CS 246::Biquadris

Chosen Project: Biquardis
Group Members: (Name | Quest ID)

- ❏ Jagrit Chaitanya (jchaitan)
- ❏ Jin Barai (jsbarai)
- ❏ Kimia Shaban (k3shaban)

Plan of Attack:

In our code, upon constructing our UML we decided to have *abstract* Block and Level classes, in order to combat this assignment. These abstract classes would then have *subclasses* (with inheritance), where the Level class would have subclasses for all levels from level 0 to 4, and the Block class would have 7 block children for all block types we are to implement. Through doing this, it would become easier to use pointers without potential splitting. We decided this was quite an easy solution in order to add further level classes as bonus features, which we intend on having up to 6 levels (starting from 0 up to and including a 6th level as well). However, having blocks was simply not enough in order to create the entire game. Hence, we decided that every Block would contain 4 cells with a Has-A relationship, with a seperate Cell class in our code. Then there would exist a Grid class that contains all of the cells within the 11x18 grid within a vector field of cells. A cell would have the coordinates for its placement on the Grid, and furthermore information regarding which type of Block it may be, in terms of what to display for graphics and text purposes. Given the game is 2 players, we will also have a Player class that will store each player's score, their name (as an additional feature) and have methods to accurately update the player and their current level as they play. A Grid would connect to a class called TextDisplay, which would actively be used to properly output the correct Grid and further details on each player as well, such as their name, score and level. The grid will also be used in order to determine the particular graphics of the program as well, which we are hoping to have a stronger structure for closer to Due Date 2, we are currently still working on fixing rotate and clearing blocks correctly using Text output, from our TextDisplay class.
At this point, we have figured out how to get these mostly working from a Text output, and working to produce a graphics solution soon as well.

Based on our preferences and experience, we decided to split it up such that Jagrit does the Block and Cell, Jin does most of Levels and Kimia does most of the Grid, TextDisplay and Player. We also all coded the Controller, main game loop and debugged many lines of code together as a team. We love our Biquadris family! The breaking up of our code was as follows:

Breakdown:

Git: Jin (taught the rest of our group members as well)

Level (all 7 levels): Jin
Description: The levels are implemented using the factory design method. Each level contains the following methods - createBlocks, blocksFromFile, makeBlocks, isHeavy.

createBlocks - is a method that does not accept any parameters. A pointer of the particular level class makes a method call to createBlock. Now createBlock does not take any argument. The logic behind generating a block on the mathematical probability takes place here. After knowing which type of block has to be generated, a call to makeBlock is made and makeBlock is passed a character, and the value of the getter function isHeavy() for that particular level.

makeBlock - makes a heap call to generate a block of type depending on the character that is passed to the function.

blocksFromFile - basically takes in a string filename, checks if the file exists using a helper function (is_file_exists), reads the sequence of characters and stores it into a vector of characters.

isHeavy - as described above is a getter function that returns a boolean representing if the blocks for that particular level are heavy or not.

So basically the level class has been implemented in such a way that only calling a createBlock method on the level pointer returns a block pointer, hence following Factory Design Pattern.

Block (all 7 blocks): Jagrit
Description: The Blocks are made using the coordinates they should be assigned to via where they must spawn on the Grid. A Block contains some of the following methods, where some are pure virtual, making Block an abstract class used where each particular Block IS-A child of the Block class.

move - Is used to update the cells the block has given a direction, i.e LEFT, RIGHT, DOWN, which have been defined as global constants.

setCoords - set Coordinates to be the ones given in, mainly used if the grid is unable to accept the Block's new location given a move or drop command.

isHeavy - Used to determine whether or not the Block's are heavy, used when other methods like move or rotate are called to move down.

setOrientation - To determine the orientation of a Block on a grid.

getType - Returns a character given the particular type it is. Example, an I-Block would return an 'I'.

Cell: Jagrit
Description: This Class is used for all of the Cell's in a grid, and all the Cells in the game are owned by the Grid, using a OWNS-A relationship. A cell contains an x and y coordinate, and a type which is a character. A cell contains several methods in order to set, and obtain the x and y values, which for space are not shown here in this plan.

