Project Breakdown:

Since Quatris is a large project, it requires the collaboration of all three team members in order to meet the deadline.

We decomposed the project to smaller milestones based on our UML diagram, in order to tackle the challenge one step at a time, initiation, component building, component linking, testing and polish up. For some stages, there are multiple tasks to be completed, we've assigned each of them to one of the team members to wrok on.

We estimated equal work loads across our team members to minimize the stress for everyone. Below is a detailed responsibility chart with our internal deadlines to follow. The projected finish date will be July 15th, and we will have one extra day to do the final fine tuning.

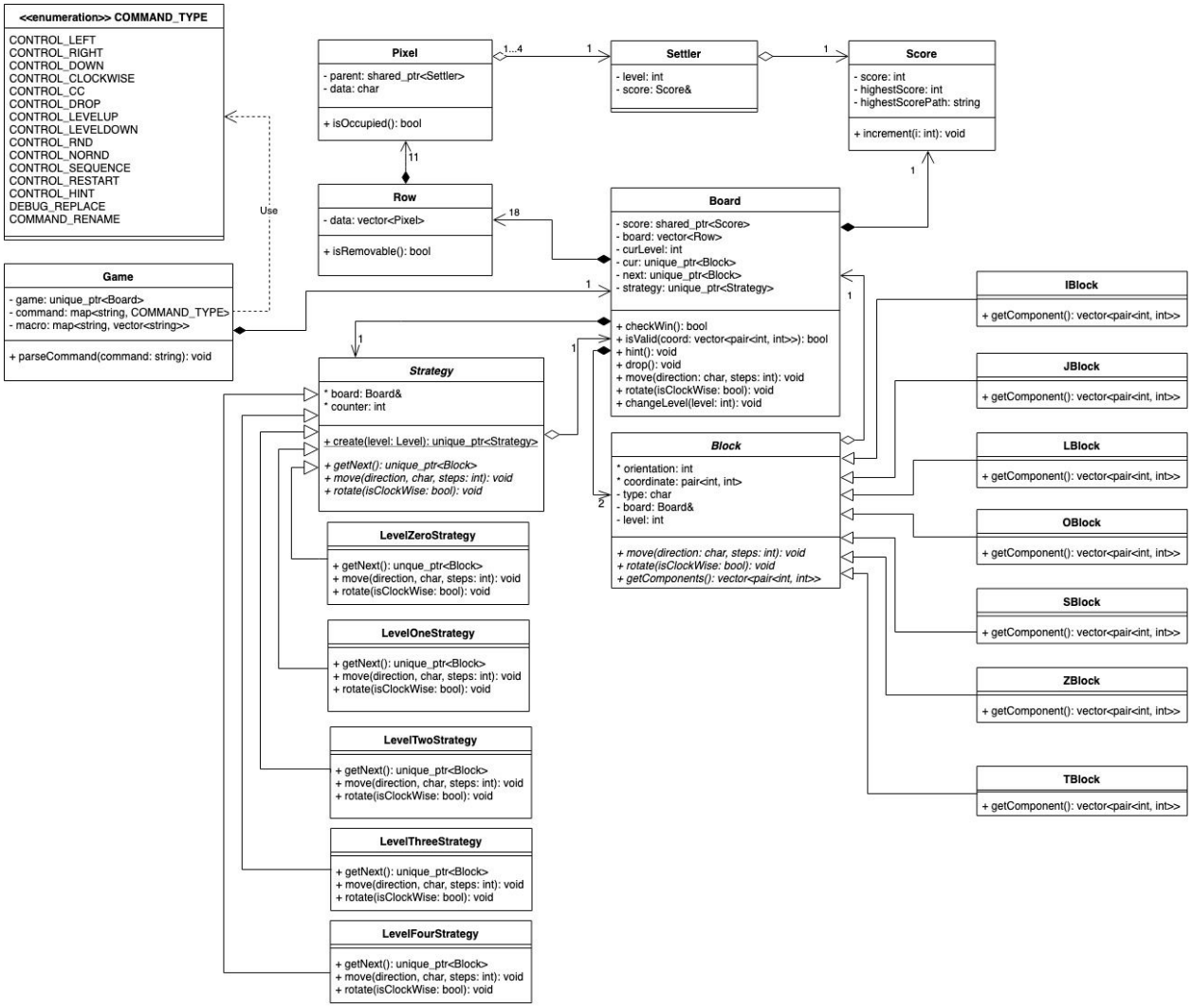
1. Initiation Stage: This includes designing the software architecture, writing a Makefile that minimizes recompilation and write simple test cases to solidify client API's. This state requires the participation of all group members. (Deadline July 1st 11:59 PM)
2. Component Building Stage:
    a. Row, Pixel, Settler & Score: Lucy
    b. Strategy: Saidi
    c. Block: Haoyang
    d. Command parser: Haoyang

