Biquadris: DD2

Dean Zhang (d29zhang), Leo Shi (l7shi), Hayden Lai (lh2lai)

Introduction:

Using the knowledge of object-oriented programming learned from class, we constructed Biquadris and implemented to our best every feature in the document. To make it more challenging, we decided to implement smart pointers to avoid memory management. Biquadris is a game that is similar to Tetris, but is turn-based and competitive. There are levels that the player can select, where lower levels are easier and higher levels are harder but reward more score.

Overview:

We separated our game into different classes and implemented the foundational classes (class block) and worked our way up to the most general (class game)

1. Block: We used an abstract class for blocks so that we could easily implement a decorator pattern to add more types of blocks to our game (eg: exploding blocks) for future implementations. A block contains the coordinates of its tiles in a vector of pairs, the character, and some special properties if it had some (eg: exploding Blocks). It also contained the level when it was created, so that when it is completely cleared, we would be able to calculate the amount of points obtained. It has methods to retrieve the data of its fields.

2. PlayerBlock: It is a class that contains the block that will be controlled and displayed on screen. It has coordinates of the tiles of the block, the coordinates of the bottom left corner and the top right corner to fit the block into a rectangle (used for rotation). The class contains elementary transformation methods for the block that will be eventually called through board. Finally, it contains accessors to retrieve the data of its fields

3. Level: The class Level contains all the properties of a level. It holds an int to denote the number of the level, a boolean to describe if the blocks are generated randomly and with that comes an array of odds for each block if it is random, and a sequence (map) /file of blocks if it is not random. Level has a field called cycle, which is a file that contains a sequence of blocks to be repeated. It has methods to generate the next block (based on the odds or the sequence), to create a new block, to level up/down, to reset level, or to change the randomness, in which case we would need to provide a file of a sequence of blocks if we change to “no random”.

4. Board: Board is the class where most of the stuff is processed. It has the 2-dimensional array that represents the board, which is useful in printing the screen as well as checking if a certain space is blocked off. In our scenario, we use a 2d char array for printing, which includes the PlayerBlock, while using a 2d bool array for checking if a certain space is blocked, which does not include the PlayerBlock. Board also has a vector of Blocks that compose the screen. This is useful for

calculating the score when all tiles of a block have been deleted. It contains all other methods to manipulate blocks, clear rows, count score/high score, keep track of the level etc.

5. Game: The class Game contains a vector of pointers to boards (in this case, we have 2 players so 2 boards) and is also a subject to observers (discussed in design). It also contains the Trie of commands where the default commands and new commands will be stored (added through "rename"). It is where the input commands are processed, which is called through board. It also has the update method to update itself after every command which notifies all its observers (text/graphics for display).

Other classes:

1. Trie: Trie is used for processing commands and is the root of TrieNodes. Given a word that will potentially be used as a command, it will search through the Trie to see if the word exists and execute whichever command that is associated with it. For example, the word "down" and the word "dow" would both be associated with the command "down". It also has the capacity to autocomplete a word and execute the respective command only if one autocompletion exists. Additionally, it only supports lowercase letters, so for the case of the commands "I", "J", etc, we converted the command lowercase first so it could be fit in the Trie.

2. TrieNode: They are the nodes of Trie.

3. Subject: In Design

4. Observer: In Design

5. Graphics, Text, Xwindows: To display our game in stdout or in a graphical display. We associated the respective color to each character (eg: a T-block is pink in typical tetris so a block with char "T" would display pink tiles). We also displayed the score and level of each player, as well as drawing the next block underneath the board.

Design:

Decorator pattern: The decorator pattern is used for blocks to decorate different types of blocks such as exploding blocks, blinking box, but we decided to omit it for the demo.

Observer pattern: The observer pattern is used for text and graphical screens. We have two observers, one text and one graphical, for each player.

MVC(Model View Controller): We implemented separate model and view, where model is board, view is the Observer and Game is the controller.

Accessors & Mutators: We used a lot of accessors to keep private fields safe. We kept Board::width public and knew it was safe since it is a const.

Iterator Pattern: We used iterator patterns when traversing through vectors and arrays of blocks, or through a word to traverse the nodes of our Trie.

Resilience to Change **(describe how your design supports the possibility of various changes to the program specification)**

Our code is very adaptable to change as a lot of our data structures are arrays that are either initialized with variables that you can easily change, or are vectors that can simply change in size. For example, we made it so that everything is dependent on some variables that can be changed later. For example, we generalised our game to be able to play against any number of players. We also attempted to separate each part of the program. For example, playerBlock is responsible for movement, Block is simply a container for coordinates and Level manages creating blocks. We also tried to use as many patterns as we could since they allow for easy understanding and better flexibility.

