CS 246

Final Group Project

Constructor Final Report

Group members:
Chengxu Han (Infinite, c38han)
Yangzhou Han (Bernadette, y236han)
Zijian Feng (Alfred, z63feng)

Dec. 12th, 2020

Introduction

Constructor is a variant of the game Settlers of Catan (also called Settlers for short), with the board being based on the University of Waterloo. This game allows 4 players to play together. Players are free to choose their starting point at the beginning, which affects the materials they can acquire and where they can build in the future. Players can build three different types of houses, earning one building point for each house they build. Each player takes turns rolling the dice. The number of dice determines the tile of the production resource. Players can decide whether to use resources to build roads and residences on their own dice rounds. The first player to receive 10 building points will be the winner.

Overview

We used both shared pointers and original pointers together in this project. The shared pointers are used to avoid memory leak and the original pointers are used to prevent circular references that shared pointers might appear.

UML Classes sketch:

1. Class Board:

Implementation of game components – Board (2.1). This is a main class of our game and it stores the current state of the game. Board acts as a game panel and map, recording the entire game as it stands. It is responsible for the process of the game and the actions that the game can take as well. When the player input command, the main function will pass the required information and let Board to execute the corresponding action. The fields in Board class are *td*, *dice*, *tiles*, *vertices*, *edges*, *builders,* and *geese*. Board has 18 Tiles. Those fields store the information of the game status. The important methods that related to our game operations are:

Init: initialize the board when the game start

buildRoad: build road at the position the player decides

buildRes: build Residence at the position the player decides

roll: let player to roll dice

trade: allow players trade their resources with each other

improve: allow players to improve their residence at the position they decides

getDiceNum: get the current number of dice to determine which plate produce resources

2. Class Tile:

Implementation of game components – Tiles (2.2). Each Tile has 6 Vertices and 6 Edges. The fields in Tile class are *num* represents the tile number, *value* represents the Tile value, *resource* represents the type of resource, *isGeese* indicates whether the Tile has geese, *vertices* stores the vertice in the Tile in a vector, and *edges* stores the edges in the Tile in a vector. Players can build residences, roads and get resources through it.

3. Class Builder:

Implementation of game components – Builders (2.7). This Class acts as the controller in MVC design pattern, also Builder is the derived class of Subject and Observer. Responsible for recording players' corresponding color, building points, and resources data. *roads* is a vector of integer stores the position of the built roads. *housing* is a vector of pairs, each pair stores the position and the type of the built residence. The Board class will pass the required information to Builder, and Builder class have methods to support command *build-road <road#>* (notifyRoad), *build-res <housing#>* (notifyRes), improve <housing#> (notifyimprove), trade <colour> <give> <take> (trade). The names in the parentheses are the corresponding method in class Builder.

4. **Class TextDisplay:**

   This class helps the panel to realize visualization, which is embodied in "printing" the board.

5. **Class Dice (LoadedDice & FairDice):**

   Implementation of game components – Dice (2.8). Realize the player roll the dice. There are two modes to be choose from: class LoadedDice allows builder to choose an integer from 2 to 12 (inclusive) and an error message will be outputted if it is out of this range; class FairDice chooses two integers randomly, between 1 to 6 (inclusive) and the sum of the two generated integers are used for the turn.

6. **Class Resource:**

   Implementation of game components – Resources (3.5). Record the name of the resource in the player's hand and the amount of the resource held by the player. There are 5 resources: Brick, Energy, Glass, Heat and WIFI.

7. **Class Vertex:**

   Vertex records the information of vertex. The fields are *num* represents vertex number, *type* represents the residence type, and *owner* represents the residence owner.

8. **Class Edge:**

   Edge records the information of edges. The fields are *num* represents edge number, *road* indicates whether the road is built, and *owner* represents the residence owner.

Design

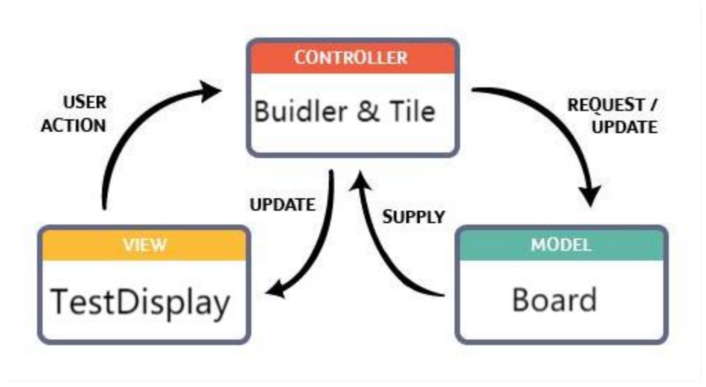
To achieve good scalability and efficient implementation of this project, we used the following design patterns:

1) **Observer Design Pattern:**

   Implemented by class Subject and Observer. Class TextDisplay observe class Builder. The TextDisplay will be notified when the Builder acts. For example, when Builder improves their residence, class TextDisplay will be notified what type should be changed and the location of it. Thus, TextDisplay can change it directly.

