Dongwoo Son (20420487) Haejung Choi (20429107)

SquareSwapper5000 (ss5k) - Final Design Document

CS246 – Assignment 5, Group Project

Outline of the Final, Actual Design

Design patterns used:

Singleton Design was used for the Board class to ensure that there is only one Board created and used during the game play.

Factory Design was used to initiate the appropriate board for each level. More specifically, this design was used because we needed a way to configure the Board in the SquareSwapper class using the pure virtual method from abstract Level field which can be an object lv0, lv1, or lv2.

Observer Design was used to notify the text display and graphical display for the changes for individual squares.

Description of each class and its relationships to other classes:

SquareSwapper – is the core class that is used from the main function.

Public method includes *initiating method* to activate all the command-line options and all the *command functions* that can be called such as swap, hint, scramble, levelup, leveldown, restart and test. Also, public method includes all the *core functions* such as check, chainCheck, chainMatches, dropSquares, generateSquares, and removeMatches which are all used during the swapping phase. In addition, there are getter and setter for the score functionality and operator method to print the current game (board with the score and level).

SquareSwapper “has a” Board, Level, TextDisplay and Xwindow.

Board – is the core class that does the actual functions done in board inclusively.

- Its field has all the necessary information about board itself: board, score, level, genSequence and etc. Board class “has a” TextDisplay and Square.
- It has singleton Board methods such as getInstance and cleanup.
- It has method to clear all the memories allocated for each squares in the board
- Most importantly, it has the implementation to check if the match is 3-match, 4-match, 5-match or L-match along with the dropSquares and generateSquares for after crushing the squares.
- Core functions such as swap, scramble and hint are implemented in this class.

Square – is an abstract class to store basic information about each individual squares such as row, col, lock, special, colour and etc. Majority of the public methods are getter and setters for each fields. This class has two pure virtual method for its subclasses: crush and draw, which are used for doing each special square’s special ability and drawing on the board according to the field information.

The subclasses of Square class (Basic, Lateral, Upright, Unstable and Psychedelic) has only its ctor, crush and draw as its methods.

Square class “has a” TextDisplay.

Level – is an abstract class to store basic necessary information for each level: initScore, TextDisplay. It has the implementation to read the saved board configuration. This class has three pure virtual methods for its subclasses: initBoard, isComplete, and setInitScore, which are used for initializing the board for that level and to be assigned to a Board field in SquareSwapper class using the factory method and checking if the level completion requirement is met.

The subclasses of Level (Lv0, Lv1 and Lv2) has the implementations for those virtual methods.

Level class “has a” TextDisplay.

TextDisplay – is a class to store the information about the texture display.

It has a public method to notify this texture display about all the changes occurring to the individual square using observer’s pattern. In addition, it has an operator to print out the

board information in the text.

Xwindow – is a class to facilitate the graphical display using X11.

It has public methods to draw a filled Rectangle, to draw a string as default. In addition, we modified these functions and implemented two new methods: fillSquares to draw individual squares according to the field of the square and drawAccessory to draw the Score and Level features graphically.

How it differed from initial design

SquareSwapper

- All public methods that we first included in Board moved to SquareSwapper such as swap(int, int, int), hint(), levelup(), leveldown(), scramble(), and restart().
- We added public methods called dropSquares() and generateSquares() so that they can be called after a match has been crushed.
- We added a public method check(int , int) so that it can be used in main function to check whether or not the swap can be done.
- We added public methods called removeMatches(), chainCheck(int, int, int) and chainMatches() which are mainly used to check if there is a chain match or if there is already a match before swapping or after generating squares.
- Added some mutators and accessors.
- Related to chain matches, we added getScore(), scoreUpdate() and returnScore() to update score according to the rules.

Board – Singleton Pattern

- Basically Board has most methods that will be called from SquareSwapper. For example, we have generateSquares() in SquareSwapper and it calls the public method of Board, called generateSquares() to actually generate squares.
- A few setters(mutator) were added.
- Added few public methods that help to check if there is a match and if there is a match with how many squares that it was built.

Square – Inheritance Relationship with subclasses

- At first, we planned to use an observer pattern in order to check a match and crush each squares by making an array called “Neighbours”. However, as we proceeded, we realized that this is not efficient. This is the reason why we moved public methods for

checking matches to Board class.

- Other than this change, fields and public methods are mostly the same.
- Added some mutators and accessors for private fields.
- We made class Square abstract by announcing pure virtual method: crush().