Answers to Question

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

   If we had more time, we would have done the Factory Pattern for generating new Blocks. We would have an Abstract class AbsLevel with a pure virtual method generateBlock() and a normal method generateRandom(). generateRandom() would work very similarly to our implementation but instead of storing all the probabilities in a map, we have each Level have an array of NUM_BLOCKS where we store the probabilities in CNF to allow memory easy random block generation. As for generateRandom(), each Level will have their own version of it since they have their own quirks such as Level 3 and 4 being capable of taking blocks from a file in the middle of the game. This allows minimum recompilation since we would simply have to create a new subclass of AbsLevel and compile it.

2. **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?**
   To allow multiple effects, we made each effect have their own method, thus if we wanted to apply multiple effects on one board, we would just call each method and each method would apply the effect separately. We also used booleans (for generality, we could have used integers denoting the number of turns it lasts) to tell the board that it has been affected. If the boolean is true, the effect is applied on the display, on the Board, or somewhere else, depending on whether it is a visual effect or affecting playerBlock or the Board itself. To add more effects, you simply have to create a method for that effect and modify Board through booleans or directly manipulating some fields so that it can react accordingly. We do not need one else branch for every single combination, because the effects are applied simultaneously and independently.

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?**
   Given a function that we want to implement as a command, all we need to do is to add a case for it in game.cc that calls the function, then add the command name into the trie of commands. To rename an existing command, we can use the function addCommand for the Trie. This function adds the nodes of the word in the Trie if they don't exist, and then set command name to the word. For example, using "rename rc clockwise" will add the nodes "r" then "c" into our Trie and change the command to "clockwise" at the node "c". If we wish to rename rc again as such "rename rc counterclockwise", then through the nodes "r" and "c", it will path in the Trie to the node "c" and change the command name to "counterclockwise". If we wish to delete the old name, simply overwriting the old name with a dummy such as the empty string command will suffice. It is very simple to implement, and should not take more than 3-4 lines.

4. **How difficult would it be to adapt your system to support a command whereby a user could rename existing commands ?**
   We already have this implemented so there is little adaptation. "rename x y" makes x an alias of y, which means typing x and pressing enter will do the same thing as typing y and pressing enter. Upon lookup on the trie, it will return a string that is the same as itself for normal, non-user-defined commands, but for user-defined commands, it will be the complete form of the shortcut. Thus, by looking through the tree, we can know what each shortcut means.

5. **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.**

Following from the previous question, if y is a sequence of commands instead, we can store y in the TrieNode of x. Then, when we look up the command of the TrieNode representing x, we can find a space-delimited sequence of commands, which we can then execute with an istringstream.

Extra Credit Features

We implemented smart pointers to avoid having to delete dangling pointers. std::unique_ptr automatically does all freeing for us once it is out of scope and std::shared_ptr does the same when all pointers to the same object go out of scope We also implemented ExplodingBlock, a decorator of Block, but it is not actually used in the demo. ExplodingBlock is a block that will delete itself after 10 turns. We also the ability to rename commands and play with any number of players simply by changing the numPlayers constant. We can also easily change the size of the Board, or add new shapes of Blocks by simply adding them in the map of Blocks where we store the default coordinates for each block..

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?**

   Working in a team requires a lot of communication and organization to make sure that everyone is on the same page. For example, some of us had different coding styles so our code would look slightly different. In addition, sharing an idea that has many layers of complexity was challenging since it requires the sustained focus of everyone and often leading to a deadend. Oftentimes, we had to check on one another to make sure that everything was on track since once we start being behind, it becomes increasingly difficult to keep up as we cycle through many ideas that might stack on each other. Also having different opinions can be difficult sometimes, but it can also help us determine a more efficient and simple solution. We often changed ideas of the implementation of our classes which explains why we progressed slowly in the beginning.

   It was also a struggle to find time to work together since we had different schedules. As we are nearing the finals period, we are sometimes busy with preparing for our exams and finishing our coursework. We learned to accommodate our fellow group members and set a schedule for focusing on work.

