-kh3chen-y266zhao-

Plan of Attack

On Tuesday, July 16, a 1.5 hour long meeting took place, which was the first meeting regarding the CS246 Assignment 5. Together, we began discussing their thoughts on how to approach the project, **Quadris**, namely how to divide the program into suitable, organized classes. We began sketching out a rough UML diagram of the classes for this project. Using previous assignments for reference, we were able to complete the UML diagram. This allows us to begin writing code.

First, Joey will write header files for all of the classes that we have planned out in the UML diagram. Since the UML diagram only contained public fields and methods, writing the header files helps us organize the coding by making the private fields and methods concrete.

After writing the header files, the next step would be to actually code the functions in the header files. What will be coded first is the command interpreter, written by Kevin. This is a good starting point because being the user input, it simplifies testing for other components of code. For instance, this allows us to input `right` to test shifting a block to the right rather than hard-coding it into the program to test, or using some other type of input that would eventually need to be replaced anyway. The expected date of completion of the command interpreter is Friday, July 19.

Upon completion, what obviously follows is to begin coding the true bulk of the program; that is, designing the game board. However, since `Board` contains a 2D-array of `Cell`s, the `Cell`'s methods should be written first. Specifically, we want to make sure that each `Cell` contains information regarding its location, is able to set its corresponding character (e.g. T, L, whitespace), and can return it's current state. Joey will be in charge of coding this class, although collaboration will likely occur. The expected date of completion of the `Cell` class is Sunday, July 21.

What follows is the `Board` class as mentioned earlier. The Board class is in charge of handling relationships between all the `Cell`s that it contains. It is in charge of giving feedback on whether shifts and rotations of tetrominoes are legal and handles the modification of `Cell` data when these actions take place. Since this class contains many of complicated methods, both members will work on the board class. The expected date of completion of the `Board` class is Tuesday, July 23.

Since there are exactly 7 different configurations for tetrominoes, we decided to create an individual class for each of these types of blocks. First, we will write the code for the `AbstractBlock` class, which is the superclass of the 7 different blocks. Currently, our plan is to have each block have its own implementation of the `Rotate(…)` and `Shift(…)` methods, which are virtual in `AbstractBlock`. These block classes will be designed and coded collaboratively. The expected date of completion of the `AbstractBlock` class and its 7 subclasses is Thursday, July 25.

With all of the components of the game in place, the `Game` class will now be written, which contains the `Board` and sends commands to it based on user input. It is also here where the levels will be coded and designed as seen in the project description. Kevin will write the code for this class. The expected date of completion of game components and class is Friday, July 26.

At this point, the mechanics of the game board are set up and complete. What follows is the implementation of the scoring system. Several changes may need to be made to `Board`, `Cell`, and the block classes to meet the requirements of the scoring system. To ensure correctness of these changes, the scoring will be designed and possibly written by both members collaboratively. However, not much trouble is anticipated—the expected date of completion of the scoring system is Saturday, July 27.

After the scoring system has been implemented, the final aspect of the project to be implemented is the GUI. The `Board` will be modified to provide information to `Xwindow` and call its corresponding functions, but once again, not many issues could arise (due to previous experience with GUI in assignment 4). Kevin will be on charge of implementing this. The expected date of completion of the GUI is Saturday, July 27.

Lastly, command-line interface will be implemented by Joey. Final testing will be done by both members to ensure that all aspects of the program are working and bug-free. Testing will be done until there is great confidence in the program, which is expected by Sunday, July 28.

## Quadris Questions

**Question**: How could you design your system to make sure that only these seven kinds of blocks can be generated, and in particular, that non-standard configurations (where, for example, the four pieces are not adjacent) cannot be produced?

**Answer**: To ensure only seven configurations can be generated, create an abstract block class and seven concrete block classes for the seven block configurations. When each concrete block instance is created, it generates that particular block on the game grid. Thus only those seven specified configurations can be initialized since only those seven configurations are available to be initialized by the program.

**Question**: How could you design your system (or modify your existing design) to allow some game levels to occasionally generate transparent blocks (which can overlap other blocks), or skippable blocks (if you don't like it, you can skip to the next block)? Aside from these properties, transparent and skippable blocks behave like regular blocks. Could a block be both transparent and skippable?

**Answer**: Assuming that the transparent block is completely transparent, then it should fall through all existing blocks and drops to the bottom of the game grid. Implement this by adding a new char field to the abstract block type. If a block was initialized with this parameter set to 't' for transparent, the block would not check for collision with other blocks on the grid and only check for collision with the game grid when the player sends a move command to the block. Skippable blocks is relatively simple to implement. Add a new function in the game class that simply destroy the current bock on the board by setting all the cells on the grid where the block was to be empty and call the function to generate the next block.

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

**Answer**: We can introduce new levels by creating a next level function and a new integer array for probability of each block in the game class. The function clears the game grid and sets the probability array to predetermined values for each level. To introduce newer level simply add a set of configurations to the next level function. The function can take a parameter to indicate which level to create and it will play though the levels in numeric order otherwise by using a static counter. Thus only the game class needs to be recompiled and other classes are unaffected.

**Question**: 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)?

**Answer**: Separate the commands into a single command function in the game class. When the main class passes on the command from main to the command function in game, the game command simply looks up which command to execute and executes it. this require changing and recompiling the main class and the game class to accommodate the command change, but it could require the change of other classes since it might not be a trivial command to execute. To change the name of a command, simply change the string to be compared to in the if condition that selects the command in main. There is no need to change anything else since the selected command is passed through to the command function in game to execute the command.