This is the plan of the CS 246 Fall 2020 final project Constructor, developed by Zhilin Zhou (Catherine) and Xinru Cheng (Meredith).

1 Task Breakdown

1.1 Planning

We will plan the project by having an initial UML diagram of the structure as well as an outline of our timeline finishing the project.

The UML diagram is enclosed at the end, and the timeline will be outlined in the following sections.

To organize our coding, we have set up a git respository to save all our commitment to this project. We also have set up a discord group to make our communication more convenient (especially in the current social distancing situation).

1.2 Make Tests

We will first make some tests of our program. After developing a runSuite file, we will include about 20-30 tests with the given sample executable.

1.3 Develop Project

After setting up the tests, we will start developing the project. We adopt the MVC design pattern, which is also shown in the UML diagram, to structure our program.

Part distribution:

- Meredith: Model, 15 tests, Makefile, runSuite
- Catherine: View, Control, 15 tests, design.pdf

1.4 Testing and Debugging

We will use the tests developed previously to test our finished program. We will enter the debugging process when identifying any errors.

2 Timeline

Month of December:

S	M	T	W	T	F	S
		1 Finish planning and UML	2 **DD1**	3	4	5 Finish writing tests
6 First meeting, current progress and discussion	7	8	9 Second meeting	10	11	12 First draft, start testing
13	14 Finalize program	15	16 **DD2**	17	18	19

3 Answers to Listed Questions

1. 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?

   We will use the **Factory method** design pattern to implement this feature. We will decide the resource of setting up the board at runtime, and using the factory method can help us decide which object to create.

2. 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?

   We could use the **Template method** design pattern to immplement this feature. We will have a abstract `dice` class, which contains a virtual `roll` function. We will have `load_dice` and `fair_dice` inheriting the virtual method and override the function by their methods.

3. 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 numbers of players). What design pattern would you consider using for all of these ideas?

   We will use the **Factory Method** design pattern to implement this feature. We have discussed to add graph display to this game as the add-on feature. We will have a abstract `display` class, which includes the general functions for view. We will also have `textDisplay` and `graphicDisplay` classes to inherit `display` and decide to use which object at runtime.

4. 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.

   We will use the **MVC** design pattern to implement this feature. We will have a `controller` section, which reads input from the user and decide whether or not the input is legal before passing it to the `model`.

5. 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?

   We can use the **Decorator** design pattern to adopt more advanced and smarter strategies from computer players. We can add more functionality to our program by using the Decorator during runtime.

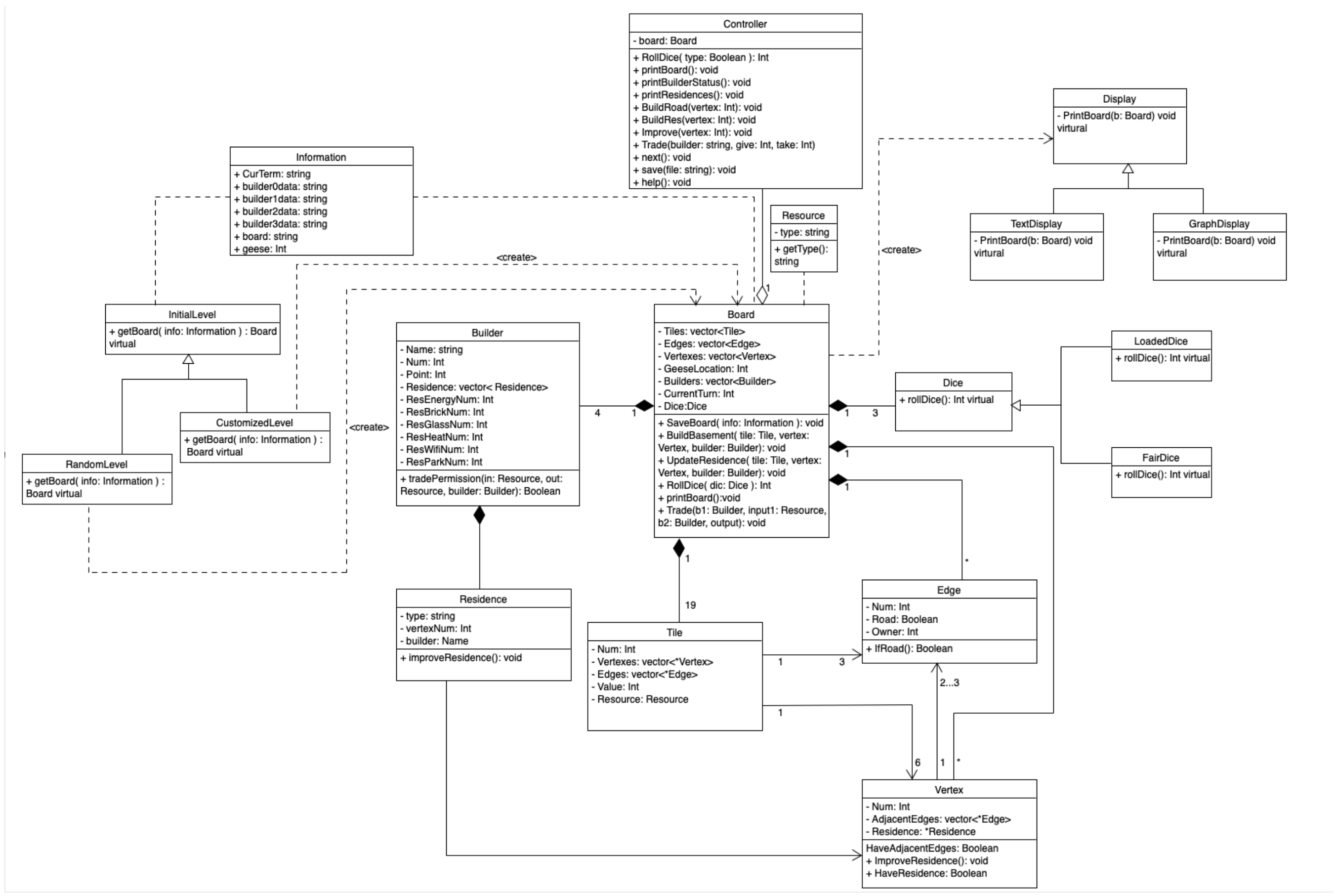
6. Suppose we wanted to add a feature to change the tiles' production once the game has begun. For example, being able to improve a tile so that multiple types of resources can be obtained from the tile, or reduce the quantity of resources produced by the tile over time. What design pattern(s) could you use to facilitate this ability?

