CS 246 Final Project – Constructor

Documentation and Design

Xinru (Mereidth) Cheng
Zhilin (Catherine) Zhou

December 16, 2020

Contents

1 Introduction **1**

2 Overview **1**
- 2.1 Model . . . 1
- 2.2 View . . . 2
- 2.3 Control . . . 2

3 Design **2**

4 Resilience to Change **3**

5 Answers to Questions **5**

6 Extra-credit Features **6**
- 6.1 STL smart pointers . . . 6
- 6.2 Bank . . . 6
- 6.3 Market . . . 6
- 6.4 Number of Players . . . 7
- 6.5 Computer Players . . . 7

7 Final Questions **7**

---

1 Introduction

This document first gives an overview of this project, including the describtion of the structure and the explaination of how the design patterns helps implementing the existing features. Then, it includes the anticipation of several changes/new features. It goes on to explain how the program would accomodate these modifications. Additionally, it contains answers to the questions as well as the extra-credit features we add to the program.

2 Overview

The overall structure of this program splits into three sections: Model, View, and Control. We decide to implement the MVC design pattern to organize our classes and their relationships to make the program with higher cohesion inside the classes and lower coupling between classes.

2.1 Model

- The model section of the program is centered at `Board`, which contains all the information on the game board and implements most of the actions during the game. It contains vectors of `shared_ptr` to the `Tile` objects, `Edge` objects, `Vertex` objects, and `Builder` objects. It has a composition relationship with all these classes. To minimize coupling and maximize cohesion, `Board` is the only class that holds the pointer to these objects, while other classes hold the index of these objects inside the vector.

  For example, when the player rolls a dice, `gainResources(int roll)` is called. It goes to the `Tile`(s) that matches the number of the dice and finds out if any player has build residences on that Tile by getting the vector of the indices of these `Vertex`. The `Board` then uses its private `Vertex` vector to access these `Vertex` objects to see if it has residences or not.

- `Builder` class holds all the information of a player, including the number of resources, the building points, and the vector of `Residence`(s). It has a composition relationship with the `Residence` class.

- Tile class holds all the information of a tile on the game board, such as its number, value, the resource(s) it represents, and the vertices and edges it associates with. As mentioned above, the vector of Vertex and Edge contains the indicies instead of the object/pointer itself, to enhance programming design.
- Edge and Vertex hold information of the edges and verticies of each tile.
- Residence class obtains information of each type of residence the player can build. The UpdateResidence() and ImproveType() function updates the Residence type and the building point of the current object.

2.2 View

We use an abstract Display class with the subclass TextDisplay to implement features of the View section. Display contains printBoard(), printResidences(int builder), and printBuilders(). When Board needs to print these information, it calls the corresponding function in the Display class to output the messages. We use the Template Method here, and we will discuss the details of this implementation in the Design section and Resilience to Change section later.

The textDisplay class holds the layout of the game board. We use the rectangular_layout text file as the template and add specific information to the template when the game starts. The Board gives the Information object it receives to initialize Display, and Display uses the informaiton stored to fill in the board.

We use the Observer design pattern to implement the feautres in Display. Everytime Board makes a change to the state of Builders, Vertices, or Edges, it will *notify* Display by calling the corresponding functions. Display will then change the information it stores and the corresponding coordinates on the board. The Board can also potentially choose to *attach* or *detach* to different display layout.

2.3 Control

The Control section consists of two parts: the main function and the Controller class.

- Main contains a Controller object. It takes in input from the players and calls the corresponding Controller functions to implement the actions. It uses several exception structs to implement exception safe, making sure that the program will not crash or goes into error even if the user does not follow a legal sequence of actions. Controller goes between the main function and the Board class. It contains a pointer to the current Board and calls the corresponding Board functions when the main function calls its functions.
- We made an Information class to contain all the information the board needed at the beginning of the game. The Information class has a composition relationship with the BuilderData class, which contains the information of each builder from the loaded data. It takes in a string (from the given file) and translates them into the way that Board could read later.
- We use Template Method to implement the features that the player gets to choose what type of board they want during run-time. We have an Level class and three subclasses, RandomLevel, PreviousLevel, and CustomizedLevel. PreviousLevel and CustomizedLevel both reads the file given by the player stores the information in the Information object. RandomLevel uses a default_random_engine to generate random sequences of the tiles and their values based on the setted distribution. The Level class uses the function getBoard() to return a shared_ptr to the Board object it constructs using the completed Information object.

