Overview

Our program supports all the required command-line and the -computer command for extra features. Main is used to get the command line message, compile and submit this message to the controller. Controller, gamemodel and gameframe build as MVC. All the logic staff of the game is done in the controller. Gameframe is just responsible for printing the staff in gamemodel to cout. Gamemodel contains all game data and objects (also method to modify them). All game object instances (class) are created by each factory method. SetBoardStategy(abstract class) is used to set the initial board, so transfer a different setBoardStrategy to the controller and let the controller set the board by different strategy. For example, -load xxx command uses setfromfile strategy to set the board and players using the datas from the provided file.

Design

The first design challenge is that we need to separate the functions required to implement the game into different parts and maintain low coupling between parts so that group members can do their own part of jobs at the same time. Hence we used MVC technique, which separates the main logics to run the whole game and all small parts to implement the required functionalities. The main logics of the game are in MVC(Controller, gamemodel and gameframe), others just do the supporting jobs.
Another challenge is that we need to provide high flexibility to create and use different kinds of objects. Also, we need to consider the possibilities of adding new subclasses or new functionality in the future. These changes in the existing code should be minimized. The technique we used to solve it is using the factory method. We have three factories, the board factory, the player factory, and the dice factory. The first advantage of using this technique is that when we need to add a new class, none of the calling client or the unit implementation needs to be told. We can just create a new class and extend our factory method. For example, if we want a different kind of player “smart”, we can just create a new subclass inherited from player class and create it using playerfactory by giving parameter smart. Moreover, the factory acts as a knowledge center producing needed objects, so that the places that need the products do not need to know how to construct one. When we need to add a new object, the only thing we need to do is to change the instance returned from the factory, other parts of the code will stay the same. This technique not only helps to maintain the low coupling, but also helps to maintain the high cohesion.
There is one place that differs from the original design. We planned to create the dice in gamemodel class, however now we created the dice in player class. When we actually implemented the code, we found that it is more reasonable to place the dice in the player class because in this way we can indicate that each individual player has his own set of dice.

Resilience to Change

Our design can accommodate new features and changes to existing features easily.

- If we want to use a different way to print the game, we can just directly add a new printing method and change the method to use in the gameframe. If we want to print the tile in a different way, we can just change the printing method in tile class. Same as the board printing method.
- If we want to support a new command-line, we could add that command option in main and controller without influencing any existing parts.
- If we want to add a command for the user to use during the game. For example, add -road command, which prints the roads that has been built of the builder, add the command option in controller and add one specific that implements the command in gamemodel.
- If we want to use other ways to generate the initial board, we can just create a new strategy inherited from setBoardStrategy, let main transfer that into the controller.
- If we want to set up a different kind of board(such as different sized or hexagonal), we create new board subclass inherited from the Board and let boardfactory produce its instance in gamemodel. If we want to use different player behaviour, we let the new player inherited from the Player and playerfactory produce it. If we want to use a different set of dice(such as 12 sided dice, 24 sided etc) , we let the new dice inherited from the Dice and dice factory produce it.
- If we want to support a different number of players, we set different number of instances in gamemodel.
- If we want to support different numbers of players and computer players. For example, using 2 computer players and 2 human players, we can add a new command line option: -computer2, where the number indicates the number of computer players. By passing this command string from the controller to gamemodel, the gamemodel is able to use a different number of each player created by playerfactory in its constructor.
- If we want to change computer players' strategy, we can just change the specific *choose* method. For example, if we want to use a different strategy to choose the geese tile after rolled 7 for computer players, we can change the chooseTile function to check which tile has more builders and does not have itself's residences. This way, we can easily change the strategy of computer players by changing the methods of choosing in computer player class
- If we want to change the input syntax, for example, read all the users' input from a file, we could overload the operator >> in the humanPlayer class.

- If we want to change the rules on building a residence or roads on the map, we can simply change the specific function that implements the rule. For example, to change the resources that need to upgrade the residences, we can simply change one function called attemptBuild in the Player class. If the rules to build roads or vertices needed to be changed, simply implement the canBuild method in Board class differently.
- If we want to use a different way to save the game into a file or another place, we only need to change the implementation of the function savefile() in the GameModel.
- If we want to implement a different rule For example, when we roll 7(moving geese), we just need to change the implementation of the function update() in the GameModel. If we want to implement different rules to trade resources, we just need to change the implementation of the function chooseToExchange() in the Player.
  Overall, if we want to change the high level logic, we can change the implementation in MVC, if we want to change a specific rule or behaviour, we can simply change the single method which implements that functionality.

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?

I used Strategy. By using this design pattern, I could create a strategy interface in the game so that when I put in different instances and implement them, I can directly change the strategy to use. In this game, each strategy that loads the board is inherited from the class SetBoardStrategy. When the controller receives different strategy instances from main, it can load the board in different ways by calling methods in different strategies. In this way, if you want to add a new strategy later, just let the new strategy class inherit from SetBoardStrategy and let main pass a different instance.

