CS246 Final Project: Constructor

Yiheng Ye Linus Zhang Mingxiao Zhang

Contents

1 Introduction

Constructor is a modified version of the board game Settlers of Catan. It is played a board representing the University of Waterloo, where four players build residences and roads, obtain resources from tiles, trade or steal resources with or from others and move geese. The goal is to be the first player achieving at least ten building points.

2 Overview

The structure of the Constructor follows the Model View Controller architecture.

2.1 Models

Models manage data in the game and present it to the user through View. In this project we have the following Models:

Player Model for the state of a builder. It stores the resources, residences and roads owned by the user and is manipulated by the Controller and other related Models through public methods.

Board Model for the entire game board. It owns the tiles, vertices, edges and geese and all access to these Models is done through Board with its getter methods.

Tile Model for the data of a tile. It contains the resource type, index, and value of a tile as well as the indices of the vertices surrounding this tile. In the event of obtaining resources, Tile accesses Vertex through Broad by vertex index and manipulates Player's resource data accordingly.

Vertex Model for the data of a vertex. It contains the index of a vertex as well as the indices of the edges connected to this vertex. It can be mutated to a building in which the building type and owner are also stored. When the user gives commands of `build-res` or `improve` to Game, Game invokes Vertex. Then, if all conditions are satisfied, Vertex will self-mutate to a building and add its index to the Player's residence list or improve itself to the next type. Otherwise, View will be notified to print an error message.

Edge Model for the date of an edge. It contains the index of an edge as well as the indices of its adjacent vertices. It can be mutated to a road in which the owner is also stored. When the user gives the command of `build-road` to Game, Game will invoke Edge. If all conditions are satisfied, Edge will self-mutate to a road and add its index to a Player's road list. Otherwise, View will be notified to print an error message.

Geese Model for the data of geese. It contains the position of geese.

2.2 View

View defines command-line user interface and is observed from Models.

View prints messages, prompts, errors and game board status on output and error streams (`cout` and `cerr` by default). For printing messages and prompts, View ensures the correct format internally and other classes only need to pass a message string when invoking View. Also, for Board printing, the caller only needs to pass a Board and View generates the required format internally. This design ensures the compliance of the Single Responsibility Principle as only View is responsible for the details of the display.

2.3 Controller

Controller accepts user input and manipulates models. In this project, we have an Controller interface with an input stream (`cin` by default) and a utility method for clearing fall bits. This interface is implemented by the following concrete Controller classes:

Game Primary Controller of during the game. The play method starts a game and calls other private methods for different stages of the game. These private methods accept user commands such as build-res and board and inputs, and manipulate Models accordingly.

LoadedDice Auxiliary Controller of the game. It accepts the user's input and returns it to Game.

3 Designs

In the design of this project, we applied the following techniques:

3.1 Factory Pattern

According to the project specification, the Board's layout should be initialized randomly or from file. To realize this function, we created a BoardLayoutFactory abstract class with a virtual `createLayout` method and let two concrete factory inherit the abstract class and implement their own internal logics for the `createLayout` method. At startup, according to the command-line options, a specific concrete factory instance is passed to Game as a BoardLayoutFactory, and Game calls the virtual `createLayout` method to create layout without knowing what specific concrete class is used or its internal mechanisms. If the layout should be created in a new way, we only need to implement a new concrete factory and pass it to Game at some point, and the rest of the program remains unchanged.

3.2 Template Method Pattern

According to the project specification, a user must be able to switch between loaded and fair dice at runtime. These two types of dices have some common methods like yielding a number but have some different implementations as well. To realize this function, we implemented an Dice interface for their common methods and let the two dices be derived from the Dice interface. Other parts of the program using them would treat them both as the dice interface (instantiate both types of dice and store these instances both as the interface), and when the user switches the dice at the runtime, we simply switch which object to use and rest of the program remains the same.

3.3 Polymorphism

For Dice and BoardLayoutFactory, we created a abstract class or an interface with pure virtual methods first, and then implemented the concrete classes with `override` keyword on methods inherited from the abstract class. This allows us to use a single interface when using these concrete classes and let the dynamic dispatch mechanism determine the specific type of the object at runtime.

