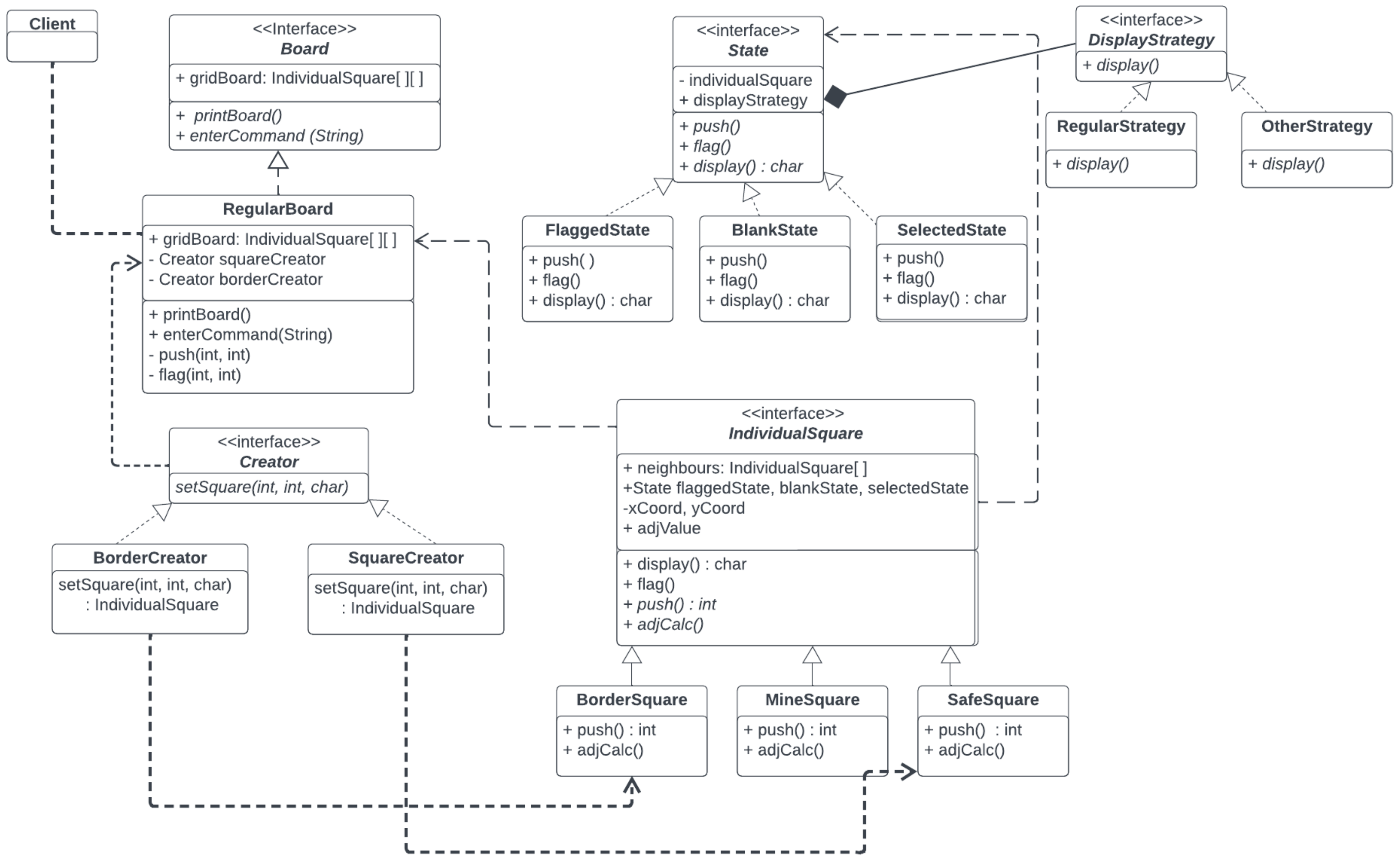
Paul Seward CS-3620 Assignment 3 April 6th, 2023
142012



Paul Seward CS-3620 Assignment 3 April 6th, 2023
142012

1. Design Explained:

Upon brainstorming ideas on how to create a well thought out design for Minesweeper that follows OOD and SOLID principles, we started with a first round of a naïve implementation without any such principle. This type of design would be a hard coded grid of individual squares of fixed size and fixed functionality. No new games could be generated from this. Following this approach, my design aims to stray as far as possible from this fixed functionality approach.

In this design, the RegularBoard class will be the regular type of square minesweeper board that constructs a board from a given size, and percent requested of squares to be created as mines. This uses Java's random number generator using the percent threshold. This class implements a Board interface in case there needs to be other types of boards of different shapes, islands, that use future different types of square factories creating different types of squares. This specific RegularBoard will take string commands in the form x8/y12/push [or flag]. This class then dissects the command and provides the proper implementation for command requested on the specific square. In the case of explosions, this class will allow the user to retry if the user replies a certain keyword/button.

This board will use two factories, SquareCreator and BorderCreator, to create minesquares, bordersquares and safesquares. The factories implement a creator class for future factories to extend from. There currently is only one method for creating a square, but other methods can be added to the interface with a default implementation.

