CS246 Final Project: Constructor
Final Design

Yuanhao (Jim) Liu
Shuchen (Mason) Liu
Yuetong (Ivy) Jiang

Introduction

The project Constructor is a variant of the game Settlers of Catan, where four housing developer players earn building points towards winning by building residences. There are three different types of residences, each required different amount of resources. The location and level of the residence determine how many materials the players could receive. During the game, the geese may move throughout the board when a specific dice value was rolled. Players at the geese location or with too many resources will lose a certain amount to the geese or the other players. The first of the four players who obtains 10 (or more) building points wins the game.

Our Progress

We finished the core components of the program before Aug 30th, and full implementation without compilation error on Aug 1st. We hold an extra meeting at Aug 2nd for testing. After that, we allocated more functional and regression tests for each one for different sections they worked on. We adjust our meeting on Aug 4th to Aug 8th because of the time conflict while we kept debugging and adding new features. We finished the UML, report, and demo part on Aug 11th.

Overview:

This section will briefly discuss the relationship and interactions between all the classes, also a brief explanation of the core methods. For details, please check the uml-final.pdf.

We classified the structure in four groups of classes:

- ◊ **Board Components**:
  Subject, Observer, Tile, Vertex, Edge
- ◊ **Dice Generator**:
  Dice, Strategy, DiceRand, DiceLoad
- ◊ **Aggregation of Core Components**:
  Player, Board
- ◊ **Game Controller** :
  CtorGame

Group member Yuanhao Liu is responsible for all the Board Components classes and all methods related to the construction and information printing methods. Group member Shuchen Liu is responsible for the Dice Generator classes and all random-generator-related methods. Group member Yuetong Jiang is responsible for the trade feature, loading-saving related features, and main and Game Controller classes. We split the projects into relatively independent groups for each member but more integrated within each group, which allowed us to work simultaneously with the provision of only header files.

The classes in each group can communicate with each other by the fields and methods. This enabled the program be designed with high cohesion, while the groups are relatively independent with each other, then make the program contain lower degree of coupling.

Board Components

Subject:	An abstract class which notifies the Observer class. Used for Observer Design Pattern
Observer:	An abstract class which will be notified by Subject. Used for Observer Design Pattern
Vertex:	Where one can construct buildings. Inherited from the Observer class. Worked as the observer of each tile. Worked as observer of neighbour Edges.
Tile:	A component of the board, containing specific resources, values, and 6 vertices. The class is inherited from the Subject class, worked as a subject which was attached with a few observers.
Edge:	Where one can construct roads. Worked as observer of neighbour Vertices.

Methods with interaction:

- Vertex::wasNotified(Tile& whoNotified) override
  Called to increase building owner's corresponding resource point.
- attachEdgeDoubly(Edge* ptre)
  Attaches Edge pointer to a vertex's neighbour; attaches Vertex pointer to an Edge's neighbour.
- Tile::notifyObservers() override
  Used to notify all observers of an subject (Tile).

- Vertex::notifyObservers() override
  Called to notify neighbour Edges when this Vertex is built
- Edge::notifyObservers() override
  Called to notify neighbour Vertex when this Edge is built
- Vertex::wasNotified(Edge& whoNotified) override
  Called to modify the Boolean field canBuild. If no adjacent vertex exist, set Boolean to true.
- Edge::wasNotified(Vertex& whoNotified) override
  Called to modify the Boolean field canBuild. If have adjacent buildings, set Boolean to true.

Dice Generator

Dice:	Concrete class to roll a dice and record strategy. Used for Strategy Design Pattern
Strategy:	Abstract class to generate a dice based on the strategy. Used for Strategy Design Pattern
DiceRand:	Use strategy "Rand" generate a random dice, inherited from Dice class. Used for Strategy Design Pattern.
DiceLoad:	Use strategy "Load" to generate a load dice, inherited from Dice class. Used in Strategy Design Pattern.

Methods with interaction:

- Dice::roll()
  a virtual method to roll a dice and get dice value.

- DiceRand:: rollDice() override

  Override from Strategy class. Generate the random a number from 2-12, which will be set to the current dice value.