3.4 Single Responsibility Principle

We designed our codes so that every class only has a specific part of responsibilities in our project, and modification of a single function only affects only one class. For example, Game is only be responsible for accepting user commands and invoking Models according to the commands. It does not know how these Models work to fulfill user commands or how results is presented to the user, changes to Models' behavior or presentation do not affect Game.

3.5 RAII Idiom

We used `unique_ptr`, `vector` and `map` to store our Models so that the memories of Models are automatically freed at the end of our program without the use of `delete`. This practice also ensures that even if an exception causes the program to terminate, all the memories will still be freed.

3.6 Reduced Compilation Dependency

To avoid cyclic compilation dependency, we forward declared the View class in the Board, Edge, Vertex, Tile and Player classes for the View pointers in these classes, and similarly forward declared the Board class Edge, Vertex, Tile and Player classes for the Board pointers in these classes.

3.7 Low Coupling and High Cohesion

Aiming to achieve low coupling and high cohesion, we designed our codes so all classes interacts with each other through method invoking instead of direct field manipulation. Therefore, there is no public field in our codes and the use of the `friend` keyword minimized to a single methods for Board layout creation. Abiding to the Single Responsibility Principle also helps achieving low coupling and high cohesion, as every class only has a specific part of responsibilities in our project and only very few public methods are required for every classes.

3.8 MVC Architecture

Controllers are responsible for accepting and interpret user commands and manipulate the Models accordingly. Models, including the Player, Board, Edge, Vertex, Tile, and Geese, manage the data and invoke View for presentation. And View accepts invokes from Models, manages the mechanisms of presentation and present Models' data to the user. Models' data to the user.

4 Resilience to Change

4.1 Input and Output

Since this project uses MVC architecture, it is flexible to changes in the types and methods of user input commands, as well as the presentation of game to the user.

If the specification requires additional user commands, they can be added to the Game class while the Model portion remains the same. If a new method of input is required other than the terminal, then we can create a new children class of Controller that interprets the new type of input and manages the Model classes accordingly.

The project design also supports changes to the content of the text output and the method for presenting the output. All text outputs are specified in the View class, so changing the content would only require changes in this file. If new ways of output are required, such as a GUI, then we change the view class to a parent class and move all of its current code to a child class for text output that inherits from it. This way, we can create a new child class of View for graphical output and add all the required features in this class only. This way, the only change required in the main program is the initialization of the View class, where we can specify which mode of output is required.

4.2 Board

Possible changes to the specifications of the board may include creating boards of different sizes or adding ways of inputting the board layout.

Changes to the board size would result in different neighbouring structures for the tiles, vertices, and edges. This project design uses a separate file to store the adjacency tables of the elementary components of the board, which is generated by an external program. Static changes to the board's size can be made by directly changing the adjacency tables. If the specification requires different board sizes at run time, then we can add the program that is used to generate the adjacency tables into the project.

Since this project uses the Factory design pattern in creating the board layout, to add new ways of loading the board layout, we can create new children classes to the BoardLayoutFactory class. Since the use of Factory allows the program to decide which factory to use at run-time, we can simply add a new case in the main file that specifies the new factory before the game starts.

4.3 Resources and Buildings

This project uses enumeration classes to store the resource types and building types. The only changes required for changing the enumerations are within their classes and in minor parts of the factory classes and the player class. The specific type of buildings and resources are mostly hidden in the game program. Both enumeration classes also encapsulate the methods of conversion between their types and string, so the program for the output can also remain the same when changing the enumeration.

4.4 Players

This project uses a Player class to store the data of each player builder and uses an enumeration class Color to store the color for each builder. The number of players in the game is stored as a constant. If the number of players change, then we can change the enumeration of colors and the constant for number of players. No changes are required in the game's main program.

5 Answers to Project 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?

The ability to choose between randomly setting up the resources of the board and reading the resources used from a file at runtime indicates that the program must have two methods to create our game board layout.

For this requirement, we used the factory design pattern in which we implemented two factory classes to create the game board layout. Both factory classes accepts a blank Board instance, but one fills the board with data from the file and another fills the board randomly.

