CS246 Final Design Document

Overview

When implementing the game we focused on the four distinct stages of the game, and handled their respective inputs and cases accordingly in a number of classes. The classes we implemented in our program are Dice, Builder, Vertice, Edge, Tile, and Board. We also have the Info, Subject and Observer that were used to implement the observer pattern in our game.

Observer Pattern
The overall observer design pattern didn't change from our original design. The Tile object (subject) notifies its Vertice objects (observers) when the value of the Tile was rolled. Then, the Vertice object (subject) notifies its Builder object (observer) to collect the appropriate resources. The only minor change is when we attach the vertices to the tile object. We initially planned to attach the vertice to the tile when it was initially built, however for simplicity we attached all the vertices to each tile at the start of the game. Furthermore, we will use this section to highlight significant changes in our UML from our initial design.

Dice
Dice is a class that is relatively short in terms of the code. The most important function is the **roll** function. It deals with the random number generation for a fair dice, but also takes in an int as a parameter in case the value of the roll needs to be set for a loaded dice. We also chose to have an enumerated class Die that would be used to set the dice is loaded or fair. We chose to use an enumerated class rather than a number value for each type as it made the program more readable, and in part helped in the debugging process.

Builder
The builder class deals with all of the player specific details such as the colour, points, resources, dice type, as well as resource allocation, losing resources to geese or having them stolen from other players, and trading. In large part, many of our functions stayed the same from our original UML in DD1, but we did make a few changes that account for some cases we missed, or were tweaked for consistency and simplicity sake. First, rather than having a build function for all three residence types we decided to just have two functions; **buildBasement** and **upgrade**. We found that it was easier to deal with the resources and restrictions of building a house and tower in one function, and that the upgrade function overall was more cohesive with the **improve** and **build_res** functions in the builder class. Additionally, we originally overlooked the option of loading a board from a file. We introduced the functions **setRoad**, **setResidences**, and **setResources** that set up all the details for the player when the game starts from loading a board from a file. Another thing to note is that anywhere where we initially passed a raw pointer as a parameter (ex. **buildRoad**, **trade**, **buildBasement**), we decided to instead pass a smart pointer. As mentioned, we did this as a way to better manage our memory and lower the risk of memory leaks.

Edge
The Edge class differs very minimally structure wise from our original design. Apart from a few getters and setters added, the design stayed the same. An important part of the edge class was the **buildRoad** function that set the owner of the road to the current builder, and also updated the name of the road on the board. Our initial plan was to have **buildRoad** check for the requirements of a valid road placement in addition to setting the owner. However, to do so it would need access to the vector of vertice and edge positions (in the board class), so our plan was to make the class a friend within the board class. After further consideration we decided to check for validity of building a road in the board class, so that by the team **buildRoad** was called, it would be guaranteed to be a valid choice.

Vertice
In large part our vertice class was modified to be more simple. We removed any functions that dealt with building or improving residences and only kept functions related to the owner of certain vertices, the types of residences on the vertex, and attaching the respective observers. In the case of the vertice, the builder was the observer, so when residences were built we would attach the builder with the **attachBuilder** function. We also had overridden methods to notify the builder what resources they would be gaining when that tile and respective vertice was rolled. Similar to the edge class, we initially planned on checking for a valid basement within the vertice class, but due to the same reasons as explained in the edge class, we decided to move that functionality to the board.

Tile
The tile class also remained relatively the same, with a few small modifications. We removed the **getBuildersOnTile** method in favour of doing the calculation in the board class. We did have to add a few methods relating to the printing and formatting of the tile when displaying it, but kept the general structure of the class that we had in our initial UML the same.

Board
The Board class was the most extensive of all. Every class in some form had to interact with the board class. Therefore, as changes in other classes came to light it usually meant a required change in the board class. For the most part these were small changes, like adding/removing parameters. Larger changes were made in the addition of private member functions where we had to check for valid moves made throughout the game. By separating some code into private member functions we achieved more encapsulation, readability, and modularity. We also changed our member variables into smart pointers to manage memory and moved the hard coded relationships between the tiles/vertices/edges into a global file to avoid cluttering the class. A significant change was needed for adding member functions for printing, loading a board, and loading an entire game. Initially, we thought that having 3 public functions for these functionalities would be sufficient. However we decided to make our code more modular by separating the key functions into further methods. We created three different functions when loading a board from a save file, board layout, or from a seed. Doing this allowed us to easily distinguish between the three, as there are differences when setting up a game from the beginning, or from a save file. We also made a **print** function for the board, which we initially didn't plan on doing.

