Course: CS246
By: Benson Chen (zc3chen), Sarah Liu (x686liu), Qiyuan Mai (q4mai)
Date: December, 13, 2020

Final Design Document

Introduction

The Constructor game is a game similar to Catan. Players build residents on the tiles in the board and gain resources on the board according to the number from the dice rolled and gain resources to build more buildings. Each building is worth some points and, in the game of constructor, the one who first has 10 points wins.

Overview

The game has five major modules, the component module, controller module, player module, startup module and view module.

The Dice class from the player module uses a strategy design pattern to let the player choose which dice they decide to use, a load dice or a fair dice. If they do not choose, then by default it's a load dice, then the dice receives a wrong input rolling number (or an out-of-range one), then the exception will rise, and it will output an error message (saying that "it isn't a valid number"), then the player needs to input a valid roll number between 2 to 12.

The Player class from the player module composites the Dice class. The player class contains the details of the current player (eg. color, resources, buildings), and the Controller will use those details to determine which action will be taken, for example, the Controller will use the colour and resources of the current player to determine if this player can add a basement. Every time an action has been taken, the information of the player and the board information will be updated.

The StartTemplate class in the startup module uses the template design pattern method. The GenerateBoard function is used in this class. By using this template design, it lets the subclasses redefine certain steps of an algorithm without changing the algorithm's structure. The player can choose to start the game with a randomly generated board, set the random number generator's seed, or load the game from a file.

The Controller class in the controller aggregates Edge, Vertex and tile class which are in the component module. The Vertex class tells the controller who the owner is, what the building is and gives the position of the adjacent edges and tiles of the current vertex.

The Edge class tells the controller if it has a road on it, and what vertices the current edge are adjacent to.

The Tiles class tells the controller what the tile value is, what the resource type is, whether it has Geese, what the adjacent vertices are and what the adjacent edges are based on the current tiles. Controller is the composition of the View class. The View class prints out the board information for the Controller.

The Controller class calls the Player class to update the information that has been changed by the player.

The Controller class calls the StartTemplate class at the start of the game. When StartTemplate is called, it generates a board, but which kind of board that has been generated depends on the player's choice.

The Controller class itself can control user actions. When the print command is used, the View class will be called by the controller to print out the current board. When the other commands are used, the controller class will call the corresponding functions inside it to implement. Notice that when the improve or build-res command is used, the player class is used, and when the load or fair command is used, the dice class is used. The printStatus function prints out the current status of all builders in order from builder 0 to 3. The residence function prints out the current residences the current player owns. The roll function allows the player to get a rolled number and decides what kind of resources each player gains, and if the rolled number is 7 the current player can choose to place the geese somewhere on the board. Notice that all of the other features about GEESE are implemented in the roll function: the player can steal one resource from another player each time, and if any player has greater than or equal to 10 resources, he/she will automatically lose half of his/her resources(rounded down). Build-road function allows the player to build a road, the build-res function allows the player to build a residence on some vertices, the improve function allows the player to improve his existing residences, and all of those three functions depend on if the player has enough resources. The trade function allows the current player to trade some resources that he has with other resources that others own, and the other player can choose to accept this trade or not by answering yes or no. The next function switches to the next player in order. The save function saves the current current game state to <file>.

The main.cc file receives commands, it takes in arguments and sets up the board. The Decorator design pattern is used here to ensure that each input is valid; if an invalid input has been received, the exception will be caught and the error message will be printed out to let the users know that they have input an invalid command.

As soon as the game starts, this game will only end once one player has more than 10 points, or an EOF has been reached.

Design

Since the game allows players to either start up using a random board or a specified board in a file, we need 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. Notice that there will be code duplication. In order to solve this design challenge, we used the **Template Method** design pattern.

Since the game has the feature that asks the players to make a decision between using loaded dice or fair dice which are two different strategies, we need to implement the ability to to switch between loaded and fair dice at run-time. We used the **Strategy** design pattern to allow players to have different strategies.

Since the game proceeds only by getting valid inputs from the standard input, we used the **Decorators** design pattern to add a functionality that does input validation to an object at run-time rather than to the class as a whole.

