Introduction

For our final project, our group choose the game constructor, which is a variant of the game Settlers of Catan based on the information provided by University of Waterloo. The program we implemented supports exactly four players (builders) and allows them to build roads or residences in the given board by using five kinds of resources. Each player will gain building points when successfully build roads or residences. The first builder who has at least 10 building points will be the winner.

Overview

Our group choose to implement constructor as our final project. We split this program into several components and use design patterns to implement them in order to achieve the property -- low coupling and high cohesion. In addition, we use smart pointers to avoid memory leak.
Before we started, we discussed how to write this program, we drew UML diagram and made a preliminary plan. When we implement this program, we found that our initial assumptions and analysis were somewhat different from the actual methods. So, we made some changes and subsequently revised our UML diagram and plan. Now we have completed this project and started to summarize the process.

Design

We construct eight classes, one structure and three enumerations. For enumerations, we define Color, Residence, Resource. In this way, type safety is improved since we restrict the value an enum variable can take in programming. For structure, we choose to define RES as a struct to store the number of resources each player has since it is an easier way to access to structure. As for classes, we define Builder, Vertex, Edge, Tile, Board, Game, Observer and Subject. The main component of the game consists of the first 6 classes mentioned above. We use observer design patten when implement the class Vertex and Edge since we need to notify observers when some player builds a road or a residence at some edge or vertex.

Struct:

RES

We collect all data for each type of resource, number of Bricks, Energies, Glasses, Heats and WIFIs, into this Struct. It is easier to access to structure than classes and using structure is a good way to reduce the complexity of our program. We find out that each builder knows the kinds of resources and when we implement the Class Builder, if RES is a class, it would be difficult to change the current resources. As a result, we implement RES as a structure instead of a class.

Enumerations:

Colour, Residence, Resource

In this program, we restrict the types of each player, resources, residences and commands. Therefore, we choose to use enumerations. We add the type “None” to Colour since when initialize each vertex and edge, none of the player builds residences or roads on them. We add the type “Nothing” to Residence for the same reason. The only difference between the struct RES and Enum Resource is the type “Park”. Therefore, when dealing with builders, we always use the struct RES and when dealing with the initialization of the tile, we use the Enum Resource.

Class Builder:

Class builder is an important part of our program since each builder represents a player of this game. We use it to store the color they represent, the resource they have and which kind of dice they use. At the beginning of a game, our program will use method **setColor()** to set four players with their corresponding color. The class can execute the commands such as **addRes()** and **removeRes()** when the current resources they have need to be updated. Also, we can get the current status of each player by using the command **getRes()**. Besides, **getColour**() allows us to get the color of player. **showColour()** is necessary to print the abbreviated string representing the color of each player. **haveRes()** shows whether the builder have any resources. **getLoaded()** and **setLoaded()** are used to get and change the state of dice.

Class Vertex and Class Edge:

For Class Vertex and Class Edge, we choose to implement them by using the observer design pattern. For each vertex, it checks whether player can build a residence at current vertex and changes the vertexColor after player successfully builds a residence. When a player builds a residence at current vertex, it will notify its vertex neighbors and road neighbors to imply no one can build roads or residences. When a player wants to improve residence at current vertex, the class will execute method **buildingByColor()** and, residenceLevel will be changed by using **setValue()** to update the data. **getColour()** is quite useful when checking whether player can build a residence or road at specific location. Moreover, methods **show()** and **showResidence()** allow us to print the board easier. Also, we can get information about current vertex by using method **getID()** and **getResidenceLevel().**

Similarly, for each edge, if a player wants to build a road at current edge, the class will check whether players can build a road at current edge by using the method **buildingRoadByColor().** If the player have the authority to build the road, the class will change the edgeColor of current edge by using **setValue()** to indicate the owner of the road. When a player builds a road at current edge, it will notify its vertex neighbors and road neighbors to indicate that the player can build a road or residences. In addition, printing the board becomes easier as we implement methods **show()** and **showResidence().** In addition, we can get information about current road by using method **getColor()**, **getID()** and **getResidenceLevel()**.

Class Tile:

A tile represents one of the nineteen tiles in the board. We use it to store tile number, type of resources, tile value, vertices and edges it has. Since for the whole board, some edges and vertices are shared and in order to avoid memory leak, we use shared pointers for edge and vertex. Apparently, method **getResourceID()** returns the type of resources of the tile. Similarly, **getResourceNum()** returns the value of the tile. After dice are rolled by players, we use this method to find out the matching tile where players can gain resources. **ShowID()**, **showResourceNum()** and **showResource()** are useful while printing the information of each tile via the layout specified in the game rule. When initializing board, we use **setValue()** and **init()** to set tiles according to the information which are either read from files or randomly chosen.

