Design Document: Constructor

Group Members: Yan Zhao, James Hong, Terry Zhang

Overview:

The core structure is divided into three main components: board, player, and game.

The *Board* class reflects the state of the board. It holds a vector of *Vertex* objects, a vector of *Tile* objects, and a vector of *Edge* objects, where each *Vertex*, *Tile*, and *Edge* represents one of the elements on the board. The *Board* class provides public methods to build and improve residences and roads, as well as changing the geese's location.

The *Player* class reflects the state of a player. Every player has their own instance of *Player* object. The *Player* class holds the quantity of resources that they own as integers, and holds the buildings and roads that they have built as a map of *Building* and *Road* objects respectively. The *Player* class provides methods to check if they have the resources to build or improve a residence or road at a certain location, methods to actually build or improve those residences or roads, and methods to facilitate the gains or losses of resources from resource distribution, geese, and trade.

The *Game* class holds one instance of *Board*, and four instances of *Player*, and it provides a public method to allow the game to run. Within the public method "run", it loops through its private methods that correspond to the various stages of the game, such as the initial basement-placing phase of the game, the beginning of each turn, and during the turn itself. Each of these private methods will be able to ask for the appropriate input from the player, display the appropriate prompts on screen, and relay the appropriate commands to the Board and Player when necessary.

In order to facilitate these core classes, the *CreateGame* class is responsible for creating an instance of *Game* in the way that we intended. The *TextDisplay* class holds a pointer to the Board class, and has a dedicated print method to print the board via std::cout. The *main.cc* file holds the main function and contains the code that selects the correct game option.

There are additional methods within these classes, such as accessors, mutators, and type converters to allow these classes to retrieve the necessary information. For more details, please refer to the Design section of this document, or see the code. The core structure has not changed from the plan of attack document from DD1, But some of the finer details of implementation did change from our original plan.

Design:

The first challenge for us was to select the right way to represent the relationship between the vertices, edges, and tiles. The rules of the game heavily rely on the concept of "adjacency" between vertices, edges, and tiles, since the construction and resource distribution rules around the roads and buildings are all dependent on its adjacent elements. However, as mentioned during lectures, there is no obvious mathematical relationship between adjacent elements, and we needed some way to keep

track of which elements are adjacent to each other. For this reason, every *Vertex* object has three vectors of integers, each corresponding to its adjacent vertices, edges, and tiles. Additionally, every *Edge* object also has two vectors of integers, which track its adjacent edges and vertices. When a game mechanic requires us to check the state of one of its adjacent elements, the code will go into one of these vectors to identify the index of those adjacent elements, and then use these indexes to make the appropriate computation. As a side note, the reason we chose to track the indices by index rather than by holding a pointer to the object itself is because these objects are held in a vector from the *Board* class, and the vectors can get resized during game creation, which would invalidate the pointers.

The second challenge was to keep track of the interactions between the various components of the project. Towards this end, we made extensive use of immutability assurance measures. Object fields were made to be private, and public accessors were only implemented if they were necessary and were declared const whenever possible. Mutators were made to be as specific as possible so that they reflect the way the class is supposed to be used. For example, when a player loses half of their resources to the geese, we have a “lost_to_geese” method for this specific use case to ensure that the resources are lost in the right way, rather than calling “lose_resource” individually.

The third challenge was to enforce the proper sequence of steps for game creation through the use of the Factory Method design pattern. First, the board needs to be created with the right distribution of tile attributes, depending on whether it’s read from a file or randomly generated, and all the elements within the board need to have the adjacency indices properly loaded. Then, this *Board object* gets passed to *Game*, and *Game* will set the states of the *Player* and *Board* to an initialised state when starting a new game, or to the corresponding state when loading from a save file. The *CreateGame* class and the Factory class make up the design pattern, and are entirely dedicated to creating an instance of *Game* according to the correct sequence, by using a dedicated “create” function.

The fourth challenge was to enforce the proper sequence of steps when the game is running through the use of Template Method design pattern. As mentioned in the previous section, the “run” method in the game loops through “turn_start” and “turn_middle”, and each of these methods are then calling the appropriate dice roll method, geese method, build methods, save methods, etc, which correspond to a specific action that a player can take. This ensures that the sequence of actions that the player takes within the game is correct, and that these actions are implemented by calling the right sequence of functions.

The fifth challenge was error handling. When std::eof(end of line) is called, the game is presumed to save and then exit the program. However, there are many different places within the code that ask for user input, so we decided to turn on exceptions for std::cin, and have the main function catch the std::eof, which then calls for the game to be saved. Another possible point of failure was that the user might input an invalid command, such as inputting a non-numeric character when the game needs an integer from the user, resulting in input failure. We threw std::logic_error whenever user input fails, or when the user inputs an unrecognised command, and strategically placed the catch block to resolve the failure and ask for another input. The third point of failure was that the user might

try to access the board location that doesn't exist, and we simply put in a range check in the *Board* methods to account for this. The fourth point of failure was for an invalid file read. When the user passes the filename to the executable, the main function will check the file if it can be read, and then start building the *Game* class. Note this does not enforce the proper format of the file.