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?

Here I am using a factory method. Different dice instances are obtained through the factory method, and different functions can be realized by calling the methods of different dice instances during use. In the game, I made some changes to this design pattern. By passing different parameters to the dice factory let them create different dice based on this parameter. In this way, if we want to add more kinds of dice in the future, we only need to pass different parameters to the factory to get other dice without changing other code.

4. At the moment, all Constructor players are humans. 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 always followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.

I use the factory method, the factory method will generate player instances and all players inherited from Player. We only need to change instance return from playerfactory to computer which overrides the methods in Player. The *controller* controls the overall procedure of the game. It checks the players' command and calls the corresponding methods in *gamemodel*, The gamemodel instantiates the needed objects(human players or computer players) then implements every specific phase procedures.

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?

I will use the factory method, each level of computer will be a class inherited from Player which overrides the methods(with different playing strategies) in Player. For example, if we want to have a smarter computer player, we could create a new smarterComputer subclass inherited from the player class. Then override chooseToExchange function to see if the new resources can help the computer build new basements/roads or not then decide. Also, to decide when to build a basement or road we can also change the override functions. By implementing the methods in a different way, the computer player can have different strategies.

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?

I can still use the factory method to create a new class, say newBoard which is inherited from Board and the instance created by boardFactory. The newBoard can have different fields and process data differently by overriding methods in Board differently.

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.

No. The place I could use exceptions is in human player class, where the program reads the players' inputs**.** I could have used exceptions to catch players' invalid input and give error messages. However, if the player inputs are invalid or undefined commands, there will be several problems. Most importantly, if the inputs are invalid, we need to guide the players to input valid ones. First, if using exceptions, it is not easy to implement, makes code complicated, hard to debug and no big use in this program. As the program requires throwing many different kinds of corresponding error messages to the cout (invalid roll, invalid command etc.). Throwing exceptions will stop running subsequent code which can easily cause bugs. There is an easier way to do it is just to use it to check the command. If not getting the correct command, print the error message and continue reading.

Extra Credit Features

We added a computer player feature. This allows the single human player who can not find another three people to play with three computer players. This feature helps to relieve loneliness of the player and provide more possibilities and fun to this game. The computer players' actions follow the objective of winning the game, however they also have unexpected actions since we set some valid random actions for them.

The particularly challenging part about implementing this feature is to make the computer behavior rational. The main difference between the computer player and human player behaviour is that the computer player should give valid commands(such as choose the valid vertex and edge). We need to find a way to let the computer choose which command to process in each turn and how to let the computer choose which vertex, edge or tile. Another challenging part is to coordinate the computer player with its logic control class(gamemodel). Since the gamemodel controls both human players and computer players. It reads the input from players and outputs corresponding messages for human players, but these messages shouldn't be output for computer players.

The method we use is to put all number choosing features into Player(class) itself. Computer players are also inherited from Player so it has methods which are used to choose vertex, edge or tile. What we need is to call the specific methods in each situation which needs the computer to make decisions. Whatsmore, computer players shouldn't cheat so they always use fair dice. To show the player the detailed updated situation, the computer player also always uses residences and status commands. Overall, when active the computer mod, each computer can have its own logic which will make the game become more fun.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you
   Previously, I thought that every part of a game is closely connected. This will make it difficult for many people to complete the project together because I have to wait for another person to finish his part before I can start. But now by separating the different parts and allowing them to achieve different functions, the team can work at the same time without affecting each other.
2. What would you have done differently if you had the chance to start over?
   I will implement the function of gameframe(print board) by using strategy, so it can support switching the print strategy during the game.

Schedule

Task Contents	Estimated completion dates	The partner who will be responsible for the task	If sticked or not

Generate the design pattern of the whole project roughly	July 23	All members	Yes
Discuss the details of the design pattern	July 25	All members	Yes
Make a draft of the initial UML class diagram	July 26	Andrew	Yes
Finish the initial plan of attack	July 27	Andrew	Yes
Revise the initial UML class diagram and the initial plan of attack (fixing typos, adding details)	July 27	All members	Yes
Implement the Board class, Dice class and Player class	August 1	Bella	Yes
Implement the GameModel class, Controller class,	August 1	Steven	Yes
Implement the main for command-line Interface	August 1	Andrew	Yes
Integrate different class modules together and fix any bugs in order to meet project requirements and avoid any memory leak	August 4	All members	Yes
Revise the initial UML class diagrams to the final one	August 6	All members	Yes
Write a draft of the final design document	August 6	All members	Yes
Add additional feature for bonus	August 8	All members	Yes
Finish the final version of the final design document	August 9	All members	Yes
Test if the additional feature works and double check if the basic function works.	August 11	All members	Yes

Fix any bugs.			
Work through the demo plan in the last few days	August 13	All members	Yes