Special Squares (Subclasses of Square)

- Implemented crush() on each subclass which crush a square according to its special character. If the square has a lateral sign, it automatically calls the crush method in Lateral class.
- We were able to approach to the board from each subclass of Square by implementing as the following:

```
Board *board = Board::getInstance();
```

TextDisplay

- Implemented the class as we planned.

Level – Inheritance relationship with Lv0, Lv1, Lv2

- Implemented each level by factory method pattern.

Xwindow

- Added a few public methods to noticeably show special squares.

In general, we implemented the most of the fields and methods as we designed in our initial UML. However, we ended up having more methods with some extra fields. Also, we have added more classes as we moved on. We compared our UML diagram and actual code in detail.

The biggest change we made from our initial UML diagram was introducing a new class SquareSwapper to divide the role of Board class in the initial UML. Initially, this was done because we could not find a way to return Square* [10][10] type from initBoard method in Level class.

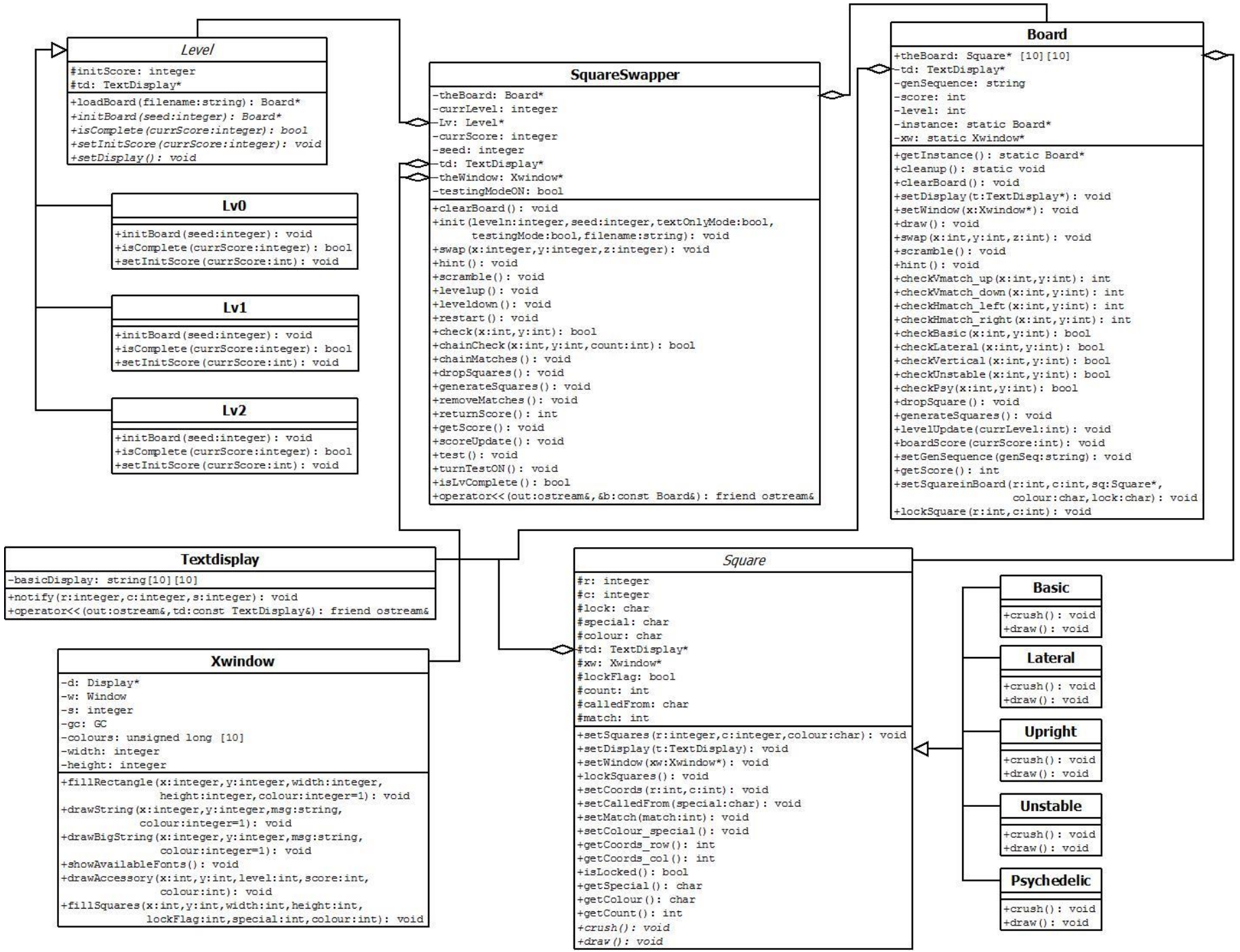
Another noticeable difference are the new methods regarding to the graphical display. For

this to be done, we had to modify the Xwindow class and added new important methods in the the subclasses of the Square class and the Board class to draw the squares in the board as well as to draw the level and the score graphically.