3 Design

- At the start of the program, the user need to choose whether they want a random board or a customized board layout by giving the command-line instructions. We implement this feature by using the Template method on Level.

  Level class contains a shared_ptr to an Information object. It uses the getBoard() function to generate a shared_ptr of Board by passing info to the Board's constructor. Level has three subclasses: RandomLevel, PreviousLevel, and CustomizedLevel.

  When the user gives the command-line instructions on what kind of board to create, main will pass these instructions to Controller by giving its constructor the seed and input files. The Controller will decide which Level object to create at runtime. Since for whatever type of Board to create, they all need an Information object and pass it to the constructor. Thus, we decide to use the Template Method, such that he three Level subclasses have different implementation on `updateInfo()`, ad the `Level` class has a public `getBoard()` function. Then at run-time, `Controller` will get the corresponding board by calling `getBoard()`.

- At the beginning of the turn, the player chooses to roll a loaded dice or a fair dice. We implement this feature by using the Factory Method, that we have an abstract `Dice` class and two subclasses (`LoadedDice` and `FairDice`).

  We will determine which object to create at run-time. The `Controller` class passes the `seed` parameter to `FairDice` and the player-chosen number to `LoadedDice`. Both classes will then return the rolled number by the overriden `rollDice()` function, either using a `default_random_engine` to shuffle the dice, or directly return the chosen number. To make sure that the rolled number is random while using the `FairDice`, we shuffle the list of numbers (1 to 6) two times, and take the first element on the shuffled list as the index of the number rolled in the shuffled list. We then add the two results we have to return.

- During the game, each builder will make changes to their state by rolling a dice (gaining resources), building a road, or building a residence, etc. We want to udpate these changes immediately to the `Board` as well as the `Display`, since the player can choose to print the board or player status right after they nake these changes. Therefore, we use the Observer design pattern here to implement this feature. Everytime the builder makes a change, the `Controller` calls the corresponding function in `Board` to conduct this action. `Board` will then notify the `Display` object it attached to at the beginning of the game to make the corresponding updates. Subsequently, when the player wants to print the board, `Display` can immediately print the up-to-date information.

- During the game, each builder can choose to build a basement or a road, which definitely changes their state on the resources they hold, as well as the state of the board. We adopt the Observer design pattern here to implement this feature. We contain mappings from verticies to their adjacent vertices and edges in `Vertex` and `Edge`. Everytime the builder build a road or makes a change to their residences, `Board` will *notify* the adjacent verticies and edges of the corresponding vertex or edge to make some changes to their state. We decide to add the builder's name to these adjacent verticies or edges to show that such builder has built a basement or road. Doing so helps us keep track on the current state of each vertex and edge, so that when the builder want to build a road or another basement later, we can check their validity using the most up-to-date information.

- We noticed that this program requires a lot of interaction with the user, by taking in input from the user, conduct actions, and output the corresponding messages to the user. The entire process matches the MVC design pattern, and thus, we decide to adopt this design pattern to implement the user-interaction features of this game.

  We separate our classes into three parts: `Model`, `View`, and `Control`. The `Control` section includes classes that takes in user input, translates it into the format that is understandable by other functions, and calls the corresponding functions in the `Model` section. It acts like the deliver between the user and the program itself. The `Model` section includes classes that actually conduct the instructions and calculates the correct output. After the calculations are done, it delivers the raw result to the `View` by updating the output information stored in `View`. The `View` section then prints the message or shows the display to the user, letting them know how their instructions are fulfilled. We have the follwoing example to illustrate how we use this design pattern to enhance our program:

  One of the features of this game is Geese. When the dice is rolled to a 7, `Control` calls the `Geese()` function in `Model`. `Model` first finds a list of builders that have at least 10 resources on hand and give this list to `View` to let these players know how many of their resources have been lost to the Geese. `View` then asks the current builder to choose a new position to place the Geese. `Control` takes the chosen vertex and pass it to `Model` by calling the `UpdateGeese(new location)` function. `Model` searches for the builders that have built residences on such tile and delivers the list of such builders to `View`. If there are no builders to steal, then `View` will just output the message, and `Control` will move on to the next instruction. If the list is not empty, then `View` asks the player to choose one builder from the printed list. `Control` gets the name from the player and passes it to `Model`. `Model` generate a randon resource from the chosen builder and gives the resource to the current builder. It updates `View` with this change on state, and `View` then prints the message to notify the builders.