Structure of main.cc
The main.cc is very important to touch on as it is the test harness, and is where we explicitly deal with the commands, as well as the 4 stages of the game. In the beginning of the file we have a function called **startGame** that takes in a reference to a board as well a bool that indicates the game has started. The bool parameter start indicates the 8 initial basements have to be built and the game is in the *Beginning of Game* phase. As soon as the basements are built, start is set to false and a new bool variable beginning is set to true indicating the *Beginning of Turn* phase for a player. Once the builders roll successfully, beginning is set to false, indicating to the function to move on to the *During the Turn* phase. This function is dealing with all of the commands and is checking after every successful call of build-res or improve that the game has ended (as points are added to the players). Once a player wins, the main moves on to the *End of Game* phase where the player is prompted to either play again or end the game. The startGame in theory is a helper function that is called in main, after checking for all of the command line arguments, and is used in a while loop to constantly check if the game is being played over, or if the board has to be reset when the game is eventually over. We have also nested the **startGame** function in a try-catch block, so that we can save to backup.sv and exit the game when we read in an EOF character.

Design

This section will highlight how we gave each class a common purpose that allows us to meet the several design challenges. In addition, we will go over how each class interacts with their surrounding classes.

Board
The board is the central class in our design. The board owns (composition) all the key objects in our program: Builder, Tile, Vertice, Edge. Part of the board's purpose is to delegate many of the game's functionalities to the appropriate classes, while also completing many different checks in terms of valid moves using its private member variables. Another significant part of the board also takes care of loading the game by setting its private member variables and calling the correct member functions to do so. Similarly, the board objects take care of saving the state of the game by gathering data from the board's private member variables.

As mentioned earlier, the board class interacts with many key classes:

- Builder: In general, the Board communicates to the Builder object when the curBuilder attempts to change the state of the game, which would affect the Builder's points or resources. This includes numerous operations, like building or the GEESE removing resources.
- Vertice & Edge: The Board class interacts with the Vertice by calling its getters/setters for when the board has to save or load a specific state of the game.
- Tile: The Board interacts with the Tile class to implement the observer design pattern. At the beginning of the game the Board class attaches the appropriate Vertice objects as observers to each Tile object. In addition, it also calls Tile::notifyObserver for the Tile object when its value is rolled. In addition, the other interaction is when the GEESE is moved, the board has to change the values to the corresponding Tile Objects.

One of the other design decisions that we made when implementing the board class was the process in which we printed the board. Instead of hardcoding the format of the board, we read the format of the board from a board.txt file. We then took advantage of the way the board is set up (vertices are in increasing order), and was able to populate the information from the vectors of our vertices, edges, and tiles on the screen. This approach helped to make our code easier to understand, as well as potentially support the format of reading in different board types, as we have separated the logic of our code from the display.

Builder

The Builder class is a crucial part in our design. The purpose of Builder is to interact with their resources, keep track of their points, and their respective die. Therefore, it plays an important part in the functionality of the game which deals with resources, such as displaying the status of a builder, taking resources away for building or when the GEESE is in play.

The Builder interacts with the following classes:

- Board: Only sends data back to the Board class when a Builder member function was called.
- Vertice: The Builder object [observer] will call Vertice::getInfo to obtain information from the Vertice object [subject] when the Builder has been notified to claim resources from that Vertice. In addition, the Board class sets the appropriate variables to the Vertice object when it's either built or upgraded.
- Edge: The Builder sets the appropriate values to the Edge object when it's first built.
- Dice: Each Builder object directly owns their individual Dice object. The Builder interacts with the Dice by calling its roll function to obtain a value, dependent on if it's loaded or fair.

Edge

The purpose of the edge class is to store and then send information specific to the location of an edge on the board. Its main function is to initialize its builder, location, and name.

The Edge class only interacts with the Board:

- Board: Only returns information about the state of the Edge when their getter has been called.

Vertice

The purpose of the Vertice class is to set and communicate key member variables that make the Vertice object unique. In addition, it's a central piece in the implementation of the observer design pattern.

The Vertice class interacts with the following classes:

- Board: Only returns information about the state of the Vertice when their getter has been called.
- Builder: Communicates to the Builder object [observer] when resources can be obtained from the Vertice object by calling the Builder::notify via Vertice::notifyObservers method.
- Tile: The only interaction is when a Vertice object [observer] calls Tile::getInfo to obtain information from the Tile object [subject] who notified the Vertice that the Tile was rolled.

Tile
Similar to the Vertice class, the purpose of the Tile class is to set and communicate key member variables that make the Tile object unique. It also serves as an important part in the observer design pattern.