The final challenge was random number generation. We needed a way to randomly generate numbers for both the *Game* class and the *Player* class, but we wanted them to be able to share the same seed, and reshuffle the same seed whenever a random number is generated. Both of these classes have a *Random* object, but now the seed number of the *Random* class is made to be static. Calling the constructor of *Random* no longer sets the seed, and now the seed is set during game creation. This implementation allows us to make sure that the two *Random* objects are not using the same sequence of seeds as they generate numbers. As an aside, if the seed isn't set by the user, the seed is generated from the system time, which would be a truly random experience for the user.

Resilience to Change:

The structure of the board can accommodate changes in the size and layout of the map. If we were to implement a new map, the *Vector*, *Edge*, *Tile* classes wouldn't change, since those elements themselves didn't change. The *Board* class wouldn't change either, only the contents of the vectors within it. Since these vectors of *Vertex*, *Edge*, and *Tile* are sequentially read in from file in the *CreateGame* class, we only need to provide another set of text files corresponding to the new layout and adjacency structure of the new map, and tell the *CreateGame* to read from those files instead.

The structure of the *TextDisplay* can also easily accommodate change. The way that each map element is displayed is codified within its private methods, and the sequence of these methods to call when printing these map elements is also read in from a file. A new size or layout of the map would only require a new text file for *TextDisplay* to read in, and the board will still print properly.

If we were to add more improvement options for the residences, such as different tech trees, we could add in the new type in *BuildingType*, and the logic into the improve method from the *Building* class to support the new tech tree.

If we were to add more players, we'd only need to add more colours into the *Colour* class, and tweak the operator++ for the *Colour* class. The loop in *Game* loops through all the different colours using the operator++ both during game creation and during gameplay. Thus, the creation of the additional *Player* object is handled automatically, and the player action of the additional player will also be handled automatically.

If we were to add a computer player, then we would need to keep track of which players are computers within the *Game* class, and make the "turn_start" and "turn_middle" functions ask for the computer player to make the appropriate decisions. The computer player will then call the appropriate action-specific method, such as "roll_dice", "steal", "trade", "build_road", etc. The core structure of the *Game* class would remain intact, and it would behave as if it's an interface for the computer player to use.

If we were to add online play multiplayer play, we would simply allow each player to call their action-specific function when it's their turn, in a way similar to the change we can make for a computer

player. Then we have the central server read the packets they received to advance the state of the game and notify each user.

Answers to Questions:

Question 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 see this design pattern? Why or why not?

To implement the ability to choose between randomly setting up the resources of the board and reading resources a file at runtime, the Factory Method design pattern would be a great choice. This design pattern encapsulates the creation rules for an inheritance hierarchy.

In this scenario, we can define an abstract factory class to declare methods for creating the resources on the game board for each tile. We then implement concrete factory classes that are responsible for creating the resources of the game board according to a specific method, such as random generation or reading from a file. We can create a factory mechanism that allows users to select the desired factory at runtime. Doing so allows us to instantiate the corresponding concrete factory class by choosing which methods to create the game board.

We are using this design pattern for game creation, but not for this purpose. We have the CreateGame class being the abstract class, but we now only have CreateGame as the concrete class. We are passing in the filename and the load type (random, savefile, or board layout), which changes the way the input file is handled. As for the random board generation, we have decided to leverage existing code, and randomly generate a sequence of integers that follows the tile distribution rules as if it's a board layout file, and we simply read this string and proceed as normal. The reason for this decision is that there is a lot of overlap in the code between the different modes of game creation, and we didn't see enough of a benefit from using this design pattern in this particular case.

Question 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?

To implement the switch between loaded and fair dice, we can use the template design pattern. A template method encapsulates an algorithm where steps must occur in a particular order.

In this scenario, we can create an abstract class representing the dice with methods that outlines the general algorithm for rolling the dice. Within this abstract class, we can have two concrete subclasses: the loaded dice and the fair dice. The specific implementation of generating numbers would differ between the subclasses.

We are not using this design pattern in this case, because there are only two different ways to roll the dice, and it would be easier to store the state of the dice as a boolean and use one method dedicated to the dice roll based upon the state of this boolean.

Question 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 board for a different number of players). What design pattern would you consider using for all of these ideas?

For supporting various game modes and configurations, the Factory Method design pattern is the way to go. This design pattern encapsulates the creation rules for an inheritance hierarchy.

We have an abstract Factory class, and a concrete CreateGame subclass that is entirely dedicated to creating an instance of Game according to the intended sequence of steps. If we were to implement different game modes, we would create more subclasses to inherit from Factory, and setup the game according to their own rules.