   We will use the **Decorator** and **Observer** design patterns to implement this feature. Using the decorator, we can add functionality and features to our program at runtime and withdraw it at any time. Using the observer, once one class has changed its feature/state, we can notify all the other classes to adopt this change using the observers.

7. 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.

   We will use exceptions in our project, such as throwing `InvalidCommand` when the player inputs an invalid command, or `NoPermission` when the player cannot build house at a certain vertex. Using this strategy could make our program exception safe and will not crash on these illegal actions.

4 UML Diagram


Controller
- board: Board
+ RollDice(type: Boolean): Int
+ printBoard(): void
+ printBuilderStatus(): void
+ printResidences(): void
+ BuildRoad(vertex: Int): void
+ BuildRes(vertex: Int): void
+ Improve(vertex: Int): void
+ Trade(builder: string, give: Int, take: Int)
+ next(): void
+ save(file: string): void
+ help(): void
Display
- PrintBoard(b: Board) void virtural
TextDisplay
- PrintBoard(b: Board) void virtural
GraphDisplay
- PrintBoard(b: Board) void virtural
Information
+ CurTerm: string
+ builder0data: string
+ builder1data: string
+ builder2data: string
+ builder3data: string
+ board: string
+ geese: Int
Resource
- type: string
+ getType(): string
<create>
<create>
<create>
1
InitialLevel
+ getBoard(info: Information) : Board virtual
CustomizedLevel
+ getBoard(info: Information) : Board virtual
RandomLevel
+ getBoard(info: Information) : Board virtual
Builder
- Name: string
- Num: Int
- Point: Int
- Residence: vector< Residence>
- ResEnergyNum: Int
- ResBrickNum: Int
- ResGlassNum: Int
- ResHeatNum: Int
- ResWifiNum: Int
- ResParkNum: Int
+ tradePermission(in: Resource, out: Resource, builder: Builder): Boolean
4
1
Board
- Tiles: vector<Tile>
- Edges: vector<Edge>
- Vertexes: vector<Vertex>
- GeeseLocation: Int
- Builders: vector<Builder>
- CurrentTurn: Int
- Dice:Dice
+ SaveBoard(info: Information): void
+ BuildBasement(tile: Tile, vertex: Vertex, builder: Builder): void
+ UpdateResidence(tile: Tile, vertex: Vertex, builder: Builder): void
+ RollDice(dic: Dice): Int
+ printBoard():void
+ Trade(b1: Builder, input1: Resource, b2: Builder, output): void
1
3
1
1
Dice
+ rollDice(): Int virtual
LoadedDice
+ rollDice(): Int virtual
FairDice
+ rollDice(): Int virtual
1
19
*
Residence
- type: string
- vertexNum: Int
- builder: Name
+ improveResidence(): void
Tile
- Num: Int
- Vertexes: vector<*Vertex>
- Edges: vector<*Edge>
- Value: Int
- Resource: Resource
Edge
- Num: Int
- Road: Boolean
- Owner: Int
+ IfRoad(): Boolean
1
3
2...3
1
6
1
*
Vertex
- Num: Int
- AdjacentEdges: vector<*Edge>
- Residence: *Residence
HaveAdjacentEdges: Boolean
+ ImproveResidence(): void
+ HaveResidence: Boolean

Using the MVC design pattern, we divide our project into three part: model, view, and controller.

The controller section includes `Controller`, which contains a pointer to the board, takes in input from the user, and calls the corresponding board functions to operate the commands.

The model section includes `Board`, `Builder`, `Tile`, `Residence`, `Edge`, `Vertex`, and `Dice`. After the controller calls the functions in `Board`, the `Board` will calls some other functions to finish the commands and turns the output to view.

The view section includes an abstract `display` class and two subclasses, `textDisplay` and `graphicDisplay`. We will decide to use which display mode (create which object) during the runtime. After executing the commands, the model will call the corresponding function in `display` to output the display.