We used such a design pattern because with the factory pattern, we can isolate the internal details of layout creating, the selection of creating methods, and actual invoke of the creating methods in three separate places to comply the Single Responsibility Principle and minimize the side effect of code change.

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 and used the template method design pattern.

The two types of dices have some common methods like yielding a number but have some different implementations as well. For example, the loaded dice would have to prompt the user for some input, but the fair dice would have a random number generator inside. Therefore, we could implement an interface for their common methods and let them be derived from the dice interface.

Other parts of the program using them would treat them both as the dice interface (instantiate both types of dice and store these instances both as the interface), and when the user switches the dice at the runtime, we simply switch which object to use and rest of the program remains the same.

We used such a design pattern because with the template method design pattern, the internal mechanism of a dice, selecting or switching the dice and rolling the dice can be isolated three separate places to comply the Single Responsibility Principle and minimize the side effect of code change.

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., hexagonal tiles, a graphical display, different sized boards for different numbers of players). What design pattern would you consider using for all of these ideas?

We would consider using the bridge pattern in which we separate abstractions and implementations, and different implementations would allow different game modes. Taking adding graphical display as an example, previously, methods for notification would be a mix of notification details (e.g., the text of the notification) and displaying details for a specific user display (e.g., CLI, WinForm, WPF, Qt, GTK). This would be a mess if we continue to add new displays as all stuff is mixed.

To solve such a problem, we could separate notification details and displaying details into two types of classes: notification class for abstraction, and implementor classes for specific details of different displays. The notification class would have a display implementor derived from an interface. To add a new graphical display (e.g., we have CLI and GTK for Linux GUI, but we want to have a macOS GUI), we now only have to create new display implementor classes (which should be derived from the implementor common interface) and attach it to the notification classes at runtime.

4. 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 could use the decorator design pattern. Basically, we would implement an abstract tile interface with getter methods for the production of each resource (resource getters) and then implement the concrete basic tile class and its decorator classes. The basic tile class would have resource getters that return quantities specified in

the project document and the decorators' resource getters will add or subtract resources returned by the last element in the chain of effects.

5. 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.

Generally, we will not throw exceptions from the models in our program.

If the user input and program output are handled in one interaction class, exceptions should be thrown in other classes to notify the interaction class when the user's operation through the interaction class is illegal. Otherwise, other classes other than the interaction class will also have to handle error output directly, which is a clear breach of the Single Responsibility Principle and will cause problems when adding new user interaction methods (e.g., adding GUI to a command-line program).

But in our program, we used the MVC design model, in which the user-input handling (Controller) and program output (View) are separated. If a user feeds an illegal operation to the models through the controller, the Models should not throw an exception as the controller does not provide output functions. Instead, the Model will directly notify the View through function calls.

But we tried to catch exceptions from `std` objects and functions in our program.

An exceptions may occur from `fstream` during the reading and writing of the game board storage file since the filename given by the user may not be correct, and an exception may occur from `istream` if we are expecting an integer but the user gives a string. We need to catch these exceptions to prevent the program from crashing and to provide useful prompts to the user.

Furthermore, the project specification requires the program to quit when receiving `EOF` while waiting a command during a turn. For this function, we tried to catch exception from `istream` to check if the user gives an `EOF`.

6 Changes to the UML Diagram

In the process of coding, we changed our UML diagram on due date 1. The following are the main differences between our diagram on due date 1 and on due date 2:

1. **Added a static Random class for random number generation**
   A static Random class allows the random engine be initialized and managed at a single location. With this static class, classes that need to generate random numbers do not need to instantiate their own random engines hence improve performance. Also, with this static class, we can set the seed only once and the seed will have effect on the entire program.

2. **Added an abstract Controller class and let Game and LoadedDice to derive from it**
   Our Controller class is an abstract class with an input stream (`cin` by default) and a utility method for clearing fall bits. By deriving Game and LoadedDice from it, we defined the basic structure of a Controller and made it possible to use another source as user input without the need of change our code (e.g., we can let the input stream to be another stream different than `cin` by constructing the Controller part with a different parameter, and other codes remain the same).

