Quadris - Report

Our overall structure of program employs *Model-View-Controller (MVC)* design for its obvious advantages.

Our **Model** comprises of following classes:

Cell, **Block** and its child classes, **Level** and its subclasses. These form the central component of the pattern. They express our application's behavior in terms of the problem domain, management of data, logic and rules of the application.

Our **Controller** comprises of following classes:

GameBoard, and ScoreBoard - This is the part which tends toward whatever the use input it. It has a knowledge of what the basic commands are and it controls the main board of the game using the commands and the rules of the game. There are of course additional parts such as the ScoreBoard which updates to store the new score as soon as a row clears out in the game board, block clears out, etc. This newly updated information is then sent to the View part of the program to inform the outputs.

Our **View** comprises of following classes:

CommandInterpreter, TextDisplay and **GraphicsDisplay** - View displays the level of game, current score, high score, game board, dropped blocks, the current block to be set, and next block. The View component also contains the *CommandInterpreter* which notifies the controller components of the game of the different commands sent to the main game. For the displays, the TextDisplay displays this information on the command line, while GraphicsDisplay employs **XWindow** utilities to create a more aesthetic graphical window. MVC design pattern was very critical for our project's implementation. It gave us following advantages:

<u>High Cohesion</u>: MVC enabled us to logical group related actions on a controller together. The views for a specific model were also grouped together. For example, model components level and block work in coordination to generate appropriate blocks according to the difficulty level set by the user.

<u>Low Coupling</u>: MVC helped us in achieving lower interdependence between software modules. For example, while GameBoard maintained the active status of the game, it wasn't required to know how the display presented its information. So, there was low coupling between GameBoard and TextDisplay.

<u>Simultaneous development</u>: Since were facing tight deadlines, only way to manage work was to divide it among the group members so that all of them can work in parallel. So, while Vinit was working on setting up the controller, Kinshuk was able to work on model, and Jimit on view.

<u>Flexibility in modification</u>: Because of the separation of responsibilities, future development or modification is easier. Also, our CommandInterpreter has a feature to add new command, so that future developers could reuse our code and carry out their intended functionalities. Any changes to the view won't affect the model and controller of our game.

Command Interpreter

The *CommandInterpreter* is designed to handle the reading, parsing and execution of commands. It reads input from the standard input and then parses the command according to the format specified. Incomplete commands are mapped to complete commands for user's flexibility. Illegal inputs are rejected. It consists of a mapping from the command name to the actual command object, which enables developers to expand the existing set of command and support command renaming (which is a bonus feature we expand on in the later sections). We used observer pattern in which *CommandInterpreter* is a concrete *Subject*. This is to ensure that whenever a new command is entered by the user, the observer object (*GameBoard*) is notified and it performs the function accordingly. Ultimately, the relationship between the *GameBoard* and the *CommandInterpreter* is of an Observer/Subject pattern. This design supports the principle of loose coupling between objects that interact with each other, i.e. *CommandInterpreter* itself executing different unrelated actions didn't make sense, so the responsibility is vested upon the observer to bring out the desired effect of command. This design allowed us in sending data to other objects effectively without any change in the Subject or Observer classes. Furthermore, Observers can be added/removed at any point in time without affecting other regions of program. Another key plus point about this design is that we would be able to change the input type to the main game easily, as we'd only need to add an additional *CommandInterpreter* to the main program module.

Question: 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 initially had the idea of making a hard-coded set of commands that can be run on the *GameBoard*. Doing this, we realized that we would need to add new commands, maybe add commands that execute multiple other commands, etc. Hence, we decided to maintain a map of commands which has the key as the string and the value as a vector of string commands. This is great now, because firstly, as the value is a list, we are now able to send whole lists of commands to be executed to the *GameBoard* from the *CommandInterpreter*. So we decide to append to the map a unique new command name with the list of commands to be executed on the *GameBoard*. Additionally, if we need to rename a command, of course, we can rename a key of an element in the map. Here, we leveraged the fact that we have the *GameBoard* as an Observer to the *CommandInterpreter*. By taking advantage of the fact that *GameBoard* is an observer of *CommandInterpreter*, we were able to design a system resilient to changes.

Hence, by using the Observer/Subject design pattern and some of the previously outlined design choices we made, we were able to establish a *CommandInterpreter* with really usable new features.

Cell

Cell stores the position (x-y coordinates) and block type (letter of block). Here, we have again used the Observer/Subject design pattern. Each cell is a subject to the displays of this program. Every time a cell is changed, it notifies the displays, which update the cells in the display grid according to the coordinates sent with the *CellData*.

ScoreBoard

ScoreBoard keeps track of current level, current score, high score, next block that's going to be spawned. Of course, *ScoreBoard* is a subject to the displays, because it is in charge of notifying the view that the score/level/hiScore has changed. Once the game is over, ScoreBoard maintains a flag, which is set to true, which notifies the displays about the game being finished.

Level

Level Class employs the factory method pattern to generate block objects. This is to deal with the problem of dynamically creating the specific types of blocks that we want (According to the rules). This also allows us the create blocks without having to specify the exact class of the object (block) that will be created. Also, the frequency of different letter blocks differ in each difficulty level. For example, Level 0 takes in a predefined sequence of blocks, whereas all other levels have added functionality of randomizing the creation of blocks with equal/unequal probabilities. The Factory Method design pattern is perfect for our use here because this way, we are better able to control the rules while at the same time minimize compilation. Let's suppose that we'd want the probability to change. We could simple alter the block generation functions in This variation in level prompted us to design our system in this way. In fact, we also have a specific block generation function inside the parent level class, which returns a desired block (for testing, different levels, etc).

