**Game: Square Swapper 5000**

**Teammates: Jeremy White and Yifan Song**

**Board**

The board is a simple 2-dimensional array containing the squares. Its functions contain no game logic; they merely provide an interface for other classes to use to manipulate the board.

Since locked squares are based on board positions, not squares themselves, the board is actually a 2D array of Cells: a structure that contains a pointer to a square and a Boolean which indicates whether the cell is locked or not. This implementation detail is hidden from the other classes.

**Squares**

Squares keep track of their position and colour, can draw themselves (using text or graphics), and remove themselves from the board during a match. Special squares inherit from the basic square.

For drawing, the basic square performs the tasks common to all squares (drawing the basic square with the correct colour, drawing an outline if the square is in a locked cell), and calls a virtual function to do any special drawing required by the special square.

Removing from the board has a somewhat recursive nature: when a match is made, a "destroy" function is called on all the squares in the match, with an integer argument representing the number of squares destroyed passed by reference. For basic squares, the argument is incremented by one and the square is destroyed (this is why the argument can't be a return value: the square is destroyed at the end of the function). For special squares, the square calls destroy on all the squares that the square's special ability affects**.**

**Matches**

The BoardManip class is where most of the main game logic happens. It runs the main match algorithm, which works as follows:

1. Iterate through a list of "updated" squares (square are marked as updated when they move or are generated)

2. Check for a match involving the current square

3. Destroy any squares involved in a match, and generate special squares depending on the match

4. Plug the holes created by destroying squares (this includes generating new squares using the current level), and mark all squares that fell or were generated as updated

5. Score points based on the number of squares destroyed, and increment the current chain

6. Repeat the above steps until no matches are found

This was designed to work with both swapping squares and resetting the board - the only difference is that there is an “initializing mode", which disables scoring, while the board is being reset.

For finding matches, an effort was made to make the functions as reusable as possible. This is because there are different features that use the concept of matches in slightly different ways: swapping needs to check if a move makes a match before performing the move, clearing matches needs to analyse a match for its size and shape to determine which special square should be created, and the hint and scramble features need to check if any possible move would result in a match. As such, many of the match-related functions use a list of positions of squares involved in a match, which makes the results of those functions more generally usable, increasing the overall cohesion.

**Display**

The TextDisplay and GraphicalDisplay inherit from a simple abstract BoardDisplay class, which has a pointer to the board and a draw function. Both classes just loop through the board and call the appropriate drawing function on each square that needs to be drawn.

At first, GraphicalDisplay kept its own 2D array of pointers to squares, which were copied directly from the pointers held by the Board class. A square was redrawn if GraphicalDisplay's pointer did not equal Board's pointer (indicating that the square has been changed). However, this method seemed to not work all of the time, so a rather inelegant method is now used: squares are set as modified whenever they need to be redrawn. GraphicalDisplay draws all squares that are marked as modified, and sets them as not modified after drawing each one.

**Levels**

Most of the functionality for levels occurs in the abstract base class Level. Since script files work the same for all levels, they are mainly dealt with in the base class. When Level is told to use a script file, it parses the entire file and creates a queue of squares, which is consumed when the board is initialized. If there is a sequence line at the end of the script file, it makes a list of the colours, which is used once the queue is empty.

Generating the next square makes use of the Factory pattern: squares are generated according to the specifications of the current level. If a script file is being used, a new square is generated using the queue made while parsing the script file, or, if that has been depleted, using the list of colours from the sequence line. If there is no script file (or no sequence line), it then defers to the implementing class to generate a square, and returns that.

For locked squares, a list of squares to lock is provided by a function that works similarly to nextSquare. If a script file is being used, Level returns the list of locked squares, which was filled while reading the script file. If no script file is being used, the implementing class generates a list of locked squares, which is optional to implement: by default, an empty list is generated.

Checking the conditions for advancing to the level is a simple virtual method that goes straight to the concrete level.

Some advanced features, such as levels specifying a size for the board, or a restriction on the number of moves, work in a similar way to locked squares: there is default functionality (10x10 board, no move limit), but levels have the option of specifying these options (level 3 has a move limit and uses an 8x6 board).

**Scoring**

The Score class is fairly simple. The current score can be increased by a score function, given the number of squares cleared in a match and the current chain. It also uses a file for recording the highest score, which is written to in the score function if the current score goes higher than the high score.

**Commands**

The command-line interface and in-game commands are handled in the main method. The main method creates instances of all the main classes (Board, Score, Level, BoardManip) and sets up their references to each other. Aside from calling the classes' various functions in response to commands, it also keeps track of the current level (this includes associating level types to numbers), and draws the game information such as the current level and score.

