CS 246 Final Project – Constructor

Final Design Document

Trevor Chen (h586chen)

Eric Liu (e44liu)

Introduction: About Constructor

The game, Constructor, is a variant of the game Settlers of Catan. The game involves four players. Players will try to build constructions to gain building points and the first player who gets 10 building points wins the game. The game can be automatically or manually saved if the game is not ended if the program was called to end itself. Several command-line options can allow users to choose which game and types of the game board before starting the game.

Overview: The Overall Structure

The game has 7 main classes: Board, Builder, TextDisplay, Dice, Tile, Edge, Vertex.

In order to avoid memory leaks, we used smart pointers. The Board class takes ownership of all the unique pointers and will destroy them automatically when the board class is deleted. Unique pointers can avoid circular reference and achieve secured memory management. In other classes except for Dice class, we use the raw point as they don't take the ownership and are not responsible for deleting them. For the Dice class, as both Board instance and Dice instance share the ownership, the shared pointer is used. Thus, RAII is achieved using smart pointers.

Strong guarantee is ensured to make the game fair. All methods in classes except Board are no-throw guarantee and error handling is done by the main function, Board (the controller) and Builder (the Model). The game program is a no-throw guarantee.

Class: Board

Related Design Pattern: ***MVC***

We designed the Board as the controller of the program. It manages most of the interaction to modify data in other classes and call other classes methods to realize proper functionality. At the initialization of the game, the main function will process the command line argument and set up the Board according to different command line options.

During the game, the main function will call Board's method *readBegin(String)* and *readDuring(String)* repeatedly, which represents two stages of the game. These two functions act as read input functions and process the inputs in order to control the game process. The Board class takes the ownership of all class instances, and smart pointers allow the board to automatically delete other instances when Board is being deleted.

This class is the first class to be created and last class to be destroyed. We treat the Board as the controller to apply the MVC design pattern. Also, as the controller, it interacts the most with other classes.

Important fields:

turn: Current turn in the game. We also use it to decide whose turn is by finding the remainder of the turn after dividing by 4 -- then we can find the position of the builder in the current turn from the vector of four builders.

Important methods:

Init(…): We use this method to initialize the board after reading the command line, and doing four different choices -- “-seed”, “-load”, “-board”, and “-random-board”.

readBegin(…): We use this method to accept commands from input in the beginning state of the game. This method can handle the options of set and roll the dice, print the status of all the builders, and help information of the valid command.

readDuring(…); We use this method to accept commands from input in the during state of the game. This method can handle the options after rolling the dice and accept all the commands until the input of “next” which represents the next turn of the game.

Class: Builder

Related Design Pattern: MVC

The Builder class records each player’s game data, responses to commands passed from the Board via return value, and execute the commands such as build road or residence, trade with the player, and lost resources to the geese. All methods of this class are no-throw guarantee.

The Builder class acts as the model in MVC design pattern, the Board is ***composed*** of Builder. In the game, four Builder instances share the board and can change contents on the board via pointers. The methods of this class consume pointers to other class instances to make changes to other classes. As Builder does not take ownership of any other classes, methods consume raw pointers.

Some of the game features, such as trade, include interaction with user input. However, letting the Board to process all inputs and call various methods in Builder may lead to too many methods in Builder and hard to manage functionality changes. Thus, some Builder methods read input and print outputs to integrate specific features (input trade resources, etc.) into one function. This can result in a cleaner coding in the controller (Board) and easy to manipulate function changes.

Important Methods:

buildRes(…) / buildRoad(…) : These two functions did the construction on the board when initializing the game, loading the game and during the game play. The Board can simply call this function without worrying about violating the game rule, causing undefined behavior, or generating errors. It can handle all game stage conditions to make proper change to the board while keeping the game rule section 2.7, 3.1, 3.2, and 3.3 on the project handout, such as not being allowed to build at adjacent vertices during the game play.

trade(…): This function realized the trade feature in the game, it follows the sample executable and game rule section 3.5.4 on the project handout.

geeseAttack(…): This function realized the feature when a player rolled a dice with 7. Each player with resources sum equal or more than 10 will lose half of their resources automatically. The resources lost are random and the possibility follows uniform distribution.

Market(…): This function realized the additional feature, market. This function is originally present in the game Settlers of Catan, where players can choose to lose four resources of the same type in exchange with one resource of any type with the market.

printStatus(…): This function prints the builder’s status according to section 2.7 on the project handout.

