STUDENTS OF WATAN - PLAN OF ATTACK

Francis Wu (f46wu : 20639597)
Ang Zhu (a7zhu : 20479729)
Reginald Cui (r5cui : 20625355)

Order of Completion (List of Priorities from Highest to Greatest)

First: The highest priority is making sure the board can be loaded and all the criterion and goals are properly initialized, and criterion are built according to player input at the very beginning of the game. In a nutshell, making sure the board can be played on and displayed.

Second: After that, we will work on the dice functionality and making sure that the Tiles properly distribute resources based on the dice rolls.

Third: Make sure players are able to purchase criterion / goals and the game ends at 10 points.

Fouth: Work on saving and loading the game. Ensure that the game can load a given save point, and the game properly saves all the data required.

Fifth: Geese implementation.

Complete List of Classes that need to be implemented:

Board, randomBoard, savedBoard
Will be done by: Frank
Estimated completion date: November 29th

TextDisplay
Will be done by: Frank
Estimated completion date: November 29th

Tile
Will be done by: Reginald
Estimated completion date: November 29th

Criterion
Will be done by: Ang
Estimated completion date: November 29th

Goal
Will be done by: Ang
Estimated completion date: November 29th

Player
Will be done by: Frank
Estimated completion date: December 1st

Dice, LoadedDice, FairDice
Will be done by: Reginald
Estimated completion date: December 1st

Geese
Will be done by: Ang
Estimated completion date: December 1st

SubTypes
Will be done by: Frank
Estimated completion date: December 1st

Subject
Will be done by: Reginald
Estimated completion date: December 1st

Observer
Will be done by: Ang
Estimated completion date: December 1st

Testing phase: December 2nd – December 5th

Meeting

We will meet every two days to share progress and code. We will attend as many tutorial sessions as we can.

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 can implement the Factory method pattern to decide whether to load from a save file or to randomly generate the board. We can have an abstract class with a virtual function that initiates the board, and two sub-classes that override the init function. One sub-class would init from a save file while one sub-class would initialize normally and randomly.

We used this design pattern because we can encapsulate the board instantiation process based on whether we need to load the board from a file or from scratch.

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" which consists of an abstract class with a virtual function, and two sub-classes that override this function. The behavior of the function depends on the class using it.

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

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 did not use any exceptions in our current iteration of our project.

An example of a place it would make sense to use an exception is for our function purchaseCriterion which takes an integer, and attempts to purchase a criterion at that position (int). We can use an exception to catch when the user tries to call purchaseCriterion using an integer that is not represented as a position on the board. For example, in Watan the last position on the board is 53, so if the user tries to purchaseCriterion(54), we would throw an exception.

We can generalize this for any dynamically sized board because we stored Criterion in a vector, so if we are unable to access the vector at that index, then it is invalid input and we throw an exception.