Quadris: A Plan of Attack

In order to complete Quadris, we have divided the assignment into several components, based on our UML design.

First, we plan to brainstorm some of the potential corner cases, that may be encountered at runtime. Throughout the coding process, we plan to add to this list based on our implementation.

Next, we begin coding. The first part that we plan to code is our abstract classes. We have identified 'Level', 'Subject', 'Observer' and 'Block' as our abstract classes. Subject and Observer will follow the standard structure (based on what we were taught in class). Level will contain methods such as 'level_up();' and 'level_down();', which are necessary methods for all instances of level. Block will contain all the methods that need to be applied to a block such as 'Rotate', 'Left', etc. We have chosen to implement the abstract classes first, since almost every component of our UML diagram stems from one of those abstract classes. Reuben will implement the Block abstract class while Joshua will implement Level. We plan to have this completed by July 16.

Similar to in the Reversi game in A4, every cell will function as both a subject and an observer, communicating with the adjacent cells to the left and right. When a cell is 'set', it will send a signal to its left neighbour which will send a signal back if all the neighbours to the left are filled. This is how the game will recognize when a row has been completely filled. When it recognizes that it has been filled, a signal will be sent back along the row instructing each cell to clear and instructing the grid to shift all other cells downward.

Next, we plan to implement the grid class. The grid will be the main board which holds a 2-dimensional vector of cells, which are initialized as empty. All of the cells initialized at runtime will be all of the cells that will ever exist in the game. Following this, we can implement the class 'cell'. Fields of cell will hold information such as whether the cell is empty, and if not, which block currently occupies the cell (so that we will know which colour to print). Then, we can create our main class 'game'. Game will have the method 'start_game();' which will call the constructor for grid, initialize it with cells and provide the first block. Joyce will implement the grid, cell and game classes by July 17. After, we plan to instantiate the subclasses of level (levels 0 – 4), and block (all the different shapes). Again, Reuben will implement each subclass of block and Joshua will implement the subclasses for level. This allows us to code our block generator and implement the appropriate probabilities according to the assignment instructions. Joshua will also implement a command interpreter class. We want to finish all of this by July 18.

Finally, we can implement our graphical and text displays. These will be implemented last since the functionality of our program can be implemented before we display. Joyce will implement the graphical and text display. This will be completed by July 19th. This leaves us several days for the appropriate testing. We will utilize our list of test cases that we create throughout the coding process and turn it into a test suite. From there, we will attempt to brainstorm some further test cases before submission.

Question: 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: In our existing design, grid currently holds ‘block_vector’, a vector of blocks which contains all of the blocks that are held within the entire grid. Each block has ‘cell_vector’ which has a length of four, with cell pointer to the four cells, which contains the four cells that the block overlays. If we wanted to modify our existing design to allow for the feature where generated blocks disappear from the screen if they are not cleared before ten more blocks have fallen, we could modify the vector to be a queue of maximum length 10, we could have a ‘remove_block’ function in our grid, which would pop the front block and remove the generated blocks.

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

Answer: To design our program to accommodate the possibility of introducing additional levels into the system, we can create an abstract level ‘level. Each level is an instance of level and in order to introduce additional levels into the system, we simply add another child to ‘level. This minimizes recompilation since only the new instance of level will need to be compiled, before being linked to the remainder of the files.

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)? 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: By creating a separate command interpreter class, we will be able to read in strings which call the appropriate methods. This allows for us to easily add new command names, as we can simply add new fields. This will require minimal changes to source, since this does not modify any existing commands and minimal recompilation since only the single command interpreter class must be recompiled. It would also allow for us to easily adapt our system to support a command whereby a user could rename existing commands by providing setters to each of the command fields. To support a “macro language” we could create a stack of commands which the user would be able to create and run.