- DiceLoad::rollDice() override

  Override from Strategy class. Using the integer (between 2-12) read from the input as the dice value.

Aggregation of Core Components

Player: gathering all player's information including dice, resources, buildings, roads. Mainly interact with class Dice, and class board.

Important Methods:

- beStolen:

  randomly lose one resource (be stolen by the other player who rolled geese)

- loseHalfResource

  current player lose half of the resource. Called when geese rolled and conditions met.

- rollDice

  return dice value based on current strategy of current player.

Board: gameboard which stores the pointers of all tiles, vertices, edges, and players.

Important Method:

- initAttachBoard

  Attaches all the vertices and edges for the board construction.

- initRandBoard, initLoadBoard, initSeedBoard

  Assign value and resource type to all tiles, based on the game setting requirements from the command line arguments

- trade

  When current build wants to trade resources with another user

- moveGeese

  When the current dice is 7. Lead to a sequence of actions: lose resources (if applicable), move geese, steel resources (if applicable)

Game Controller

CtorGame: works as the controller of the whole game. Applied Bridge Design Pattern

Important Methods:

- setUp

  Used when initial a new board, each player chooses two positions for their first two basements

- play

  Play the game thoroughly, the main controller of the game

- endGame

  End by save the game in the target files

Resilience to Change

This section will discuss the capability to support possibility of various changes and new features, based on each class.

Tile:

We can add more resources to a single tile. For example, before we only have one resource type for one tile. Now we can enable some tiles to have more than one type of resource (i.e. more affluent tiles). This can be easily achieved by modifying the field for resources type from a string to an object of class Resources. Class Resources is designed with Decorator Design Pattern, then each time we want to add a resource, we can add a new concrete class.

Vertex:

We can add field neighbourTiles which is a vector of pointers pointing to the nearby Tiles. When the abundance of the building near the Tile decides the resource abundance of the Tile, then when the new building is built, we can check change of the nearby building abundance of the nearby Tile of the new building through field neighbourTiles.

Edge:

Edge can also contain resources. We can achieve that by simply adding a new field in class Edge, and called function Player::addResources() when the player built a road.

Strategy:

We can create new strategies for the new game rules and more functionalities. For example, we can change the difficulty of the game by adding new biased dice, moving geese when rolled 2, 7, 12; steeling others' resources when rolled the same number consecutively three times, and so on.

Player:

We can enable to add or drop players by using a Queue, then pushing or popping the players as we want. All other methods related to player sequence will all depend on this queue. We can easily retrieve the current first player in the queue at run time, and we can pop it and emplace it to the end of queue when moving to the next term.

We can also set new game rules toward the players. When the game has already been played for a long time, say 10 rounds, then we can automatically update the game rule to an "easier mode" – all players can get one of all the resources (i.e. one Brick one Heat one Energy, one WiFi, one Glass), to speed the game up.

Board:

When we want to re-shape the board like board with hexagon tiles, we only need to change the following methods:

- Board Constructor
  This is quite intuitive. Emplacing new (or removing unused) components (edges/vertice/tiles) to the end of each vector.
- Board::initAttachBoard
  Since this method works to link all edges/vertices/tiles to the board. We have to modify it by linking the new edges/vertices/tiles with the board.
- Board::printBoard

Since the board layout changed, the print methods need to be slightly modified. While since we didn't totally rely on the hardcode, but using a few helper functions to improve efficiency. This enables us to print out a new board easily.

The rest methods which are related to the board will keep the same. For example, Board::initRandBoard, Board::initLoadBoard, Board::initSeedBoard, Board::loadGame, Board::saveGame. Because when we implement those methods, we pay special attention to the way we retrieve each component. We didn't explicitly use the number of tile/vertices like 19 or 54, since those numbers will narrow the board into this single layout. Calling them implicitly avoids redundant modifications when we want to change a new board.

For the other methods in class Tile, class Vertex, class Edge, class Player, they worked on an individual object, with no specifications. Thus they do not need to change at all.

Some other possible changes:

.

