Our groups have two members, Yixing Zhang and Yilin Mo. The following shows our plan for this project. It specifically indicates which person is responsible for which parts of the projects and our goals for completion for different dates. Our group is going to make the Quadris game.

Step 1 (July 25): We have an online meeting for about 1 hour to determine which project we are going to work on and discuss the general plan for the project.
Step 2 (July 26): we have an online meeting for about 2 hours to determine how we are going to implement the program. This includes the programming patterns and structures that might be useful for the program. After the discussion, Yilin Mo makes the UML class diagram based on the idea that we came up with during this meeting. We also decide which person is responsible for which parts of the projects in this meeting.
Step 3 (July 28): Yixing completes Q2, Q3 and Q4 for DD1. Yilin completes the UML class diagram and Q1 for DD1. Submit both documents on July 28 for grading.
Step 4 (July 30): Yixing implements the main function and uploads them to GitHub. Yilin implements the interface for structures, Cell and Grid.
Step 5 (Aug 1): Yixing writes the implementation for functions “void play()” and the function for finding the next block according to “level”. Yilin writes the implementation for the method functions for the command interpreter, such as “void left()”.
Step 6 (Aug 3): Yixing implements a function for scoring and Yilin implements functions for applying special actions.
Step 7 (Aug 4): Each group member comes out with 10 tests and records the results.
Step 8 (Aug 5): We will have a meeting on Aug 5 and debug our code together to make sure all 20 tests will pass.
Step 9(Aug 10): Complete the final design document. Yixing works on four parts of the design document, overview, design, resilience to change and extra credit features. Yilin works on three parts of the design document, answers to questions, final questions, and conclusion.
Step 9(Aug 12): Make any last-minute change and submit the project.

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?

   As displayed in the UML diagram, each BlockCell object has a field of type vector<BlockCell *> called friends. We create every block at the default position in the grid by function setDefault(). To achieve this, we simply turn every BaseCell objects at the block’s default position into BlockCell objects of specific type using the Decorator design pattern. Then, for every newly created BlockCell object, we store the pointers of the other 3 BlockCell objects which belong to the same block in the variable friends. When we may need to delete a row in function compScore(), the BlockCell objects are deleted. Before they are deleted, we access their “friends” by the vector friends, and for every “friend” we change the pointer of the object that is going to be deleted to nullptr in the vector friends.

   For example, we created an O-block by setDefault(). Suppose the block consists of 4 BlockCell objects called A, B, C, and D. Then after setDefault(), the fields of A, B, C, and D will be like:
   A:
     type: O   friends: pointer of B, pointer of C, pointer of D
   B:
     type: O   friends: pointer of A, pointer of C, pointer of D
   C:
     type: O   friends: pointer of A, pointer of B, pointer of D
   D:
     type: O   friends: pointer of A, pointer of B, pointer of C

And suppose that the block is like:
A B
C D

Then if the row that C and D are in has no gap anymore, and we need to delete it by function compScore(), we first delete C and then D. Before deleting C, we access A, B, and D by friends, and change every pointer to C in A's friends, B's friends, and D's friends to nullptr. The same operation happens before deleting D. After C is deleted, the fields are like:
A:
  type: O   friends: pointer of B, ~~(pointer of C)~~->nullptr, pointer of D
B:
  type: O   friends: pointer of A, ~~(pointer of C)~~->nullptr, pointer of D
D:
  type: O   friends: pointer of A, pointer of B, ~~(pointer of C)~~->nullptr

After D is deleted, the fields are like:
A:
  type: O   friends: pointer of B, nullptr, ~~(pointer of D)~~->nullptr
B:
  type: O   friends: pointer of A, nullptr, ~~(pointer of D)~~->nullptr
In addition, every BlockCell object has a function called checkDelete(). This function checks if all pointers in friends have been deleted. If yes, it returns true; otherwise, it returns false.

To add the feature described in the question, we can add an int data field called roundnum and set it to -1 by default. In every setDefault() call, after we create the BlockCell objects, we add 1 to the round num of every BlockCell object in the grid, including the newly created ones. Then we loop through the entire grid to find the BlockCell objects with a round num equal to 10. Then we delete them one by one, and fill their positions with new BaseCell objects.