Class: TextDisplay

Related Design Pattern: MVC

The TextDisplay class is responsible for displaying the whole game board. It acts as the Viewer in MVC Design Pattern that manages the interface to present data. The Textdisplay stored in the controller Board is notified and updates itself when some information on the board is changed. This Class can be modified into a template method design pattern in order to allow different display options. (text, graphic, etc.) As we only implement the text display option, it is not necessary to expand the class into more classes that obtain the template design pattern and provide more display options.

Class: Dice

Related Design Pattern: Strategy

The Dice class stores the strategy type and dice points. We want to implement the feature to switch between two different modes during the game, the loaded dice mode and the fair dice mode. Hence, we apply a strategy design pattern by creating an abstract class named Strategy and two different concrete subclasses named Loaded and fair. These two different concrete classes represent the two different strategies. Allowing us to perform dice roll differently.

Important Methods:

setStrategy(…): This method allows us to switch between two different dice modes during the game.

roll(…): This method can produce an integer which represents the number we roll the dice.

Class: Tile

The Tile class stores information about tiles on the game board (such as the resources type), dice points value, vertice on the tile and whether the geese is present. This class can also add resources to the builders according to the game rule section 3.5.2 on the project handout.

Important Methods:

addResource(...): This function performs the game rule section 3.5.2 on the project handout, when the dice roll matches the tiles value, specific types of resources are distributed to the builder who built the housing with an amount according to the building level. This function returns a vector of Int, representing the number of resources gained by each builder. This helps the controller(Board) to print out the information, more details shown on the demo.pdf section [Roll Dice and Give Resources to Builder] on page5.

stoleFrom(…): This function tells the controller (Board) when the builder places the geese, which builder the current builder can steal resources from. This function returns a vector of Builder* to help the controller print information to the interface.

Class: Vertex

*Related Design Pattern: **Observer***

The Vertex class stores the information of whether it is built, allowed to build or improve, residence status (stored as H_Info class), and the vertex owner (Builder*). This class checks whether the builder can construct on the vertex using the observer design pattern. Each vertex observes other edges and vertices, and each vertex is being observed by edges and vertices, to make sure all construction actions are valid and obeys the game rules of section 3.1~3.3 on the project handout.

Important Methods:

checkIfAllow(…): This function checks if the current builder can build at the specific vertex. It returns true or false to the Model (Builder) and makes no change to the data.

Build(…): This function builds on the Vertex without checking if the vertex can be built. When initializing the board from a loaded game, the controller (Board) calls this function to force the building process as the loaded game is reading from a valid game board (Invalid board is rejected by the main function and program will terminate when reading from a invalid board). During the game, the Model (Builder) called this function, the model must have checked whether the current builder can build at this vertex except at the beginning of the game when each builder takes turns to build their initial constructions.
If the vertex is successfully built, the vertex will notify its observers.

Class: Edge

*Related Design Pattern: **Observer***

The Edge class stores the information of whether it is built, and the road's owner (Builder*). Only the Model (Builder) will call build methods to build. Each edge observes other edges and vertices, and each edge is being observed by edges and vertices, to make sure all construction actions are valid and obeys the game rules of section 3.1~3.3 on the project handout.

Important Methods:

checkIfAllow(…): This function returns a boolean to the caller (Builder), indicating whether the current builder can build a road at the edge.

build(…): This function builds the road without checking if the edge can be built. The Model (Builder) will call this function after checking if it is allowed to build here. If the edge is successfully built, it will notify its observers.

Design

1. MVC

To keep cohesion high, we used the MVC design pattern. The Board class acts as the controller, it receives input messages from the main function, translates input into valid commands, and calls the Model to perform game functions. Builder classes act as the Model in MVC, as its methods are called by the controller to modify the game data. TextDisplay class acts as the view, even though we did not design an abstract observer class, TextDisplay still acts as an observer because it is being notified frequently during the game to modify its own data. As the View, TextDisplay only displays the board to the interface, and does not participate in modifying the game data in other classes.

2. Strategy Design Pattern:

To implement the feature to switch between two different modes of dice during the game, the loaded dice mode and the fair dice mode, the strategy design pattern fits this very well. By creating an abstract class named Strategy and two different concrete subclasses named Loaded and Fair, which represent the two different strategies, we can perform dice roll differently and switch between them easily during the game.

3. Observer Design Pattern:

To make sure when a Builder constructs a building or a road only on valid positions and obeys the game rule. Observer Design Pattern is used on Vertices and Edges to achieve these requirements. Vertices and Edges observe each other and notify all its observers when successfully built a construction. The attaching process is completed before the start of the game, Vertices and Edges that are adjacent to each other are attached to each other and observe each other to obtain the game rule stated in section 3.1~3.3 from the project handout.

