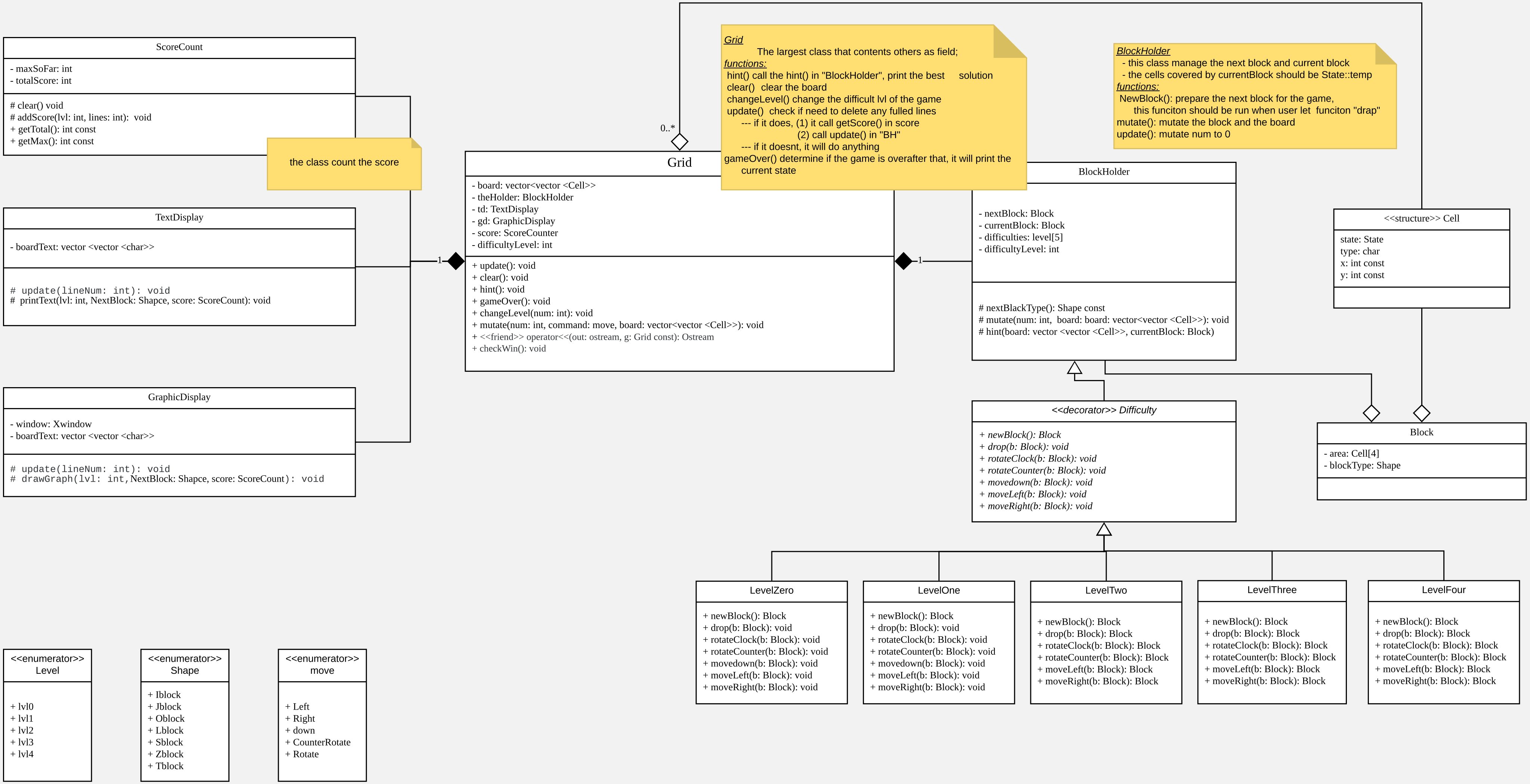

ScoreCount
- maxSoFar: int
- totalScore: int
clear() void
addScore(lvl: int, lines: int): void
+ getTotal(): int const
+ getMax(): int const
the class count the score
TextDisplay
- boardText: vector <vector <char>>
update(lineNum: int): void
printText(lvl: int, NextBlock: Shapce, score: ScoreCount): void
GraphicDisplay
- window: Xwindow
- boardText: vector <vector <char>>
update(lineNum: int): void
drawGraph(lvl: int,NextBlock: Shapce, score: ScoreCount): void
0..*
1
Grid
- board: vector<vector <Cell>>
- theHolder: BlockHolder
- td: TextDisplay
- gd: GraphicDisplay
- score: ScoreCounter
- difficultyLevel: int
+ update(): void
+ clear(): void
+ hint(): void
+ gameOver(): void
+ changeLevel(num: int): void
+ mutate(num: int, command: move, board: vector<vector <Cell>>): void
+ <<friend>> operator<<(out: ostream, g: Grid const): Ostream
+ checkWin(): void
Grid
The largest class that contents others as field;
functions:
hint() call the hint() in "BlockHolder", print the best solution
clear() clear the board
changeLevel() change the difficult lvl of the game
update() check if need to delete any fulled lines
--- if it does, (1) it call getScore() in score
(2) call update() in "BH"
--- if it doesnt, it will do anything
gameOver() determine if the game is overafter that, it will print the current state
1
BlockHolder
- nextBlock: Block
- currentBlock: Block
- difficulties: level[5]
- difficultyLevel: int
nextBlackType(): Shape const
mutate(num: int, board: board: vector<vector <Cell>>): void
hint(board: vector <vector <Cell>>, currentBlock: Block)
BlockHolder
- this class manage the next block and current block
- the cells covered by currentBlock should be State::temp
functions:
NewBlock(): prepare the next block for the game,
this funciton should be run when user let funciton "drap"
mutate(): mutate the block and the board
update(): mutate num to 0
<<structure>> Cell
state: State
type: char
x: int const
y: int const
<<decorator>> Difficulty
+ newBlock(): Block
+ drop(b: Block): void
+ rotateClock(b: Block): void
+ rotateCounter(b: Block): void
+ movedown(b: Block): void
+ moveLeft(b: Block): void
+ moveRight(b: Block): void
Block
- area: Cell[4]
- blockType: Shape
LevelZero
+ newBlock(): Block
+ drop(b: Block): void
+ rotateClock(b: Block): void
+ rotateCounter(b: Block): void
+ movedown(b: Block): void
+ moveLeft(b: Block): void
+ moveRight(b: Block): void
LevelOne
+ newBlock(): Block
+ drop(b: Block): void
+ rotateClock(b: Block): void
+ rotateCounter(b: Block): void
+ movedown(b: Block): void
+ moveLeft(b: Block): void
+ moveRight(b: Block): void
LevelTwo
+ newBlock(): Block
+ drop(b: Block): Block
+ rotateClock(b: Block): Block
+ rotateCounter(b: Block): Block
+ moveLeft(b: Block): Block
+ moveRight(b: Block): Block
LevelThree
+ newBlock(): Block
+ drop(b: Block): Block
+ rotateClock(b: Block): Block
+ rotateCounter(b: Block): Block
+ moveLeft(b: Block): Block
+ moveRight(b: Block): Block
LevelFour
+ newBlock(): Block
+ drop(b: Block): Block
+ rotateClock(b: Block): Block
+ rotateCounter(b: Block): Block
+ moveLeft(b: Block): Block
+ moveRight(b: Block): Block
<<enumerator>> Level
+ lvl0
+ lvl1
+ lvl2
+ lvl3
+ lvl4
<<enumerator>> Shape
+ Iblock
+ Jblock
+ Oblock
+ Lblock
+ Sblock
+ Zblock
+ Tblock
<<enumerator>> move
+ Left
+ Right
+ down
+ CounterRotate
+ Rotate

WORK DELEGATION:

CHRIS CHEN: UML elementary design, window display, graphic display frameworks with implementation, interface and header files elementary design.

YUFENG LI: Block section implementation, level section partially implementation, UML redesign, difficulty section implementation, interface redesign.

TINGDA DU: Redesign for the implementation's pattern, extraneous features design, level section partially implementation, score section design, interface redesign.

The completion date (expected schedule)	
July 15th	**Complete .h files, UML, partially implementation for the framework.**
July 17th	**Complete block section totally and fix the final framework**