The individual square interface provides the required functionality of each square. The three current types of squares are, MineSquares, SafeSquares, and BorderSquares. This design includes the use of BorderSquares to prevent any chain reactions going past the constraints of the size of the array and preventing the use of significant amounts of if then else hard code. Upon creation of the board, first each square is created, and then the adjacency count of mines is then calculated.

In this design, the job of adjacency counting is placed on each individual square. Each square holds an array of references to its neighbouring squares. Because of this choice, each square can count the number of adjacent mines. Of course, since BorderSquares and MineSquares are different types of squares, this adjCount() method can have unique or empty implementations of the adjCount(). While it is possible and initially intuitive to defer the task of counting adjacent mines to the underlying RegularBoard class, each square needs to hold references to its neighbouring squares because selecting an empty safe square is suppose to cause a chain reaction of neighbouring empty safe squares to be selected.

Each square also holds a state references which follows the State Design Pattern. The individual squares will keep a reference of each possible state and a reference variable, state, which keeps track of which current state of the object. The individual square interface also provides implementation for getters and setters to use in the State Design Pattern. The individual squares methods will allow the state to provide the proper implementation of the needed behavior but can intervene or add functionality in certain circumstances such as keeping track of number of safe pushes to monitor when the game is won/lost.

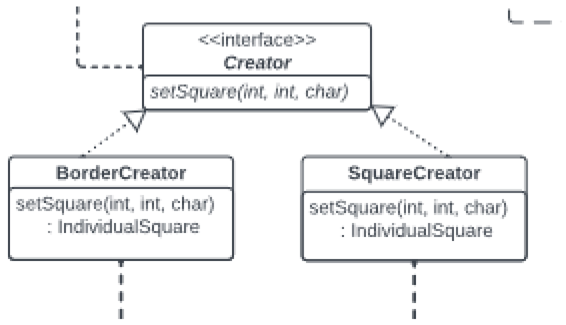
Paul Seward CS-3620 Assignment 3 April 6th, 2023
142012

There are three states in this design: FlaggedState, BlankState and SelectedState. The state interface holds a reference to the individual square and uses its own getters and setters to rotate between classes. The three methods are push(), flag(), display() to display what the square should display on the visual board.

Finally, this state interface also uses the Strategy Design Principle to hold a strategy inside its state. Now, how a square is displayed is not fixed and is open for extension. The display method will call on its strategy reference variable to use the DisplayStrategy's implementation.

2. Design Principles Followed:

I. Factory Method Design Pattern:

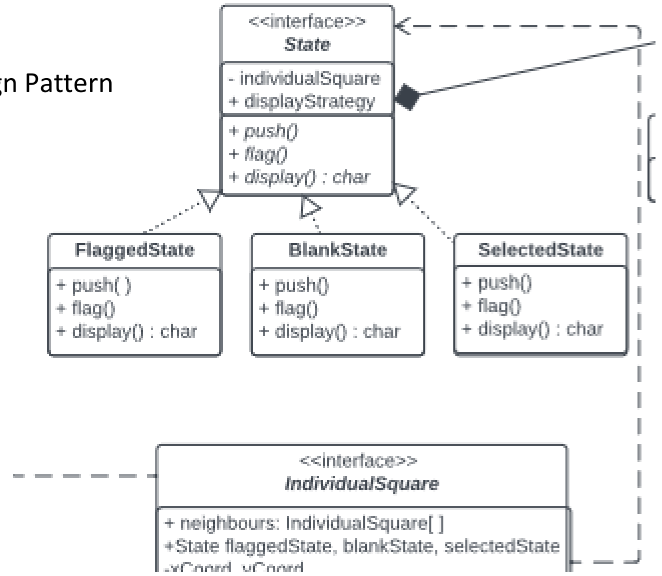


<<interface>>
Creator
setSquare(int, int, char)
BorderCreator
setSquare(int, int, char) : IndividualSquare
SquareCreator
setSquare(int, int, char) : IndividualSquare

We chose to incorporate this design pattern because we required the use of inheritance to decide object instantiation. As other tweaks to this came come, control can be kept inside the individual factories with additional methods that adjust how the object is created. Moreover, since there are multiple square classes to create, we can defer the creation of these types of square to the subclass factories and prevent significant if then else hardcoded logic to decide which class to instantiate. The use of this pattern allows for extended modification as more square types are included and more factories can be easily added following the Open/Closed Principle.

A drawback of including this factory method design principle is additional classes to implement a simple behavior. However, the use of the pattern follows the Dependency Inversion principle to decouple the board and how it creates individual squares. Following that, this pattern also supports the Single Responsibility Principle since the game board's role is storing the objects, not deciding how each is created.

