Laxman Elisha

Design Document

Overview

The centre piece of the project is the Board class. This class is where most of the game interactions happen. The Board class will have pointers to a level, text display, window, and the squares on the board, so it can make the necessary changes to those objects.

When the board is created, it needs to initialize the board with squares. This initialization depends on what level you are currently on, so the factory design pattern was used to implement this. Each level class has an “init board” method that will initialize the board in a different way. Depending on what level you are on, the board object will call a different “init_board” method. The same design was used for the generation of new squares. Each level class has a “generate square” method that will generate a square in their own way.

When a swap is made, the squares that were swapped will notify the text display and the graphical window of the changes. The board will calculate how many points were made by looking for matches on the board, and calculate the score based on the matches made. The board will then update the scoreboard on the changes to the score.

The board will go through the board looking for matches, so it can turn the squares into holes. It will also create special squares if a certain type of match is made. While it is creating the holes, it will activate the effects of the special squares, such as destroying a whole row or column. These effects will be included in the score calculation. After the holes have been made, the board will fill the holes by dropping all the squares above it downwards, and then generating a new square at the top.

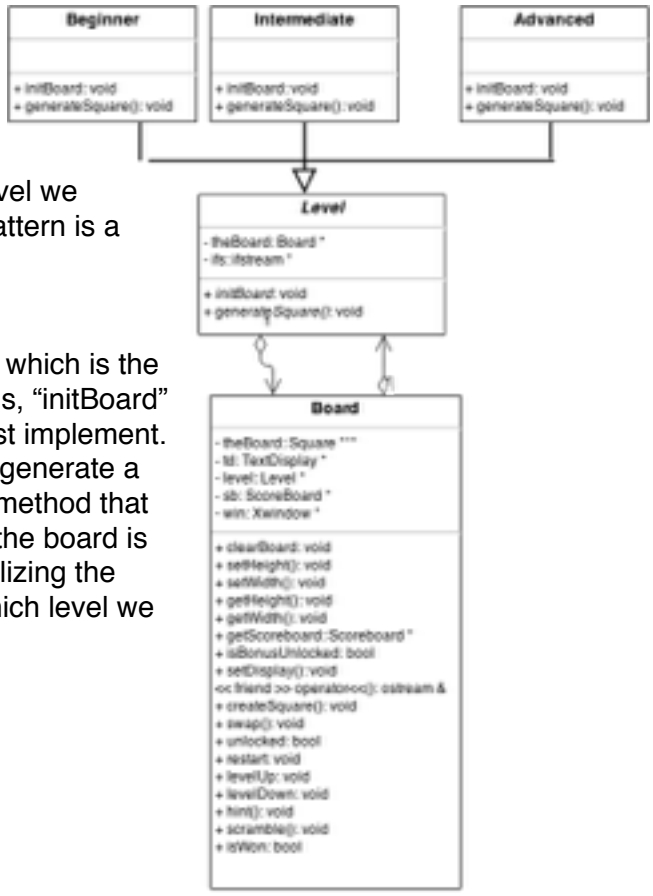
Finally, the board will update the scoreboard on the changes to the score. The scoreboard will check if the level is complete by checking the scores and the level requirements of the current level. If the requirements, are met, the scoreboard will tell the board to level up. If the requirements are not met, the board will check for any matches, caused by the generation of new squares, if it finds a match, the process of creating the holes and updating the score will repeat.

