CS 246

Object-Oriented Software Development

Final Project

Constructor

Final Report

Charles Zheng (j243zhen)

Jiqi Hu (j334hu)

Kevin Xu (y574xu)

December 16, 2020

Introduction

In this project, we implemented the game Constructor, which is a variant of the game Settlers of Catan (often called Settlers for short), with the board being based on the University of Waterloo. Many of the rules are similar to rules in the game Settlers of Catan. More details are listed in the project instruction.

Overview

We have controller.cc, file.cc, player.cc shuffle.cc, strategy.cc and their respective header files. The main.cc is responsible to analyze the command lines arguments and determine the proper initial board and sequence of dice. Then, all information will be sent to controller.cc.

controller.cc

controller.cc contains a class Controller in which the front controller design pattern is being implemented. It is the main implementation file throughout the game where it controls the progress of the board and keeps track of players' status. The private instances in controller.cc are setups of boards and players such as player_list, order, grid, etc. They give an initial state of the game and we can use functions to modify states throughout gameplay. There are three types of methods in controller.cc that handles three different tasks: printing, checking and modifying. First of all for printing, there are methods such as print_board, print_all_buildings_position and print_gain which can both exist as the form of helper function or member functions that print out the specific detail of the game board and user statistics. Next, we have functions that return a boolean value such as canPutBasement, canStealFrom to determine the conditions of modifying the board or resources, e.g. are we able to steal from a certain player or is this a valid location to

set up a basement, etc.. Finally, we have getter and setter functions that modify certain states in the game such as buildBuilding, getResource, setGeese and steal that either add housings to the board, change players' resources or manipulate the goose to steal resources. These three types all combined make our controller.cc a functional and organized class to fully control the game.

file.cc

-load xxx and *-board xxx* are two important features in the game. Both features involve the input of a file, so this file contains functions responsible for dealing with these. When the game starts, main.cc determines if there is an input file, and sends the file name to file.cc, then file.cc returns the vector containing all content in the file separated by space and checks if the input file is valid for the *-board xxx* command.

player.cc

player.cc contains a class Player, which stores the information of the player (such as name, points, brick, dice_type, res…) and also provides methods that change the information of the player or returns value based on the information of the player (such as win(), addResource(string target, int adder) ….). For example, if we want to add two GLASS resources to the current player, we can call by addResource("GLASS", 2). If we want to check the player's points to see whether he wins, we can call by win() and get a boolean value. The methods can all be used by the controller to increase the cohesion of the program.

shuffle.cc

This file is implemented to ensure the randomness of the board and dice. If *-seed xxx* is used, then *xxx* will be used to generate the sequence of the number of dice. Since one seed number can only return one random number between the range, the actual seed to generate number is the sum of seed and turn. This function will be called by a function in strategy.cc and send the number to the controller, which uses the Strategy design pattern. If *-random-board* is also used, the seed number will also be used to generate a board. It generates a vector of integers which represents dice number and resources.

strategy.cc

There are three classes in this file, Strategy, Load and Fair. The idea of Strategy design pattern is used in this file. An implementation of checking whether the upcoming dice is loaded, or fair in the Strategy class. Load class returns the value of dice if load dice is chosen. Fair class returns the value of dice if fair dice is chosen. One function in the Fair class gets value from the shuffle.cc.

Design

To maximize cohesion, several design patterns are used in the implementation of the program.

1. Front Controller design pattern

We implemented Front Controller design pattern where main.cc is the front controller class that handles all requests from the user and forwards them to the dispatcher. In this case, the dispatcher is controller.cc where board states, player status and everything is monitored by it, and changes are being dispatched and executed by it as well. One of the handlers is strategy.cc

where the dice information received from the dispatcher is being handled. By using this design pattern, we can achieve high cohesion by the unification of information of the front controller, and thus resulting in a more flexible and organized program.

2. Strategy design pattern

The Strategy design pattern allows the algorithm to vary independently of the client. In our program, we need to have fair dice and load dice. After reading the input from standard input and verifying that it is a valid input, the main file would pass the information to a separate file, Strategy.cc, to determine whether it is load or fair. If it is load, the Load class reads the next input number and sends it to the controller. If it is fair, then the Fair dice would take the random number from shuffle and send it to the controller.

Additionally, since we need to read the content of streams, iterator from STL Algorithms was used in our program, to separate the text in file by whitespace, and push to the vector. That is, we have used the idea of the Iterator design pattern to traverse the vector.

Resilience to Changes

We made the following changes to our program after Due Date 1.

1. We planned to use the Observer design pattern, as described in the Plan of Attack, submitted in Due Date 1 part of the project. We thought this is similar to the Game of Life question in Assignment 4. However, after some discussion within the group, we found it is hard to implement the Observer design pattern to control the board, as there are too many things we need to update each time. A tip from an instructor is there are not two many ways to initialize the board, including the random board. We decided not to use grid.cc, but wrote some helper functions to initialize the board, and we found that the board can also be updated by using those helper functions, or with some modification. Therefore, we decided not to use the Observer design pattern.

2. We also planned to use the Model-View-Controller (MVC) in this project, together with the Observer design pattern. Since grid.cc and the Observer design pattern were not implemented, it is not realistic to use MVC. Instead, we used the Front Controller design pattern. It also receives user inputs, makes appropriate decisions based on the inputs and manages interaction to modify data. They shared the same idea.

