Introduction

The game controller we built is a four people competition game that is similar to monopoly with different rules. On each player's turn, the player can choose to use a normal dice or a loaded dice to cheat, and the player can obtain resources or trade with other players. With the resources, the player can choose to build different residences: basement, house and tower on vertices and road on edges. Each residence counts as certain points scored for the player, and as long as one player has scored 10 or greater than 10 points, that player wins, and the game ends.

Overview

In order to avoid memory leak, we chose to use shared pointers. To make sure there is no circular reference, we used raw pointers to do so. We mainly used observer design pattern for the tile and the vertex, strategy design pattern for the dice and the residence and factory design pattern for the board.

Design

We will use the main to set up the game using the given command-line options.

Class: Controller

The Controller class is designed to manage the whole game, and the class mostly follows the Model-View-Controller Architecture (MVC). Commands read from input after the initialization will be accepted and executed by the Controller class; outputs generated by the commands will be displayed by the Controller class as well. Most of the reading and processing inputs is done inside the playGame() method. The Controller class plays an important role in the interaction of human players and classes.

Class: Board

A Board object is composed of a product of the Factory Method design pattern. There are three choices available for players now, random board, layout board and loaded board. The concrete RandomBoard class will randomly create the resources, and the concrete LoadBoard and LayoutBoard will create the resources based on the given file name. The board object is based on the choice of players, and players will not create the board by themselves. The reasons for using the Factory Method design pattern is we can easily add new types of the board in the future.

Class: Tile, Edge, Vertices

The Tile, Edge, Vertices classes belong to the Board class. Since we need to check if a tile is rolled, and then distribute resources to the vertices that has building, we implement an observer pattern to tell Player that they acquire the resource. On ddl 1, we said we need an observer design pattern on vertices and tiles to observe each other. However, we did not implement it since it makes the program more complex and does not benefit us much. The relationship among Tile, Edge, and Vertice class is that a tile contains few edges and vertices, and the same Edge or Vertice may be shared by different tiles.

Class: Player

The Player class is designed to execute commands passed by the Controller class, such as building residences, building roads, and trade resources. There are four players by default, and each of them keeps track of the resources they have, the residences and road they built, and their colour. Also, each of the players has their own Dice object.

Class: Dice

The Player class is composed of a Dice class. The Dice class follows the Strategy design pattern so that players are able to switch between loaded dice and fair dice at run-time. There are two strategies, FairDiceStrategy and LoadedDiceStrategy, and we can implement more strategies for different dice features in the future.

Class: Residences

The abstract Residences class has three subclasses, Basement, House and Tower. Each of the subclasses overrides the returnNumResource() method in order to return the corresponding number of resources depending on their type of residence. More types of residences can be added as subclass of the abstract Residences class. A strategy design pattern is implemented for residence class, which is not stated in ddl 1. The different number of resources will be distributed based on the different residences; a strategy design pattern will make this easier. In the future, if we want to change the setting such as distributing different resources or multiple resources, it can be easily achieved based on strategy design pattern.

Resilience to Change

We have implemented various design patterns in order to support the possibility of changes to the program specification. Moreover, the Board class is implemented with a factory design pattern. There are three choices available for players now,

random board, layout board and loaded board. Which type of board is used is the choice of the players, and players will not create the board by themselves. The factory design pattern allows us to add more types of board in the future without notifying players, which means if a new type of board is created, players can choose to play with the board directly.

We also implement the Residence class into a strategy design pattern. Therefore we can have more specific subclasses created. We can simply add other more types of residence in the future by switching the strategy we use for Residence.

