F46WU, A7ZHU, R5CUI – WATAN PLAN OF ATTACK REV2.0

INTRODUCTION

The game we implemented, Watan, is a simplified version of the popular board game, Catan. It consists of a hexagon grid, with 6 vertices and edges on each hexagonal tile. Players can build on each of these vertices and edges (with various conditions) in order to earn resources, with the end goal of earning reaching 10 points.

OVERVIEW

In Watan, the greatest obstacle by far is constructing and setting up the grid. The nature of hexagons makes it much more difficult to set up as opposed to something like a square grid.

In our implementation, the **Board** class is in charge of this setup. This is where all the **Tiles**, **Criterion** (vertices), and **Goals** (edges). It will create all the appropriate objects as well as attaching their neighbours, which is necessary for implementing the *Observer* pattern. Naturally, we used the observer pattern for many our classes (such as Tiles), since they would have to frequently update other objects whenever their state changes.

After the Board class sets up the grid, the **TextDisplay** class translates this information into a visual display.

Naturally, in order to simulate the real game, we created **Player** classes which hold information such as number of resources, number of points, etc. The **Controller** class is the class that parses input and changes the actual state of the board/model.

DESIGN

As previously mentioned, the most difficult part of this assignment was setting up the board. Calculations and algorithms are much more difficult when performed on hexagons as opposed to squares / rectangles.

Originally, we intended to calculate things like object adjacency (which is necessary for building structures within the game) by finding algorithms, but we discovered that the algorithms were unintuitive and difficult to implement. To circumvent this, we hard-coded things such as adjacency relationships and the base template for the TextDisplay into text files, and read from them in our actual program. For example, we convert the TextDisplay (text file) into a vector of strings, and augment it that way. This saves a large chunk of time.

The actual nature/process of the game (rolling Dice -> getting Resources depending on number rolled -> adding points) was easily solved using the Observer pattern. For the most part, it was a convenient chain of events.

One major road-block was figuring out how to initiate the game from a file vs initiating the game randomly. This was solved using the Factory Method Pattern; we created a base Board class and two derived classes which overrode the initiation function in the Board base class.

RESILIENCE TO CHANGE

Because we implemented our program using the MVC pattern, our Model, View, and Controller are clearly separated. The Model refers to the Board, Criterion, Goal, etc. which represent the actual model / state of the game. The Controller, the class responsible for parsing input and performing moves, changes the state of the model. Finally, the View is updated by the Model and changes the display correspondingly. Due to this separation, we can change the View, add a Graphical display, add functionality to the controller, etc.

EXTRA CREDIT FEATURES

- We did not use new or delete anywhere in our program.
- Implementation of "Development Cards" from actual Catan game (if time permits)

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?

A design pattern that could be used to implement this feature is the Factory Method Pattern. By having an abstract class with a virtual function that performs differently depending on if the Board is randomly generated or loaded from a file, we can easily decide which type of Board to create since they share many of the same fields.

In our game, we implemented the Factory Pattern by creating a base class, Board, and two derived classes, savedBoard and randomBoard. The Board class contains all the essential public / private fields, but has a pure virtual init() function that is overridden by savedBoard and randomBoard. In randomBoard's init(), the Tile's values and resources would be generated at random, while savedBoard's tiles were blank, since they would be loaded in later.

We used this design pattern since at the beginning of the game we must decide whether to randomly generate a board or load-from-save, and the Factory method allows us to easily choose which derived class to create. It also allows reduces dependency and abstraction, allowing us to create other Board types in the future by just overriding the pure virtual function in Board.

2. You must be able to switch between loaded and unloaded 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 this feature we could (and did) use the "Strategy pattern" because it easily allows us to switch between loaded and regular dice at run-time.

For example, our implementation has an abstract class named "Dice", with a pure virtual function called "rollDice." The Dice class has two sub-classes that inherit from it, named "loadedDice" and "fairDice." As their names suggest, loadedDice.rollDice will output a specific, non-random number while fairDice.rollDice will output a random number.

Alternatively, we could have implemented this feature using several if/else cases, but using the Strategy pattern increases readability / generality and makes it easier to change / add new "strategies" or types of dice in this case because it doesn't involve rewriting the if/else block.

3. We have defined the game of Watan to have a specific board layout and size. Suppose we wanted to have different game modes (e.g. square tiles, graphical display, different sized board for a different numbers of players). What design pattern would you consider using for all of these ideas?

The design pattern that would best cover all these different game modes would be the MVC (Model View Controller) design pattern. In the MVC design pattern, the classes that are in charge of rendering and displaying the model doesn't need to know *how* the model is being implemented, and the controller will change the state of the model. All three components are separate and changing one will not greatly affect the other.

We can combine this with the pImpl idiom, which contains a pointer to the implementation. We can change the implementation without having to refactor existing code. For example, we could point to an implementation with a different sized grid, or one with a different shape. This combination provides us with the flexibility we need to implement these ideas.

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 can use the Decorator design pattern to implement this feature. We could start with an abstract class named "Tile" and a sub-class named "basicTile" which simply grants one resource. We can then add decorators, such as "doubleTile" which gives two resources rather than one, or "multiTile" which gives multiple resources.

An easy way to design this would be as follows: In the abstract class, have virtual functions with integer return values for every resource such as

numCaffeine, numLabs, etc. The base class would return 1 for one of these functions. Decorators would change the values of these functions.

Ex. int numCaff() const override { return 0; }

In Decorator: int numCaff() const override { return 1; }

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.

We used an exception to catch EOF or when a player has won, because those scenarios signal the end of the game. If either situation occurs, we throw a string, "Game Over!" which is caught + thrown by numerous functions until it reaches main, where we can save and end the game (in case of EOF) or end/restart (in case of winning move).

This process of catching and throwing until it reaches main allows us to easily deal with end-of-game scenarios.

Initially, we did not think of using exception handling to change the actual state of the game. It actually proved to be very useful.

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?

This project taught us the importance of having a proper repository / versioning tool where we can share and update our progress. We ended up using Google Docs due to lack of experience with other services, and ran into scenarios where code would get overwritten, and it was difficult to keep track of what files were up-to-date. Having a more efficient manner of sharing code would have been very beneficial.

It also taught us the importance of splitting up code properly. Initially we thought that we could assign one class to each person, but we quickly learned that many of our classes relied on functions on other classes, so we had to work together in groups.

It also stressed the fact that you should start assignments early ☺

What would you have done differently if you had a chance to start over?

- We would like to reduce the number of times we pipe to cout.
- Reduce the size of Board.cc (upwards of 700 lines)
- Reduce the dependency on loading from .txt files
- Start the project earlier

In our currently implementation of the game, the I/O wasn't handled the best way we could think of. If we had the chance to start over, we would have implemented a class called GameMessage to collect all the game messages (such as, "invalid command") from each class that generates them, and to be initiated with any types of istream/ostream as opposed to limit them to std::cin/std::cout. This way not only would it be easier to enforce MVC but also help collcted imporntant usage statistics. However, due to time contrain, we implemented many direct std::cout and std::cin in our classes, which undermined our MVC design.