We also made some last minute changes to fix the bugs of special squares' crushing not working properly when it was chain reacted by some other special squares. To deal with some of the bugs, we introduced new fields such as calledFrom and match in the Square field to let each squares know which square the crush() was called from and what the number of match was.

Bonus Feature

1. **Moves Remaining**: we added extra level to show how many moves are left in order to win the game. The already-existed three levels (level 0,1,2) have unlimited moves but the last one has 10 moves to swap. To win the last level, the player has to get score over 700. If the player fails to finish the game in 10 moves, it restarts the game but the score remains the same.
2. **Colour Scheme :** we have added another colour scheme for a bonus version. The original colours of graphic seem too bright so we changed to soft pastel colours.


Level
#initScore: integer
#td: TextDisplay*
+loadBoard(filename:string): Board*
+initBoard(seed:integer): Board*
+isComplete(currScore:integer): bool
+setInitScore(currScore:integer): void
+setDisplay(): void
Lv0
+initBoard(seed:integer): void
+isComplete(currScore:integer): bool
+setInitScore(currScore:int): void
Lv1
+initBoard(seed:integer): void
+isComplete(currScore:integer): bool
+setInitScore(currScore:int): void
Lv2
+initBoard(seed:integer): void
+isComplete(currScore:integer): bool
+setInitScore(currScore:int): void
SquareSwapper
-theBoard: Board*
-currLevel: integer
-Lv: Level*
-currScore: integer
-seed: integer
-td: TextDisplay*
-theWindow: Xwindow*
-testingModeON: bool
+clearBoard(): void
+init(leveln:integer,seed:integer,textOnlyMode:bool,
testingMode:bool,filename:string): void
+swap(x:integer,y:integer,z:integer): void
+hint(): void
+scramble(): void
+levelup(): void
+leveldown(): void
+restart(): void
+check(x:int,y:int): bool
+chainCheck(x:int,y:int,count:int): bool
+chainMatches(): void
+dropSquares(): void
+generateSquares(): void
+removeMatches(): void
+returnScore(): int
+getScore(): void
+scoreUpdate(): void
+test(): void
+turnTestON(): void
+isLvComplete(): bool
+operator<<(out:ostream&,&b:const Board&): friend ostream&
Board
+theBoard: Square* [10][10]
-td: TextDisplay*
-genSequence: string
-score: int
-level: int
-instance: static Board*
-xw: static Xwindow*
+getInstance(): static Board*
+cleanup(): static void
+clearBoard(): void
+setDisplay(t:TextDisplay*): void
+setWindow(x:Xwindow*): void
+draw(): void
+swap(x:int,y:int,z:int): void
+scramble(): void
+hint(): void
+checkVmatch_up(x:int,y:int): int
+checkVmatch_down(x:int,y:int): int
+checkHmatch_left(x:int,y:int): int
+checkHmatch_right(x:int,y:int): int
+checkBasic(x:int,y:int): bool
+checkLateral(x:int,y:int): bool
+checkVertical(x:int,y:int): bool
+checkUnstable(x:int,y:int): bool
+checkPsy(x:int,y:int): bool
+dropSquare(): void
+generateSquares(): void
+levelUpdate(currLevel:int): void
+boardScore(currScore:int): void
+setGenSequence(genSeq:string): void
+getScore(): int
+setSquareinBoard(r:int,c:int,sq:Square*,
colour:char,lock:char): void
+lockSquare(r:int,c:int): void
Textdisplay
-basicDisplay: string[10][10]
+notify(r:integer,c:integer,s:integer): void
+operator<<(out:ostream&,td:const TextDisplay&): friend ostream&
Xwindow
-d: Display*
-w: Window
-s: integer
-gc: GC
-colours: unsigned long [10]
-width: integer
-height: integer
+fillRectangle(x:integer,y:integer,width:integer,
height:integer,colour:integer=1): void
+drawString(x:integer,y:integer,msg:string,
colour:integer=1): void
+drawBigString(x:integer,y:integer,msg:string,
colour:integer=1): void
+showAvailableFonts(): void
+drawAccessory(x:int,y:int,level:int,score:int,
colour:int): void
+fillSquares(x:int,y:int,width:int,height:int,
lockFlag:int,special:int,colour:int): void
Square
#r: integer
#c: integer
#lock: char
#special: char
#colour: char
#td: TextDisplay*
#xw: Xwindow*
#lockFlag: bool
#count: int
#calledFrom: char
#match: int
+setSquares(r:integer,c:integer,colour:char): void
+setDisplay(t:TextDisplay): void
+setWindow(xw:Xwindow*): void
+lockSquares(): void
+setCoords(r:int,c:int): void
+setCalledFrom(special:char): void
+setMatch(match:int): void
+setColour_special(): void
+getCoords_row(): int
+getCoords_col(): int
+isLocked(): bool
+getSpecial(): char
+getColour(): char
+getCount(): int
+crush(): void
+draw(): void
Basic
+crush(): void
+draw(): void
Lateral
+crush(): void
+draw(): void
Upright
+crush(): void
+draw(): void
Unstable
+crush(): void
+draw(): void
Psychedelic
+crush(): void
+draw(): void