2) **Model-View-Controller Design Pattern (MVC):**

   Since MVC separates the program state, presentation logic and control logic, we plan to use MVC to increase the reusability of our code. When the user enters something, or when the game starts, either the Builder will change the tiles (e.g., building a house), or the tile will change the builder (e.g., changing the number of Builders' resources). So, class Builder and Tile are controllers. Our logic pattern is shown below.



4) Strategy Design Pattern:

We have an abstract class called Dice, it has two derived classes named LoadedDice and FairDice. This class Dice is the context class and contains a reference to the strategy. This allows Builders to switch between loaded and fair dice during game time.

Resilience to Change

Compared with our implementation plan at deadline 1, we have made the following changes:

1) Change in Visitor Design Pattern

Previously, we implemented the visitor design pattern by making TextDisplay act as a visitor to distinguish object types and implement corresponding actions according to different types. Now to simplify our code, we have decided not to apply the Visitor design pattern. TextDisplay is notified to make changes directly using the existing observer design pattern when the player acts.

2) Change in Observer Design Pattern

Our original plan was to have the Class Builder and Tile observe each other and double-inherited, so that tile would be notified when the Builder acted, and vice versa. However, this implementation is too cumbersome. For example, Tile is notified when the Builder improves the home and then calls the method *buildimprove* to implement this action, whereas the *buildimprove* essence is to have class Vertex and Edge change accordingly and notify TextDisplay to change. This means that Builder will implement a chain of notifications for any changes it makes, which not only increases our workload but also makes debugging very difficult. So, we abandoned the original plan and began to look for new ways. In the implementation, we found that we could have let the Builder notify TextDisplay to make changes directly, thus avoiding a lot of unnecessary code and achieving the desired effect as well as simplifying our workload.

Answers to Questions

The question number here corresponds to the numbers of the question in constructor.pdf.

Q1. Strategy design pattern could be used to implement this feature. We choose this design pattern because the context class (the ***Board*** class) can use the desired strategy directly without need to worry about the implementation of the strategy, and the change in strategies will not affect the context class. We could have an abstract class called ***BoardStrategy***, and it should contain a method ***setBoard***. Randomly setting up the resources of the board and reading resources used from a file at runtime are two concrete strategies and thus we should have two derived classes of ***BoardStrategy*** called ***RandomSetBoard*** and ***ReadFileSetBoard***, those two classes will implement their own ***setBoard*** method. Our ***Board*** class can be considered as the context class, it should contain a reference to one of the strategies.We did not use this design pattern because it only introduces the complexity to our program.

Q2. Strategy design pattern could be used to implement this feature. We choose this design pattern because it is easy to change strategy during the run-time. We could have an abstract class called ***DiceStrategy***, and it should contain a method ***rollDice***, and two derived classes of ***DiceStrategy*** called ***LoadedDice*** and ***FairDice*** which will have their own ***rollDice*** method. Our ***DiceStrategy*** class can be considered as the context class, it should contain a pointer to the strategy, which will allow the player to switch between loaded and fair dice at run-time. We implemented this design pattern in our project, since it allows the player to switch between loaded and fair dice by creating different derived classes to replace the pointer in the context class at run-time.

Q3. Template method design pattern is a good fit for this feature. We choose this design pattern because it defines the steps of an algorithm in an operation but let the derived classes redefine certain steps without affecting the overall structure, so by using it we can implement different game modes. We can have an abstract class ***Board*** with the basic fields and methods, and have concrete classes ***RectangularTilesBoard***, ***GraphicalBoard***, ***NumberOfPlayersBoard***(e.g. ***FourPlayerBoard***), those classes might override some methods in the ***Board*** class or having their own method to achieve certain features.

Q4. This is a typical strategy design pattern problem. When we want to replace players with different realizations, we actually need different strategies to give commands to the board. For computer player ***Greenhand***, the ***Greenhand*** strategy is used to give commands and instructions to the board. ***Greenhand*** is designed for new players who do not know the rules of the game for the first time. In this strategy, ***Greenhand*** advises the player on the next step after each turn. The player can choose to follow his advice or play freely. For computer player ***Newbie***, the ***Newbie*** strategy is used to give commands to the board. ***Newbie*** can have a function called *makeMistake*, so that when the mode is selected it calls the equation itself. In this way it leaves the player with obvious mistakes, which greatly reduces the difficulty of the game and serves the purpose of being newbie friendly. This strategy is suitable for beginners who are not familiar with the game to familiarize themselves with the rules and gameplay of the game. For a computer player ***Master***, the ***Master*** strategy is used to give commands. *MakeLessMistake*, the function called in ***Master***, will no longer leave the player with an obvious mistake. ***Master*** will increase the difficulty of the game and bring some challenges to the player, making it impossible for the player to win easily. This strategy is suitable for those who are familiar with the game, to increase the playability and fun of the game. Finally, we have ***Pro*** strategy, which is an updated version of Master. Its *MakeLittleMistake* function greatly reduces the probability of making mistakes based on *makeLessMistake* and reduces the probability of winning to the lowest (not zero, the player can still win). It is suitable for players who think the ***Master*** is not difficult enough to challenge themselves. Thus, the strategy design pattern is in good behaviour.