**Implementation Questions**

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

Each special square should be a subclass of the Square class. Thus, whenever a square of any type is generated, the constructor of the appropriate subclass is called. If the generated square is a basic square, then only the Square constructor is called.

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

The squares will be generated with the factory pattern during initialization and when plugging holes in the board. In addition, every type of square is a subclass of the square class. At first, we thought that all the logic for generating square is compartmentalized in the level classes. Thus, we only need to create a new square subclass and modify one method in each level class if we want to add new squares to our game. In reality, the BoardManip class also contains part of the logic as certain matches create special squares. If we want to add new special squares, we would have to add some code to the BoardManip class.

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

Each concrete level is a subclass of an abstract level class. Implementing new levels thus only requires the addition of a new concrete level subclass and some modification to the main program. We note that the level classes are not implemented using the decorator pattern as levels do not add more features to a basic level. Instead, each level has its own unique features and some unique implementation of some common features. For this reason, we cannot combine different levels, as the Decorator pattern would allow.

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?

The main program should handle as little of the game logic as possible. As such, the commands given to the main program should translate into the main program calling methods of the Board, BoardManip and/or Level classes. Thus, the addition or modification of commands only leads to changing and adding case in the main program’s switch statement.

How difficult would it be to adapt your system to support a command whereby a user could rename existing command?

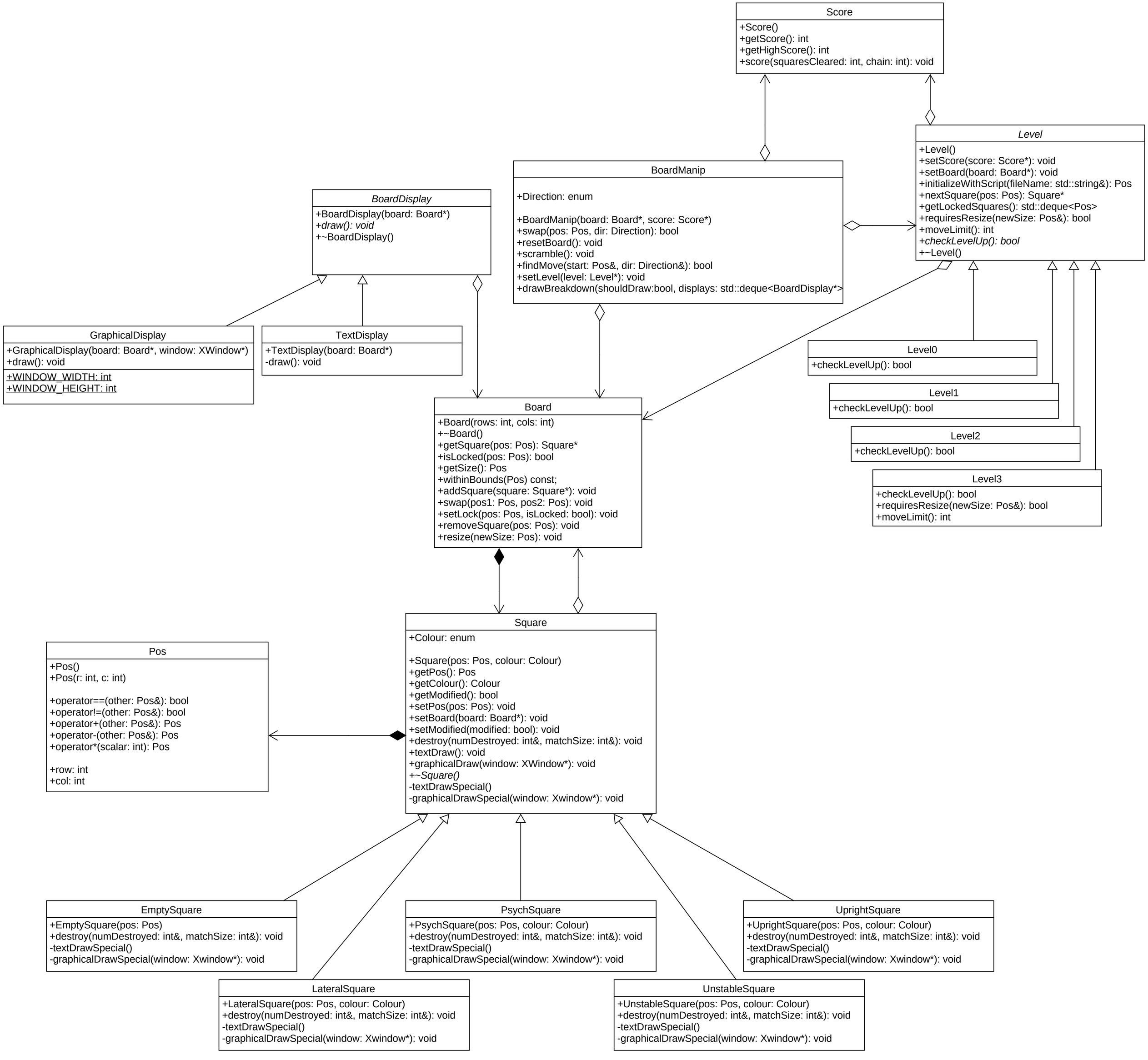
We expect this to be a simple task.  The command names can be saved as strings in the main program. When the user inputs commands, the main program compares the input with those strings. It is possible to add a “rename” command that would reassign a given command string with the new command name given by the user.