Class Board:

A board consists of all vertices, edges, nineteen tiles, the location of geese, four players and four RES resRoad, resBasement, resHouse, resTower to store required resources while building or improving houses. Most of the gameplays are in this class. When a player tries to build(improve) a house or a road, **enoughToBuildRoad()**, **canBuildRoad()**, **canBuildHouse()** and **enoughToBuildHouse()** are used to check whether the player has enough resources to build roads or houses. If the player has enough resources, the program will execute **BuildingRoad()** or **BuildingHouse()** to build the corresponding items and then **payForBuildHouse()** and **payForBuildRoad()** will be executed to deduct corresponding resources they should pay. Subsequently, **setTile()**, **setHouse()**, **setRoads()** and **addRes()** are used to update the data regarding built roads and houses.

InitGeese() will initialize the location of geese and **getGeese()** will help us find the tile number where geese is currently at. **ShowGeese()** and **showColor()** allow us to print the board easier. When 7 is rolled by a player and the geese event is triggered, method **dropResToGeese()** is used to randomly deduct resources of players who have more than ten resources totally. After that, the active player can choose a tile to move the geese, **setGeese()** is executed to move the geese to the tile the active player wants, and then **stolen()** will be executed to deal with the stole part.

When a player inputs the commend "fair" or "load" at the beginning of the turn, **setLoaded()** is used to set the dice to the type the player wants. Also, **getLoaded()** returns the type of dice player<color> has. **GetBuilderResNum()** are useful while printing the information of resource number each player has. We choose to implement **strToResource()** and **strToColour()** to help us handling inputs, these two methods can change the input string to corresponding type. During the game, when each player rolls a die, **addResByDice()** will be used to calculate the number of resources each player required.

When a player inputs the command "trade", **trade()** will be called to handle this part. An important part of this class is **show()**, which can print the board. **GetVertexLevel(),**, **showVertexLevel()**, **showExistHouse()** and **showBuilderResidence()** are used when a player inputs the command "residences". When a player inputs the command "status", **showBuilderStatus()** will be executed to print the current status of all players including the information of building points and the amount of each type of resources. We implement **getBuilderResPoint()** to calculate building points each player has such that we can easily find out whether there is a winner. **GetTileSaveData()** and **getBuilderSaveData()** helps saving the tiles' and builders' data if the player inputs "-save <filename>".

Class Game:

We design this class as the main controller of this program. It stores the board of the game, which player is in current turn and we use a vector of string to store all inputs. At the beginning of the game, the game notifies each player to build their first two basements in a sequence – "Blue, Red, Orange, Yellow, Yellow, Orange, Red, Blue". Then once the first residences are chosen, it is the Blue builder's turn, followed by Red, then Orange and then Yellow. This sequence then repeats until the game ends. Consequently, **turn()** is used to determine the sequence. Apparently, **showCurTurn()** allows us to know who is the active player. Besides, **boardFromFile()** and **loadFromFile()** are used to implement the command "-load xxx" and "-board xxx" (i.e. loading the game from the file or loading the game with the board specified in the file respectively). save <file> can be achieved by using **saveToFile()**. We use **init()** at the start of the game -- each builder needs to build two initial basements. **Roll()** is used to get the result after rolling a die. Obviously, **getAFileName()**, **getACommand()** and **getAInt()** are used to get the user's input based on different type of input. When player is confused with the commands they can input, **help()** will print all valid commands.

During the game, each player needs to roll the dice and update resources according to the dice. So **beginTurn()** is only used to handle the part of rolling dice after the initialization of two basements for each player. When rolling a seven, the active player encounters the geese event. In other circumstances, the player will gain resources based on the tile number and the type of their residences in that tile. After rolling dice, players can have diverse actions during their turns. So **duringTurn()** handles several conditions, for instance, building a road, building a basement, improving a residence, trading with others, saving the current game status, etc. **gameIsOver()** checks whether there is a winner, the first player who has achieved no less than ten building points. And finally, the last method **playAgain()** is used to prompt players that someone has won the game and whether they want to play again.