The files shuffle.cc and file.cc deal with the various initialization boards independently. Changes of the initialization to the board can be processed quickly. Additionally, features of the player can be added through the class Player, with a few modifications in the controller. Since we used the Strategy design pattern, we can easily add a new way to roll dice, by changing those classes involving the Strategy design pattern, and the main.cc, for the command line arguments.

Answers to Questions

1. You have to implement the ability to choose between randomly setting up the resources of the board and reading the resources used from a file at runtime. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?

The Strategy design pattern could be used to implement this feature. It is intended to define a family of algorithms. In this case, algorithms would be different ways of obtaining resources. During game runtime, the game would ask players for an option to choose between randomly setting up the resources and reading from file, and after receiving player's input, main will pass this information into controller.cc and set up the designated board of user's choice. Strategy design pattern also factors out common behaviours of two grids and makes our code more loosely coupled.

However, we chose not to use the Strategy design pattern. There are too many options for the board, such as *-random-board*, *-board,* with seed or without seed. Loading a game from a file would also create a board. However, in our implementation, *-board* is implemented with *-load* together in file.cc as they all involve read inputs from a file. *-random-board* is developed together with *-seed* in shuffle.cc as they all need randomly generated numbers. Although we used the same function,makeChangeGame(), to initialize the board, we did not implement the necessary classes for the Strategy design pattern, as splitting functions to different classes is extremely complicated when we finish our program.

2. You must be able to switch between loaded and fair dice at run-time. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?

We used Strategy design pattern in this case. By definition from the lecture, the strategy design pattern is intended to define a family of algorithms, encapsulating each one, and making them interchangeable. Allows the algorithm to vary independently of the client. We have decided to use this design pattern. After reading the input from standard input and verifying that it is a valid input, the main file would pass the information to a separate file, Strategy.cc, to determine whether it is load or fair, and determine appropriate function for further input. Using this design pattern could make my code more elegant, improve the readability and evolve more easily.

3. We have defined the game of Constructor to have a specific board layout and size. Suppose we wanted to have different game modes (e.g. rectangular tiles, a graphical display, different sized board for a different number of players). What design pattern would you consider using for all of these ideas?

We would consider using the Factory Method design pattern. It provides an interface for object creation, and lets the subclasses decide which object to instantiate. Superclasses will specify generic behaviours. In this case, different game modes will have different aspects such as tiles, display, size, number of players, etc. By using Factory Method, we have a general superclass which is the overall game and based on different modes, we can let subclasses choose objects like board to initiate based on the mode (e.g. set board to a different size). Factory Method thus removes instantiation of actual implementation classes from client code and makes our code less coupled. In our program, we did not use this design pattern due to the complexity it brings.

5. What design pattern would you use to allow the dynamic change of computer players, so that your game could support computer players that used different strategies, and were progressively more advanced/smarter/aggressive?

We would use the Strategy design pattern. It is intended to define a family of algorithms. In this case, algorithms would be more difficult game modes (smarter computer players). During game runtime, it would be effortless to switch modes (choose computer difficulty) using Strategy, and the current algorithm (mode) does not depend on other modes since we are using the abstract base class. The strategy hierarchy lets inheritance factor out common behaviour from the families of related algorithms, and in programming, Strategy helps eliminate conditional statements to select behaviour so we don't need to consider the use of switch statements. In the coding of the game, we did not use Strategy specifically for this scenario, however we implemented it for the dice scenario.

6. Suppose we wanted to add a feature to change the tiles' production once the game has begun. For example, being able to improve a tile so that multiple types of resources can be

obtained from the tile, or reduce the quantity of resources produced by the tile over time. What design pattern(s) could you use to facilitate this ability?

We would use the Decorator design pattern. Decorator extends the functionality of subclasses at run-time, and we can add features incrementally with new subclasses. Here, the subclass which we want to add the functionality is tiles. We want to add a feature to change the tiles' production once the game has begun. This implies that we are adding features to a subclass at run-time. In this case, we add concrete decorator classes such as addMultipleResources, reduceResources, etc. which changes the resources object from tile. To conclude, using Decorator here extends the functionality of tile -- a subclass, and we can add as many concrete decorators as we want to increase the flexibility of the program rather than using inheritance.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

In terms of developing software in teams,

Each teammate has different advantages. Before starting to write the program, each individual in the group should make sure they are assigned to parts they could do well in. This could help to group to reach its highest potential and get better performance.

Different people may have different ideas in terms of the design and implementation. Such discussion in a group is important and would help the group to get the optimal design, which may say a lot of work for all teammates.

For debugging, each teammate should be responsible for all debugging and further modification for the code he/she wrote. Other people may need to spend a long time to understand how the code works, which wastes a lot of time.

The global view is important while writing functions and classes. When you define a function or a variable, you need to consider how this will benefit not only your function, but also the entire program. This will save a lot of time, when you combine your part with others.

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

Design patterns is a significant topic in CS 246. If we have an opportunity to start again, we will make more effort on overview design, especially the use of some advanced design patterns, such as Observer. As I have mentioned above, we had no idea how to implement the Observer design pattern, so we used a more complicated method to control the board. However, we still believe the correct use of the Observer design pattern will make our code shorter.

Another improvement will be doing testing for a feature as soon as it is done and write proper comments for complicated code. When we were finishing the program, we found some small errors in some code we wrote in late November! The teammate who wrote the code took a long time to fix the bug. If we did test at late November, we would save a lot of time. In addition, writing comments would help to do modification to the code.