    (Deadline July 6th 11:59 PM)
3. Component Linking Stage: Build out the `Board` class and link all the components together to obtain a MVP
    a. hint(): Lucy
    b. drop() & checkWin(): Saidi
    c. Build sstream and overload `<<`: Haoyang

    (Deadline July 8th 11:59 PM)
4. Testing Stage: Test the text output display and perform necessary bug fixes (Deadline July 9th 11:59 PM)
5. The following tasks can be completed concurrently
    a. XWindow Graphical Display: This includes building a GUI for the Tetris with appropriate use of input devices: Saidi
    b. Write the design documentation for this entire project as specified in the project requirements: Lucy
    c. Implementation of additional features for the game. Haoyang

    (Deadline July 12th 11:59 PM)
6. Review the application and complete all the questions (from project specification) together.
    (Deadline July 15th 0:00 AM)

UML Diagram



Please see the submitted uml.pdf for a clearer view of the entire diagram.

Discussion Questions:

1. To allow for some generated block to disappear from the screen if not cleared before 10 more blocks have fallen, a lifetime counter should be added to each `Pixel` in the constructor to record current counter in `Board`. And after each drop command, in the check function, one should iterate through all pixels to check if the difference between its lifetime counter and current board counter is greater than 10, if so, destruct the pixel, and clears it. To confine it to more advanced level, this logic could be added into the specific level of `Strategy` class as a helper method: `clearDeadBlock()`. This function will be called at the beginning of `getNext()` in this child `Strategy` class and should not be included for lower levels. In the child class of `Strategy`, this function should be implemented as stated above to scan through all pixels and clear the ones with lifetime longer than expected.
2. We used the factory method pattern to accommodate the possibility of introducing additional levels into the system, while keeping recompilation minimal. Each additional game level will be an additional children the `Strategy` class, while the strategy class has a static method that returns an instance of one of its subclass. With this pattern, adding a level is as simple as implementing the new level strategy in a class that inherits from the parent `Strategy` class, and then add the new level into the `create factory` of the parent. We would now have a new level. Since all the rest parts of the program only holds a pointer to our `Strategy` class, they only need to declare the `Strategy` class to use it. Therefore, they do not need to be recompiled to accommodate the change. With this design, we can easily introduce additional levels and only recompile the `Strategy` class, which is responsible for creating new strategies.
3. **Modifying/Adding Commands from Source**
   Based on the current UML design, all commands are processed by the function `parseFunction` in class `Game`. The following definition is the command map that stores all commands. All original string of commands will be mapped to a unique integer, which will then be processed by a switch-case structure:

   ```
   map<string, COMMAND_TYPE>
   ```

   where `COMMAND_TYPE` is an enumerator of integers which will make the code more readable for project members.

   For adding new features, a new `COMMAND_TYPE` integer, the name string of the new command will be added to the enumerator and the map respectively. Then a new case block will be added to implement the new feature.

All changes above will only affect the class `Game` and its implementation, so only one part (file) of this project needs recompiling.

Support Renaming

The simplicity of supporting command renaming can be maximized based on the implementation mentioned above. Suppose the example with corresponding renaming keyword `rename`:

```
rename counterclockwise cc
```

Firstly, the parser lookup the existing command `counterclockwise` (with error handling, and the complexity of lookup is O(log n) in general, so the operation will not be costly) and fetch the corresponding enumerator value. Then the parser will add a new entry to the map such that the new command `cc` will be mapped to the original enumerator value that operates `counterclockwise`.

Support Macro Language

1. Support Command Sequence

Add a new keyword `&&` (in the same way described in the first section) to concatenate multiple commands. The only difference is that this keyword does not perform any operation to the game, it only acts like a separator so that the parser will not stop parsing and it will wait for more command input.

2. Support Macro

Another macro map will be introduced to store all user-defined macros:

```
map<string, vector<string>>
```

And all macros starts with a name (without spaces) followed by a colon, then a sequence of commands, for example:

```
dance: left && left && right && right && down && right
```

When the parser receives a macro, it creates a macro map entry to the above map implementation, while the key is the name of the macro, and the value is the sequence of commands that has been validated (see below).

Validation of a command sequence will check each command in the sequence is valid, i.e. every command in that sequence can be found in the command map or the macro map.

Parser also recognizes embedded `sequence` command:

```
dance: sequence seq.txt
```

Which will parese the sequence from the file.

To support macro, the lookup method needs to be modified to maintain a lookup order: the parser lookup the command map first. If there is no such command, the parser will then look up the macro map.

Core operations (`down`, `right`, `left`, etc) are not allowed to be overridden by default, and the lookup method described above already satisfies the requirement. If the overriding is allowed, it can be accomplished by simply reversing the lookup order of the lookup method.