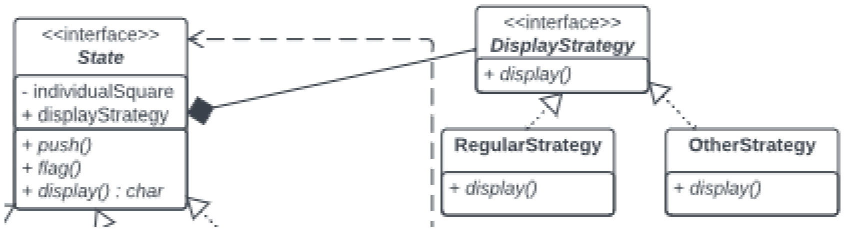
II. State Design Pattern



The next pattern used is the State design pattern. This pattern was chosen to be implemented since we desired to encapsulate state based behaviors such as flag, push, and display, and delegate the specifics of the behaviour to the current state. Using this patter, allows the board of individual squares to appear to change a squares state dynamically when certain operations are incurred. As an object goes from flagged, blank, and selected, each of the operations should alter its behaviour accordingly. Again, this allows a dynamic design that scales with larger boards and prevents any hardcoded logic.

The drawbacks of the state pattern are similar to factory since they both create extra classes and increased complexity. However, the higher complexity is required to allow individual squares to have varying behavior when they are flagged, selected, or blank. A significant drawback to this pattern is tight coupling between context and state classes. Adding additional states become problematic since how an object changes states is defined in each state. A single state change would likely involve violation of the Open/Closed principle and could even involve modifying multiple classes for a single change.

III. Strategy Design Pattern



Finally, the last design we included is the Strategy Pattern. While there is overlap with this pattern and State Pattern, the specific differences of strategy allow for significantly advantageous functionality in how an object is displayed. Here, strategy is used as an alternative to subclasses of a game board and how they are displayed. Instead, the modification of displaying squares is kept here, using an polymorphism of a reference variable to a DisplayStrategy that is called inside the display method. The strategy pattern's role in this design to handle only a single specific task and provide the underlying implementation.

This pattern will also provide a setter method so the behavior can be set/changed at runtime. People with visual impairments can choose a display strategy the verbalizes the square they opened, can use capital letters, allow the border spaces to display a visual border of '#' instead of blank padding, and much more. The customization of how a board is displayed is enabled through the use of this pattern while allowing further display implementations without requiring any other classes to be modified.

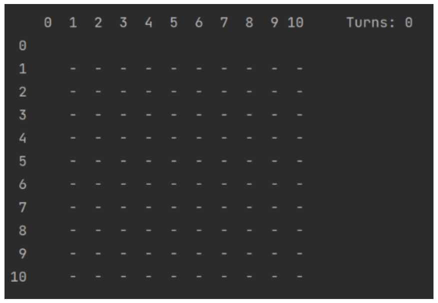
A drawback of strategy pattern is that the client will need to be aware of the display strategies. Although this may violate some encapsulation principles which prioritize hiding the implementation from the user, the advantages of allowing the user to customize their game board display significantly outweigh this slight negative.

3. Additional Patterns:

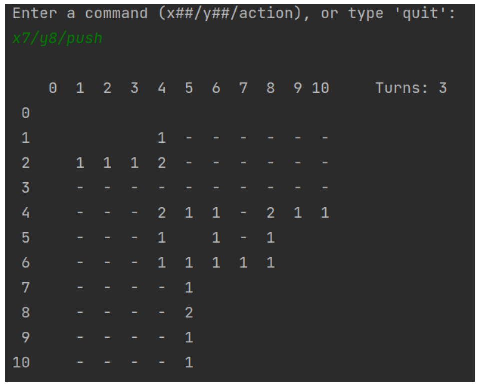
IV. Façade Design Pattern.

One final design decision we included was the use of the Façade Design Pattern. Since this is a simple game, we wanted to provide a simple interface for the user. In the board game class, a single method, enterCommand, is used for the user to interact with the game. From this, the game's complex subsystem implementation details are hidden and decoupled from the user. This uses the Principle of Least Knowledge.

Here is a screenshot of the game's implementation. I also included the IntelliJ project of my working minesweeper game. I understand this was not required but using my UML diagram of well designed infrastructure, coding the implementation came very quickly.



```
    0  1  2  3  4  5  6  7  8  9 10     Turns: 0
 0
 1     -  -  -  -  -  -  -  -  -  -
 2     -  -  -  -  -  -  -  -  -  -
 3     -  -  -  -  -  -  -  -  -  -
 4     -  -  -  -  -  -  -  -  -  -
 5     -  -  -  -  -  -  -  -  -  -
 6     -  -  -  -  -  -  -  -  -  -
 7     -  -  -  -  -  -  -  -  -  -
 8     -  -  -  -  -  -  -  -  -  -
 9     -  -  -  -  -  -  -  -  -  -
10     -  -  -  -  -  -  -  -  -  -
```



```
Enter a command (x##/y##/action), or type 'quit':
x7/y8/push

    0  1  2  3  4  5  6  7  8  9 10     Turns: 3
 0
 1              1  -  -  -  -  -  -
 2     1  1  1  2  -  -  -  -  -  -
 3     -  -  -  -  -  -  -  -  -  -
 4     -  -  -  2  1  1  -  2  1  1
 5     -  -  -  1     1  -  1
 6     -  -  -  1  1  1  1  1
 7     -  -  -  -  1
 8     -  -  -  -  2
 9     -  -  -  -  1
10     -  -  -  -  1
```