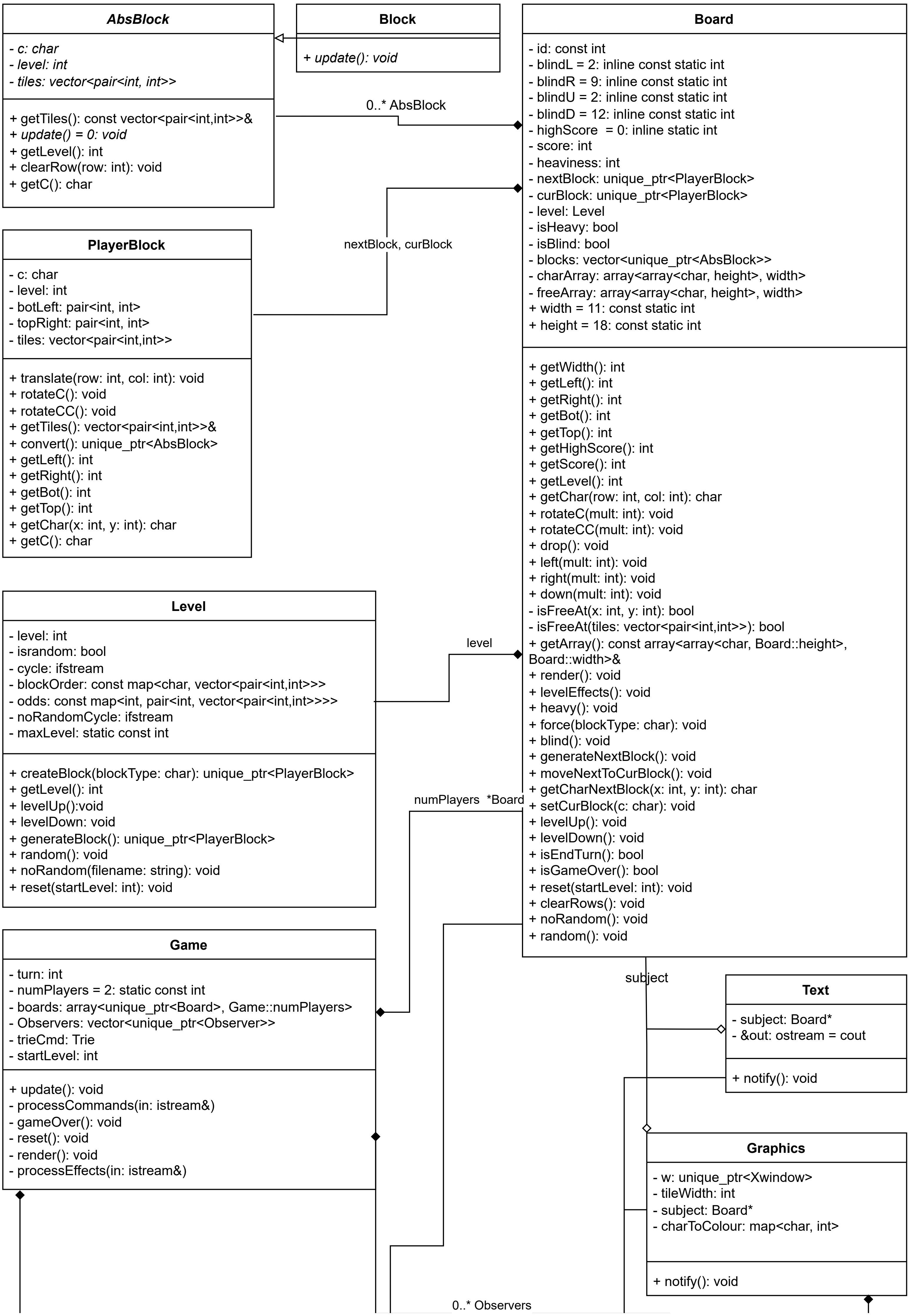
2. **What would you have done if you had the chance to start over?**

   We would plan it more meticulously and make sure to stick to the plan. Since we used LiveShare on VsCode, it was necessary for all 3 of us to be there in order to work. Also the connection to LiveShare was not great, we would frequently disconnect, or our scripts may sometimes not be in sync.

We would also try to do things ahead of time by scheduling a certain number of hours to work on it everyday. Since many of us were busy, we did not realize until a few days before the deadline that it was going to be close, and we needed to pull an all-nighter to finish the work. This would make it so that we would have more time thinking of different concepts and clever solutions that would have made our lives easier when programming as we had to modify our code multiple times after discovering a detail that we originally missed.

