CS 246 A5 Final Project
Group Members: Tiffany Lin, Olivia Liu, Rebecca Lai
Project: Biquadris

Introduction

For this CS246 group project, we chose to code the game Biquadris using object-oriented programing design patterns. All features of the game are implemented, including working levels 0 to 4, special actions (blind, heavy, force), as well as text and graphical display using Xwindow. This document will provide an overview of the structure of our program, our updated UML, how our actual design differed from our planning, resilience to change, and answers to questions.

Overview (overall structure of project)

Overall, we constructed different classes for board, block, level, unit and command interpreters. The Block parent class has 8 subclasses for each of the 8 types of blocks(I, J, L, O, S, Z, T, and the * block needed for level 4). The Level parent class also has 5 subclasses for each level(0, 1, 2, 3, 4). For text and graphical display, we used the subject and observer pattern.

In main, we created two boards, one for each player. The current board points to either one of them depending on the turn count. When it is player one's turn, they can choose to enter commands such as levelup, leveldown, right, left, down, drop, clockwise, and counterclockwise. We also implemented a command interpreter so that short forms of the commands are recognized. The players can also put an integer before the command, for example 3ri, to indicate the number of times they want to execute that command, in this case, being right 3 times.

After the player enters a command, the board will update accordingly. If the player moves a block, we used methods in Board to check if the move is valid for the current block in the current board. If the move is valid, we use another method to move the current block. When the player drops the block, we check if the row is filled with another method in the class Board. If it is filled, we remove that row and update the score accordingly. Then, it becomes the other player's turn.

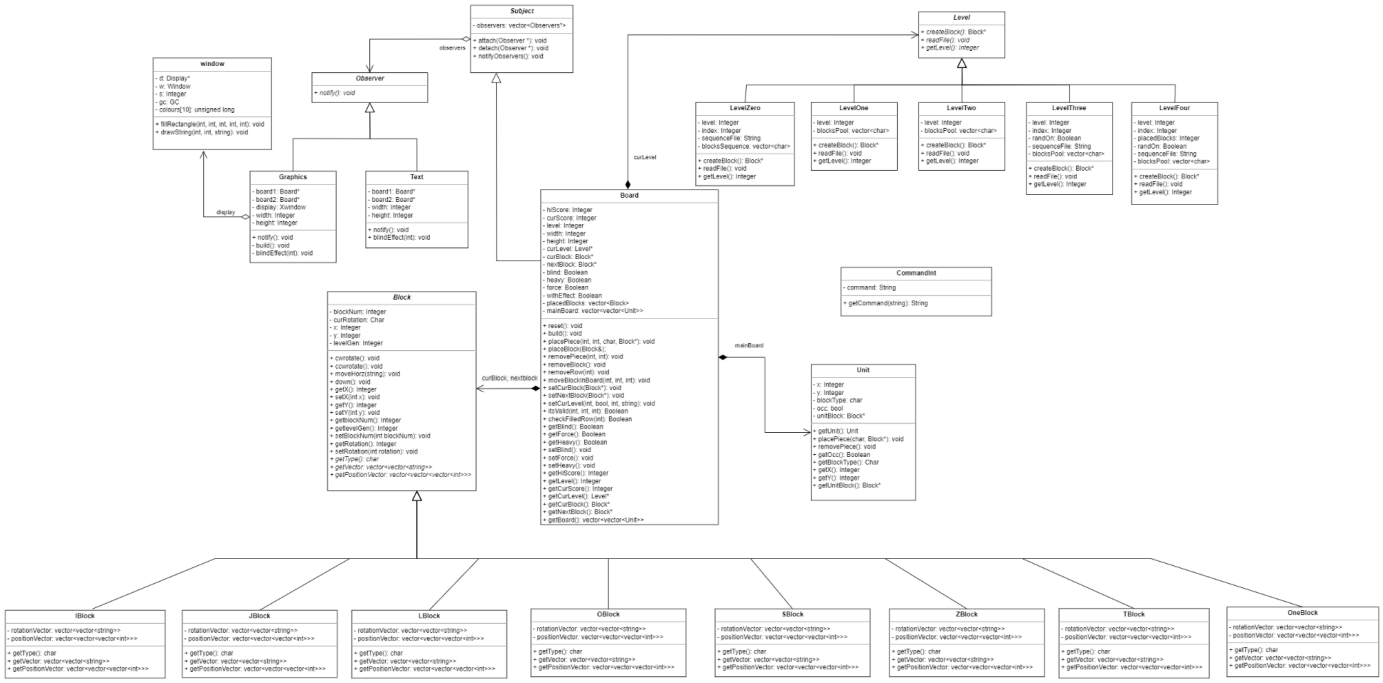
Furthermore, the Block class is used for the creation of new blocks. The current block and next block of each board is generated depending on the level that the player is on. If level is 0, the program will read in from a file to determine the sequence of blocks. If the level is 1 or 2, the program will randomly generate blocks depending on the required probabilities. Levels 3 and 4 can randomly generate blocks as well as take input from files. The difficulty level increases as the level goes up.