Resilience to Change

Differences compared with the original plan

Compared with the original plan, we found out that there are two main parts that changed greatly – the structure of the whole program and the implementation of some methods. To begin with, we added four enumerations, deleted class Building, changed class Resource as a structure in our program. The logic behind Class vertex, Class edge and Class tile changed greatly. Before we started writing code, we though that Class tile should consist 6 vertices and 6 edges. If we want to change the state of vertices or edges, when players successfully built a house or a road or improve a residence, we first need to find which tile the vertex or the edge is located at. Then, we could change the status of the vertex or the edge by methods we implemented in the Class tile. Since most of the vertices and edges are shared by more than one tiles, we chose to use shared pointers. In this way, when we modify the status of a vertex or an edge in one tile, the corresponding status will also be updated. However, when we were trying to implement our design, we found some shortcomings in our plan and our code just messed up. Thus, we reimagined this part and changed our plan from finding vertices and edges by tile to finding the corresponding tile by vertices and edges. In this way, we can avoid many problems caused by improper use of shared pointers. Also, this plan makes it easier for us to debug and thus more convenient to use.

Also, at first, we used a string to store the state of each vertex and edge whether players can build here. Since we used the observer design pattern for Class edge and Class vertex, when a residence is built at a vertex, it should notify all vertex neighbors and change the status of them to "N", which indicates that no residence can be built. Also, it should notify its edge neighbors that this builder can build roads. Similarly, when a road is built at an edge, it must notify all edges neighbors and vertex neighbors that the current builder has successfully build a road such that all of its neighbors would know which builder can build residence or road. When we write these parts, we found there is a problem. For instance, imagine that there is a vertex that has three edge neighbors. Road is successfully built by three different players at each neighbor of this vertex. This vertex would has received messages regarding which player can build residences. If we just record whether residences can be built rather than who can build, scenarios will be quite complex. Therefore, storing the state is not the best solution. That's why we introduce the Enum Colour and use it as a field for both class Edge and Vertex. In this way, when updating the status of every observer of the current subject, the vertex or the edge, all problems are solved easily. If a vertex or an edge has no building or road built at it, the default color of them is None. Once a builder builds a road or a residence at a vertex or an edge, the color of the vertex or edge changes to the builder's color.

Possible features

Low coupling and high cohesion allow out program to add more features by changing little code. For example, if we want to change to the number of players, we only need to add more colors in the Enum Colour, which is the representation of players, and change the number of players in the class board. Similarly, if we want more kinds of resources, we need to change the RES, Resource and some methods regarding displaying them. In addition, in the case that we want more kinds of boards, for instance different sizes, shapes etc., we can use a decorator design patten to implement this feature. We need to overwrite the function **show()** which is a method used to display the board. We can also add more gameplays by adding more methods in the class Game. For instance, we can introduce the role of "market" -- players can get the scarce resources by giving many resources to the merchant. Players can also buy weapons so that if the geese event is triggered, they can use them to attack the geese and as a result, they will lose less resources depending on the level of their weapons. In this case, we can use a factory design pattern and add a class Weapon to handle different cases.

Answer to Questions

Question 1

Answer:

Strategy design pattern can be used to implement this feature and we choose to use it in our program. We implement one base class Strategy and two derived classes, namely **randomBoard** and **readingBoard** to apply strategy design pattern. There is only one method to override in the derived classes which is very straightforward to accomplish and very intuitive to read. In this way, we can switch between different algorithms in runtime easily, in specific, switch between randomly setting up board and loaded board in this program. If we choose to not use this design pattern, we need to implement this feature by using lots of additional methods and conditional-infested code which will make our code look messy.

Question 2

Answer:

We can use strategy design pattern to switch between loaded and fair dice at run-time. At first, we choose to use it in our program, but when we are implementing this part, we decided not to use the design pattern. We found out that if we decide to add a class Dice, there will be only one method for both loaded dice and fair dice – **roll()**. For loaded dice, we only need to check whether the input is valid (i.e. check whether the input is a integer between 2 and 12). For fair dice, the points should be the sum of (rand() % 6 + 1) and (rand() % 6 + 1). Therefore, we decide to implement this feature by adding a field loaded in the Class Builder. loaded is a Boolean representing whether the player uses a loaded dice. For each player, we can easily change the status of dice if they input “loaded” or “fair” by using **setLoaed()**. Obviously, strategy design pattern increases the complexity of the code and thus we do not use it.

Question 3

Answer:

We will consider using decorator design pattern to implement different game modes in our program. Decorator design pattern allows behavior to be added to an individual object dynamically without affecting the behavior of other objects from the same class. For example, if we want to have a different sized board for a different number of players, we can add additional features. To implement different sized board, we can add methods **addTile()** and **deleteTile()** in Class Tile based on our original code and to implement different number of players, we can add methods **addBuilder()** and **deleteBuilder()** in Class builder. We can extend different features to objects without affecting other objects. In addition, decorator design pattern is an alternative of subclasses which adds behavior at compile time and may affects other objects of original class; decorator design pattern can provide new behavior at runtime for individual objects. Besides, decorator design pattern allows us to define a simple class and add functionality incrementally with decorator objects instead of a complex customizable class.

