Plan of Attack by Peter, Gursirat and Ayman

Due Date	Task	Group Members Responsible
July 17	Complete all header files	Everyone
July 18	Implementation of Observer pattern framework	Ayman
July 18	implementation of Blocks and Factory Method Pattern for Blocks	Gursirat, Ayman
July 19	Implementation of Board Class	Everyone
July 20	Cell implementation	Ayman
July 20	Score + NewBlock implementation	Peter
July 20	Interpreter Implementation	Gursirat
July 21-22	Meet and debug/fix code, and implement main.cc, and finish TextDisplay/GraphicsDisplay	Everyone
July 22-24	Continue debugging and write-up final report, test code	Everyone
July 24	Add enhancements and edit final report	Everyone
July 25	Submit!	Everyone

Breakdown:

Prior to any of the tasks mentioned above, we collectively as a group created an ample set of tests cases. These tests included white box and black box testing, to cover every conditional branch in our game. Furthermore, we kept in mind specific edge/boundary cases, for example: if a block is at the rightmost end of the board and thus cannot be rotated. We decided to create tests immediately because we know that is one of the most crucial steps to creating an effective program, and it helped is conceptualize ideas for future design and implementation.

First we will create all the header files for our classes so that we can setup the necessary interfaces, and create the composition, aggregation and inheritance relationships between classes as is outlined in our UML. We gave this a due date of **July 18th.**

The core element to our overall design is an observer pattern, so it makes sense to implement the framework for it first. We believe the observer pattern is appropriate as both TextDisplay and GraphicsDisplay must react according to when the items of the board are updated, so it made sense to make the Board our subject and the two different displays observers of the board. Peter seems to have a good grasp on how the Observer pattern works and performed well on Assignment 4 question 4, so he will be primarily responsible for implementing the subject and observer classes and their functions as well as defining the observer-subject relationships between Board and both Display classes before **July 18th.**

At the same time Gursirat and Ayman will be responsible for creating the Block Factory Method Pattern. We figured that a Factory Method Pattern would be appropriate because there are multiple fields and methods which are common to all Blocks, but some aspects (like shape and colour) that are unique and distinct for each block. For example: the “I” block may be able to rotate in certain circumstances where the “O” block may not, so it is important to define different methods for each. Thus we made the Block class the abstract superclass, with each type of block (e.g. O, S, T, etc.) a subclass of the superclass. The Block class is essentially our factory which produces the specific type of block (from the concrete subclasses) that is asked for. We expect this to be done also before **July 18th.**

Arguably the most centralized class in our design is the Board class: nearly every function of the game has a direct relationship with it, and it is the only concrete subject in our observer pattern. Each board class either has-a/owns-a instance of Score, NewBlock (the next block that will be put onto the Board, as displayed in the top left corner), Block (the current block being controlled by user commands), and Cells. Due to the importance of this class all three group members will work together to integrate this class with all the other individual components of the game. Much of the implementation should be relatively straightforward, as many of the functions are accessor methods for the observers, or accessor methods for the Block rotation/movements. We anticipate this to be done before or on **July 19th.**

There are a few parts within Board that we decided to divide up because they largely act independently of each other. This will help us achieve low coupling by creating classes specialized to do a specific task. First, Ayman will implement Cells: Cells are what make up the grid, and also what make up the Blocks. As this is relatively straight forward he will also implement the Score class: which keeps track of both the current score, and the score algorithm (which changes based upon the level). Peter will implement the NewBlock class: the board class has a NewBlock instance that represents the nextblock that will be placed on the board. The most difficult part of this implementation will be the algorithm that generates different probabilities of Blocks based on the level. Gursirat will be responsible for the Command Interpretation, as he had the best grasp on potential solutions during our previous brainstorming

sessions. This class does not have a relationship with any other class, but an instance of it will be created within main in order to interpret the user input and decide how the game should respond. We expect these classes to be implemented to the best of everyone's ability by **July 20th.**

After this it will be important to meet on the 21st to fix any inconsistencies from integrating the individual separate classes, for the last due date. Once that is done, main.cc will have to be implemented. Due to the modularization, the main file will have very little logic in it. It simply must initialize a board and interpreter object, then enter conditional loop where the user input is interpreted and the appropriate Board method is called to continue the game. Then the displays (our observers) within the observer pattern will have to be implemented so they are notified of changes on Board and update accordingly. Now that the displays are set up, for the next couple of days until **July 22nd** we will be vigorously debugging and testing the code together.

