**Introduction**

Our group chose to implement the Quadris game. In this document, we will explain our implementation of the game, discuss our design and answer the provided questions.

**Overview**

| Class/Function Name | High Level Description of Implementation |
|---|---|
| Controller | Controller's main responsibility is to takes user input and converts it to valid commands that the rest of the program will understand. Controller then passes these converted commands to Game.<br><br>Controller is also responsible for creating and restarting the Game; setting the game-state according to the command line arguments. |
| Game | Holds information of the entire game and splits tasks to corresponding components of the game. It takes in commands from the controller and calls the functions responsible for making the correct changes.<br>Game also does score calculation for line removal and block removal according to the rules and updates it accordingly. The high score will also be updated if at the end of a game, the score is higher than the previous high score.<br>Game will end the game once the Grid cannot move a Block (ie. When a Block has nowhere else to move to).<br>Game is also responsible for printing out of the text display of the game. |
| Level | Responsible for storing level specifications and generating Block chars according to those specifications (ie. Random vs. non-random, block probabilities, isHeavy conditions). Level passes the generated Block chars to Grid to be created.<br>Level also responsible for level switching and printing out next block for the text display. |
| Grid | Grid takes commands from Game and apply appropriate changes to the game board. This includes, move, rotation of selected block, line clear, creating new block when previous block has been placed, printing game boards. |
| Block | Block is responsible for keeping track of which cells belong to which block. It ensures that the cells maintain the appropriate collective shape through all the movements and rotations. |
| Cell | Holds the information of its coordinates in the Grid and is stationary. Each cell points to the Block it belongs to, if doesn't belong to any Block, points to nullptr. Each time a Block moves or gets removed (partially/entirely), the cells adjusts to reflect the change. |
| GraphicDisplay | GraphicDisplay uses Xwindow class to display game board and score graphically. |

**Design**

Controller design

Controller interprets user input by, first creating a vector of valid commands (commands that the rest of the program will understand). This is done by retrieving the commands from a text file that contains all of the valid commands. Controller then retrieves the user's input and splits the user's input into two strings: the prefix part and the command part. Controller then interprets the command by comparing, character by character, the user's input to each command in the list of valid commands. Controller continues to eliminate non-matches until it finds a matching command or realizes that there is no single command that matches. Controller then passes the interpreted command to Game prefix number of times, or does nothing if Controller was unable to find a valid command.

To handle game creation Controller takes in a vector of string, representing the command line arguments, as a parameter. Controller then creates the Game class using the command line arguments to determine the constructor parameters. Controller constructs the Game class inside of a while loop. Thus when we want to restart we just enter a new iteration, the old Game class goes out of scope and and a new Game class is constructed.

Grid design:

We designed our game in such a way that Cells are only owned by Grid. Blocks only have cell pointers pointing to the Block they are part of. During Block movements, Block would first get the cell coordinates it would move to and check for collision (with other Blocks and the Walls). If no collision would occur, the Block frees its original cells and claims the cells with the new coordinates.

Due to our implementation of deleting line mechanism, which erases the row of filled cells and create new row at the beginning of game board, and efficiency of the graphical display, we choose to call notifyObserver function of cells manually when lines are getting deleted.

Level design:

Since the number of levels are predetermined before the game start, we decided to put specifications of levels in a .txt file with each line representing a level starting from level 1. Our txt file stores the probability for each block to be selected and has the format:

gameSpecs.txt

| (GCD of denominator) (numerator of I) (numerator of J) … (numerator of T)<br>… |
|---|

Example:

Level 1 (S and Z blocks are selected with probability 1/12 each, and other blocks are selected with probability 1/6 each)

| 12 2 2 2 2 1 1 2<br>… |
|---|

This format is to make random generation easier.

"Heavy" in Level 3 & 4 is implemented by adding a field in the Level class

"*" block is added as a new block in the Block class and treated differently in Grid.
We designed level in this way so the changed to block selection probability can be easily made with minimum recompilation.

Block design:
Block handles the movements by iterating through all of its cells and providing coordinates of the block given any movement or rotation. For the movement functions Block adds 1 to the row or col coordinate of the cells based on the given movement command. For rotations Block rotates its given block using vector multiplication: multiplying the given block by a vector that represents a 90 degree turn to its side.