Question 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 always followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.

To ensure that all player types always follow a legal sequence of actions during their turn, we will be using the template method. A template method is useful when multiple algorithms share common steps but may have variations in certain steps.

To achieve this, we can first define a computer class that serves as the superclass that contains the methods that represent the sequence of actions a computer player takes during their turn. Then, concrete subclasses will be defined for the different versions/difficulties of the computer player. However, both the human players and computer players must interact with the game_start, turn_start and turn_middle functions within the Game class, and they are only allowed to call certain action-specific methods within the Game class when it is legal. This ensures that all sequences of actions during a turn are legal.

Question 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.

Using exceptions in a project is a common practice to handle error conditions or exceptional situations that may arise during runtime. In this project, exceptions can be used in handling invalid commands, file reading errors and invalid player actions.

Firstly, exceptions are used to handle situations where invalid commands are provided by the user. We do have a dedicated method to request for a specific type of input, as discussed in the previous document. But now, when there is any command that is not being recognized or syntactically incorrect, an exception std::logic_error is thrown to signal the error. The code block that requested for the input can ask the user to try again. We decided to use exceptions in this case because it was easier to redirect the control flow to a desired location whenever we encountered this error.

Second, exceptions could be used when reading from files such as savefile.txt and layout.txt, when potential errors may occur. For example, the file might not be found or accessible. Or, the data within the file could be invalid (for example if a residence location is out of range, or if a resource type character is somehow invalid). Then, the exception can be thrown to indicate the error. We are not

using exceptions for this case, since we did not have time to build the full error checking capability. Instead, we will only check the filename inputted by the user to see if it can be read.

Third, exceptions can be employed to handle invalid player actions. For instance, if a player attempts to perform an action that is invalid given the current game state, even though the command is recognized, an exception could be thrown to signal the error, to indicate that the attempted player action did not succeed. We are not using exceptions in this case, since the methods that represent player actions can use a boolean as the return type, to indicate whether it succeeded or not. Since the outcome of an attempted player action is simple - it either succeeds or it fails, it's easier to implement this without using exceptions.

Lastly, we added in exceptions to handle std::eof signals, so that the game can be saved before the program exits. This is a very convenient way to make sure that the game is saved upon exit when we want it to, but not otherwise, such as when the game is won. We did not foresee this problem when writing the previous document, and avoiding the use of exceptions to implement this feature would be much harder than simply turning on the exception and catching it.

Extra Credit Features:

Ineffective memory management often causes memory leaks when handled improperly, this is especially true when the code is under many changes. Thus, we eliminated the risk of memory error by eradicating the use of raw pointers. The data in the program are handled through the std::vector, std::map, and std::set containers provided by the STL. Its usage greatly reduces the chance of memory leak, since it allows automatic memory deallocation under the RAII principle. With the reduction of memory leaks along with robust functionalities, the effective memory management measures allow for better handling of any new changes to the data structure.

There is one pointer in the entire program, which is that the TextDisplay class holds a pointer to the Board class, so that the TextDisplay can have access to the print functions from Board. But TextDisplay does not own Board, because Board is always a field in the Game class, which is always on the stack. Since *TextDisplay* is also a field in the Game class, whenever the Board gets destroyed, TextDisplay is also destroyed so will never be a dangling pointer.

Attempts have been made to implement the computer player to the point where a beta version was implemented. In the beta version, the two subclasses *HumanPlayer* and *ComputerPlayer* are hosted under the player superclass, and behaviours of function in the game class change based on the player type. The beta version also introduced new memory management techniques such as smart pointer and dynamic binding. However, due to concerns about the uncertainty around the possible errors that could be caused by these techniques, we decided to revert to the basic version.

Final Questions:

Question 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?

The first lesson we learned is the importance of communication. Throughout the project's duration, effective communication was carried out to exchange ideas during discussions and update each team member's progress with the rest of the group. This experience of collaborating within a team environment for a large-scale software project provided us with invaluable insights. We learned the significance of planning, designing, and communication facilitated by version control software, all of which will be invaluable for future endeavours.

The second lesson when working on this software is the importance of using version control technology to streamline the development process. Throughout the project's duration, GitHub assumes a pivotal role by facilitating access to the latest code iteration for all team members. This expedites feature iteration and delivery, minimising redundant efforts. Furthermore, GitHub provides the capability to revert to prior code versions if necessary, aiding in the effective management of code changes.

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

We would have given more thought into choosing the appropriate STL container for the various classes. We had to rewrite some code because we had to switch between vectors and maps, due to unforeseen issues. For example, in the Player class, to keep track of the residences and roads they had built, we originally used a vector to store that information. But it was later discovered that using a map with the index number being the key yielded much cleaner code, and we had to make that transition after making significant progress with the *Player* class.