If everything goes according to plan then from **July 22- 24** we should be able to solely focus on writing up the report. If we are still debugging then we will have one person dedicated to debugging, while the other two focus on the final written report. On **July 24th** we should have a working implementation of Quadris and be in the editing stage of our written final report. We will add enhancements to the program at this time. But we will still maintain version control, so we can revert back to a previous version at anytime, in case our new implementation doesn't work. Once any enhancements are implemented, we should be set to submit an amazing final project on **July 25th.**

Question/Answer:

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?

We could make it so that each Cell object within the Board has a counter field initially set to 10 through the constructor. Everytime a Block is dropped the counter field of all Cells on the Board will decrease by one if the Cell is not empty. The program will also check to see if any cells have reached a counter of zero yet. Every Cell identified that has a counter of zero and has a colour, will be set to no colour by Cell public method setType(), invoked within class Board. The Board class will then invoke the notify() method to update the TextDisplay and GraphicsDisplay, thus effectively making the blocks disappear from the screen.

The generation of such blocks could easily be confined only to more advanced levels. The Board class will hold the integer field level. We would include an if statement within the drop method that checks if the level is above a certain threshold. If and only then will the cells counter field be decreased by one, when the drop method is called.

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

Within quadris there are three classes that are affected by levels: NewBlock generation, Score generation, and the TextDisplay. The classes NewBlock and Score are owned by the Board class (as outlined in the UML). Board, NewBlock and Score will all have the integer field level, they also have public methods that change the level. Therefore since Board owns both classes it has access to those pubic methods, so if the level changes it will update both classes immediately. Thus whenever a new level is added, we simply alter whichever of the two classes the new level affects. For example: if you want the new level to have different block generation, we make a new if statement condition for the NewBlock class that implements the correct probability algorithm. For TextDisplay whenever there is a level change the Board class runs notify(), this notifies the TextDisplay class of any changes. Thus the majority of our functionality would continue as implemented, if additional levels were introduced. There is minimum recompilation required as the classes dependent upon level are modularized separately with minimal dependencies, through the good design practice of low coupling.

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.

In order to accommodate new command names, changes would have to be made to the Command Interpreter class. The class would have to have a map dictionary, with keys being the functional/actual command (i.e. clockwise()), and the value being a string array of command name aliases (i.e. clockw, cw, clo). If a new command name is to be added then the dictionary’s size is increased by one, and a new key value pair is created. If a command name is to be changed, by using the command line function “rename [oldname] [newname]”, then we will iterate through the dictionary to find the key equal to the oldname, and then change the key to the newname. As the interpreter class does not have a relationship with the rest of the classes, there will be minimal changes to other source code, and minimal recompilation required. All command names from user input will be processed here, and the only output will be the correct functional/actual command (i.e. clockwise()) being called to run the game.

In order to support a macro language, we would create and store an array of macro commands. Each macro the user creates will be a dictionary: the key is the name of the macro command, the value is an array of all the commands that will be run, in the correct order. So once a key is identified based on user input, then we would run a for loop through the array of corresponding

commands. Again this minimizes any changes in functionality to source code because the same actual commands will be called, their order will simply be controlled based on the array value of a given macro command key.

This would have minimal effect on the available shortcuts for existing command names. We would still be able to implement the feature of “Only as much of a command as is necessary to distinguish it from other commands needs to be entered.” The algorithm for determining this would have to loop through a series of keys within the dictionary rather than all the command names by themselves.