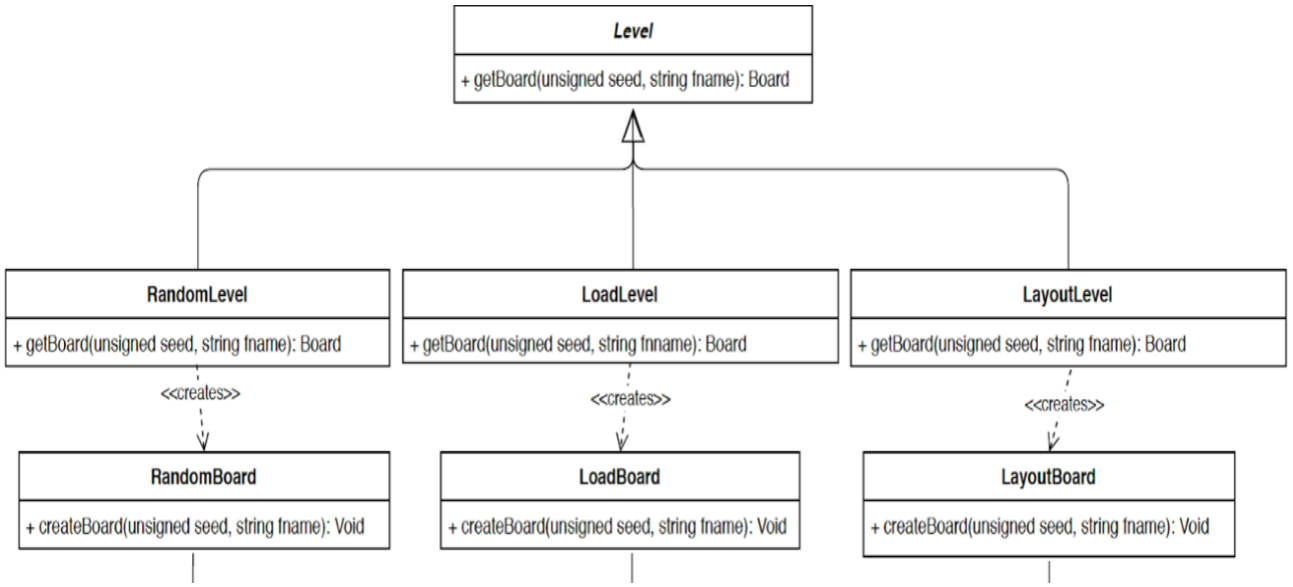
Our program is designed to maximize cohesion and minimize coupling. Most of our files serve a single purpose; for example, the Vertex, Edge and Tile class only deal with one specific point, so the functions in each class are related to each other and work on the specific object. The design pattern we used can also maximize cohesion and minimize coupling. For instance, the factory design pattern used to create boards makes the file more independent, and each board subclass only serves one purpose which is creating boards. If we want to change the features of one type of board, we only need to simply change the file that contains that type; this limits the amount of change required and reduces the chance of introducing new errors.

We also used bridge design pattern for the controller and the board class. The controller class stores a smart pointer pointing to the board. The board serves as a pimpl class, and it stores all the information of the game. This way if we want to change our implementation for handling inputs, we would only need to recompile the controller class.

Answers to Questions

1. After we finished the code, we use the same idea of what we first planned. We initially planned to use factory design pattern in order to accommodate changes. As we are implementing the code, we found this design pattern makes it easy for us to create different board. This design pattern is reasonable since the player does not need to create a board by themselves or create a board that is not what we want. Furthermore, in the future, if we want to add a new board, we can simply add a new file without changing other files.
We could use a factory design pattern to achieve setting up resources of the board randomly or from a file at runtime. Factory design patterns have higher level flexibility so that we can create additional concrete creators if we want to make new rules on how to set up the resources. Therefore, we can easily add new features such as new types of board in the future. Also, we can hide constructor from the clients,

and we can create objects without specifying their concrete classes. We have implemented a factory design pattern by having three concrete creators, RandomLevel, LayoutLevel and LoadLevel, that inherit from the abstract base class Level and each level can create different boards based on its specification. We can have a ConcreteBoard class that implements getBoard(). The getBoard() method is used to create a RandomBoard, LayoutBoard or LoadBoard object depending on the level.



2. We did used strategy design pattern, and we found it is suitable for achieving switching dice at run-time. This is because we can set the strategy at run-time depending on what type of dice that player wants to use. To implement this We created a concrete RandomStrategy class and a concrete LoadedStrategy class, and both of these classes inherit from the abstract Dice Strategy class with pure virtual method rollTheDice(). In these two concrete classes we can implement the rollTheDice() override method. Then we can create a Dice class with rollDice() method, and this method will call the rollDice() method, changing the diceNum field depending on which strategy is set. Then at run-time player can switch between loaded dice and fair dice by switching between the FairDiceStrategy and LoadedDiceStrategy.