Course: CS246
By: Benson Chen (zc3chen), Sarah Liu (x686liu), Qiyuan Mai (q4mai)
Date: December, 13, 2020

Design Module Break Down

Player-Module

The Dice class is part of the player module, it is an abstract class and has a method which returns a number from 2 to 12. It has two classes which inherits from the Dice class, a LoadedDice class and a FairDice class. This class is implemented as planned.

The player class stores all information of a player including all its building and resources and a Dice. The player class has methods which improves a building of it's own and a method which returns the current point of the player depending on the building it has. This is done differently compared to the plan. The original plan is to have an unsigned int field to store the points, and whenever a new building is added, the points are updated. However, this way it needs the coder to manually add the points whenever adding or improving the building, increasing the risk of having bugs.

Startup-Module

The startup module has an abstract class StartupTemplate, and several concrete classes that inherits from StartupTemplate which are LoadGame, Seed, Random, and Board. The purpose of this module is to allow different command line arguments such as -load, -seed, -random-board and -board. Each command line argument has its own concrete class which will set up the board and player information accordingly. The module was not implemented as planned. Originally, all the concrete classes are simply inherited from StartupTemplate, however, during implementation, we found that -random-board and -seed are very similar, furthermore , -board and -load are similar. Therefore, we have Board and Seed inherits from StartupTemplate and LoadGame inherits from Board and Random inherits from Seed instead to reduce duplicated code and code complexity.

The Board class simply has a method which reads the layout file, and modifies the board accordingly. It will throw exceptions when the file cannot open or if the layout is incorrectly formatted. For example, when any tile that is not a park has a value of 7, it will throw an exception and exit the program.

The LoadGame class has a method which will read the current player and player information, and also throw exceptions for incorrectly formatted plate information, then, it will call the method which it inherits from Board class to read the board then it will read the geese location.

The Seed class will generate a random sequence based on the seed and set up board and player information. It has two predefined vectors which represent the tile value and tile resource and use the code in shuffle.cc file to shuffle the two vectors and then assign each of them to a tile.

The Random class inherits from the Seed class, which the only difference is that the seed is randomly generated.

Component-Module

The component modules contain a Vertex class, a Edge class and Tile class. These classes are used to store information. Vertex class has a method, ReturnEdge, that can return the edge number of all edges incident to it in a vector, and a method, ReturnTile, which tells which tiles the vertex is in. The vertex also stores the building and owner on this vertex.

The Edge class has a method, returnVertex, that can return the Vertex number of vertices incident to it similar to the vertex class, and the road and owner on this edge.

The Tile class has a method returnVertex that can return the vertex number of the vertices in the tile similar to the vertex class, and stores resources and the value the tile has.

This module is implemented as planned.

Controller-Module

The controller module is where all the operations are done. The Controller class has vectors of edges, vertices and tiles and the constructor of the Controller class will generate all the vertices and tiles and edges for this board. Furthermore, it also has a vector of players and tracks the geese location.

The Controller class has one method for each command option, for example, trade, roll, save, and ect.

The setFairDice and setLoadedDice method is used to set the current player's dice to fair or loaded.

The resident method prints the current players information using the player object in the controller.

The roll method includes the geese feature. It takes a random number from the Dice class of the current player, and if the number is a 7, then randomly take half of the resources from all players with more than 10 resources, then relocate the geese, and use the player information to check if any player has a building where the geese now located and has resource to steal. If the rolled number is not 7, then search for the player which has a building at a tile with such tile value and does not have a geese, then reward such player with the resources on the tile.

The buildRoad method takes in an edge number and checks if the edge with the edge number already has a road, and if the current player has enough resources before building the road on the edge. Similarly, buildRes method checks the tiles and current player resource and decides if the command is valid.

The improve method improves the building of the current player and updates the vertices and player information. Originally, the plan was to implement the improve method in the player class, however, later when implementing we found that in order to improve a building, we also need the vertex and edge objects. Therefore, the improved method was moved to the controller class.

The next method set the current player to the next player by incrementing the current player field. Since the current player is represented by an integer, which the current player object can be accessed by the player vector in the controller using the current player integer as the index.