4 Resilience to Change

- The player might want to switch to a different display, such as a graphic display, in the middle of the game (at run-time). Considering this potential change, we use the Template Method on the `Display` class.

  We have a `Display` class, which contains the current up-to-date information about the builders, vertices, and edges, including mapping from builder to their number of resources and residences, mapping from verticies to their position in the board layout, etc. The information can be accesses by all its subclasses, so that they could construct their layout and update the information when get notified by the `Board`. `Display` class contains public functions to print error messages or output results from the `Board`. It also contains pure virtual functions for updates and printing. We choose to leave them as pure virtual functions, since the different subclasses will have different implementation on how to update the information as well as printing the layouts.

Inheriting `Display`, the `TextDisplay` class obtains a vector of *strings* (we use vector of chars here, so that it will be easy for us to access and update the layout). It implements the virtual methods from `Display`. Since the `Board` only contains a `Display` object and only decide which specific display type it is when constructing this object at the beginning, both the non-virtual methods and the virtual methods will work at run-time, and the virtual methods will have different display output when using differerent display type.

We originally plan to add another display type such as changing the board layout, but due to the time constraint, we didn't implement such feature. However, we still keep the Template Method design pattern, just so we can always implement such features with the smallest amount of change to our program – we only need to add another subclass to the `Display` class and change the implementation of the virtual methods in the new subclass.

- The players might want to change the settings of the game when restarting the game at the end. For exmaple, they originally set the game to a previous game status, and after one player wins, they decide to use another board layout or game status instead of the original one. To implement this potential change, we use the Template Method on setting the board.

  At the beginning of the game, when the player gives the command-line instructions, we store the information we get from the file in the `Information` and `BuilderData` class. We have a `Level` class and three subclasses to separate the cases when the command-line is 'random', 'board' or 'load'. `Level` class has a public `getBoard()` method. Calling which will return a `shard_ptr` to the `Board` with the stored `Information`. It contains a private virtual `updateInfo()` method, and the three subclasses all have different implementation on this method. When `Controller` calls `getBoard()`, the `Board` it returns depends on the type of `Level` object `Controller` creates based on the user input.

  `Controller` holds the `Level` object it originally creates, so when the game finishes and restarts, it can use the public `restart()` method in `Level`, which returns a new `Board` pointer with the same `Information` as the beginning. Thus, if we want to implement this change, we only need to change the `restart()` method in `Level` and the `main` function at the end, which will ask the players to input a new command-line instruction. `Controller` will decide the new type of `Level` object it needs to create and pass the corresponding information to this new object. Then, calling the new `getBoard()`, it shall return the udpated `Board` with the new settings the players just choose.

- We add a new feature to the game called `Bank`. The detail of this feature will be explianed in the extra-credit section. We did not change our program a lot when adding this new feature because we adopted the Observer design pattern in `Model`.

  When the builder want to mortage his/her residence to the bank, `Board` will notify the current `Builder`, and `Builder` will detach the corresponding `Residence` at the given vertex. The detach function will first delete this `Residence` from the vector of residences stored in the `Builder`. It will then notify the vertex on which the `Residence` locates and the adjacent verticies and edges to make changes to their states.

  Adopting the Observer design pattern divides the work of implementing each instruction into different parts, which will be done by different classes. This maximizes the cohesion within a class and minimizes the coupling of different classes.

- We add another feature to the game such that the player can choose the number of builders in the game by inputing `-players x`, where x is the number of players. We make minimal changes to our program since we implement the MVC design pattern. Since we separate our classes and functions into three sections, making change to the number of players barely changes the `Model` section. We only make minimal change to `main`, `controller`, and `Display` to add player size as a private member and change the vectors of players into the correct size.