4. Probability Generator

In the fair strategy of dice, we need to generate a random number between 2 and 12. In that situation, I first used the system_clock to generate a seed, and then use the default_random_engine to produce a random generator according to the seed we have generated before. At last, we can use the uniform_int_distribution to produce a number to generate an integer between 2 and 12.

In board.cc, I also use a similar method to produce a random generator. In the seed function, I first construct two vectors which separately represent the source-type and value of each tile. Then, I use the default_random_engine to produce a random generator according to the seed the player entered. At last, I use the shuffle to disarrange all the elements in the two vectors. However, above all, I have set the premise that the park always has a value of 7.

In the randomBoard function, I first use the system_clock to generate the seed, and input the seed I generated to the seed function.

Resilience to Change

1. **Extra Dice Option:**
   Since in my Dice class, I used the strategy design pattern which can be added with more strategies in the future. For example, we can add larger dice from 1 to 25 through adding a new strategy class in dicestrategy.h. By using the strategy design pattern, I do not need to change the original codes to suit the new dice option. As all methods in the new strategy class can be inherited from the Strategy class directly.

2. **Extra Display Method:**
   To have a new display method such as graphic display, we only need to create a new class that is a derived class of the displayObserver. As the print method and notify methods of the new class are inherited from the displayObserver, not much modification is needed to the original code. Only to store the new graphic display class in the Boad class is enough, as the calling the displayObserver methods can call both the textdisplay class and graphic display class as the same time. Both textdisplay and graphic will act as the View.

Answers to Questions

Question 1

Strategy design pattern is useful to implement this ability, but we didn't use it. Strategy design pattern allows us to change strategy, in this case, generate a random board or read board from an input file. If using strategy design pattern, the Board class needs to store a Strategy* field and use methods to change the board generating options. A significant benefit for strategy design pattern is that the user can change the strategy from one to another during the game process. However, strategy design pattern is not necessarily in this game because the user sets up the game board option from command line input before running the program. More importantly, the user will never change the board option during the game. In another word, we won't change the board strategy anymore after the board is set up once. Thus, the benefit of using the strategy design pattern is not significant. Instead of using strategy design pattern, the main function processes the command line option settings from argNum and argVal, which is enough to set up the according board options. A benefit for main function to handle the option is that when user entered several option, we can process the most reasonable option instead of the last option on the command line.

Question 2

I could use the strategy design pattern to implement this feature. We used this design pattern, since we need to implement the feature to switch between two different modes during the game, the loaded dice and the fair dice. Hence, we created an abstract class named Strategy, and since there are two different strategies, we created two different concrete subclasses named Loaded and fair. These two different concrete classes represent the two different strategies. After adding methods in these classes, we created a class named Dice, which is composed of Strategy. Now, the dice can call public methods to choose which method to use. Since we want to create a set of different behaviors, the loaded dice and the fair dice, in Dice and make them interchangeable. The strategy design pattern is very useful to implement this feature.

Question 3

I would use the Template Method design pattern for all of these ideas. We have different game modes to implement but the game rule is unchanged. It is very useful to apply the Template Method design pattern to let one game fit for different game modes.

For example, If we want rectangular tiles, we can add another recTile class to inherit an abstract class Tile. We can let the derived class, the recTile and the hexTile, override the methods which we define in the abstract class Tile and we do not need to change the overall algorithm structure. If we want to add a graphical display, we can add another GraphicalDisplay class together with the TextDisplay class to inherit an abstract class Display. We can let the derived class, the graphicalDisplay and the TextDisplay , override the methods which we define in the abstract class Display, Hence, we can both implement the feature of displaying the board in the graphical form and in the text form. If we want a different sized board for a different number of players, we can add another LargeBoard class together with the SmallBoard class to inherit an abstract class Board. We can let the derived class, the LargeBoard and the SmallBoard , override the methods which we define in the abstract class Board to implement different features, Hence, we can change the original board size into a different board size for different numbers of players.

In conclusion, once we want to add some additional method, we can locate the class we want to add the feature at and add an abstract class and some derived concrete subclasses to implement the features we want to add. This fist for the Template Method design pattern, and I would use this design pattern for these ideas.

Question 6

The strategy design pattern can be used to facilitate this ability. As we only want to change some of the function of the class Tile while keeping other methods unchanged, we only need to switch strategies to allow different resources generating features instead of creating different tile classes and replace the old ones on the board. Each feature is a concrete class of the abstract class Strategy. By using the strategy design pattern, we don't need to redefine the already existing methods and we can apply different features to different instances of the same class type(Tile). Thus, by using the strategy design pattern, we can easily implement the additional features to change the properties of Tile instances.