The trade method uses the player objects to check if a trade is valid, that means go the the current player object and the player object that the current player is trading to, and fetch the status of each player and check if both player have enough resource to trade, and if user inputs yes then updates both player information.

The save method uses fstream to output the current player information and board information using player, edge, tile, vertex objects in the controller. The save method is called whenever a eof is

received by cin. Whenever a cin eof occurred, it throws an exception, which the program will catch and save the game and quit the program.

One change to the controller class is adding the gameInit method, which will prompt the user to enter the initial basements. Depending on what concrete starupTemplate subclasses are called, it will allow each player to put 2 basements on the board. We choose to implement this method because it allows more variety of ways players can start up the game.

Another method implemented that was not planned is the win method. The win method in the controller class will decide the condition for the player to win the game. In this case, the win method will calculate the point of the current player and will tell the main function if the player won.

View-module

The view module will take in a controller object reference and uses the vertex objects, edge objects, and tile objects in the controller object to print out the board.

Resilience to Change:

Using the **Template Method** design pattern, players are allowed to have more different ways to start up the game and generate the board.

We know that the Template Method design pattern defines the algorithmic steps in an operation and its subclasses will redefine certain steps but not the structure of the entire algorithm.

By default, the program will generate the board from the file *layout.txt.* However, there is also a command-line option to change the file that will be read and a command-line option to generate a random board layout.

In order to satisfy this feature, we used the Template Method design pattern by first initializing the resources of the board that is loading, and players can then change/redefine those resources in subclasses.

This design can also support the possibility of various changes to the program specification. For example, if we would like to provide more ways to start up the game, changes will be only made by adding more subclasses of *StartTemplate* in the *startup.h, startup.cc* files which have negligible impact on the implementation of other classes.

Using the **Strategy** design pattern, players are allowed to have more different strategies.

Known that the Strategy design pattern is designed to define a set of algorithms, encapsulate each one, and make them interchangeable, thereby allowing the algorithm to change independently of the client.

In our original program, players have their own set of dice to roll. One is loaded dice, and the other is fair dice. If a player chooses to use a loaded dice, he/she can decide what roll they will get between 2 and 12 for the turn. If a fair dice is used, the roll for the turn is the sum of two randomly generated

numbers in the range between 1 and 6. In order to satisfy this feature, we used the Strategy design pattern.

Strategy design can also support the possibility of various changes to the use of dice. For example, if the specification changes, and the program would like to provide a "*NotDice"* instead of the loaded dice described before. The "*NotDice"* will ask the player to enter a set of numbers that are between 2 and 12, and does what fair dice does until it gets the first number that is not in the set. We can make this change by removing the algorithms of loaded dice, defining the algorithm for the "*NotDice"*, encapsulating the algorithm of fair dice and that of "*NotDice"*, and making them interchangeable which allows the algorithm to vary independently of the client.

Using the **Decorator** design pattern, the program can do input validation by adding functionality to an object at run-time rather than to the class as a whole. It is flexible since it can combine multiple classes together to satisfy multiple requirements of user inputs.

If now the program is case-sensitive that accepts only the lowercase input, that is insteading of inputting the player Blue as Blue, it has input the player Blue as blue. This change to the program specification can be done by modifying the conditions of being a valid input in the Decorator.

The win method in the controller class will allow future changes to change the condition to win the game. Currently, the win method will return a string of the player if the player has 10 or more points, later if we want to modify the condition to win, we can do so by modifying the win method.

The Player class will allow the game to add more players or even to have an AI player if needed. What it needs to do is to modify the constructor for the controller to set up more players on the controller.

The view module and component module will allow the game board to change. Only need to modify the view module to have a different printing method for different controller objects of different sizes, then we can allow the game to have different size boards.

Answers to Questions:

Q1
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?

Ans:
The template method design method could be used to implement this feature and we did use it on our project. We learned from the class that the template method design pattern defines the steps of an algorithm in an operation, but lets the subclasses redefine certain steps though not the overall algorithm's structure. Hence, we assigned the initialized resources of the board, but also allowed players to modify those resources in subclasses.