  The reason why we almost do not need to change the `Model` section is that everytime `Controller` calls a `Board` function, it passes the index of the current player to `Board`, such that the `Board` always uses the correct index of player to conduct the following actions, and will not get an invalid player index. `Controller` keeps track of the number of players and the index of the current player to make sure that the index number it gives to `Board` is valid. The only thing we need to change in `Model` is to add player size as a private member, so that when `next()` is called, it can change `curTurn` to the correct player. We did not give much changes to `Display` either. We only add player size as a private member and change the vectors' size to match the player's size.

5 Answers to 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**~~ **Template Method** design pattern to implement this feature. We have three ways to create the board: random, using previous game status, or using a board layout. Since the `board` class takes in an `information` object, these three types differ only when preparing the `information` object. They all have the same last step – passing the `information` to the `board`. Therefore, we could use the **Template Method** here to help us implement this feature.

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**~~ **Factory 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.

   *Note: at DD1, we stated to use **Template Method** here. We decide that **Factory Method** should be more accurate to implement this feature, since using a fair dice or a loaded dice does not have a similar algorithm. If we use the fair dice, we have a `defualt_random_engine` to take a random roll, and if we use the loaded dice, we just output the number the player just chose. Therefore, we just need an abstract `Dice` class to declare in `Controller` and choose which object to create at run-time, which uses the **Factory Method**.

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**~~ **Template 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.

   *Note: Orignally, we plan to use the **Factory Method**, but later we found that `Board` class needs to use `Display` to output error messages or the result of an instruction . Thus, we add some public functions to `Display` to output messages, since with different display type, these functions will be the same. For the functions that need to update the baord layout, we put them as virtual funtions so that with different tyep of `Display` objects, they will have different implementations on those.

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.

*Note: We add a `Exception` class to our program to store all the exceptions we need for this program. The class includes `alreadySpecified`, `unableOpen`, `invalidArg`, `invalidCommand`, and `EndOfGame`.

6 Extra-credit Features

We have implemented several extra-credit features to this project. Here are the details of their features and descriptions.

6.1 STL smart pointers

In this project, we choose to use STL smart pointers instead of normal pointers to implement RAII. We designed our classes so that the lifetime of the objects in the classes are bound to the corresponding class. For example, in our `Board` classes, we have a `shared_ptr` of a `Display` object. Since the `shared_ptr` will automatically clean up its allocation in heap when the object is gone out of scope (reference count is reduced to 0), which in this case, is when the `Board` class goes out of scope. Similarly, the `Controller` class obtains a smart pointer to the `Board`. When we restart the game, we replace the old board with a new board containing brand-new information. The `shared_ptr` to the old board will then go out of scope, since `Controller` is the only class holding the reference to it. Then the replacement will automatically clean up the allocation of the old `Board` as well as the `Display` object, since the lifetime of `Display` is bound to `Board`, and create new allocation for the new `Board` and `Display`. The entire process can be done by the STL smart pointers, without us explicitly *delete* the objects. Using the smart pointers has significantly helped us to keep our codes clean and designed our classes to implement RAII.

6.2 Bank

We added a new functionality to this game: the player can mortgage one of their residences to the bank to get resources. Just like the mortgage in real life, we set a different price for each type of residences, and we print the price table to the display for the player to decide if they want to continue with the mortgage or not. The price of each residence is half of their price before. For 'House' and 'Tower', we add the number of resources one builder need to pay from 'Basement' as their price.

The description of 'bank' in 'help' is:

```
~ bank : attempts to apply for a mortgage using an existing residence.
Further instructions will be given when the command is chosen.
```

During the game, when the player type `bank`, they decide to initiate a mortgage with the bank. Then the program will print the following price table to the player:

```
Here is the rules for applying a mortgage.
          Original price                                What you will get
Basement: 1 BRICK, 1 ENERGY, 1 GLASS, 0 HEAT, 1 WIFI => 1 BRICK, 1 ENERGY, 0 GLASS, 0 HEAT, 0 WIFI
House   : 0 BRICK, 0 ENERGY, 2 GLASS, 3 HEAT, 0 WIFI => 1 BRICK, 1 ENERGY, 1 GLASS, 1 HEAT, 0 WIFI
Tower   : 3 BRICK, 2 ENERGY, 2 GLASS, 2 HEAT, 1 WIFI => 2 BRICK, 2 ENERGY, 2 GLASS, 2 HEAT, 1 WIFI
Enter the vertex if you want to continue, or else 'quit'.
```