Conclusion:

Overall, the project was long and required a lot of problem-solving. cooperation, and communication to succeed. This assignment has deepened our knowledge of C++ programming, design patterns, and object-oriented development. It was also a great opportunity for us to experience the feeling of working in the industry where multiple people work on one product as we enter our coop semester. We are very proud of our final product that we spent many hours and days planning, writing, and debugging.



```
AbsBlock
- c: char
- level: int
- tiles: vector<pair<int, int>>
+ getTiles(): const vector<pair<int,int>>&
+ update() = 0: void
+ getLevel(): int
+ clearRow(row: int): void
+ getC(): char
```

```
Block
+ update(): void
```

```
Board
- id: const int
- blindL = 2: inline const static int
- blindR = 9: inline const static int
- blindU = 2: inline const static int
- blindD = 12: inline const static int
- highScore  = 0: inline static int
- score: int
- heaviness: int
- nextBlock: unique_ptr<PlayerBlock>
- curBlock: unique_ptr<PlayerBlock>
- level: Level
- isHeavy: bool
- isBlind: bool
- blocks: vector<unique_ptr<AbsBlock>>
- charArray: array<array<char, height>, width>
- freeArray: array<array<char, height>, width>
+ width = 11: const static int
+ height = 18: const static int
+ getWidth(): int
+ getLeft(): int
+ getRight(): int
+ getBot(): int
+ getTop(): int
+ getHighScore(): int
+ getScore(): int
+ getLevel(): int
+ getChar(row: int, col: int): char
+ rotateC(mult: int): void
+ rotateCC(mult: int): void
+ drop(): void
+ left(mult: int): void
+ right(mult: int): void
+ down(mult: int): void
- isFreeAt(x: int, y: int): bool
- isFreeAt(tiles: vector<pair<int,int>>): bool
+ getArray(): const array<array<char, Board::height>,
Board::width>&
+ render(): void
+ levelEffects(): void
+ heavy(): void
+ force(blockType: char): void
+ blind(): void
+ generateNextBlock(): void
+ moveNextToCurBlock(): void
+ getCharNextBlock(x: int, y: int): char
+ setCurBlock(c: char): void
+ levelUp(): void
+ levelDown(): void
+ isEndTurn(): bool
+ isGameOver(): bool
+ reset(startLevel: int): void
+ clearRows(): void
+ noRandom(): void
+ random(): void
```

