FINAL PROJECT-PLAN

Siming Xu Amanda

Yihui Zhang

12.1 2020

Plan of Attack：

Plan of Attack			
	Tasks	**Responsible**	**Estimated Time**
1	Draw UML	Amanda, Hui	4 hours
2	Plan of Attack	Amanda, Hui	2 hours
3	Implement Header files	Amanda	2 hours
4	Implement builder.cc building.cc basement.cc house.cc tower.cc	Amanda	6 hours
5	Implement resource.cc tile.cc board.cc	Hui	6 hours
6	Implement gameplay.cc dice.cc vertex.cc edge.cc	Amanda, Hui	10 hours
7	Implement main.cc compiling	Hui	6 hours
8	Debugging	Amanda, Hui	Undefined behavior
9	Testing	Amanda, Hui	3 hours
10	Implement graphics	Amanda, Hui	2 hours

Design of Classes:

Class Gameplay:

We design this class as the main controller of this program. The first two methods are used to begin and decide whether there is a winner of the game. Other methods are read from input, accepted by the Gameplay and finally will be passed to other classes.

Class Builder:

Each builder represents a player of this game. We use it to store the game data of each player, for instance, current points they get, all buildings they built, total amount of resources they own, all roads they build, which color they represent. Also, it can execute the commands such as building residences, building roads, trading with other players, stealing resources from other players when rolling a seven and thus geese event is triggered and printing the current status of each player.

Class Building:

A building represents the residence that each player can build at each vertex. We use it to store the location and type of each building. It has three sub-classes:

Class Basement, Class House and Class Tower.

The reason for separating them into three classes is that for each type of the building, it has different properties. For instance, they need different amounts of resources to update to the next level of building. Tower cannot update since it is already the highest level. All of them can execute the command printBuildings().

Class Tile:

A tile represents one of nineteen tiles in the board. We use it to store tile number, type of resources, tile value, vertices and edges it has and whether it contains geese. Thus, Class tile is composed of Class vertex, Class edge, Class resource. The reason for separating the key components into different class is to implement high cohesion and low coupling. Also, when player roll a die in the player's turn, the method getValue() can help us lock the tile quickly.

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 v, it checks whether players can build a residence at v and changes the name after player successfully builds a residence. When a player builds a residence at v, it will notify its vertex neighbors and road neighbors to imply how the state need to be changed. Similarly, for each road r, it checks whether players can build a road at r and changes the name of current road. When a player builds a road at r, it will notify its vertex neighbors and road neighbors to indicate how the state need to be changed.

Class Resource:

We collect all data for each type of resource, number of Bricks, Energies, Glasses, Heats and WIFIs, into this Class. Whenever the resource of each player needs to be changed, we just call the method updateResource(). When a seven is rolled by some player and geese event is encountered, for inactive players who owns more than ten resources in total, the player will lose half of resources by calling the method loseResource(). Also, the print method is helpful when players call the command status to show the current information of resources they owned as well as others'.

Class Board:

Class board is only used to store the data of nineteen tiles in the current game, and therefore, it can display the image of the entire game board. We use the strategy design pattern to implement the source of the board, randomly setting up or reading from an existing file.

Questions:

Question 1

Answer: Strategy design pattern can be used to implement this feature and we choose to use it in our program. The first reason is that using the strategy design pattern can make it easy to switch between different algorithms in runtime, in specific, switch between randomly setting up board and loaded board in this program. The second one is using it can make our code cleaner since we can avoid conditional-infested code. Also, using the design pattern allows the algorithms to vary independently of the clients since it uses the abstract base class. In addition, the algorithms in use can be dynamically changed.

Question 2

Answer: We can use strategy design pattern to switch between loaded and fair dice at run-time and we choose to use it in our program. Encapsulating the algorithm into separate strategy classes allows us to vary the algorithm independently of its context, making it easier to switch between types of dice, in specific, loaded

dice and fair dice. In addition, it eliminates conditional statements to select behavior. However, it increases the complexity of code since there is only one method in the class Dice and we only need to override roll() for loaded dice and fair dice. The merits of using strategy design pattern outweigh the disadvantages and thus, it should be a good idea to implement it.

Question3

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.

Question4

Answer: Since we want actions of all players to form a legal sequence, we consider using chain of responsibility design pattern. Under this design pattern, a chain of objects receives requests passing from the client and processes them. Therefore, the computer player should be the client. The following is an example of the chain that should be processed by the handler.

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 chain 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 of the resource 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 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.

Question 7

Answer: We use exceptions in many places in our program to handle error conditions. At the beginning of the game, each player needs to build two basements, so they need to input a number which represents a valid vertex. If the player inputs an invalid vertex, we need to throw an exception and continue asking them where to build the residence until a valid vertex is entered. The second one is in Class Dice, 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”. Besides, when player tries to complete an action which requires more resources than the player currently owns, we throw an exception and print “You do not have enough resources”. Since the player fails to complete the action, the state of the resource should remain unchanged.

As a result, all methods regarding updating resources should offer strong guarantee. 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, if the builder wants to improve or build a residence, or build a road and the space is invalid, we should use exception. 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. Finally, if the player input an unrecognized command, an exception is needed, and the program should print “Invalid command”.