Question 4

Answer:

Since we want actions of all players to form a legal sequence, we consider using template method design pattern. Under this design pattern, we define the skeleton of the algorithm in the superclass and choose to override some of them. So, in the abstract class, we declare methods that act as steps of the algorithm, for example:

roll a dice -> check whether can upgrade the residence -> check whether can build a residence
-> check whether can build a road -> end of the turn

In the real scenario, the steps would be much more complex than this example. The following scenario should follow the assumption that all players use fair dice. For instance, the computer player should consider the maximization or resources and when the player asks to trade, computer should analyze which resource is most scarce for him and then make the decision. Also, when the geese event is encountered, the computer builder should place the geese at the tile where the number of residences (except his own residences) is maximum if some or all of the players except himself have more than ten resources in total. If the computer finds out that there is nothing to do, then it should end this turn. In addition, the possibility of successful trade should be 0.5 and the result should be random since if the computer can always trade successfully, it will “plunder” all resources from others and thus would always be the winner.

Question 7

Answer:

Before we started to write code, we mistakenly thought that we should use exceptions in some places. One of them takes place when the player decided to use the loaded dice and then the player input a number outside the range of 2 to 12. In this situation, the program will output "invalid roll". When the input is not valid or out of range, we find out that we can handle these conditions by using a while loop (i.e. until we get a valid input). However, we still using exceptions in many places. For instance, the trade part. When a player inputs "trade", there are several possible cases. The active player may trade with the player himself or give a resource to take a same resource. The active player may give a resource he does not have to take a resource from other player or ask other player for a resource they do not have. Thus, we use exceptions in this part to help us handling these cases since it makes our code more intuitive to read and easy to understand. Also, when the geese event is encountered, the active player should choose a tile to place the geese. If the player responds with the number of the tile where the geese is currently placed or a non-number input, we throw an exception. After the builder successfully choosing a tile to place the geese and there are some players who can be stolen from, if the active player chooses a player not on the list or input invalid commands, an exception is also needed. In addition, when a player fails to complete the action which requires more resources than the player currently owns, we throw an exception and print "You do not have enough resources". All methods regarding building or improving residences or roads should provide strong guarantee since if the player fails to build or update the residence or road, resources should not decrease, which represents that any modification in the program state made by these methods needs to be undone.

Final Questions

Question 1

Answer:

It is such a great experience working as a team to finish this project. Since our group is made of only two, work per person would be heavier than those three people groups. However, it allows us to learn more and become more familiar with C++. At the beginning of the program, we first reckoned that we could separate the program into some parts, describes like the plan of attack. Thus, we implemented basic.cc and basic.h these basic parts together and then separately wrote our own part. However, as we wrote some hard parts, for instance the Game and Board class, we concluded that it would be better if we can share some knowledge so that we can come up with solutions immediately by communicating with each other rather than wasting several hours but still cannot figure out a solution. Moreover, when writing class Board, we found out that when initialize the board with default layout.txt, geese is not shown in the board, and we came up with two solutions. The first one is to place the geese in the park and when outputting the layout, we ignore it while the second one is to give the position of geese a very large number (in our program, 100). We thoroughly discussed the merits and drawbacks of these two methods and we finally used the latter one. Since it is the first time that we implement a program completely by ourselves without some well-completed parts that school provided, we imitated the project completed in the previous assignment and analyzed which part we need to add or delete. However, when we started coding, we realized that there are lots of problems since we do not have the "absolutely" correct header files to refer to. Thus, we can only write the code and modify methods in header files at the same time. This is a huge challenge of our patience and when we overcome this difficulty, we have a deep sense of pride and accomplishment.

Question 2

Answer:

If we could start over, we would start this program earlier. In this situation, we would implement some features, by using design patterns we have learned so far. If we have enough time, we may learn some other useful design patterns by ourselves. We will not give up the bonus part and will try our best to implement more features, such as the market. When we finished all basic parts of the program, we did not have enough time to implement market this command. Besides, if we have the chance to start over, we will choose to find another GENIUS for our group who is only responsible for design.pdf, demo.pdf and UML. Both of our existing members of our group strongly agree that these three parts are much more difficult than writing code. :(

Conclusion

This program allows us to review most part we have learned so far and brings us more knowledge not only about C++ but more importantly the skill of self-study, communication and cooperation. It allows us to realize the importance of teamwork, which is quite helpful to our future. :)