- Board: Only returns information about the state of the Tile object when their getter has been called
- Vertice: Communicates to the Vertice object [observer] when resources can be obtained from the Vertice object by calling the Vertice::notify via Tile::notifyObservers method

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

We designed and implemented our project to easily accommodate change. One of these implementations was the use of the observer design pattern. The observer design can allow easy change to add new subjects or observers. Say we wanted to add complexity to the game, by allowing a residence to be shared by two or more builders. This can be easily achieved by attaching an extra Builder object [observer] to the respective Vertice object [subject]. Or, say we wished to make the adjacent Tile objects to the Tile that was rolled also distribute resources. This can be done by simply making the Tile inherit from the Observer class and then making each Tile object observe their adjacent Tiles.

As mentioned earlier we effectively separated our code into different classes to represent different properties of the game, like the Builder, Edge, or Vertice class. By doing so we can easily add new behaviour to these classes. For example, say we wanted to add an AI to play for the Builder object. This can be achieved by adding member functions to make decisions on where the Builder object should build next. In addition, the Builder class was designed to own their individual Dice object. By having a separate class for the Dice, we can easily expand the class to include different types of dice. We could implement the Dice to specify the probability of each value when using a loaded dice. Overall, we can easily add new member variables to the various classes in our program to create new characteristics.

Furthermore, our design could allow for an easy implementation of a graphical interface. The Board class owns the key objects of the game: Tiles, Edges, Vertices, Builders. Therefore, it would be easy to read the values using getters from the objects and communicate them to the graphical interface.

Other aspects of our code that accommodate change is our extensive use of enumerated classes. One of the reasons we used an enumerated class was to allow for the ability to add different types of resources and building types. For example, if you wanted to add another type of residence, all you would have to do would be to add it to the enumerated class and change the upgrade function in builder to accommodate the new change.

Questions from Project Specification

1) **Question 1:** After implementing this feature, we did not use a specific design pattern to read the board from a file or set up the board from a random seed. Instead we created specific functions based on if the command line argument was specified or not. This approach allowed for a simpler and easier way to set up the board rather than implementing a design pattern.

2) **Question 2:** We decided to not implement a specific design pattern. We found that the differences in the type of die is minimal and we could easily implement the same thing quicker and in fewer lines of code. Though it has its benefits, in this case we found that a design pattern would just complicate the solution. Instead, we used an enumerated Die class that contained the two types of different dice, Fair & Loaded.

3) **Question 3:** A design pattern that we can use to allow for different game modes, a graphical display, or a different sized board would be the decorator design pattern. We can use the decorator design pattern, and depending on what options were provided, add the attachments to the various classes. This way, the main components or classes won't have to worry about the various options that can be passed in, and instead focus on the functionality of the game.

4) **Question 6:** Although we did not end up implementing this feature, a design pattern we could use to implement this type of behaviour is the factory method. In this case, our Tile objects created in the interface, but could have numerous subclasses that could make alterations to the Tile that will ultimately be made. This design pattern not only extends the functionalities and features of the Tile class, but also allows us to save resources by giving us the ability to alter existing objects as well.

5) **Question 7:** Throughout our program, we did not end up using many exceptions. Most functions, as we expected, only needed to return a boolean to indicate the validity of attempting to change the state of the game. Therefore, we avoided the extra handling of throwing exceptions throughout our code and only dealt with the simple primitive type, boolean. However, there were instances that arrived in our final code, which required the use of throwing exceptions. We used exceptions mostly in the command line arguments, which would throw errors if we could not find or open the file specified. These exceptions helped to provide use with an easy way to terminate the program if one of those errors occurred.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

Throughout the project, we only used smart pointers (both unique & shared) to manage memory. This initially took some time to get used to, but allowed us to not have to worry about freeing memory that we allocated.

Final Questions (the last two questions in this document)

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

   Developing software in teams was definitely a different experience than doing an assignment by yourself. We definitely learned the need to have clear communication to make sure everyone was on the same page in what they were doing. One of the other problems we ran into was not merging our changes often enough, which would result in huge merge conflicts that would take forever to sort through. Towards the end of the project, we merged more frequently and it definitely helped to speed up development and allowed us to resolve errors quicker.

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

   There isn't too much we feel like we do differently if we had a chance to start over. We set a plan from the beginning that left us with a good amount of time to test and debug our program. However, something that we did notice was that we often saved most of the testing for the end of the project. If we were to start over, we would have tested each piece more rigorously separately, rather than wait to test the components together at the end. One of the other things that we could have done was start a few days earlier to leave us time to implement some of the bonus features.