Group Members:

Paranjay Agarwal (p38agarw), Daksh Prashar (d3prasha), and Jaskirat Pabla (j6pabla)

Introduction:

Initially, all three of us are big fans of Tetris and thought that it would be very fun as well as a great challenge to code out a game similar to the one that we grew up playing. While reading the biquadris pdf, we realized just how complicated things can get for such simple maneuvers. While reading, we decided to jot down all the important implementation instructions that were written. By the time we were finished reading through the document thoroughly, we had a clear idea of what was required from us. Although it was a little overwhelming at first, we decided to take our time and focus on one problem at a time. For example, we decided to first break down the game to identify different objects that will have to be a class while preparing to create the UML. In the big picture, we knew our main class would be the Gameboard class, and objects such as levels and blocks will be subclasses of it.

Overview and Design:

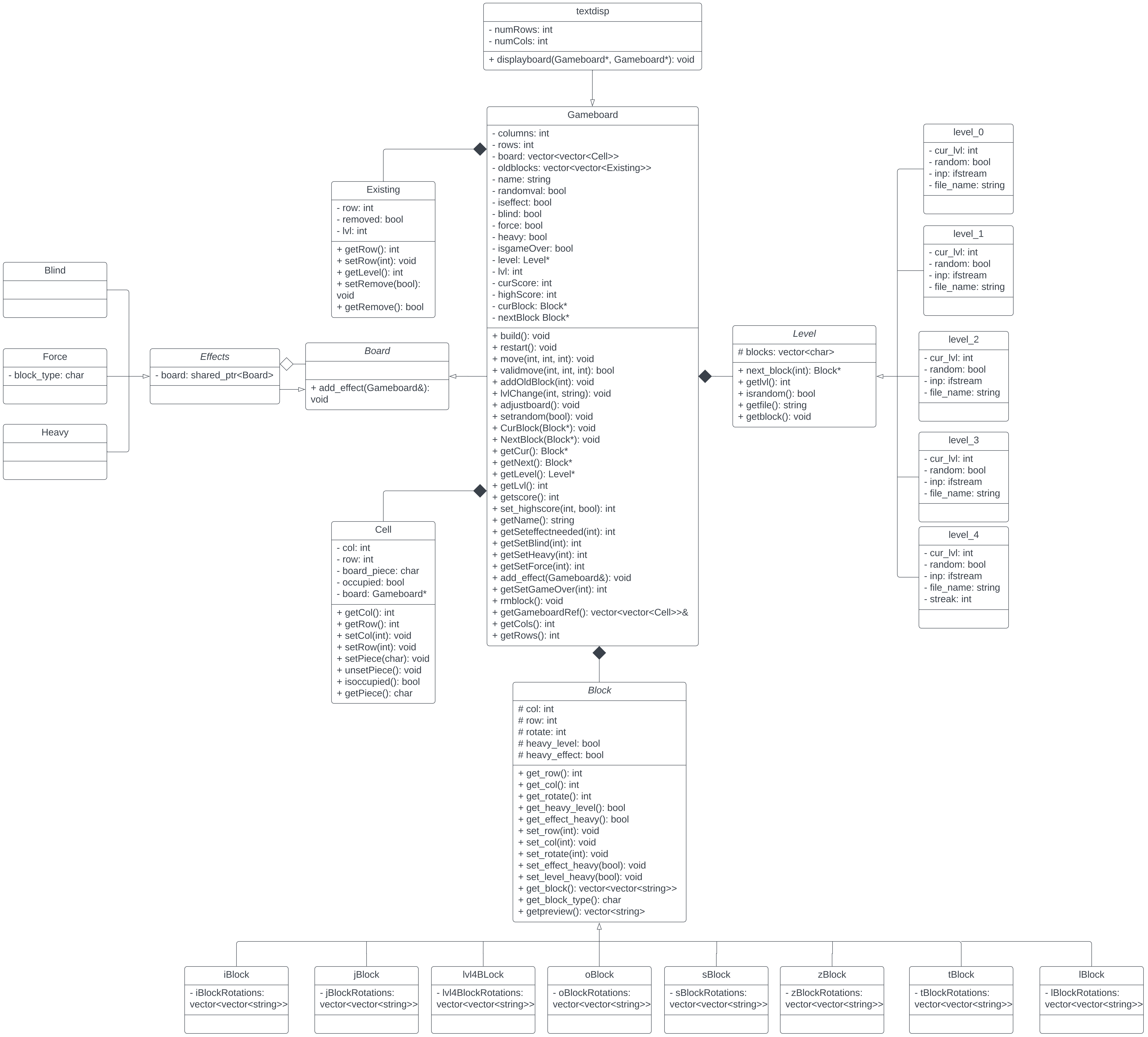
This game of Biquadris has five different levels ranging from 0-4, it contains 8 different blocks, and has 3 special effects that occur when a player clears a more than one line. These functionalities are implemented using five different essential classes, as well as two design patterns, namely, the decorator pattern and the factory method. The important classes mentioned consist of:

- Gameboard: This class controls the entire gameboards for each player. This includes the movement of blocks; applying effects when appropriate; updating the scores, level, and the board itself. These actions are performed with the help of methods such as:
    - build: Builds the gameboard in the beginning.
    - restart: Restarts the game by resetting the gameboard, but updating the high score if necessary.
    - move: Moves the block according to the player’s instructions.
    - validmove: Determines whether a block is allowed to be moved or rotated.
    - add_effect: Applies the effect on the opponent’s gameboard.