1. Trade with more than one resources: just adding a new function parameter
2. More than one group of Geese: modify the previous geesePos field from an integer to a vector of integer.
3. Lose a certain amount of resources when geese rolled: just adding a new function parameter as target percentage, and change 0.5 (a half) to that percentage
4. ……

Difference between DDL1 and DDL2

We have make a lot of changes as we proceeding the implementation. Some changes are quite valuable which improve the functionality and program efficiency to a large extend.

In DD1, we added the field to store all neighbor vertices of the current vertex, and neighbor edges of the current edge. We wanted to do that since we expected to check if building action is valid by retrieving those values in the run time. However, we found that the time we used on attaching neighbors is quite more than just checking them in a loop with no more than 3 iterations (at most 3 edges connect to one vertex; at most 2 vertices connect to one edge). Therefore, we removed those non-necessary fields.

Similarly, we used a single method to doubly link the edges with vertices, which improve the function efficiency a lot and avoid too much effort spending on the linking of classes.

Before in DD1, we put the dice in class Board, while later we found that each player should have a single dice, with different strategies. Using a common dice will make us hard to keep track individual's strategy. Therefore now we put Dice into the class Player.

We changed a few stack-allocated fields/objects to heap-allocated fields/objects. For example owner of a Vertex/Edge became a pointer pointing to the corresponding player.
and applied smart pointers on them to avoid memory leaks.

We had also added a Bridge Design Pattern class to hide information in board. We defined a few more methods like Board::printAllChoice(), Board::autoCorrect() for our extra features.

Design

Observer design pattern

Observer design pattern is used to connect each tile and its vertices. We create two classes Subject and Observer. Each tile acts as a subject and we attach 6 vertices as observers to the tile when initializing the board. During the game, when a tile is selected, it calls the method notifyObservers() to check whether its 6 vertices have owners. If any of the 6 vertices has an owner, then these vertices call the method wasNotified() which is a pure virtual method under Observer class. In the method wasNotified(), we update the residence of the current vertex.

Also, vertex and edge can be seen as observer and subject relationship. Vertex acts as an observer of edge. When a road is built on an edge, the edge notifies its observer vertex that player is able to build a residence on the vertex, if no adjacent vertex exist. The method Vertex::wasNotified(Edge& whoNotified) will modified the Boolean field `canBuild` to `true.` Similarly, when a residence is built on a vertex, it will notify its subject edge that a road can be built on that edge, if there exist adjacent edges. The method Edge::wasNotified(Vertex& whoNotified) will modify the Boolean field `canBuild` to `true.` This enabled us to check if the building action is valid much more efficient!

Strategy design pattern

Strategy design pattern is used to achieve the switch between loaded and fair dice. In the implementation, we create a class Dice which acts as a controller to show the dice point we rolled, the current strategy we are using to roll the dice and provide a method to swap between two strategies. Then, we have an abstract class Strategy which is composed form Dice. Under the Strategy class, we have two concrete classes DiceRand and DiceLoad which provide the algorithm of generating random rolled dice and loaded dice respectively. By calling the pure virtual method rollDice in Strategy class from the method in Dice class, we achieve to roll dice in different strategies.

Bridge Design Pattern

Bridge design pattern is used when implement the game. We create a class CtorGame as a controller to collect the information from user input such as the type of board the user wants to create and the seed. The class CtorGame will create a heap-allocated board in method CtorGame::play().

The information collected from users is passed to several methods which connect to Board class to set up the game. In this way, all functionalities are stored in the Board class. Even though the user can access to ctorgame.h, they will not be able to see more detailed information, cause we hide details from the private fields in Board class. Moreover, this keeps the resilience to change by simply adding properties in CtorGame class and adding corresponding methods in Board class.

To achieve low coupling, we reduced the re-compilation times when making any changes by using forward declarations, as much as possible throughout the whole program.

For example, we used forward declaration for the class Board in ctorgame.h; forward declaration for class Tile, Vertex, Edge, Player in board.h. Thus, if we make any modifications on either board.cc, tile.cc, vertex.cc, edge.cc, player.cc , only the corresponded object file will recompile.

