CS246: Final Project - Constructor Design

By: Daniyal, Giovani, Helen

Introduction:

For our final project we implemented a game called Constructor. Constructor is an alternate version of the game Settlers of Catan. The game has 4 builders (Blue, Red, Orange, Yellow) corresponding to each player. The gameplay takes place on a board consisting of tiles, vertices and edges. Builders can build roads on edges and residences on vertices based on the game rules. These vertices and edges are located at different tiles with five different types of resources attached (Brick, Energy, Glass, Heat, Wifi). Building a residence and upgrading it gives the builder building points. When a builder reaches 10 building points, the game is over, and that builder wins the game! The players can then choose to restart the game and play again.

Overview:

Our implementation of the game consists of five main classes: class Builder, Edge, Tile, Vertex, Dice and Board. The classes ensure that our code follows modularization principles. Along with these classes, we implemented multiple design patterns. These consist of the Model-View-Controller, Strategy design pattern and Observer design pattern. We chose these patterns to ensure that our code had low coupling and high cohesion. An added benefit was that our code used classes and polymorphism to avoid code repetition and made our code far more readable, which allowed our group to collaborate effectively.

The MVC design pattern was implemented to manage the data in our complex game and provide a model for the user and an interface they can interact with. This allows us to keep our controller (board.cc) and model (builder.cc, vertice.cc, tile.cc, edges.cc, dice.cc) and view (main.cc) separate. Our MVC design was invaluable because segregating the code made it easy to divide and organize our logic into our game. An added benefit was it helps to find specific portions of code quickly and allows the addition of new functionality with ease.

Class: Board

The Board class serves as the main model to connect and interact with our classes Builder, Edge, Tile, and Vertex. The board class utilizes vector arrays to dynamically store the information of all Builders, Edges, Tiles and Vertices on the Board. It also is responsible for communicating the information to our main.cc which provides an interface for the ‘view’ of our

board. It handles interactions between the different classes. For example, if a certain tile with a certain resource is called, the board is responsible for letting the builder know they got a specific resource and handles the information transfer.

Class: Builder

The Builder class serves as the main model storing the information of each builder. It keeps track of their resources, the number of points they have, their unique player identifier and the residences/roads the builder owns. Along with this, they have methods that use the information to check if the builder can do certain operations. For example, a canBuildRoad() is inside the builder class and determines if the builder has enough resources to build a road. The class also executes other commands such as steal, lose resources, move goose, and improve housing based on the player's behalf.

Class: Edge

The Edge class serves as the main model storing the information of each edge. It keeps track of whether someone owns the edge, who owns the edge and the neighbours of the edge. It uses the Observer pattern to simplify in notifying its neighbours and checking if a builder is able to build roads at certain positions.

Class: Tile

The Tile class serves as the main model storing the information of each tile. It keeps track of whether a Goose is on a tile, the neighbour checks the tile, the position of a tile and its elements. It is heavily used in the initialization of our program.

Class: Vertex

The Vertex class serves as the main model storing the information of each vertice. It keeps track of whether someone owns the vertice, which player owns the vertice and the roads corresponding to the vertice. The methods incorporate the Observer design pattern by allowing the vertex to notify its neighbours and its neighbouring roads that it has been constructed. We designed the Vertex class to build and upgrade housing. The name of the vertex is changed to the builder name and house type once a house is built.

Class: Dice

The Dice class serves as the main class for the board to get a random roll (number from 2-12). We chose to implement the Strategy Design Pattern to implement the class. The children classes loaded dice and fair dice generate a random number for the game to play, which means

they share the same purpose. Thus by creating an abstract class Dice which uses polymorphism and has child classes loaded dice and fair Dice, not only are we able to switch between loaded and fair dice at run-time, but we can also reuse or code through polymorphism. Thus, we have implemented this class along with the use of the Strategy Design Pattern.

Other Important Fields:

seed: Assists in the randomizing of the random number generator which will be used in Dice, loseResource() and steal() methods. Used with the "-seed" command-line option.

load/board: The boolean variable is true when a game has been loaded/board provided; false otherwise. Calls the loadFile(file_name) and loadBoard(file_name) functions. The "-load" and "-board" command-line options are used.

initialization: Utilized to initialize the board given the command line arguments with variables assigned and objects attached to particular vectors. It is also responsible for assigning the initial values to related fields.