Q7. We did use exceptions in our project. In main.cc we will have try-catch blocks to catch the exceptions of incorrect input command and the exceptions thrown by other classes. For example, in our ***Builder*** class, if a builder wants to build a residence at a vertex but the vertex already had a residence built by other builder, then the builder method ***notifyRes*** will retrieve builder's status and throw an exception to the main function to display an error message "You cannot build here." And then let the player input a valid number. Using exceptions not only can enforce invariances but also can provide strong guaranteed exception safety.

Extra Credit Features

(1) No memory leaks

We handle all memory management via STL containers and smart pointers. We did not use any delete statements in our program at all, and the only raw pointers we used are those that are not meant to express ownership.

This is a challenge since we have not used RAII-related operations in previous personal assignments, and we are more familiar with how to use new and delete statements for operations. Therefore, it is easy for us to subconsciously use what we are more accustomed to when writing a project. Not only that, but it was also the first time we used Shared Pointer to implement a project on such a large scale, and we encountered a lot of problems and bugs when we were writing.

(2) Market

We implemented market command in our project. This command allows attempts to sell resources on the market, give four resource of type <sell> and receiving one resource of type <buy>. Since our group was designed to add new commands, we were able to write this very smoothly. First, we implement the ***market*** by using its method *checkResource* in builder to check whether the player has enough resources to take the resource they want. Then use method *obtain* in the Builder to change the number of resources of the player and then output the changed result. Finally, we added the command ***market*** to **main** file. So far, we have implemented all the functions of market.

Final Questions

Q1. What lessons did this project teach you about developing software in teams?

This team project has benefited everyone in our team a lot. We summarize the specific receipt as follows:

1) Learned to initiate GitHub pull request and code review

Unlike previous individual assignments, when we were part of a team, we needed to make sure that our project was filled with high-quality, tested, bug-free code so that team members could write code more effectively.

When new code enters our main project, we need to conduct code reviews to ensure that the added code meets our quality requirements. Usually, our review is done by members who are not responsible for this part of the review, who need to review the added code and make changes to make sure we can find any problems that might exist.

Every time a member starts writing code, they look at other's code, read the comments left by the previous editor carefully, test new additions, and write down what we think is being ignored.

2) **Learned to split problems**

In Deadline1, we divided the entire project into classes responsible for the different functions through UML drawing. In Deadline2, we used UML to determine the functionality we were responsible for implementing, which allowed each member to choose their own section based on their merits, greatly improving the accuracy and efficiency of code writing.

3) **Learned to communicate actively to avoid conflict**

Due to the time difference between our group members, our information sharing was delayed, at which time there might be conflicts. To solve this problem, we considered all possible problems in the plan before writing, so that each section could write independently without interfering with each other. We also have a short group meeting as soon as possible when there is a conflict. The team members learned how to keep calm and actively communicate with other team members, fully understand their thinking process, and find the best solution to achieve the team's goals.

4) **Learned to trust and use meetings to share information**

On a team, with many lines of code changing rapidly every day, we learned to stay focused on our task. We must trust our team no matter how the code evolves and changes. At the same time, we want the market to update the code to make sure that we are using the latest code.

In our meetings, each member also shares what they did after the last meeting and what they plan to accomplish before the next meeting, which makes our meetings short and productive.

Q2. What would you have done differently if you had the chance to start over?

Although the cooperation of our group is very harmonious and pleasant, there are still some areas that need to be improved. Specifically, the following three points:

1) Start early

Although it is not late for our group to start the project, we would choose to start earlier if we could do it over again. We were overconfident that we would be able to complete it within the scheduled time, though we found the project was not easy on Deadline1. However, the reality was that we were running behind schedule on every feature. There are many factors contributing to this result. For example, we did not reserve enough time for review of other courses and did not reserve time for unexpected situations.

2) Think more maturely when design

In Deadline 1, we thought we had considered most cases and did not need to make any major changes to the implementation. However, in the actual implementation, we found that we still missed a lot of small details, and some features could have been implemented more optimistically. In view of our team's attempts to use multiple design patterns, we had some patterns for the sake of patterns at the beginning of design, but in the actual implementation, we found that these codes brought us a lot of unnecessary work.

Conclusion

To sum up, every member of our group benefited a lot from this group project. Our team cohesiveness, personal skills, group stress tolerance, and time management skills have all improved tremendously. Not only that, but this project is a good way to relate all the knowledge we learned in CS246, so that we can have a more intuitive sense of how the skills we learned before can be better applied in practice.