Laxman Elisha

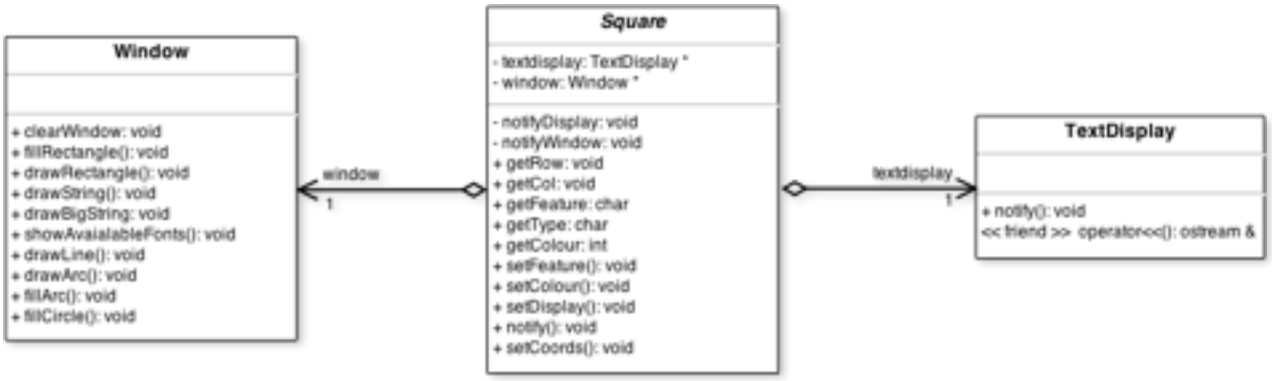
Design Patterns

Factory Design Pattern

Why did we choose this pattern?
We needed to a way to initialize the board and generate squares, but the problem is that those two things depend on which level we are on. We needed a way to initialize the board and generate squares based on which level we are on, so we decided that the factory design pattern is a good choice.



How does it work?
The Board class has a pointer to the level class which is the factory. The level class has pure virtual methods, “initBoard” and “generateSquare” which its subclasses must implement. When the board needs to initialize the board or generate a square, it will ask the level class to do so. The method that will run will depend on which subclass of Level the board is holding at runtime. This way the effects of initializing the board or generating a square will depend on which level we are on.



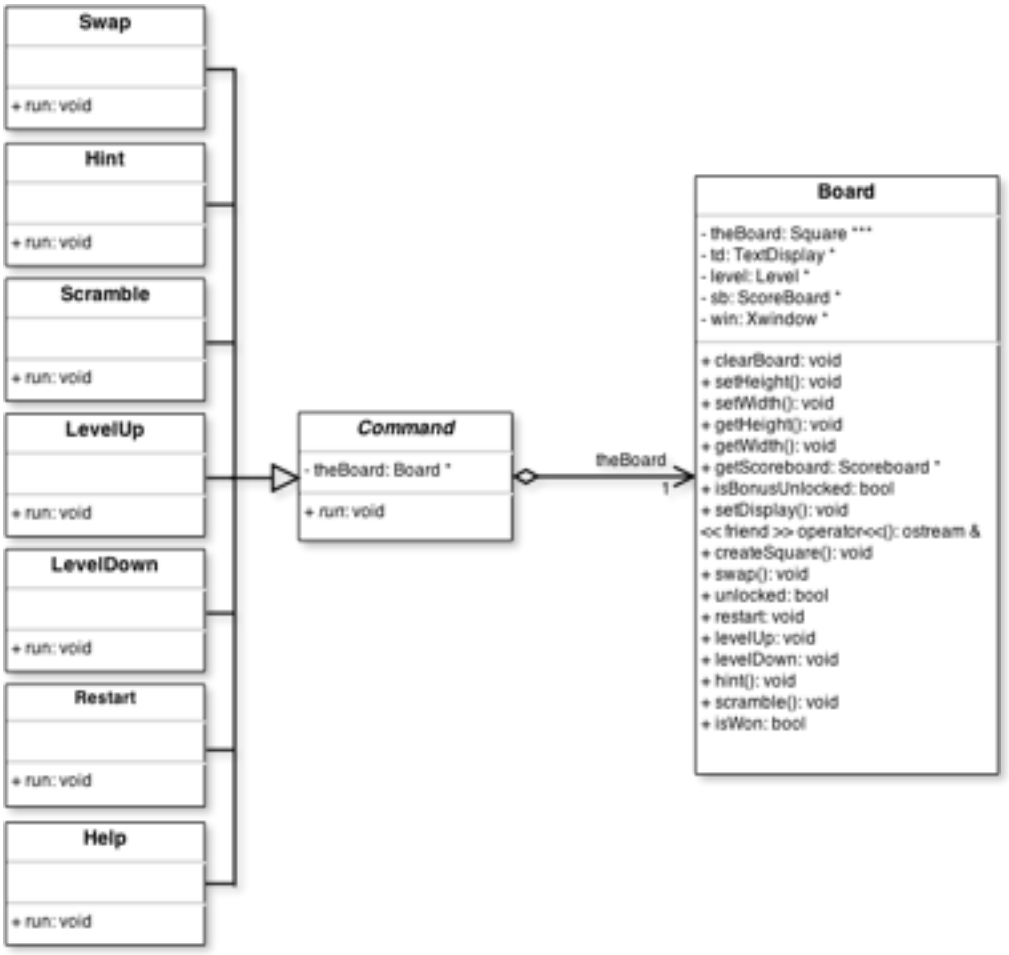
Observer Pattern

Why did we choose this pattern?
This pattern is chosen because it allows us to specify one-to-many dependencies between a Square object and its dependents (Window and TextDisplay objects). When a Square object changes state, the Window and TextDisplay objects are notified and updated automatically.