Q2

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?

Ans:
The design pattern we can use to switch between loaded and fair dice at run time is the strategy design pattern. The strategy design pattern is designed to define a set of algorithms, encapsulate each one, and make them interchangeable. We did use it on our project because we need to perform the same operation, roll the dice, but use different algorithms such as fair and load.

Q3
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?

Ans:
The factory design pattern could be used to implement all of these ideas. Known that the factory design pattern provides an interface for object creation, but lets the subclasses decide which object to create. In this example, we need the subclasses to determine whether it requires to create a new object (new game mode), since we have an interface already. Using a factory design pattern allows us to produce distinct board objects. These distinct board objects can be in many different dimensions and can have various players. In our project, since we are required to use a specific board layout with a fixed size and it is not necessary for us to add distinct game modes to the existing interface, we did not use the factory design pattern on our project.

Q4
At the moment, all Constructor players are humans. However, it would be nice to be able to allow a human player to quit and be replaced by a computer player. If we wanted to ensure that all player types alway followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.

Ans:
We could use a factory design pattern to satisfy this feature. When a human player quits and is replaced by a computer player, we use this design pattern to create a computer player object, and make this computer player object own all the functionality as a human player while the AI needs to know all the information about the current plater and inherits that. To achieve this, a factory base class is needed to generate distinct player objects. When computer objects are requested, the human player object will be replaced by the new computer object.

Q5
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?

Ans:

We could use the strategy design pattern to allow the dynamic change of computer players. Since the strategy design pattern is very useful in the cases that dynamically change the behavior of a program, it is a suitable design pattern to implement computer players that use different strategies. We can implement this design pattern by first implementing an abstract base strategy class, and then implementing the concrete strategy class with many different strategies. For instance, we could have a concrete strategy class that focuses on building only, a concrete strategy class that is wiser and will attempt to do trading when it owns too many needless resources, and a concrete strategy class that will use a loaded dice to move the geese to a tile which has the most other players.

Q6
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?

Ans:
A design pattern we could use to facilitate this ability is the factory design pattern. Known that the factory design pattern is a creational pattern which is able to produce distinct objects according to different needs, so it is suitable for allowing distinct tiles to be created at distinct times, and such tiles may have distinct resources to provide. In our project, we wrote a Tile class that refers to each single tile on the board. By using a factory design pattern, we could first create new tiles with distinct resources or with repeated resources, and then replace the old tiles when needed.

Q7
Did you use any exceptions in your project? If so, where did you use them and why? If not, give an example of a place that it would make sense to use exceptions in your project and explain why you didn't use them.

Ans:
We did use exceptions in our project. Exceptions will be raised when players enter invalid inputs. For example, if a player enters a number when a valid input should be a string or enter an unrecognized command, then an exception std::invalid_argument will be raised from the stdexcept library or if the program catches any failure when reading commands from standard input, an exception ios::failure will be raised.

Extra Credit Feature:

The enhancement that we did for our projects is that we complete the entire project, without leaks, and without explicitly managing our own memory. We achieve this by using only smart pointers. Since we did so, there are no delete statements in our program at all, and no raw pointers.

Final Questions:

Q1

Course: CS246
By: Benson Chen (zc3chen), Sarah Liu (x686liu), Qiyuan Mai (q4mai)
Date: December, 13, 2020

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?

Ans:
We need better communication with each other. This time we don't know well about what others are doing, so we sometimes mess things up. Also, each module should have high cohesion and low coupling. Through this way the dependency between each module would be minimized, thus it's easier for us to test the correctness of a single module and a single change in one module doesn't heavily impact the others. Moreover, planning is really important as it helps us to work with an aim and can improve the efficiency.

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

Ans:
If we have the chance to start over we would like to plan earlier and spend more time on making plans, since it turns out that it is difficult for us to add extra features based on our initial plan. In addition, we may want to restructure our UML to achieve higher cohesion and lower coupling. More communications between group members are also needed to ensure everyone has a good overall understanding, since we had a difficult time to combine everyone's codes due to the confusion.