# Plan of Attack(Biquadris)

John Zhu  Daniel Ni  Charles Zhang

## Plan:

Breakdown of the project:

1. The basic build up for the game, including the board, with any blocks available, with suitable output for the initialization of the board and certain function realization such as show the first block.
2. With the basis of 1, we will implment Level 0 with test inputs, at this stage, our program can show blocks of our input and move up and down within the board.
3. Implement all the functions in the board, clear rows when needed, clear blocks when cleared, return number for scoring. At this stage, the blocks appear in the sequence of input file. Basic functions for Biquadris are all realized, with proper output with scoring. Proper prompts if game is over and can restart the game.
4. Implment all the level decorators with special action heavy. At this stage, our program should be able to function with almost all the details in the description file.
5. The rest details to implement are rename command, macro command names, special action force, randomness change in Level 3+.

Estimated time for each

1. Around 3.28
2. Around 3.30
3. Around 4.2

4 and 5 can carry on simultaneously, they should be finished before 4.3. Left with 2+ days to test can find problems in detail.

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: Our design includes a function to check the “validness” of rows. If certain rows are not valid, it will be cleared from the “board”, and this allows for more blocks to fall. The validness check take place before the generation of new blocks. So I don’t think it will be a problem.

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

   Ans: We would design our program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation by coding with low coupling and high cohesion. Every level will be of low coupling and high cohesion such that each of them will not affect each other. We plan to use decorator pattern to implement the level part. In this way, adding new levels will do not affect existing levels.

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?

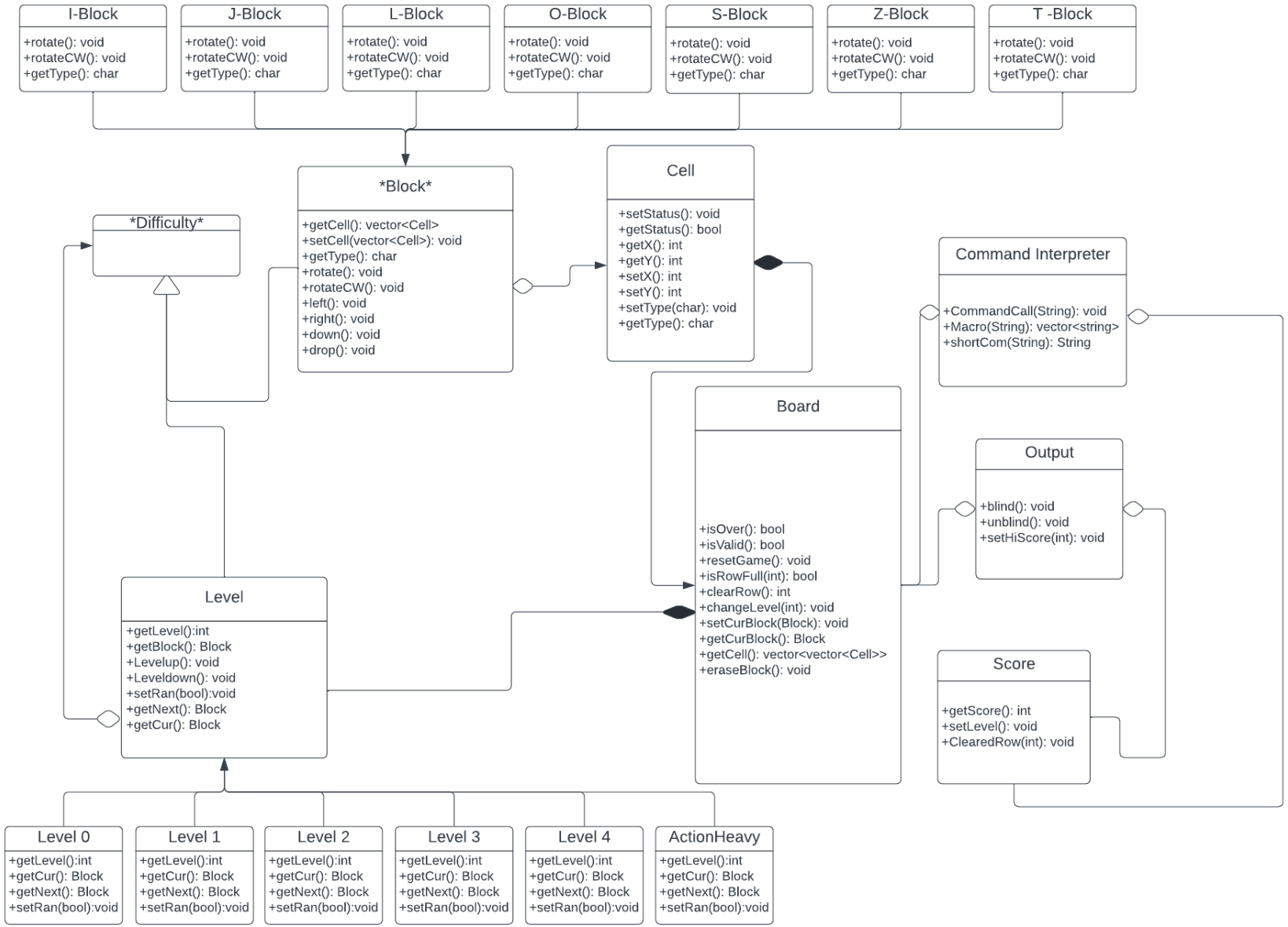
   Ans: We plan to use decorator pattern to avoid having one else-branch for every possible combination. For any new effects, we can build a new decorator and applied to the game if needed. We don’t