How does it work?
Each Square object is a subject with two observers of Window and TextDisplay objects. When a Square object changes state, the Window object gets notified and changes it graphical display accordingly (i.e.: changes the colour corresponding to a Square). Also, the TextDisplay object gets notified and changes its text display accordingly (i.e.: changes the encoding corresponding to a Square).

Laxman Elisha

Command Pattern



Why did we choose this pattern?
One of the major reasons we choose to use this pattern is that it allows us to encapsulate a command in an object and thus provides a convenient way of adding and removing commands as needed by the client. Another huge advantage of using this pattern is that it allows us to fully control what commands can and can't be called by the client.

How does it work?
The Board object is the receiver of the Command. When the client needs to invoke a specific Command to the Board object, the client creates the appropriate Command object and call its run method. The Board object will then perform the operation(s) required based on the run method.

Extra Features

ADDITIONAL COMMAND-LINE OPTIONS

- **-unlock** unlocks the following bonus features in the game:
  - Welcome & Exit Screens
  - Levels 1 & 2
    - Addition of light blue and yellow squares
    - Randomly generated grid size that ranges from 8x8 to 12x12 inclusively

- **-seed time** sets the random number generator's seed to time. If you don't set the seed, you always get the same random sequence every time you run the program

ADDITIONAL COMMANDS

- **help** provides the following features:
  - List of the available game commands
  - Key informing the player of how the encoding in the text display corresponds to the shapes and colours used in the graphical display
  - Objective for the completion of the current level which the player is on

Questions

Q: How could you design your system to make sure that only these kinds of squares can be generated?

A: You could make the Square class abstract, so that nobody can create their own generic square object. They would be forced to create one of the subclasses of the Square class. Each of these subclasses of Square will have their type predetermined, and immutable. This way, you can be sure that if a square is generated, it will be of one of the predetermined types.

Q: How could you design your system to allow quick addition of new squares with different abilities without major changes to existing code?

We have defined all our fields and methods in our Square abstract base class. In our derived classes for each of the different type of squares, we have defined the constructor and destructor for that particular type of square. If one wants to add a new type of square with a different ability into the game, they would have to define a new derived class with a destructor and an appropriate constructor to set the type field to a character representing that new type of square.

Q: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

We have defined a Level abstract base class and three different derived classes for Beginner, Intermediate, and Advance respectively. If one wants to introduce a new level into the game, they would have to modify the levelUp method in the Board class slightly and define a new derived class, which inherits from Level, with a method to initialize and regenerate squares for the grid. They would then have to recompile the Board class and the new derived class inherited from Level.

Q: How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which means adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename swap s)?

A: Initially, the plan was to have a Command class that holds a map from a string to a function pointer. The problem was that you cannot have a function pointer that holds any function. A function pointer can only hold functions with a specific argument type. The plan was changed to hold the command names in string variables. When a command needs to be renamed, all you have to do is change the string variable to the new command name. The string variables are also stored in a vector for easy searching. In order to add a new command, you would have to create a new string variable for the command and add it to the vector.

Laxman Elisha

Final Questions

Q: What lessons did this project teach you about developing software in teams?

A: The most important lesson that this project taught us about developing software in teams is definitely accurate and clear communication within the team. For example, when we meet up to discuss how the modules that we have written individually work and function, we are able to verify that the function and method calls between each module work as intended in a short amount of time. Additionally, we can plan the order in which we are going to write the modules and the time allocated for writing each module. During the process of developing the game, we found that we have different ideas a lot of the time. Through communication, we are able to work out those differences and combine the best of our ideas.

Moreover, we have learnt about goal setting and planning within the context of developing software in a team. It was indeed helpful to plan ahead ,by setting project milestones and estimate timeframe for each stage, before we jump right into writing code for the game. During the process of developing the game, we found it easier to modify our initial plan as necessary since we have something to work with.

Q: What would you have done differently, if you had the chance to start over?

A: I would have put more time into planning out the project. For instance, our UML diagram right now is very different from when we just started. This is because there were a lot of things that we didn't think about back then. A lot of new methods needed to be created because the task was complicated. For example, the hint method needs to find a swap that would result in a match. This required writing a few different methods to accomplish the task.

I would have written some test cases out at the beginning. We did not write test cases before we wrote the code, and this ended up causing some trouble. Things that initially worked stopped working after new code was added. We would have been able to find these issues sooner if we had written test cases.

UML Diagram For A5: SquareSwapper5000