Constructor

by Aiden Li, Frances Wang, Yi Bo Cheng

Overview (describe the overall structure of your project)

Our project contains several layers of classes.

Catan

The top layer is the class Catan. Catan contains all of the functionalities and objects needed for the game to run. For example, for a game of Catan to run, it requires a Board and information about the players that are playing; this is exactly what Catan contains: a pointer to a Board and a vector of pointers to Builder (the players).

As for its methods, the key method to note is playTurn(). This method goes through a single turn for the game. The function will take command inputs which are listed in 4.2 and 4.3 and will call the private methods related to the command accordingly. These private methods usually have a great relation to either the Board or the Builder. Other important methods to note are the load methods. These methods are ways to create a Board, load, and set up its Board field.

Since a lot of commands in the game mainly change either the board or the builder's information, Catan has a high relationship with Board and Builder and often uses public functions from those classes to update its Board and Builder members.

Due to the class being the top layer and the wrapper that contains all elements of the game, the main file uses this class and its public methods to run the game.

Builder

The Builder class contains information for a player which is named a "builder" in this game.

A Builder stores its Colour, Dice, a map that contains information on the Builder's residences, a vector containing information on the Builder's roads, a counter for its building points, and an Inventory object which includes information on the resources it has. Most of its methods modify its own fields as it does not have a lot of relationships with other classes

Board

The Board class contains all information about the state of the game's board.

The class contains a Geese object and lists of Tile, Vertex, and Edges, objects which are the information on the game's tiles, vertices, and edges respectively. The Geese field is an indicator of where the Geese is currently located on the board.

Most of the Board's method modifies its fields which are vectors of Tile, Vertex, and Edges objects along with their fields; these classes will be explained later. For example, here is where we can change a Vertex's residence type, build roads on Edges, and also start the distribution functionality via a Tile.

Tile

A Tile object contains information about the resource type, number, and value of a tile.

Notice that the tile does not contain information on the vertices that surrounds it. This is because we decided to implement the observer pattern where the Tile act as a subject with the vertices surrounding it are observers of this Tile. This will be explained in more detail in the Design section. This pattern allows us to notify the observers which let us distribute the resources to those Vertex objects.