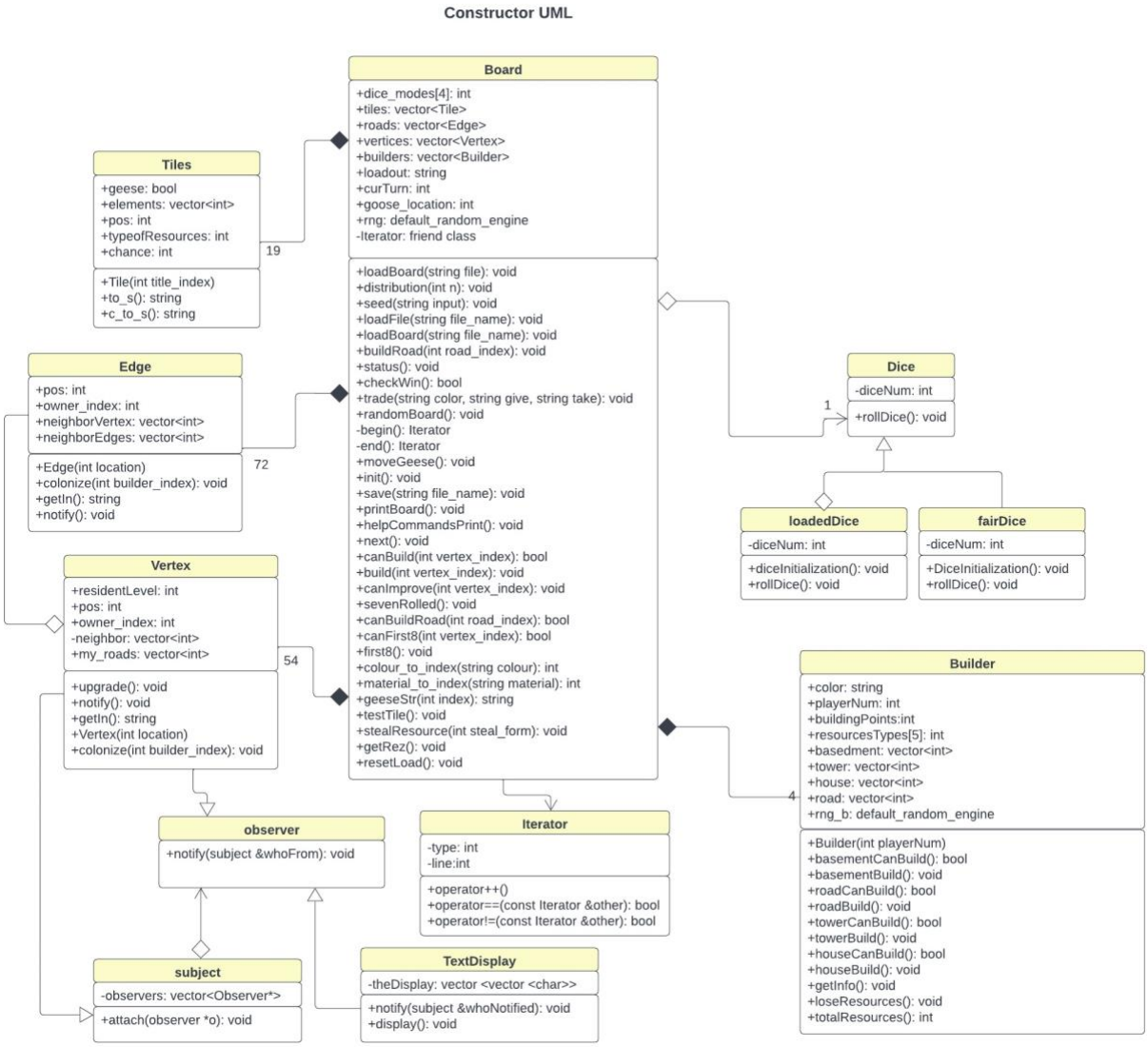
save: This method saves the board information to the assigned file. The user can then load the file and continue playing the game from its current state. Additionally, if at any point the game receives an EOF signal mid-game, the game ends with no winner and the data is saved to backup.sv.

random: for functions such as steal() and loseResources() along with the dice functionality, we utilized a random number generator which uses a seed along with uniform_distribution functions to get random numbers to make the game not deterministic.

dice: Dice methods were designed using a Strategy design pattern. It has two possible implementations, loaded dice, or fair dice. In a loaded die, the user can manually input the number they want to roll. While in a fair die, the number is randomized using a random number generator.

vectors of tiles, builders, edges and vertices: These vectors of particular objects are to store all the tiles, builders, roads and vertices in a sequence of proper indexes.