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

Since Level is an abstract base class, any new additional levels will inherit from this base class and implement its own method to generate blocks. Thus, there will not be any need to make changes in any part of program. The only main method we'd need to change for this new "Level 5" would be the block generation function, which means we'd only need to compile that file. Hence, the advantages of modularized design and using the Factory Method design pattern.

Block

Block is implemented as an abstract class, and all 7 blocks (I, J, L, O, S, T, Z) derive from this class. We implemented in such a way since all the letter blocks have very similar functions, such as shifting (left, right, down), dropping, rotating, etc. In fact, all these "actions," if you will, are implemented in the parent Block class, because the actions are the same for all the blocks. The difference between the blocks is the shape of the blocks. Each block has a different shape, so that property of the block is initialized in each specific block class.

Also, other modules that used block objects could utilize this polymorphism to their advantage by simply creating a pointer to a Block in the code and assign it as any of its child classes by utilizing the Dynamic Dispatch properties of C++ where required (i.e. maintaining the shape of the block). Additionally, the base Block class has a virtual destructor to allow deletion of its subclasses when a block pointer is deleted.

Now, let's consider how the blocks are designed.

The blocks start off with a reference point (which is the bottom left of the block) and have the other respective points constructed relative to this reference point. Additionally, at the start, the block already has its four orientations constructed relative to each of the different reference points. The elegance of this solution is that, every time there is a translation in any of the directions, we only need to update the reference point, and generate the other points likewise.

GameBoard

The GameBoard class is the core of the whole game. It manages the cells in the grid, is has the block list, and it has the primitive definitions of block placing, deletion and maintenance. As mentioned before, the GameBoard class is an observer of the CommandInterpreter and it runs a repetitive loop of the list of commands given from the notify function. GameBoard implements the core functionality of Quadris game.

Let's look a bit deeper into how GameBoard works.

Cell Grid

The 2-D grid of cells inside the GameBoard maintain the state of the board. Every time there is a change in the board, (i.e. a block translates down), the GameBoard updates the grid likewise, and the particular cells which have updated in turn notify the displays (View).

The Hint System: The Hint works so that it finds the lowest point in the grid where the block could fit legally. Additionally, the hint makes sure that the solution it gives us for the block placement and orientation gives the placement such that the lowest number of rows are taken up. We make sure to try all the different positions and orientations, and although this is processing intensive, it gives us the most ideal location to out hint algorithm.

The GameBoard has a ScoreBoard, so it also has the task of sequentially updating the score and the level upon level changes, row deletions, etc.

Display

TextDisplay and GraphicsDisplay class inherits from Observer abstract class. They observe the individual cells of the

GameBoard and the ScoreBoard.

Question: 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?

Since we are maintaining a has a relationship with the owner of the blocks (i.e. GameBoard), the GameBoard validates each block every time a user turn is finished (the current block is dropped). Whichever blocks have the count of 10 (which is incremented each turn) is removed from the grid and the list of blocks in the GameBoard, and the displays are of course notified of this.

Conclusion

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?

We have done a lot of projects in high-school but none of them were this complex. All the way from assigning duties to meeting deadlines, made us learn a lot about how software development works in real-life like scenarios. We first started by dividing duties among ourselves. It initially seemed to be a easy task but different problems came along the way. We were supposed to divide duties realistically, keeping in mind the abilities and capabilities of each of us. After that, we faced the main problem in meeting the deadlines. And soon enough we realized that the deadlines we had set initially weren't quite realistic. And by facing this problem, we get the idea of how deadlines might be set in real work places and how things can easily go ugly, if we won't keep up our work with the deadline.

Second major problem we faced was communication. We set the deadlines but didn't really communicate enough among ourselves about the progress and the problems we were facing. This indirectly delayed our progress, since the problem which could have been solved within an hour with the help of other teammates took days to solve. We learned how good communication between teammates can lead to smoothly running of tasks and creates a less problem-prone environment.

In continuation with the communication problem, we also found it hard to manage ourselves consequently, delaying the whole progress. Since this time we were not gives any .h files or any sort of pre-condition, which creates a basic foundation, we found it quite challenging to decide which methods or functions should we include, or which methods should we make private or public. In previous assignments, we used to think that some methods should be made public instead of private. However, while doing this project, we realized how making some methods public can make our program more vulnerable. Since, security is one of the most important thing that should be keep in mind while working in real workspace, we learned to make methods or functions privates as long as we can, at the same time, preserving encapsulation and invariant.

So, in conclusion, we learned about better design practices such as MVC, observer pattern, etc. Although we faced many obstacles such as lack of coordination along out journey, we learned a lot about real software development industries.

Early assignment of responsibilities

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

If given a chance to start over, we would have divided the project in equal proportions. Often, it happened that one of the group members worked on more number of components than others due to difference in abilities. However, this was an issue only because we didn't divide the tasks equally. The other members would have then been motivated to work harder as well.

Another issue was that we started a little too late (a week before the deadline). This provided us a less amount of time to create better and extensive test suites. Furthermore, we didn't check for memory leaks with all user inputs. This might have led to lower performance of our program. If we had invested more time in our project, then we could have made better program design. For example, pIMPL idiom is considered a good practice, but we didn't have enough time to make separate classes for attributes.

UML Changes

Initially designing the UML was really helpful in deciding how to structure our program. The main difference we have had in our UML is the number of methods we have added to each of the classes. The understanding behind this is the we had to think of the program before hand. Here are the main changes that we have had:

1. The rotate functions part of each block have been removed. This is because we had to account for the macro commands that could have been sent from the CommandInterpreter. Hence, we have to add up all the transformation and check if the block arrives at a valid location. Hence, all this is now done in the GameBoard, and in the end, the blocks' reference points and orientations are updated.
2. Another key functionality we have added was, the displays are observers of two different Subjects, the cells and the ScoreBoard. This allows for both of the information to be sent to the displays.