4. After implementing the program, we would still choose to use the template method design pattern. This way we can have a sequence of actions with different implementations depending on if it is computer player or human player's turn. To implement this design pattern, we can have a concrete ComputerPlayer class and a

HumanPlayer class inherited from an abstract Player class. Inside the Player class we can have beginningOfTurn() and duringTheTurn() methods. These methods will call a sequence of methods implemented in ComputerPlayer or HumanPlayer class. For example. beginningOfTurn() will first call the roll dice method, where HumanPlayer and ComputerPlayer will implement differently. For example, HumanPlayer will take input and roll the dice whereas ComputerPlayer would just roll the dice.

5. After we finished our code, we found that in order to implement the feature that allows computer players to play smarter, more aggressively or more advanced, strategy design pattern is still the appropriate for this matter. For different stages of the program, we can have different strategies for the computer players' decisions. For instance, when a human player or a computer player is about to win, we can switch to different strategies for the computer player's actions. More specifically, when the computer player rolls a seven, we can implement a concrete strategy class for the computer player's decision. The computer player also has option of putting the geese on a tile that has residences or it can also randomly put gees on a tile, and this is implemented by different strategies. To implement this class, we can use a ComputerAction class with a string field and maybe an integer filed that represents the input command or vertex of the computer player. We can use the methods in the concrete class inherited from an abstract strategy class to mutate these fields. The ComputerAction class will implement a method that will call the Abstract strategy class's method. The ComputerAction class should also have a setStraetegy() method to decide which concrete strategy class to use at different stages of the program to determine the computer player's actions.

7. After we finished our code, we used exceptions to make sure that the input is actually an integer when we are expecting an integer. For example, if we read a string when we want to read integers, we will catch this error and output invalid integer. Also, we used exceptions to catch any EOF signals as we are reading the input. We cached them in main so that we could save the file to backup before we exit the program. There are two reasons why we want to use exceptions. The first main reason is we want to preserve class invariant. The second main reason we use exception is we can provide exception safety if we need.

Final Questions

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

First, we learned the importance of the readability of our own code. Before this project, we always tend to organize our code into appropriate styles at last minute just to get the style mark for the assignments. However, while doing the project, at the beginning, we still wrote the code in an unformal and messy way, leaving it at the end to be organized. We then realized that we could not understand each other's codes because of the poor readability with barely any comments. Thus, we had to host several long meetings just to explain each other's code; this has slowed us down a lot. After this, we started to follow the proper style while coding, then we could easily understand each other's code without wasting any time on explaining them.

We also learned that communications are crucial for team projects. In order to keep each other updated, we would have an online meeting every day at a fixed time. By doing so, we can get information on the progress of each other's work, and sometimes we need other people's code first to finish our own code; this can help us schedule our own progress. In addition to this, the meetings also make sure that no one is behind, and everyone is finishing their work on time; it is a kind of supervision.

In addition to all these, this project also taught us the importance of reading other people's code. While doing all the assignments for the course alone, we did not need to read other people's code for our own assignments, and as long as we understand the questions, we start to write our own code, following our own thoughts and ideas. However, it is different for team projects. In order to make the program work, we have to cooperate our code to each other's. To do so, we have to first read and understand others' code, following their ideas and thoughts. This is a very important skill we learned, and it would be useful in our future work.

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

If we had the chance to start over, first we would have not started the whole project so late. All the members in our team had CS251 this term, and its final exam was on the 11th. We decided to focus on CS251 first and started our project after the final. This only left us 5 days to finish the entire project with other final exams going on as well. This decision was not wise, and we had to rush over many details for the project due to the bad time management.

In addition to this, we would have written our entire program into different modules at first place. For the project, we all wrote the entire program into one file because we thought it would be easier to finalize the modulization at the end. However, after writing everything in one file, it was very hard for us to find the code we needed, and we wasted a lot of time on it. In addition to this, modulization actually helps with our entire design, offering us a clear view of our entire system design. But we did not use that tool wisely and finished the modulization after we wrapper up the whole program.

UML