Updated UML:



Design:

Model-View-Controller: Given that we had to provide a model for the user and an interface they can interact on, we decided to implement an MVC design pattern. This allows us to keep our controller (board.cc) and model (builder.cc, vertice.cc, tile.cc, edges.cc) and view (main.cc) separate. Our MVC design was invaluable because by segregating the code, it was easy to divide and organize our logic into our game. An added benefit was it helps to find specific portions of code quickly and allows the addition of new functionality with ease.

Modularized Design: While creating the project, we decided to organize our code into separate modules. This ensures that our code follows the principle of high cohesion and low coupling. An added benefit was that our code was not repeated and more readable which allowed our group to collaborate effectively. Additionally, our Builder, Edges, Tile and Vertex classes only had access to the data sent from our controller (board.h). They do not have to interact or depend upon other classes or objects, reducing coupling and complexity of the system.

Observer Design Pattern: We used the Observer design pattern in both the Vertex and Edge class. All vertices and edges are both observers and subjects. When a residence is built on a vertex or a road is built, its neighbouring vertices and edges are notified and the changes are made accordingly. The observer design pattern is needed to communicate changes to its adjacent neighbours in order to alert them to keep track of which builders can build where.

Strategy Design Pattern: We used the strategy design pattern in the Dice class to implement fair dice and loaded dice. We used this design pattern because loaded dice and fair dice have related behaviours that ensure high-degree of cohesion. In addition, it's easy to switch between the two modes at run-time because of polymorphism in the interfaces. It also minimizes coupling since the client (Board class) is coupled only to an abstract class (Dice class).

Resilience to Change:

Adding New Resources to the Game:
Since we incorporated our resources as a vector of integers, we have the flexibility to add multiple new resource types. These changes can be easily added because our system has low coupling and our loops that access the resources of each builder run for the number of resources in the game which is by default 5 but can be altered depending on how many more resources we would like to add. For example, we could add a new resource SAND which could easily be kept track of by our existing code. It would be simple to refactor the initialization of our initialization code to allow for this new feature.

Adding New Residence Types to the Game:
Since we incorporated our residence types as a vector of integers, we have the flexibility to add multiple new residence types. These changes can be easily added because our system has low coupling and our loops that access the residences of each builder run for the number of total residence in the game which is by default 3 but can be altered depending on how many more residence types we would like to add. For example, we could add a new Skyscraper which would need to be built by upgrading from a Tower. It would be simple to refactor the initialization of our initialization code to allow for this new feature.

Adding New Builders to the Game:

Since we incorporated our Builders as a vector of Builders. Which are objects, any newly added builder would still follow the same rules that all builders must follow. Since the Builder class is the blueprints of the object Builder, all we would need to do is add another builder in the initialization and define the next builder as a new colour. The functionality would still work because all builders follow the same rules.

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 use this design pattern? Why or why not?

We considered using the Abstract Factory design pattern to allow us to produce families of related objects without specifying their concrete classes. We thought we needed this pattern because we did not know the resources used until runtime which is why we had to find a way to avoid specifying their concrete class. We planned to do this by first explicitly declaring interfaces for each distinct resource (e.g., brick). From there, we can have variants of the resources that follow those interfaces. This would help solve our problem of not knowing where the resources will be until runtime. However, even though the pattern reduces coupling between controller and the way of setting up resources, we chose not to implement this design pattern. We wanted to future-proof our code and adding this feature would force us to create a new class anytime we added a new feature to randomly set up the resources. Instead we had implemented different functions to initialize our skeleton board based on the arguments provided. For example, if we needed to load a file, then we would call loadBoard() to initialize our board through our controller.

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?