need else-branch for every possible combination because we decide to apply decoration to objects only when needed.

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.

   Ans: We can adopt hash map to store the user-defined command pointing to the same corresponding standard command name (string). When rename is needed, we can update the user-defined command to support the change. We can adopt hash map, key is the defined “macro” (string) pointing to the a series of corresponding commands as type of vector<string>.

(UML on the next page)


I-Block
+rotate(): void
+rotateCW(): void
+getType(): char
J-Block
+rotate(): void
+rotateCW(): void
+getType(): char
L-Block
+rotate(): void
+rotateCW(): void
+getType(): char
O-Block
+rotate(): void
+rotateCW(): void
+getType(): char
S-Block
+rotate(): void
+rotateCW(): void
+getType(): char
Z-Block
+rotate(): void
+rotateCW(): void
+getType(): char
T -Block
+rotate(): void
+rotateCW(): void
+getType(): char
*Block*
+getCell(): vector<Cell>
+setCell(vector<Cell>): void
+getType(): char
+rotate(): void
+rotateCW(): void
+left(): void
+right(): void
+down(): void
+drop(): void
Cell
+setStatus(): void
+getStatus(): bool
+getX(): int
+getY(): int
+setX(): int
+setY(): int
+setType(char): void
+getType(): char
*Difficulty*
Command Interpreter
+CommandCall(String): void
+Macro(String): vector<string>
+shortCom(String): String
Board
+isOver(): bool
+isValid(): bool
+resetGame(): void
+isRowFull(int): bool
+clearRow(): int
+changeLevel(int): void
+setCurBlock(Block): void
+getCurBlock(): Block
+getCell(): vector<vector<Cell>>
+eraseBlock(): void
Output
+blind(): void
+unblind(): void
+setHiScore(int): void
Level
+getLevel():int
+getBlock(): Block
+Levelup(): void
+Leveldown(): void
+setRan(bool):void
+getNext(): Block
+getCur(): Block
Score
+getScore(): int
+setLevel(): void
+ClearedRow(int): void
Level 0
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void
Level 1
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void
Level 2
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void
Level 3
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void
Level 4
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void
ActionHeavy
+getLevel():int
+getCur(): Block
+getNext(): Block
+setRan(bool):void