Extra features

- **Smart Pointer**

  We have applied smart pointers whenever we want to define a heap allocated object, which successfully avoided memory leaks. For example in class Board, we used shared pointers within vector of Tile, vector of Vertex, vector of Edge, vector of Player. For example in class Dice, we used shared pointers for a heap allocated Strategy.

- **Auxiliary Game Suggestions**

  Users can customize their game by enter -customize in command line, then the customizing questions will be prompted:

  1) Enter the building points to win the game.

     If the user wants to play a quick game, then enter a smaller value. The value should be between 3 to 15, since initially everyone has 2 basements, 2 building points.

  2) Do you want to have geese involved in the game?

     Geese will steal resources when dice rolled with 7. While the user can remove geese from the game, nothing will happen (include losing resources, moving geese, and steal from others) when dice rolled to 7.

  3) Do you want to get building suggestions in each turn?

     When testing the program, we sometimes forgot how many resources required to build a basement or upgrade the buildings. To avoid checking the game rules again and again, we provide a Suggestion to players. This works as an auxiliary function or a hint to the player.

     If the user entered yes, a building suggestion will show up after each player rolled the dice.

     Suggestion are determined by the recourses available to the current player.

  If the user does not want suggestions to be shown in each turn and did not set it in customize step. The user can just simply enter the command suggestion, when you need, after rolling the dice. This command will give exactly the same suggestion as talked in the previous section, but only for the current player, current round.

- **User friendly input recipe**

  Whenever the player needs to input a string/command. They can enter with any case. Also, within the trade command, We provide an auto-Correct tool that the user only needs to make sure the first letter of each type is correct. The tool will check the input and compared it with possible commands or strings and used the one with the highest contact ratio. For example: “trade O good Hi” will be corrected as the current player offers Orange one GLASS for one HEAT. This saves the time entering the inputs and user does not need to worry about the spelling problem.

Answers to Questions

Q1:

Factory design pattern can be used to implement this feature. Factory design pattern provides an interface for object creation. We have our sub-class to choose the type of objects to create, in this case, a randomly created board or a read-in board file. We can create objects without specifying their concrete classes so that a high level of flexibility will be shown when dealing with concrete classes, which also increases the resilience to change. However, we did not use this design pattern since we would like to simplify the code and make a more comprehensive controller class named CtorGame. This class records all the information we need to play the game including whether it is a random board or a read-in board and other information such as the seed. In this class, we call different initialization functions that are implemented in the class Board so that we can easily set up the board we want to use. In this way, we keep the resilience to change by simply adding properties in CtorGame class and adding corresponding method in Board class.

Q2:

Strategy design pattern can be used to implement this feature. We want to apply strategy design pattern since we can flexibly alter between loaded and fair dices by performing two different algorithms. A Strategy design pattern is intended to define a family of the algorithms, encapsulating each one, and making them interchangeable, so that we can switch between loaded and fair dice at run-time. In the implementation, we create a class Dice which acts as a controller to show the dice point we rolled, the current strategy we are using to roll the dice and provide a method to swap between two strategies. Then, we have an abstract class Strategy which composed from Dice. Under the Strategy class, we have two concrete classes DiceRand and DiceLoad which provide the algorithm of generating random rolled dice and self-defined dice respectively. By calling the pure virtual method rollDice in Strategy class from the method in Dice class, we achieve to roll dice in different strategies.

Q5:

Strategy design pattern can be used to implement this feature since a dynamic change of computer players is actually a family of algorithms. Under different circumstances, the computer will switch to a different strategy, in other words, a different algorithm. To allow them to be interchangeable, we could use a Strategy design pattern, so that the computer player would be more advanced and smarter. A strategy design pattern also increases the resilience in changing the behavior of computer players.
To be specific, we would create a class ComputerPlayer to record the current computer player mode we're using and provide a method to swap between different computer player modes. Then, we create an abstract class Strategy which is composed from ComputerPlayer. Under the ComputerPlayer class, we have many concrete classes which represent different computer player modes. By calling a pure virtual method computerPlay() in Strategy class from ComputerPlayer class, we achieve to have computer player in certain mode replacing real player.