    - adjustboard: Adjusts the gameboard once a row is cleared after it is filled, while also updating the score.
    - set_highscore: Sets the high score.
    - lvlChange: Changes the level if necessary, and sets the *Level* class appropriately depending on random.
- Block: This class is used to access the blocks in play, as well as the upcoming blocks. Along with this, it contains all the different types of blocks that can be played.
    - set_effect_heavy: This applies the heavy effect on the current block to be played.
    - set_level_heavy: This applies the heavy effect depending on the level of difficulty the block is being played in.
    - get_block: Returns a vector containing all possible rotations of a block.
- Cell: This class is used to represent a block of space on the gameboard. Thus, our gameboard is essentially a vector of vector of Cells.
    - setPiece: Sets the cell to the current block type when necessary.
    - unsetPiece: Undoes the actions of set piece by ensuring the Cell is at its initial state (“.”).
    - isoccupied: Returns true if the cell is occupied by a block, and false otherwise.
- Level: This class gets the next block depending on the level it is in. Wraps the 5 levels of difficulty that the user can play in.
    - getblock: Returns the next block given the file provided to it.
    - next_block: Returns the block randomly if the random is set to be true. Else, it simply uses the functionality of getblock.
- Effects: This class is a decorator class that treats the *Board* class as the component. It is used to apply effects, when 2 or more lines are cleared at the same time.
    - add_effect: Adds effect in a linked list manner as taught in lectures.

Apart from this, there are also two other concrete classes that aid in our implementation:

- Existing: Contains all the Cells (see explanation above) that are currently occupied.
- textdisp: Prints the board along with the name of the players and their appropriate scores.


textdisp
- numRows: int
- numCols: int
+ displayboard(Gameboard*, Gameboard*): void
Gameboard
- columns: int
- rows: int
- board: vector<vector<Cell>>
- oldblocks: vector<vector<Existing>>
- name: string
- randomval: bool
- iseffect: bool
- blind: bool
- force: bool
- heavy: bool
- isgameOver: bool
- level: Level*
- lvl: int
- curScore: int
- highScore: int
- curBlock: Block*
- nextBlock Block*
+ build(): void
+ restart(): void
+ move(int, int, int): void
+ validmove(int, int, int): bool
+ addOldBlock(int): void
+ lvlChange(int, string): void
+ adjustboard(): void
+ setrandom(bool): void
+ CurBlock(Block*): void
+ NextBlock(Block*): void
+ getCur(): Block*
+ getNext(): Block*
+ getLevel(): Level*
+ getLvl(): int
+ getscore(): int
+ set_highscore(int, bool): int
+ getName(): string
+ getSeteffectneeded(int): int
+ getSetBlind(int): int
+ getSetHeavy(int): int
+ getSetForce(int): int
+ add_effect(Gameboard&): void
+ getSetGameOver(int): int
+ rmblock(): void
+ getGameboardRef(): vector<vector<Cell>>&
+ getCols(): int
+ getRows(): int
Existing
- row: int
- removed: bool
- lvl: int
+ getRow(): int
+ setRow(int): void
+ getLevel(): int
+ setRemove(bool): void
+ getRemove(): bool
Blind
Force
- block_type: char
Heavy
Effects
- board: shared_ptr<Board>
Board
+ add_effect(Gameboard&): void
Cell
- col: int
- row: int
- board_piece: char
- occupied: bool
- board: Gameboard*
+ getCol(): int
+ getRow(): int
+ setCol(int): void
+ setRow(int): void
+ setPiece(char): void
+ unsetPiece(): void
+ isoccupied(): bool
+ getPiece(): char
Level
# blocks: vector<char>
+ next_block(int): Block*
+ getlvl(): int
+ israndom(): bool
+ getfile(): string
+ getblock(): void
level_0
- cur_lvl: int
- random: bool
- inp: ifstream
- file_name: string
level_1
- cur_lvl: int
- random: bool
- inp: ifstream
- file_name: string
level_2
- cur_lvl: int
- random: bool
- inp: ifstream
- file_name: string
level_3
- cur_lvl: int
- random: bool
- inp: ifstream
- file_name: string
level_4
- cur_lvl: int
- random: bool
- inp: ifstream
- file_name: string
- streak: int
Block
# col: int
# row: int
# rotate: int
# heavy_level: bool
# heavy_effect: bool
+ get_row(): int
+ get_col(): int
+ get_rotate(): int
+ get_heavy_level(): bool
+ get_effect_heavy(): bool
+ set_row(int): void
+ set_col(int): void
+ set_rotate(int): void
+ set_effect_heavy(bool): void
+ set_level_heavy(bool): void
+ get_block(): vector<vector<string>>
+ get_block_type(): char
+ getpreview(): vector<string>
iBlock
- iBlockRotations: vector<vector<string>>
jBlock
- jBlockRotations: vector<vector<string>>
lvl4BLock
- lvl4BlockRotations: vector<vector<string>>
oBlock
- oBlockRotations: vector<vector<string>>
sBlock
- sBlockRotations: vector<vector<string>>
zBlock
- zBlockRotations: vector<vector<string>>
tBlock
- tBlockRotations: vector<vector<string>>
lBlock
- lBlockRotations: vector<vector<string>>