Then, the builder will decide if they want to continue with the mortgage or not. They can choose to quit the process if they do not like the price the bank gives, or give the number of the vertex they want. `Board` will check if the input vertex is valid or not, and if it is valid, it will update the builder's status and `Display`. If the mortgage is successful, the program will print

```
You have successfully mortgaged your <residence> at <vertex>!
```

Otherwise, it will print the error messages to `cerr`:

```
You cannot apply for the mortgage at <vertex>.
```

6.3 Market

Another new feature we add to this game is market. The builder can choose to use four of their resources to exchange for one resources from the market. How it works is very similar to 'trade', but one main difference is that, the other builder in 'trade' can decline this offer, while the market will always accept the offer, except that the builder has to pay more to trade in the market.

The description of 'market' in help is

```
~ market <give> <take> : attempts to trade with the market, giving four resources of type
<give> and receiving one resource of type <take>.
```

When the builder input a resource for `give` and `take`, the `Board` will check the validity of `give` – whether or not the builder has four `give` resources or not. If the builder does not qualify for such exchange, `Display` will output:

```
You do not have enough resources to complete the exchange.
```

Otherwise, the exchange is successful, and `Display` will output the number and the type of resources the builder has just gained.

6.4 Number of Players

In this project, we add another command to the command-line instructions, such that the player gets to choose the number of player this game will have. They will add `-players x` to the command-line at any position, and x will be the number of players in this game. x has to be between 2 to 4, and the names of the players are still in the order of `Blue`, `Red`, `Orange`, `Yellow`, while for players less than 4, we will take the first x players.

For example, if the player have `-player 3` in their command line, then the game will have three players: `Blue`, `Red`, `Orange`. The game will proceed in the same way as before, except that each turn will only contain these three players without `Yellow`.

The command-line has the same rule as before, such that the player cannot input `-player` twice.

6.5 Computer Players

We add some computer players to this game to make it more fun. The player gets to add `-computer_player x` to the command-line, indicating that they want to include x computer players in this game. x has to be between 1 and $p - 1$, where p is the number of players in this game with default set to be 4. There has to be at least one human player in the game. We keep the rules for the command line, such that the player cannot give `-computer_player` twice or input an invalid number of computer players.

The computer player will only do four things in this game: choosing a valid vertex when building basement at the beginning of the game, rolling a fair dice (not 7), giving `next` command as soon as it is his turn, and chooses `no` when a human player asks to trade with it. Since the computer player has access to all the vertex and edge status, it will search for the smallest possible vertex to build basement on. Since the computer player is "smart", it will not choose an invalid address and makes an error. However, it is not so "smart" in the way that it will not do anything in its turn, such as building another basement or road. It is not so friendly to the human players, such that it does not want to trade with the human players.

7 Final Questions

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?

   From this project, we learned the importance of having a good communication between team members and how having a team helps to divide a large program into smaller pieces.

   We have a clear division of the work. Since we decide to adopt the MVC design pattern, we decide to have Meredith finish the `Model` part, and Catherine to finish the `Controller` and `Display` sections. We have meetings regularly to discuss the progress on finishing our parts, and made a Google document to record the changes we made to the original structure of the program while developing our own parts. We found that having different people writing different parts enhances the implementation of object-oriented programming pattern, since we will have high cohesion on our classes and little coupling between different classes. Using the Google document as well as a Discord group, we made our teamwork efficient and effective.

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

As you can see, our original UML diagram that we planned before we actually write our programs looks very different than our final UML diagrams. We did not think through the entire project and how each sections connect with each other, so we made many modifications to the internal structure within each sections while developing the program. For instance, Catherine did not plan the implementation of `Display` correctly at the beginning, such that the `Display` class and `Board` class each has a `shared_ptr` pointing to each other, which is obviously a bad practice and is not allowed for `shared_ptr`s. Therefore, Catherine has to rewrite her entire `Display` class after finishing most of it, and changing it to the correct implementation. Hence, if we had the chance to start over, we will plan the structure of the project more thoroughly and have a deeper understanding of the organization of the program before we start coding, so that we can be more efficient during the development of the game.