July 19th	**Complete most of difficulty, level and score section**
July 21th	**Plan, meet, and implement extraneous features**
July 23th	**Debugging, testing, and adjust the display and GUI**
July 24th	**Finalize & final documentation and prepare to submit**
July 25th	**Complete everything and submit, done!**

CS 246 Assignment 5 Project Questions

Question 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?

Answer:

As seen in the UML diagram, each level of difficulty is an instance of the `Difficulty` decorator abstract superclass, and has its own `drop()` method. Hence, in order to allow for some generated blocks to disappear from the screen when less than 10 blocks have fallen for a particular level of difficulty, a solution would be to pass an instance of the `Grid` into the decorator associated with that level of difficulty. Then, in the `drop()` method of the selected level of difficulty, the decorator could modify the provided instance of the Grid, thus allowing such a rule to be easily added and confined to certain levels of difficulty.

— ■

Question 2

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

Answer:

As observed in the UML class diagram, the `Difficulty` abstract superclass serves as a decorator for a Block as it specifies how an action such as rotation, dropping or moving affects an existing `Block`. Moreover, `Difficulty` has a concrete implement for each level of difficulty. Hence, the introduction of an additional level into the system could be accomplished by the adding of a new subclass for the `Difficulty` abstract class. Subsequently, in the `BlockHolder` class, the newly-added level may be integrated into the game by adding a new entry to the `difficulties` array.

By doing so, recompilation is limited to a minimum when a new difficulty level is added into the game.

— ■

Question 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? (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.

Answer:

In the `main.cc` of the project, for each command, there is to be a global vector containing the aliases of each command. As an example, aliases of the command "`rotate`" may be stored in `vector<string> rotateAliases`. Moreover, there is to be a command interpreter function, say `interpret(string cmd)` that interprets a given command.

Thus, whenever a command is entered, the `interpret` function will be invoked directly on the entered command. Afterwards, the `interpret` function would search through each of the alias vectors, and once a vector is found to contain the entered command as its member, the appropriate function associated with the command would be invoked.

Thus, in order to include a new feature, the necessary modifications would first have to be made to one or more of the object classes illustrated in the UML class diagram. Subsequently, in the `main.cc` of the project, a new alias vector would be created containing the aliases of the command associated with the new feature, and the `interpret` function would be appropriately altered to accommodate detecting an alias of the command associated with the new feature.

The advantage of such a system is that adding a new alias to an existing command amounts to simply adding an entry to the appropriate vector containing aliases for a command.

To support a "macro" language where name is given to a sequence of commands, `main.cc` would also contain a global variable that is an `unordered_map` (i.e. hash map) that maps each defined macro to a sequence of commands separated by spaces. Thus, to invoked a command defined by a macro, the `interpret` function would query the hash map to obtain the sequence of space-separated commands associated with that particular macro-defined command. Subsequently, the `interpret` function would recursively invoke itself on each of the space-separated commands that defines the macro. Likewise, adding a new macro amounts to adding a new key-value mapping to the hash map, and thus required minimal recompilation and may be done at runtime as macros do not add any new gameplay feature.

— ■