Also, the observers are notified every time the displays need to be shown. For example, we would notify the observers in the main class after the player's command is executed or when it is the end of their turn/the beginning of the other player's turn.

We also implemented the special effects, blind, heavy and force. If the player enters the command blind, we would set the boolean field, blind, to true, and put question marks to hide a portion of the display, implemented in the text class. For heavy, that is implemented in the main class. We would check if heavy is true when the user moves horizontally. If heavy is true, the

block moves down twice as well. For force, it is also implemented in main. The player who chooses force can enter the block that they would like to switch, which will replace the other player's current block with that block. Then, if the block there is not valid, the opponent loses. After the player's turn is over, blind, heavy, and force will be reset to false.

Updated UML



Design

Since this project greatly involves collaboration and the usage of polymorphism, we ensured all files had a separate interface and implementation. The main of our program includes Board, Level, Block, CommandInt, Text, and Graphics, which emphasizes on the encapsulation of our implementation.

The biggest class in our program is our Board class. In main, the two board classes would be created that represent a player each. The current board will point between the first and second board depending on the turnCount. Integer fields store the width, height, current score, high score, level, current block, next block, and whether it has any special effects on the board by using boolean to keep track of if there is a blind, heavy, or force effect. Lastly, there is a vector of blocks for placed blocks on the board and a two-dimensional vector of Units for the board. The board class supports many functions that are crucial to the operation of the game. It contains build() where it creates a new empty board and reset() which clears any information previously stored in the board. To perform moves on the blocks, there is a itsValid(int, int, int) function that takes in 3 integers which corresponds to horizontal, vertical, and rotational movement and checks if the movement is valid by checking if the new position is unoccupied and inside the dimensions of the board. If the move is confirmed valid, block movement functions can be applied such as placeBlock(Block &) which puts the block on the units that the

block will occupy on the board as well as removeBlock() that removes the current block. To manipulate any existing blocks that have been placed on the board already, the moveBlockInBoard(int, int, int) function is used which takes in the horizontal, vertical, and rotational movement, validates the move, and then moves the block by removing the current block from its old position(with removeBlock()) and then placing it onto its new position(with placeBlock(Block &)).Lastly, the two remaining functions with the purpose of clearing a filled row is checkFilledRow(int a) which checks if row a is filled and when it is (returns true), removeRow(int a) is called which removes row a from board by using a for loop to replace each row with the row above it, starting at the row that needs to be removed.

For our Unit Class, it represents a 1x1 unit on the board. The board consists of 18 x 11 Units. The class stores the unit's x and y coordinates on the board, the character type of block that it stores, the block that it points to, and whether the unit is currently occupied by any blocks. It also contains functions that place/remove a block from the unit. An example of how this is used is: if a unit calls placeBlock(char a, Block* b), the unit will now be occupied, have a character type of a and points to block b.

We also have a Block class where it is used by the parent Board class to keep track of the number of blocks left in each block, its x and y position, the current rotation, and the type of block it is. It has basic movement functions such as cwrotate(), ccwrotate(), moveHorz(string), and down() that change the fields of the block by rotating it clockwise, counterclockwise, moving it horizontally, and moving it down respectively. For each of the subclasses of block (IBlock, JBlock, etc), they each have a 2d vector of strings that stores the 4 rotations of each vector. There is also a vector<vector<vector<int>>> that stores the position of the 4 units in each block for each rotation.

Since the program requires there to be 5 different player levels, we have a Level class that is an abstract class for the level subclasses. Each of the level subclasses (LevelZero, LeelOne, LevelTwo, LevelThree, and LevelFour) are implemented following the program requirements.

To take in user commands, there is a CommandInt class that takes in only as much of a command as is necessary to distinguish it from other commands. The getCommand(str) function takes in the user input and distinguishes which command it corresponds to (e.g. "lef" is "left", "ri" is "right", etc) by using substrings.

Lastly, we used a Subject abstract class for the Board class. This allows the Observer subclasses (Text and Graphics classes) to be attached, so any changes can be notified to the observer. The Observer class is an abstract class for the Observer subclasses (Text and Graphics) that allows them to act as observers and attach to the board to be notified by any changes occurring in the board and print out the new changes. The notify() function that is part of the Observer class is called when any changes are made to the board, so the new changes can be reflected on the display in the Text and Graphics subclasses. The Graphic class and the Text class are subclasses of the Observer class that prints out the graphical and text display of the game board respectively.

Resilience to Change

We used design patterns such as the Observer pattern and Factory Method. With the use of these design patterns, we can easily add more changes related to these patterns. For example, we used the Factory Method to implement our Level class and its subclasses. Since each level differs by how they create the next blocks, different implementations would be needed for them. So, we used the Factory Method to create a virtual method in Level, called *createBlock*, to handle the creation of our next blocks. With this, creating even more level subclasses (e.g, LevelFive, LevelSix, etc.) would still be simple and easy to implement.