Vertex

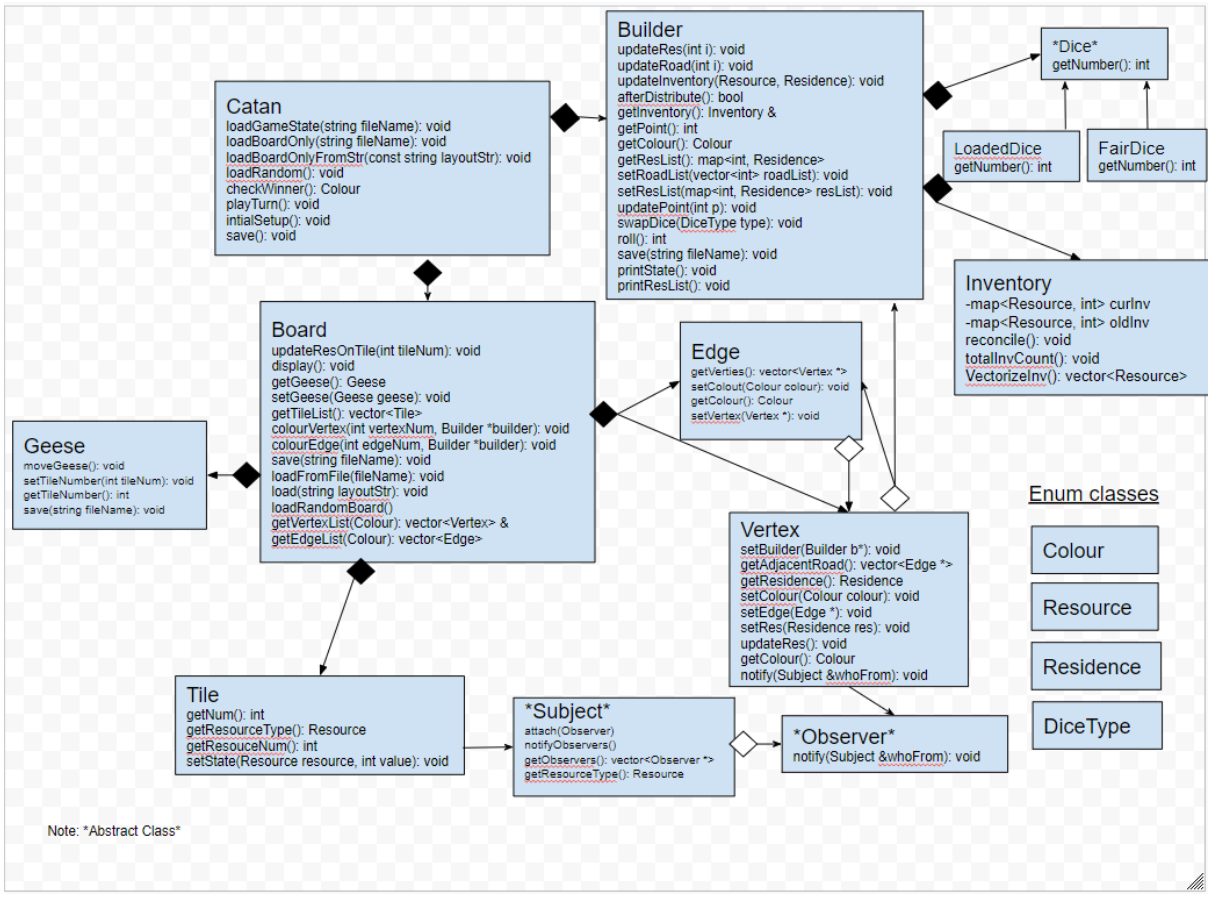
A Vertex object contains information about the number of the vertex, the Edges adjacent to it, the type of Residence on it, the Builder that occupies it, and the colour of that Builder.

As previously mentioned, this is the observer of Tile. Using the observer pattern along with the Builder pointer that occupies the Vertex, we are able to distribute resources.

Edge

The Edge class contains information about the number of that edge, the vertices that it is in between, and the colour of the edge if it's occupied.

Updated UML



Design (describe the specific techniques you used to solve the various design challenges in the project)

Modified Observer Design Pattern:

The most important technique we used to facilitate the implementation of our functions is the observer design pattern. It is considered as the core of our program, as it handles the most essential feature of the game - Resources distribution. It helps the program decide (1) which Resource(s) to distribute, (2) which Builder(s) will receive the resource, (3) how many of eac

h Resource will a player receive. Our design chose to take Tile objects as subjects and Vertex objects as observers. The reason is that Tile class contains the Resource type(resource) and the tile value(value), which are used to check if it's a matching tile and (1). Also, Tile has an observerList which contains pointers to all the vertices on each Tile (2). So, once a matching Tile is found, we call the notifyObservers() method to notify each Observer in the List. With Observer class being an abstract class (i.e. with pure virtual methods), the actual Vertex object will call the overridden notiyfy(whoFrom) method that updates the Builder object's (Builder *) Inventory if there is one bound to it. To decide (3), we simply check the Residence type in Vertex class and then static cast it to integers and add it to Builder's inventory. It is worth mentioning that rather than having a Tile * object in Vertex class to get the state(Resource type) of the Tile that notifies it, we chose to pass a Tile Object as an argument to notify(Subject &whoFrom) and add getResourceType() to subject.h as a pure virtual function. In this way, we do not need to manually initialize the associated Tile * in each Vertex object. Hence, our UML doesn't have the aggregation relationship line pointing from Vertex(ConcreteObserver) to Tile(ConceeteSubject), which is slightly different from the one we saw in class.