Q6:

A Decorator design pattern can be used to facilitate this ability as we want to add functionality or features to a Tile object. A Decorator design pattern is intended to let us add functionality or features to an object at run-time rather than to the class as a whole. To implement this feature, we can simply create a BaseTile Class which inherits from Tile class to record the basic information of a Tile such as the index of the tile, recourse type and value. Then, we create an abstract Decorator Class which also inherits from Tile class. In Decorator class, we have a protected field of a pointer to Tile object and all pure virtual methods we want depending on the feature we want to build. Then, we create all the features we want as a sub-class of Decorator with any additional field we need and an implementation of all pure virtual methods in Decorator class. Then, through these sub-classes that inherit from Decorator class, we can achieve the feature such as obtaining multiple types of resources on a single tile or reduce the quantity of resources produced by the tile when improving a vertex. This also increases the resilience in changing the behavior of a Tile that all additional features can be added as a new concrete Decorator class.

Q7:

When we answered this question in DD1, we anticipated using exceptions in Vertex Class, Edge Class, Board Class, Player Class, and the main function. However, after the implementation, we found that the board will control all the actions related to Vertex, Edge, and Player, for example, build a residence, roll dice, trade between players. Since all those methods have different specifications on types, range, format, and contents, checking those inputs methods by methods will be redundant. Also, putting everything in the main function will make the function lengthy. Therefore, we first create a new class called CtorGame, which acted as a wrapper of the board, and conduct the game step by step. Secondly, we receive almost all inputs from the user in CtorGame::play() and test the validity. Lastly, we pass the valid value to the specific functions like Board::buildRes(int pos). To avoid testing repeatedly, we defined two helper functions: askForCommand() which asks users to input a string, and askForInteger() which asks users to input a non-negative number. Also, the latter has a default parameter set. When no parameter is given, all non-negative integers are accepted. While some functions have required upper and lower bounds, like dice should be 2 to 12, we can easily call the function with the different parameters for different cases. Thus values that are out of bounds, in the wrong type, or EOF will throw an exception. Since we controlled the inputs and exceptions separately in helper functions and check validity before passing to the corresponding methods, any wrong inputs will be discarded and the current state will not be changed. If the program is aborted, the smart pointers will ensure no memory leak possible. Thus all those functions offered a strong guarantee.

Final Questions:

Q1:

It’s a great teamwork experience. We really learned a lot from this group project!

It’s really nice to make a plan in advance and follow it. We held several meetings as we planned in our DD1 plan. The meetings ensure we would not deviate from the applicable workflow and plans. Also, this enabled us to connect with each other frequently and provide feedback in time. We started the code implementation just after the DDL1 which gave us much more time to be flexible to the sudden unpredictable bug or challenges. We strictly followed the plan we made in DD1, which really helps us a lot. We left plenty of time to debug, add extra features, and adjust to any time conflicts. Besides, we all agreed that when anyone encounters problems, we will not hide the issues and team members will work together to reassign the workload. This ensured that we will not delay the process due to a single problem, and can adjust easily.

We properly assigned all sections to each team member. Everyone worked on a relatively integrated section. Since we worked in relatively independent sections, we didn't need to know too much about what exactly the method did in other's sections. Therefore, our design was developed in a “high-coupling low-cohesion” environment naturally.

Also, whenever we find there are any problems that occurred in other's section, or we want to modify any methods, we will inform that group member, and make any changes with the agreements. The changes are made by only one person and pushed to GitHub directly. Therefore we will always work on the latest and correct version. After the implementation, we tested the whole program. Each group member tested the section developed by others. This is increases the variety of edge cases and new cases. Also, we did functional testing whenever we finished a feature. Thus we ensured high robustness, and no errors will be carried on.

This is a quite valuable experience where we had applied many useful technics learned in class. The way we worked together make the teamwork efficient. We will apply this experience in future studies and workplaces.

Q2:

If we had the chance to start over, we would design the different versions of Makefile and test the main file to ensure each section works properly. This will make the functional testing quite relaxed. We would also design a more detailed progress plan for the report -- to be specific which team member works on which section. If we have enough time, we would make a thorough walkthrough and tests just as what we did in DD1 of previous assignments.