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

In our Grid structure, we have a variable, "level". Then, any method function in the grid structure can change the "level" when necessary. Also, when we determine the block coming next, we use an if-else if statement based on the "level". We would have a code like below.

```
if (level == 1) {
        ……
        next = …..;
        …..
} else if (level == 2) {
        …….
} else if (level == 3) {
        …….
}
……
```

If there is an additional level being introduced to the system, we could simply add another else-if + condition to the code shown above. For example, if we want to insert Level 5, we could add the following code.

```
else if (level == 5) {
        …….
}
```

We find the next coming block at the end of each turn. When we satisfy the condition for "level == x" where x is a natural number, we run the program within the bracket to find the type of the next block.

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?

To implement a program such that every time when the player clears two or more rows simultaneously, one or more actions could be called, we could use the while-loop. We implement the Grid structure like the following, which include the boolean variate if_blind, if_heavy, and if_force.

```
struct Grid{
        vector<Cell*>;
        bool  if_heavy;
        bool  if_blind;
        bool  if_force;
        ……
        };
```

The initial value of if_heavy, if_blind and if_force are set to be false for both grid A and grid B. When a new block is being added to board A, we will check if any two or more rows are deleted. If there is, we will have a while-loop to check any special action inputted by player A. For example, if player A calls blind and heavy, the if_blind and if_heavy will be set to false for board B. Then, when it is Player B's turn, we will first check if any of if_heavy, if_blind, and if_force is equal to true. We will apply the corresponding action while displaying board B and moving the block. For example, whenever we are moving the block left or right, we will check if the if_heavy is true. Also, whenever we are displaying the board, we will check if the if_blind is true. At the end of player B's term, we will reset all these three boolean variates to be false. In this case, our program supports us to record the multiple actions that should be applied to the board and apply them simultaneously.

We can invent more kinds of effects by simply adding corresponding boolean variate to the Grid structure. For example, to add a special effect, super_heavy such that whenever a player moves a block left or right, the block automatically falls by three rows after the horizontal movement, we add the boolean variate if if_super_heavy to the Grid structure. Then, when we find that two or more rows are deleted while a new block is being added to board A, we use the while-loop to check any special action inputted by player A. If player A inputs super_heavy, the if_super_heavy is set to be true for grid B. Then, when a block is added to broad B and the if_super_heavy for grid B is equal to true, we will let the block fall by three rows after every horizontal move.

Since I am not placing different special actions into combinations, there will not be any else-branch for combinations. We record any special actions that one player wants to apply to another player. We know that each action is applied at different points in the process. For example, the special effect, blind, is applied while displaying the board; and the special effect, heavy, is applied while moving the block horizontally. Then, we are not analyzing the special action in combination but individually; therefore, we will not have an else-branch.

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 complication? How difficult would it be to adapt your system to support a command whereby a user could rename

existing commands? How might you support a “macro” language, which would allow you to give a name to a sequence of commands?

When a new command name is added, I will add a method function to the Grid structure to implement this command. Since the method function is in the Grid structure and the vector of the pointer of the cell is also in the Grid structure, we can modify the board, which is the vector of the pointer of the cell by directly calling the method function. For example, we want to add the command, most_left which moves the current block to the left as far as possible until it comes into contact with either the left of the board or a block. In this case, we add the function “void most_left()” into the Grid structure. When the player input the command “most_left”, we then call this function and we modify the grid vector in this function. All the command names are implemented as method functions in the Grid structure. Then, when we want to change an existing command name, we could simply go to its corresponding method function and make changes to the functions.

It would not be difficult for the user to rename an existing command in the system. All the strings in this program should be stored using string variates. Then, when we want to rename a certain existing command, we could simply change the value of the string variable and all appearance of this string in the program would be changed. For example, we would implement the existing command, “left” as follows.

```
string c_left = “left”;
……
// location (1)
if (s == c_left) {
……
}
……
// location (2)
if (s1 != c_left) {
……
}
……
```

In this case, if we want to rename the existing command, “left” to “move_left”, we could do so by signing c_left to “move_left”, string c_left = “move_left”. Then, the name of the commands is changed at both location(1) and location(2).

If I want to give a name to a sequence of commands, I can implement a new method function in the Grid structure and call the corresponding function for each command in the sequence in this new function. For example, if I want to use the name “left_right” to represent the sequence of commands, “left” and “right”. Then, we add the function “void left_right()” in the Grid structure and we program this function as below.

```
void left_right() {
        void left();
        void right();
}
```