3. **Changed how Board, Geese, Tile, Vertex and Edge are constructed and set**
   We originally decided to set the state of Geese, Tile, Edge and Vertex instances in their constructors while constructing Board. However, according to the project specification, players' roads and residences (crucial information for setting the Edge and Vertex instances) are prior to Board layout in the game file. If we wanted to keep our original design, we would have to store these data first, and then use them together with Board layout to construct the Board. We thought this could be too costly and decided to use a new design.

   In our new design, Game will be provided a BoardLayoutFactory for creating Board layout randomly or from layout file, or will be provided a filename for reading previously saved game. Before generating or reading Board layout, a blank board with blank elements (Tile, Vertex and Edge will only have adjacency relationship stored and other fields remain unset) is constructed first by Game. Then, if reading a game from file, the file reading method of Game firstly will set Vertex and Edge with players' roads and residences information, then will construct and

invoke a FileMode factory to create the Board layout from the rest of the file, and finally will set Geese from file. Otherwise, the BoardLayoutFactory previous provided to Game will directly be invoked to create the Board layout, leaving Board elements blank as they should be.

4. **Added index private field to Tile, Vertex and Edge**
   The index is required to determine if geese is on a certain tile when obtaining resource from this tile. And the index is required for Vertex and Edge to checking some adjacency relationships in deciding if a road or residence can be built. We did not anticipate these in drawing our first diagram.

5. **Added some getter methods to Tile, Vertex and Edge**
   The methods `getResidenceOwners` and `getValue` are required for Tile in obtaining and losing resources. The methods `getAdjacentEdgeIdx` and `getAdjacentVertexIdx` are required for Vertex and Edge to checking some adjacency relationships in deciding if a road or residence can be built. We did not anticipate these in drawing our first diagram.

6. **Added some methods and private fields to Player**
   The vector `roads` and the method `addRoad` is required for storing a player's road. The method `getTotalResources` is required for losing resources to geese. And the `toString` method is required for serializing the player in game saving. We did not anticipate these in drawing our first diagram.

7. **Added streams for output and error in View**
   These streams were not included as we decided to hard-code `cout` and `cerr` for output and error, but we decided to add the ability to change where to output when coding. This makes View more resilient to changes.

8. **Modified how Geese is stored and accessed in Board**
   We originally designed that Geese should be stored in stack and accessed through Board. However, this is a breach of the Single Responsibility Principle as Board has a responsibility other than storing and providing access. As a result, we stored Geese in a `unique_ptr` and added a getter method for its address, allowing other classes to interact with it directly.

9. **Made some methods of Game private**
   We believe methods like `init` or `read` should be called by the constructor internally instead of by some outside callers to achieve low coupling and high cohesion.

7 Extra Credit Features

The entire project is designed following the RAII idiom and uses only `unique_ptr`, `vector` and `map` to store variables in Model classes. No `new` and `delete` are used in all parts of the program, and the program is tested to not contain any memory leaks.

8 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?**
   We learned from this project that development in teams require more careful communication in the planning stage. Planning includes deciding the functionalities of each class and the relations between classes. Planning has a greater effect in group projects since to ensure efficiency, multiple people may be working simultaneously, and making changes in the functions of one class may impact the progress of the entire project or cause unexpected issues. It is especially important to make sure everyone's ideas on every part of the program is well aligned before the development stage. Differences in ideas for the realization of each part of the program may result in incompatible code and increase the difficulty for debugging.

2. **What would you have done differently if you had the chance to start over?**
   We would extend the time we used for communication and planning. More time is required for designing the UML. We could design system and logical flow charts before designing the UML to make sure we are clear on the

process sequence of the game and the input and output methods required. This way, we may have a better concept of the game in mind when designing the classes in the UML.

It would be helpful if we made sure we understood the rules well before developing the program, since this time we had to make many changes after the development stage due to misunderstandings of various specifications if the project pdf. We would also have more direct communications throughout the entire project so that we make sure we use objects from other classes correctly.

9 Conclusion

This project created the Game of Constructor using OOP. Through using OOP design patterns and principles, the project code achieved low coupling and high cohesion, allowing for easier code maintenance and easier potential additions.