Abstract Base Classes:

We also used abstract base class(Dice) for LoadedDice and FairDice and overrode the getNumber() method.

Aggregation Relationships:

(1) Edge class and Vertex Class:
    (a) Edge class has a data member std::vector<Vertex *> twoVertices which contains the two vertex ends of each edge
    (b) Vertex class has a data member std::vector<Edge *> adjacentRoad which contains all edges that have an end of the current vertex.
(2) Vertex and Builder:
    (a) Vertex has a data member Builder* builder, which points at the Builder object who created a residence on this vertex.
(3) Observer Design Pattern

Composition Relationships:

(1) Builder class and Dice class:

(a) Builder has a data member Dice *dice, which is a use of polymorphism(i.e. LoadedDice and FairDice)

(2) Builder and Inventory:

(a) Builder has a data member Inventory, which is a struct that contains two maps: curInv and oldInv (*****)

(3) Catan and Builder:

(a) Catan has a data member std::vector<Builder *> builderList which contains the pointers to four Builder objects

(4) Catan and Board:

(a) Catan has a data member Board *board which is the board used in the game

(5) Board and Edge, Vertex, Geese, and Tile:

*****: Having both std::map<Resource, int> curInv and std::map<Resource, int> oldInv is to decide if each roll has any effect on any builder's inventory by comparing their values. If oldInv and curInv are exactly the same, then we just print "No builders gained resources", otherwise print the required lines to show the resource updates on each builder's inventory.

Resilience to Change (describe how your design supports the possiblity of various changes to the program specification)

We designed our program with its resilience to change in mind. Most methods in our program are not hard coded to check elements that can be scaled. We adopted low coupling and high cohesion in our project to account to allow better resilience to change.

An example of this is how our program checks for builders/players. Since we knew that the number of players can easily be increased or decreased, we chose to use a low coupling design between Catan and Builder as we store players/builders in a vector instead of a field for each of the four builders. With this, most of the functionalities that check for builders/players will use a range-based-for-loop. Doing so allows us to easily account for changes in the number of players in a game. Furthermore, as for players' colours, we included an enum class Colour and each builder contains a private Colour field. With this, we can add more colours and change the colours of each builder.

Another scalable element in the game is the types of resources. We knew tha t the number and type of resources can be scaled, so we decided to use a lo w coupling design between Inventory and Resources by giving each player an Inventory object which includes a std::map that records all elements; doing so allows us to avoid hard-coding each element type when it needs to update the player's inventory. Thus, if there were to be more elements added, we' d only have to modify the Inventory and the methods related to it. However, due to time constraints, not all of these features were achieved and a few methods required hard coding for types of elements. The types of elements a re included in an enum class to allow for the easy addition of new types of elements to the game.

For the "thief" of the game, we kept in mind a possible extra feature tha t involves adding more thieves and other types of thieves. Therefore, we ma de the thief Geese into a separate independent class. doing so can allow fo r other types of thieves to be added by following a similar pattern to Gees e's.

Lastly, all of our modules are highly cohesive since they are all elements required to play the game of Catan.

Answers to Questions (the ones in your project specification)

1.

We can use the factory design pattern. Since we never know exactly which wa y of setting up the resources of the board will be used, we don't want to call ctors directly, moreover, we don't want to hardcode the decision poli cy, as we want it to be customizable.

Since our resource generation is random, we can have a concreteCreator clas s along with concrete subclasses being each subclasses of Resources, which are each type of resource (WIFI, HEAT, etc.). In the concreteCreator class, we can have a function to generate a random resource for each tile as well as customize its probabilities of production.

We did not use this pattern. Due to the given shuffle function, we had a be tter method of randomizing specific numbers of each element: We put all the possible resources into a vector and shuffle them and use that vector to se t up the layout and board. This way, we do not have to calculate the probab ility of each resource and then set up the board that way.

2.