Player: Kimia
Description: A player is used to store the current score, name, level, current Block and the next Block of a player. A player contains many methods to set the Block/next Block, add to score, obtaining the score and level of a player as well, which for space are not shown.

Grid: Kimia
Description: The Grid works by owning a player via a OWNS-A relationship. It also OWNS-A TextDisplay and a Graphical Display, however, these both may change to be owned by the Controller instead.

getPlayer - Is used to provide the player that the Grid owns

clear - used to clear a Cell to be empty (if a Block moves for example)

rowClear - used to clear a row when a block is placed

isRowFull - used to determine if a row is full

isFull - used to determine if a Grid is full

move - used to move the current Block the player points to

validate - used to determine if a particular coordinate is valid (0 <= x < 11, 0 <= y < 18).

down - used to move a Block down

rotate - used to rotate the Block the player points to

drop - drops the Block the player is pointing to

TextDisplay / Graphics: Kimia

Description: TextDisplay and Graphics are both designed to use a similar design to the Observer Design Pattern, such that each time a cell on the grid is changed, it calls the notify function. Graphics also calls on the *Xwindow* class, which is given to us already.
Controller: All
Description: This is used to control all of the commands, such as starting the game, switching turns (done automatically of course), moving, dropping a Block, resetting the game, changing levels, rotating, as well as the commands for blind, heavy and force. It makes for a much cleaner test harness.

Main: All

We have learned that 2-3 hour calls a day work best for us as a group, as well call to incorporate the higher level features that bring together all of our separate code, for example how exactly we want the stack to work from when a user tries to move a block, up to when it is visible on the grid. We have made our controller class in order to make our main file more simple, and more abstract as well as a design method. Thus, based on what the user inputs, single methods on the Controller can be called to allow for a much easier and cleaner main file.

Documentation and Design:

We have added the use of unique pointers for the entire program and the Factory design method for the creation of a Block (in all respective levels). Hence, the createBlock in the levels uses the corresponding factory method such that it generates a new Block, where each level gives the respective probability as classified on the assignment. Our TextDisplay and Graphics partially use the Observer Design Pattern, and notify the blocks accordingly. Using these software engineer principles has led to a smoother design of the code.

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?

   Ans: We believe this could be done quite easily given the current classes we have constructed. If we were to place a parameter in the Cell class that kept track of the number of fallen blocks as a countdown , and simply each time a player dropped a block or a block was to reach the bottom of the grid, a method would decrease this counter field by one and another method would be called to clear all Cells with a counter value of 0. Clearing a Cell simply reverts it back to a “blank” Cell type within the grid, rather than also have a particular Block type as well.

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

   Ans: Abstract level class, we are creating new levels as a team already for bonus features, which is a lot simpler because of the inheritance. Allowing different classes to function independently, ensuring that each class' behaviour depends on the other only if absolutely necessary. For example, the block, grid and textdisplay class will be indifferent to the addition of new levels. They function separately and they only care about the parent/Abstract Level class. Hence adding new levels won't lead to recompilation of those two classes.

3. How could you design your program to allow for multiple effects to applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

   Ans: The simplest way to do this would be to use the Decorator design pattern. If there was to be a simple decoration of one effect on another, then we could easily do this in a simultaneous manner. For example, if a player already had the blind command called on it, we could also use the decorator pattern, add the force command at the same time, and simply unwrap it in order to demonstrate the desired effects simultaneously. Hence, by using this abstract Decorator pattern (similar to A4A3), we could have several effects easily demonstrated very easily.

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 dicult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise 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.

   Ans: We have a controller class that basically deals with the implementation of all the commands. With a simple call to the methods in the Controller any command can be executed. This allows us to again make sure that commands have a definite function and operate independently. Adding new commands is as easy as creating a new method inside the Controller. As far as renaming a command goes, the main reads the input from the user and calls the appropriate method inside the Controller class. To rename a command would basically mean changing a simple if statement inside the main class. Clearly, not difficult at all. In order to change a command name, we could use the preprocessor in order to do that as well.

   Supporting a macro language that would basically entail creating a condition to check the validity of the language and executing the given commands in order inside main.cc.