Design Pattern: Observer Pattern
We decided to use Observer pattern for the Graphics component of the game. Graphics is an observer of all Cells so that whenever cell is being modified, graphical display would respond accordingly.

**Changes from Initial Plan:**

Block:
Block was originally supposed to be an abstract superclass, with many subclasses representing each type of blocks (I,J,L….). We had done this originally because we felt that the movement, specifically the rotations, for each block would be unique; thus each requiring its own version of the rotation functions. However, we soon realized that all of the movement functions could be generalized, eliminating the need for inheritance. Thus Block became a single class.

Block was also supposed to have a function called predict which would take a string, representing a movement command, as a parameter and then call the correct movement functions. However, we realized that if we made the movement functions public and called the movement functions from grid, we could cut out an unnecessary middle step. Thus we removed the predict function from Block.

**Resilience to Change**

Change/addition of commands
Controller compares the user's input to a list of valid commands stored in a text file, meaning controller is able to recognize any command as long as it is in the text file. Thus, new commands can be added and removed by adding and removing the commands from the text file. This means that commands can be added and removed without recompiling controller.cc.
(assuming the rest of the program has been changed to accommodate the new command)

Change/addition of Block shapes
Since the class Block stores its cell pointers in a vector and all of it's functions iterates through the entire vector, Block is able to handle blocks of any shape or size. Also, since Block always move it's cells relative to each other, the shape of the block is always maintained. Thus you can use the Block class to control blocks of any shape or size.

Change/addition of Block movements

Since all of the Block's movement and rotation functions are generalize. It is able to handle the movement of any block, regardless of shape or size.

Change/addition of Levels

To add a new level, we will add a new line in our .txt file with the probability of each block to be selected. Other specification with new block or score rate can be accommodated in Grid and level. Since Block can be in any shapes in our implementation, to accommodate new block, only Grid needs to be recompiled. For score rate, we are currently using level value as a score rate directly. So any additional lines in .txt that keeps level info, it will increment score rate automatically.

Change/addition of scoring

To add a new scoring condition, we just need to add a new method for Game which takes in the necessary information from Grid. Grid need to add a corresponding method which passes parameters to Game's method.

Changing the scoring condition is similar.

**Answer to Questions**

1. **How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?**

   Check size of vector of Block after each additional Block has been generated. If the size exceeds 10, then remove first allocated block from the vector of Blocks in Grid and make corresponding cells empty. Else do nothing.

   Yes, we could create a bool variable that controls whether we perform the above check.

   (Answer same as before)

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

   If we were to design our program again, we would design sot that block shape is stored in external file, not specified in the code to improve flexibility of accommodating additional stages. This will result in implementation of new levels without any recompilation. Furthermore, grid would be broken down in order to reduce code complexity and less coupling.

3. **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. something like rename counter clock wise cc)? How might you support a \macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.**

Since our implementation already contains a control module, we would only need to recompile the control module and added feature to reflect the changes.
Rename of existing command can be handled by creating a map that takes key as an user renamed command and returns value as an existing command.
Macro can be implemented by creating a map that taking name of the macro as a key and returns a string that represents the sequence of command as a value. The returned value will be passed into control module using stringstream.

(Answer same as before)

**Final Questions**

1. **What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?**

This project taught me that since everyone has different skill sets, it is important for us to understand and harness them for the productivity of the team.

It is essential in the designing stage, to come up with a well thought out and detailed interface that demonstrates interaction between modules. This is a vital guiding principle that ensures the integrity of the game design. However, when it's necessary it is also important to be flexible and willing to change a part of the design for better performance, runtime, and reusability.

I have also learned that version control when developing software is critical. Without proper version control, we could be working on outdated versions and progress could be lost when trying to merge files.

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

I would have tried to make our three internal deadlines a little earlier to accommodate for unexpected problems arising in the end close to the due date. Also making a more detailed breakdown of the tasks and distribute responsibilities to maximize productivity.

**Conclusion**

Overall, we feel that there are things that we could have made better, but it was still a very successful learning experience both in developing software and working in a team.