Resilience to Change:

There are many ways we can change the rules of this game or perhaps even add new ones to create more fun challenges. For example, we can add an additional effect, a higher level, or even introduce a new block. Now, it may seem that to modify even a little bit of the game, we would have to modify many parts of our code, but it is quite the contrary. As described in the overview, we have implemented many parts of this game using design patterns, thus, to modify rules and the game as described above, would as simple as adding an additional module to our code base. For instance, to introduce a new block, we would have to add just a few conditions in the effects and levels subclass and parent classes, and a new module in the Block parent class. To add an additional effect, we need to add an if statement in the main function and a subclass in the effects parent class, to introduce a higher level a subclass in the level parent class handles it. Overall, while we are making big changes to our game, the changes in our code itself are minimal, due to our low coupling and high cohesion.

Group Answers to Biquadris Questions:

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?

- To satisfy this condition, we can implement a new method in the *Block* class. This method can be called removeOld() and it will determine based on the current level of the game whether we should remove a block that has been there for more than 10 drops or not. To implement this function, we will add an additional vector of Block pointers, which will contain all the blocks that have already been played in the order of when they were played.
- Now, if the levels are not advanced once the size of this vector exceeds 10, then we will erase the element at index 0 of the vector. Along with that, we will also remove this block from the gameboard as well and shift all the blocks that are potentially above it downwards. This ensures that the size of our vector of block pointers never exceeds the length of 10, which also means that the maximum number of blocks that can be on the gameboard at one time is 10. Note, if the levels

are advanced, then the vectors elements are not removed even if they exceed length 10, to confine this condition.

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

- We can accomplish this by implementing the factory method design pattern. This is because, whenever we add a new level, we will simply add an additional subclass to the *Level* class. In this class, we will be able to override methods to cater to the needs of Level 5 or any additional level(s) the user wishes to add. By utilizing the factory method design pattern, file dependencies are minimized, and low coupling will minimize recompilation.

Question: 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?

- We can accomplish this by implementing the Decorator Design Pattern. To add additional effects, we can create subclasses of the *Decorator* class which in turn will allow us to use these effects without an else branch for every possible combination. This will be implemented in a similar way taught in class and used in a4q3b.

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 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.

- We create a commandSetter class that has methods for every new command and if one would like to add a new command, we would just add a new method to the class. Thus, only the commandSetter class will be recompiled without affecting the other classes. And if we want to change a command, we just change the if statement that checks for commands from the old command to the new command name.

Extra Credit Features:

As an extra feature, we decided to add another way of movement for the block. We have made it possible for the block to move upwards as well. This means that there are now 5 ways to move a block (right, left, down, drop, and up). Along with this, as taught in lectures, we also attempted to use unique_ptrs and smart_ptrs as an efficient of memory. Although we were to use it in all of our program, we tried our best to implement it wherever we could.

What this Project taught us?

This project allowed us to enhance our judgement making skills as well as our teamwork as a group of coders. For any given problem or functionality additions that arise, we were required to use our knowledge and intuition to choose an appropriate design pattern that will provide the best solution. This was a great skill to develop as it is very applicable to the real working environment and allows us to grow and build intuition as software developers. Additionally, we worked in a group of 3 and divided the workload accordingly. Although having a group member to work on the same task as you can alleviate a lot of the stress, it can bring many problems when it comes to coding similar for the similar projects. Everyone has their own unique style of coding, so coding as a group can at times become a challenge. This unique style goes from having different logical ideas to get to the same result, to having a different style of typing syntax. A simple example could be the usage of postfix notation vs prefix notation in loops, snake case vs camel case, etc. Although this might not seem like a big inconvenience, it can really confuse another individual that is not used to their style of coding. This was a big problem in the initial stages of coding, however, as we started to meet up, and had the chance to see the thought process of our group mates, we became more comfortable with each other’s styles and eventually our initial problem slowly eradicated.

What would we do differently if we started over?

The one thing that we all agreed to do differently if we started over was to start talking about the project and creating the UML earlier. Although it was not the case that we started last minute, however, due to the fact that there was another deadline for the same course on the same day, we underestimated how much time each part of the assignment would take. Since it was only the first deadline for the group project, we assumed it would not take us a lot of time to complete that, thus, we prioritized the assignment 4 deadline more and put a more effort into that. This limited the amount of time we spent to thoroughly plan out all the steps that we will be taking to successfully complete the project. Although we were still able to finish everything that was required and were satisfied with our work, it still felt like we had to rush to do it. Other than that, we all had been friends prior to this project, which helped a lot as we were all comfortable with each other from the start.