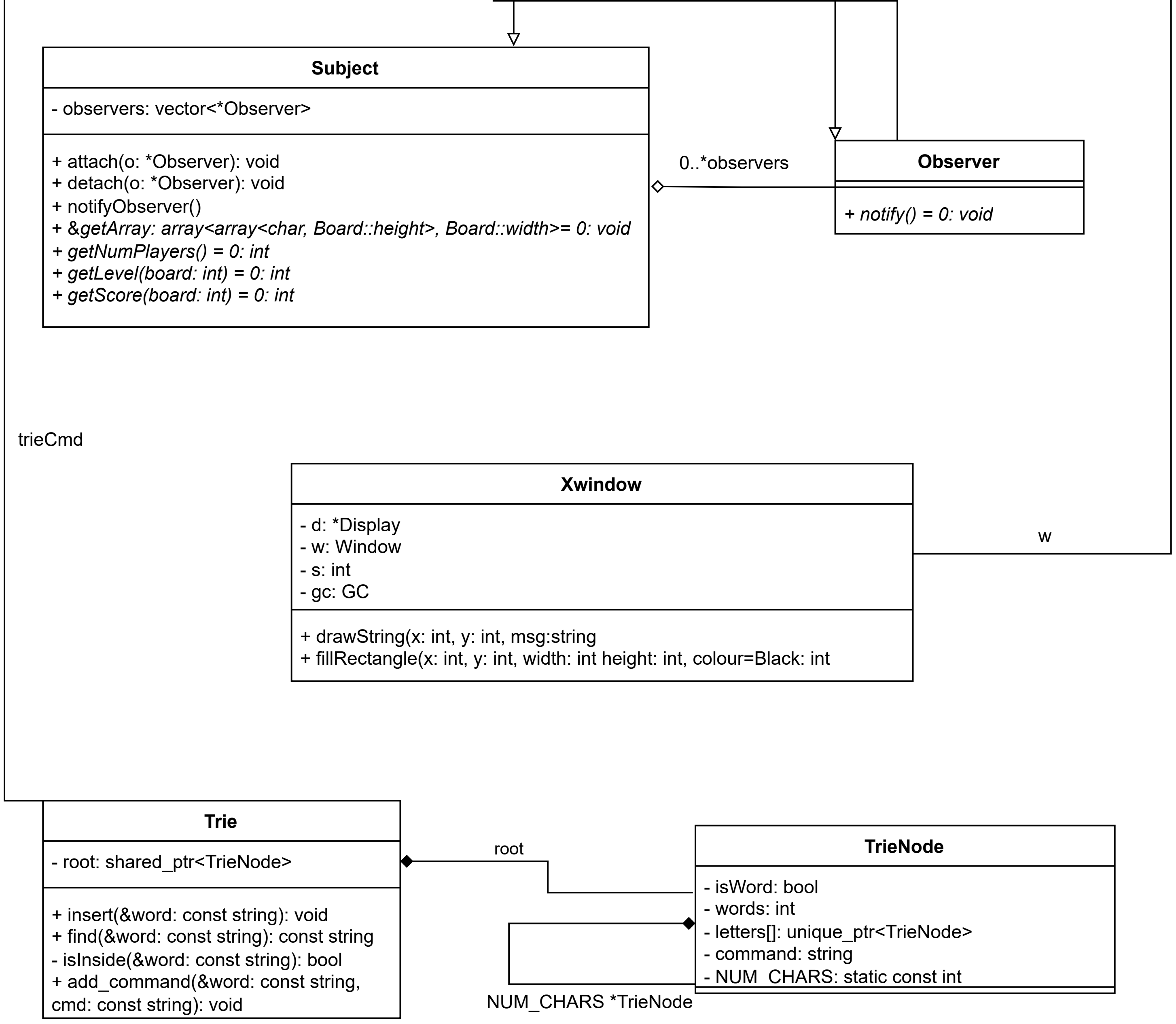
```
PlayerBlock
- c: char
- level: int
- botLeft: pair<int, int>
- topRight: pair<int, int>
- tiles: vector<pair<int,int>>
+ translate(row: int, col: int): void
+ rotateC(): void
+ rotateCC(): void
+ getTiles(): vector<pair<int,int>>&
+ convert(): unique_ptr<AbsBlock>
+ getLeft(): int
+ getRight(): int
+ getBot(): int
+ getTop(): int
+ getChar(x: int, y: int): char
+ getC(): char
```

```
Level
- level: int
- israndom: bool
- cycle: ifstream
- blockOrder: const map<char, vector<pair<int,int>>>
- odds: const map<int, pair<int, vector<pair<int,int>>>>
- noRandomCycle: ifstream
- maxLevel: static const int
+ createBlock(blockType: char): unique_ptr<PlayerBlock>
+ getLevel(): int
+ levelUp():void
+ levelDown: void
+ generateBlock(): unique_ptr<PlayerBlock>
+ random(): void
+ noRandom(filename: string): void
+ reset(startLevel: int): void
```

```
Game
- turn: int
- numPlayers = 2: static const int
- boards: array<unique_ptr<Board>, Game::numPlayers>
- Observers: vector<unique_ptr<Observer>>
- trieCmd: Trie
- startLevel: int
+ update(): void
- processCommands(in: istream&)
- gameOver(): void
- reset(): void
- render(): void
- processEffects(in: istream&)
```

```
Text
- subject: Board*
- &out: ostream = cout
+ notify(): void
```

```
Graphics
- w: unique_ptr<Xwindow>
- tileWidth: int
- subject: Board*
- charToColour: map<char, int>
+ notify(): void
```


Subject
- observers: vector<*Observer>
+ attach(o: *Observer): void
+ detach(o: *Observer): void
+ notifyObserver()
+ &getArray: array<array<char, Board::height>, Board::width>= 0: void
+ getNumPlayers() = 0: int
+ getLevel(board: int) = 0: int
+ getScore(board: int) = 0: int
0..*observers
Observer
+ notify() = 0: void
trieCmd
Xwindow
- d: *Display
- w: Window
- s: int
- gc: GC
+ drawString(x: int, y: int, msg:string
+ fillRectangle(x: int, y: int, width: int height: int, colour=Black: int
w
Trie
- root: shared_ptr<TrieNode>
+ insert(&word: const string): void
+ find(&word: const string): const string
- isInside(&word: const string): bool
+ add_command(&word: const string, cmd: const string): void
root
TrieNode
- isWord: bool
- words: int
- letters[]: unique_ptr<TrieNode>
- command: string
- NUM_CHARS: static const int
NUM_CHARS *TrieNode