Furthermore, for the implementation of our board, we decided to use a vector of vectors of Units to model it. Unit is a class we designed to represent a single space on the board, and a block would occupy 4 spaces. This allowed us to order the board in an organized way. A simple nested for loop can easily be used to print out each space of the board. Since we used a vector of vectors of Units, our design is really flexible with the increase or decrease of the height and width of the board. Simply setting the board height and width to the desired dimensions would set the change.

Answers 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?

In our design, we keep track of our placed blocks via a vector of Blocks. This would allow for us to know which blocks have already been placed down. The order of blocks placed can be tracked using a new integer field in each block. Thus, we can add a boolean field to the blocks that stores whether the block is the type to disappear from the screen after 10 more blocks have fallen. After each block has fallen, we would check each block to find the block to find the one that was placed 10 placements ago and see if the block stored there is one that would disappear. If it is, we remove it from the main board. Additionally, the generation of such blocks could be confined to more advanced levels. We would need to check the current level that the user is on when we generate the blocks and only add the generation of such blocks when the level is higher.

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

We can use the Factory Method to accommodate the possibility of introducing additional levels. With each new additional level, a class would be created. Since the level's mainly differ by the generated probability of each block, as well as level specific features, we can create a virtual method in our abstract Level class to handle the different features in each level subclass. Since the addition of new levels depends on the modification of pieces' probabilities and level specific features, large amounts of recompilation is not needed as long as there exists a probability of pieces function modifier. This function would be able to modify the probability of the generated piece directly based on what level is chosen, thus not requiring further recompilation. However, any further recompilation is dependent on the additional level specific features. If the feature itself is an extension of previous levels, they can be implemented in a

way which accommodates for changes, but if they are completely new features specific to certain levels, then new functions would need to be implemented.

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

Multiple effects could be applied simultaneously because our effects are implemented separately. We would just prompt the user to enter multiple effects and set whatever effects they want to true. Then, the corresponding implementations would work since the corresponding fields are set to true. For example, if the user wants blind and heavy, the we would set those fields to true. Then, in text, the blind effect would take action and, in main, the heavy effects would take action as well. At the end of the turn, we would set the fields to false again.

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. 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.

For making changes to existing command names, we would simply change the name of the string for that command. Then, we need to change that in the if statement in main as well. To accommodate new command names, we would add a new string with the given command name. For additional features linked with that command, we would add another if statement in the main class. In addition, we implemented a command called "rename" where it allows the player to rename their commands. The user can type rename into the command line followed by the command that they would like to rename as well as the new name. To support "macro language" or a sequence of commands, we could add new command names that would call upon a sequence of commands associated with it. Then, we would move the current block in board accordingly. For example, we might implement a command "rrddd" that would trigger the program to move the current block right twice and down three times. Specifically, we would call upon the method moveBlockInBoard(2, 3, 0).

<u>Final Questions</u>

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

We gained a lot of experience about developing software in teams. First of all, it was very interesting and exciting. Working in a team to create a game was something that we barely had experience with.

However, this was most definitely a difficult experience. In the beginning, we had a plan for each member to complete certain parts, however, we gradually strayed away from the plan as situations arose throughout the week. We used git version control to collaborate on this

project. When one of us codes something up, we would immediately push it onto the repository. The challenge of coding was that sometimes one person would be responsible for a class that depends on the functions of another class, however, it was difficult to continue when the other member did not implement that class yet.

We would try to meet up in person to work on this project together, however, most of the time, there would be one or two members who were unavailable. We would then proceed to communicate virtually, but we would not have as much work done because it was difficult to have online discussions. In addition, everyone has their own way of thinking so trying to collaborate through git version control while communicating virtually was challenging. Thankfully, we did find opportunities to meet up and work on the project together, and those moments were very fulfilling.

Finally, we learned that teamwork is extremely important. Every team member needs to contribute a fair share of work for the team to succeed. If one member does not complete their portion of the work, the other members would need to cover for them, making it a stressful situation for the team in general.

Overall, this was a great experience as we usually worked individually in previous assignments. This was a nice change of pace.

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

If I had the chance to start over, we would create a better and more realistic plan to complete this project. Each team member needed to finish their part of this project on time. Some members were unable to finish their part of the work, which created challenges for other members. Thus, our plan should be remade so that each member should be able to finish their portion of the project. Initially, we should have also set pre-planned meeting times, so every member was aware of what completed aspect of the project should be brought to the meeting table. Furthermore, while in our final implementation, we kept the core of our initial design, there were many aspects we did not consider during the planning and thus had to think as we worked on the project. We should have thought a bit harder about the specifics of the design to ease the burden on us during the latter half of the project.