Question 7

We used exceptions in Board class using the standard library <stdexcept>.

Board acts as the controller in the game. In class Board, all functions about accepting input, like init(), readBegin(), and readDuring(), will throw exceptions. In the answer of DD1, we just write that the function readInput() may throw the exception. At that time, we didn't realize that we have different inputs to read at different times. Hence, we thought that a method readInput() is enough. We find there are different states of readInput when doing the projects. Thus, we change readInput to init(), readBegin(), and readDuring(). From these methods, there are three kinds of exceptions. One kind of exception is generated from reading in a invalid input. The second kind of exception is the end of the file reached. In the answer of DD1, we thought that the violation to the game rules should be thrown. But we found that we can just solve the case of violation by a while loop and printing the error message to the standard error. In the answer of DD1, we didn't consider throwing the end of the file. When doing the project, we found that when the file is ended we need to save the game to the default file "backup.sv". Hence, we decided to throw the "End of file reached." to let the main function save the game. The third kind of exception is the end of the game or the again of the game. We decide to throw the exception when a player wins and make a decision to play again or not. We haven't considered this situation in DD1 because we thought we can handle this solution in a while loop. The load() and board() method will also throw exceptions to the main function when the file we read is in an incorrect format.

The second class is the class Builder. All build related methods and the trade method will not throw exceptions. We thought that the Builder may throw exceptions in DD1, but we found that we can handle all the error cases by simply printing the error message to the standard error.

All other classes' methods are no-throw guarantee.

The main function will catch all the exceptions and print out messages to the screen to inform the player and will not exit the game. It is worth mentioning that all the functions with exceptions are at the level of strong guarantee.

Extra Credit Features

1. **Market option:**

   We added a market option for builders during the game so that they can exchange 4 resources of the same type to one resource of any type. This solves the question when all builders(players) in the game can not get some specific type of resources to build additional residences and causes the game to have no winnings. This option is from the original game feature of Settlers of Catan. By having this additional feature, builders can have more options when all builders in the play are lacking with some specific resources.

2. **Command line/ Loaded game error input checking:**

   Our program can handle any combinations of the command line. If the command line is in an unprocessable format, an error code will be produced. If the command line is in a valid combination, the corresponding board will be produced. If we only use the -seed option without the -random-board, the board will be created by reading from a default layout.txt
   For the -load mode, our program can detect all incorrect formats like missing a character, missing an integer, the integer is in an invalid range or format.

3. **Trade**:

   When a player wants to trade, he can just enter trade and an integer from 0 to 3, or the first letter, or the colour name for any case combination of the input letter. It is a user-friendly interface.

Final Questions

1. What lessons did this project teach you about developing software in teams?

This project teaches me that every team member is very important to develop software in teams. In addition, complying with the due dates the teams settled is very important to finish the project in time.
Moreover, different parts written by all the team members are heavily connected. If one part has an error, it may cause many other parts to have a bug. In this condition, it is extremely difficult to discover the bug and costs a great amount of time. Hence, debugging and testing is significant in developing software in teams. Another important lesson I have learnt is that if I just implement a little more features in my code. My teammates may write much less code than before.

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

The first thing we would have done differently is to try to list all possible output messages when meeting different input cases, including invalid input. We also need to have a clear mind of which method can handle which error. In this project, we spend too much time adjusting the program and adding methods to handle different input and output conditions as we didn't plan out an outline of all possible cases. If we can write some pseudo-code and list control flow conditions before actually coding the functions, we can write out our code quicker and have less change adjusting huge code blocks .

The second thing I would have done differently is to create a class specifically for printing out messages other than using standard library <stdexcept> to pass messages. In the program, there are many cases that print out the same message when processing different option features. By using a Message class, we don't need to rewrite the same information displaying codes again and again. Also, by having a class dedicated to printing messages, we can manage and change the display information easily and conveniently.

The third thing we would have done differently is to communicate more often before writing out the actual code. For example, the observer design pattern of attaching edges and vertices can consume a huge time thinking of an algorithm to link them together as hard code can result in thousands of lines. If we can communicate with each other and set a different linking and notifying algorithm, the linking and attaching algorithm won't be as hard as the one we currently use. In this project, after the coding of tiles, vertices, and edges, the attaching parts becomes a huge problem. We came out with a better algorithm later when all the code has already been written, however, we don't have enough time to change our code with a better algorithm.