Project questions

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

    A. Our answer remains the same as initial answer from the plan. Creating these five kinds of square subclasses under the parent class Square, we are not able to create any other types of squares unless we create another subclass under Square to implement other type of squares. Furthermore, even if we do add and implement a subclass under Square class for new kind of square, this new type of square will not be generated unless we modify the methods for generating squares (which is designed to generate only these five kinds of squares) to consider creating this new type of square. Since these methods are not visible to the users, the system is ensured to create only the five kinds of squares.

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

    A. Our basic idea remains the same as initial answer from the plan. All we need to do to add new squares with different abilities is adding new subclasses under Square class so that they inherit basic characteristics from the superclass. Then we implement the different abilities under the purely virtual methods (in our case, they are crush() and draw()). Then modifying the methods for generating squares to consider creating the new squares will be sufficient.

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

    A. Our answer remains the same as initial answer from the plan. Since we designed to initialize the board for that level using the factory design pattern, this is really easy. Adding new subclasses under the Level class and implement the necessary pure virtual methods from Level (in our case, they are as follows: Board* initBoard (int seed), void setInitScore (int currScore) and bool isComplete (int

currScore)). Then adding few lines of utilizing these method for initiating new desired level will be sufficient.

4. 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 can mean 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. do something like rename swap s)?

   A. In our main function, we created an array of string to store all the command to be received, which are interpreted while taking input and checking if the input is one of the strings in the array. In order to accommodate the addition of new command names, we need to add this command name into the array and add a condition to check if we have received the new command name in the array. We saved the command names in array on purpose. In this way, we can easily change the existing command names and even support such command as rename swap s. All we need to do is to add the rename command name first into the array, then take input for existing command name and the command name to be changed to make necessary change. The code lines below will do.

```
if (command == commandList[7] ){ //commandList[7] is “rename”
        cin >> existingCommand >> changedCommand;
        for (int i=0; i<numCommand ; i++){
                if (commandList[i] == existingCommand)
                        commandList[i] = changedCommand;
                …
        }
}
```

Final questions

1. What lessons did this project teach you about developing software in teams?

    A. One of the key advantages of working in team was increased efficiency. We were able to find out more of several possible approaches to the project than working by oneself. Ultimately, we were able to find the optimized design. Also, since two people have different strength points, we were able to get help from each other. While developing, we were able to thoroughly understand design patterns, especially singleton, observer, and factory method. Also, we fully understood the "has a" and "owns a" relationship. By developing the game in team, we also learned more about ourselves, what our weaknesses and strengths are, and increased productivity, performance and skills (by discussing several possibilities, finding the improved design, and being responsible).

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

    A. If we had a chance to start over, we would try a different design. Currently, we are satisfied with our design, but if we had another chance, we would think more carefully to find out if there is another optimal design to shorten the running time and to target the minimum recompilation. Also, we would manage time more precisely so that we can concentrate the input and output the best result.

    B. In fact, there is one design flaw for our design. We have made the field Square* theBoard[10][10] in Board class as public, because we could not find a way to return this field value/address for the method getInstance() for singleton design pattern. We have tried all the recommended return values from Google, Piazza, and the office hour, but none of them actually compiled. If we get a chance to start over, we would definitely consider this issue and approach from a different way.