private field of Board, we use shared pointers in the vector to point to the edge and vertices class, hence we can achieve our goal to modifies them.

We also use STL like vector and map throughout the program, hence we did not use any `delete` to free memory.

(2) More specific guidelines throughout the game.

We implemented the game with extra guidelines as the game goes on to help the players know what to do next and what is the stage of the game.

More specifically, we add a divider line before the game enters a new stage. For example, when the game begins, you will see a line "—— Game setup ——". This means now you are on the game setup stage where you should build your first basement. After this you will see "—— Game starts ——", "—— Game during the turn ——" and "—— Game during the turn ——" accordingly to help you know which stage of the game you are in.

Besides, we also have: "builder has build a basement at location" at the beginning

of the game to indicate which builder build a basement at what location. "Choose your dice and roll!", "Commands: "load", "fair" and "roll", "Now you have loaded dice!", "Now you have fair dice!" and "Rolled dice is num!" guide the player to roll a dice. When there is an invalid command we will hint the player to try again "Invalid Command! Try again with "load", "fair" and "roll"". When trading with the player themselves, they will see "You can't trade with yourself! Try Again!", and etc.

(3) Through error checking and exception handling. For every section of the game, we though of and tested many possibilities a user can encounter when playing the game of our implemented version. For different type of exception handling, we provide the possibilities that user could resume the game by re-entering the designed commands or answers. When encountering conflicting command (such as command line argument -board, -load, -random-board, we notify users to provide the choice of use of flags and allow users to re-choose the options that were not intended.

7 Final Questions

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

To develop a software in a team, I find it especially essential to divided the tasks to each person clearly before we start to code. Since we are using GitHub to collaborate, there are sometimes conflicts happen between our codes. But after we refine the distribution of our tasks and each of us is very sure that who is doing which part of the program, our code's conflict is resolved easily and this helps us work more productively in the remaining time.

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

First of all, I would try to make our UML as simple and neat as possible before we start to code. To make sure the overall structure will not change very much, I will try to think about every possible other ways to draw the UML and compare which type of UML will make the program most efficient to implement. In this way, we will most likely won't change the UML in the future and we only need to follow the idea of the UML to implement the program which makes our coding process more effective.

Secondly, I would choose to use more design patterns to organize the program. As the questions answered above, we will try to think about what are the possible places to use a design pattern. However, in this project, we ended up not having enough time to implement all the design patterns we want, hence this also suggests me if I can do this project all over again I will start early instead of leaving it to the final exam week to finish.

8 Conclusion

In conclusion, this project helps us learned how to design a real program from designing UML to the final implementation. It also teaches us how to collaborate with others to make our work more productive. After four month of studying OOP, this final project concludes what we learned in this term. After seeing our game can be played by the user, it gives us a sense of accomplishment and finds that all the hard work we did in this semester is worthed it.