Decorator design pattern could be used to implement this feature, but we didn't apply it to our design. Because we found it might be easier to just have an abstract class called Dice, and two concrete subclasses of Dice, LoadedDice and FairDice which override the behaviour of the pure virtual function getNumber(): int in the Class Dice. Whenever we want to add another dice in the future, we can just create a new subclass of Dice (i.e. NewDice) which dedicates to fullfill the new behaviour of that NewDice. We don't need to change anything in Dice.h or Dice.cc. To allow the players to switch between their dices, we just created another function called swapDice(DiceType): Dice which creates a Dice object and assign it to the data member dice of the Catan object. Changing a dice will only change the matter of the number that we get during the roll, we feel there's no need to complicate the implementation.

3.

We could use the decorator design pattern for different board layouts and sizes. We can use an abstract class Board with concrete classes (e.g. hexagonal tiles) corresponding to each different board layout. By overloading the pure virtual function board(int size) in the class Board, we can successfully print the board with a different layout and size. For different numbers of players, we can add a field in the enum class colour and also add turns to match the number of players, and others need not be changed.

6.

We can use the decorator design pattern. Since the game has already begun, we can't change the tile by changing the board, thus we must decorate that tile. By having a decorator for tiles, we can easily add additional effects that change the tile via the decorator. For example, our standard tile from the original game (i.e. 1 resource with a standard multiplier of resource given) with the decorator can decorate the specified tile by adding on additional decorator subclasses via the Decorator. These subclasses can vary. To add additional resources to a tile, a decorator subclass can start by adding to the concrete component. To add more resources to that tile, the decorator can add, calculate, and return the total resources that are on the tile (similar to how rules are added to Cells from A4Q2). Similarly, to add a multiplier to the resources given out at that tile, a decorator subclass can

also add, calculate, and return the total multiplier. The concrete component tile will have a multiplier of 1, and more decorator subclasses will be added through a linked list.

7.

Yes, we used exceptions to handle the case when players entered EOF. Whenever a player entered an EOF, the executing function will throw a boolean(true). We have the try-catch block in the main function to handle this exception by saving the current game state to backup.sv and returning to end the program.

```
try
{
    catan->playTurn();
}
catch (bool)
{
    cout << "User Entered EOF - Game over. Current game state is saved to backup.sv." << endl;
    catan->save("backup.sv");
    delete catan;
    return 0;
}
```

Final Questions (the last two questions in this document).

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?

   The importance of the designing and planning phase. Without a clear design in mind, this project would have required a lot more time due to a lot of refactoring and changes if we were to design as we go. Having a design also allows each member to work on a separate part of the project for better productivity.

   Programming is very different for everyone. Everybody has their own style of coding and design and it could be difficult trying to understand what each other has in mind. For example, since all of us wrote many of our own method, we had to try our best to understand what each other wrote, even if we had a different idea or design in mind of the implementation of the function.

The importance of code communication. In a team environment for software development, the method used to communicate code played a bigger role than we anticipated. For our group, we used Google Drive to send code and files to each other. However, this led to a lot of challenges. One major disadvantage is that conflicts are much harder to solve. Conflicts must be communicated through messages. Furthermore, changes also needed to be communicated to avoid more conflicts. Also, changes are much harder to locate if there are small changes spread throughout the files; this can lead to accidentally overwriting someone else's changes or fixes. Lastly, the uploading and downloading process onto the drive unexpectedly took a lot of time.

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

The major thing is to design a better plan of attack that allocated more time for testing and debugging. In such a large project, errors are bound to occur and the smallest error can lead to hours of debugging. Furthermore, due to the scale of the project, and not all the code is written by the same person, there are parts the other members might not understand, which can lead to more time needed to debug. This is exactly what happened to us and a lot of hours have gone into debugging each other's code just a few days before the deadline. Thus, we definitely needed to plan for a lot more time for debugging if we were to start over

Another thing is to consider using other methods of code communication such as Git and GitHub. Doing so will allow us to spend less time solving and finding each other's conflicts and use the time to improve productivity on the project.