We chose to implement the Strategy Design Pattern to implement the feature. We know that loaded dice and fair dice are to generate a random number for the game to play, which means they share the same purpose. Thus by creating an abstract class Dice which uses polymorphism and has child classes loaded dice and fair Dice, not only are we able to switch between loaded and fair dice at run-time but we can also reuse or code through polymorphism. Strategy design

pattern defines a family of dice, i.e., Dice, loaded Dice and Fair Dice, which can not only keep every Dice's encapsulation but also make the loaded and fair dice interchangeable.

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 numbers of players). What design pattern would you consider using for all of these ideas?

Suppose we want to attach new behaviour for different game modes. We can do this using the Decorator design pattern. We would start by building an abstract base class with a base board class and from there we can attach new behaviours to the game by placing these objects inside special wrapper objects that contain the behaviours. Polymorphism allows us to be flexible and change the rules while reusing our current code. By using the decorator pattern, a base board class can be extended by inheritance to suit any newly added game modes. Not only would this allows us to implement new features easily, but it would also ensure our coupling remains low. An added benefit is that we use polymorphism in the design pattern to avoid rewriting unnecessary code.

Question 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?

We want to dynamically change our computer players' strategy during runtime. The Visitor pattern allows us to place the new behavior into a separate class called visitor, instead of trying to integrate it into existing classes. The original object which performed the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object. This gives us the ability to separate the different computer strategies from the objects on which they operate.

Question 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?

We want to attach new behaviour for our tiles during runtime. We can do this using the Decorator design pattern. We would start by building an abstract base class with a base board class that defines the default behaviours. From there we can attach new behaviors to the game by placing these objects inside special wrapper objects that contain the behaviors. By using the

decorator pattern, a base board class can be extended by inheritance to suit any newly added game modes. Not only would this allow us to implement new features easily, but it would also ensure our coupling remains low. An added benefit is that we use polymorphism in the design pattern to avoid rewriting unnecessary code.

Extra Credit Features:

No Use of Raw Pointers (New/Delete-Free):

The project is completed without the use of any raw pointers or any new and delete heap allocation. Additionally, our program has no memory leaks. Adding this feature is especially difficult because to run our board we need to manage lots of data and pass them through different class interfaces and modify values. Examples of this include a builder's resources, a vertex's owner and levels. We decided to solve this problem by using the include the vector module instead to dynamically handle our memory and deal with any heap allocated memory. By doing so, we followed the principals of high cohesion and low coupling by letting a separate module deal with any dynamic heap allocated memory.

Flexible Interface (Case-Insensitive Commands and Handles Invalid Input/EOF):

We have implemented our user interface to allow a user to use the interface by letting them have commands in uppercase, lowercase and a mix of both. Additionally, we have checkers everywhere in our code which checks if the input is invalid and alerts the user that they cannot input that and they get to immediately retry a different input. If at any point the program receives a EOF signal, the game is automatically saved to a backup file, which can then be loaded and played on again.

Final Questions:

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?

On the technical side we learnt the value in using a version control platform. Prior to this project, none of us had ever collaborated on coding projects before, so we had never used Github. At first we struggled with sharing and updating our code with one another. Eventually, we tried using Github and from there we were able to collaborate and push/pull changes far more easily. This taught us the importance of being able to use a software development and

version control platform to work on a large coding project with others. Another lesson we learnt was the importance of communication and clearly defining tasks we needed to do. By organizing our tasks and having certain individuals in charge of certain modules, we were able to follow the principals of high cohesion and low coupling. This allowed us to work separately and at the same time be able to work together on the larger goal. By staying in constant communication we ensured the code we were writing was useful for others.

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

If we started over we would definitely have prioritized the load functionality first. We underestimated how time consuming the specific feature would be to both test and implement the command line argument functionality. One other thing we realized was that designing the UML and design.pdf would take extra time that we should have accounted for in our plan for DD1. Since we struggled with managing our time and setting reasonable expectations, we did have to spend a stressful final 2 days debugging, creating the updated UML and the design.pdf which could have been done earlier. Another lesson we learnt was that we should be debugging and testing during our coding process rather than leaving that for the end. By conducting unit tests, we could have saved a lot of time individually testing each module rather than finding which module the bug was in.