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

Developing software in teams requires excellent communication between team members. For example, we want to avoid team members working on the same methods or classes at the same time, but we also want to share ideas or improvements to each other’s code. Thus, there needs to be a clear plan and timeline to follow. Furthermore, some pieces of code or functionality can be particularly hard to understand so it is important to explain complex code in comments.

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

We overwrote each other’s code a few times due to communication issues at the beginning. Thus, we believe that having a more detailed plan to work with would have helped us achieve a even better result.


Score
+Score()
+getScore(): int
+getHighScore(): int
+score(squaresCleared: int, chain: int): void
Level
+Level()
+setScore(score: Score*): void
+setBoard(board: Board*): void
+initializeWithScript(fileName: std::string&): Pos
+nextSquare(pos: Pos): Square*
+getLockedSquares(): std::deque<Pos>
+requiresResize(newSize: Pos&): bool
+moveLimit(): int
+checkLevelUp(): bool
+~Level()
BoardManip
+Direction: enum
+BoardManip(board: Board*, score: Score*)
+swap(pos: Pos, dir: Direction): bool
+resetBoard(): void
+scramble(): void
+findMove(start: Pos&, dir: Direction&): bool
+setLevel(level: Level*): void
+drawBreakdown(shouldDraw:bool, displays: std::deque<BoardDisplay*>
BoardDisplay
+BoardDisplay(board: Board*)
+draw(): void
+~BoardDisplay()
GraphicalDisplay
+GraphicalDisplay(board: Board*, window: XWindow*)
+draw(): void
+WINDOW_WIDTH: int
+WINDOW_HEIGHT: int
TextDisplay
+TextDisplay(board: Board*)
-draw(): void
Level0
+checkLevelUp(): bool
Level1
+checkLevelUp(): bool
Level2
+checkLevelUp(): bool
Level3
+checkLevelUp(): bool
+requiresResize(newSize: Pos&): bool
+moveLimit(): int
Board
+Board(rows: int, cols: int)
+~Board()
+getSquare(pos: Pos): Square*
+isLocked(pos: Pos): bool
+getSize(): Pos
+withinBounds(Pos) const;
+addSquare(square: Square*): void
+swap(pos1: Pos, pos2: Pos): void
+setLock(pos: Pos, isLocked: bool): void
+removeSquare(pos: Pos): void
+resize(newSize: Pos): void
Square
+Colour: enum
+Square(pos: Pos, colour: Colour)
+getPos(): Pos
+getColour(): Colour
+getModified(): bool
+setPos(pos: Pos): void
+setBoard(board: Board*): void
+setModified(modified: bool): void
+destroy(numDestroyed: int&, matchSize: int&): void
+textDraw(): void
+graphicalDraw(window: XWindow*): void
+~Square()
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void
Pos
+Pos()
+Pos(r: int, c: int)
+operator==(other: Pos&): bool
+operator!=(other: Pos&): bool
+operator+(other: Pos&): Pos
+operator-(other: Pos&): Pos
+operator*(scalar: int): Pos
+row: int
+col: int
EmptySquare
+EmptySquare(pos: Pos)
+destroy(numDestroyed: int&, matchSize: int&): void
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void
PsychSquare
+PsychSquare(pos: Pos, colour: Colour)
+destroy(numDestroyed: int&, matchSize: int&): void
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void
UprightSquare
+UprightSquare(pos: Pos, colour: Colour)
+destroy(numDestroyed: int&, matchSize: int&): void
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void
LateralSquare
+LateralSquare(pos: Pos, colour: Colour)
+destroy(numDestroyed: int&, matchSize: int&): void
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void
UnstableSquare
+UnstableSquare(pos: Pos, colour: Colour)
+destroy(numDestroyed: int&, matchSize: int&): void
-textDrawSpecial()
-graphicalDrawSpecial(window: Xwindow*): void