Our team has decided to code the game Quadris for our assignment. We have planned out a schedule for ourselves in a chart, see below. We have put the chart online and plan to complete it as the deadline approaches, marking when sections are completed and making any necessary changes.

	Start Date - Plan	Finish Date - Plan	Designated Partner	Start Date - Actual	Finish Data - Actual
UML - Outline	March 25	March 26	Will	March 25	March 25
Plan - Outline	March 25	March 26	Jacob	March 26	March 27
UML - Finalize	March 26	March 26	Jacob	March 26	March 27
Plan - Finalize	March 26	March 27	Will	March 27	N/A
qudris.cc	March 26	April 3	Jacob	March 27	N/A
Cell	March 27	March 28	Jacob	March 27	N/A
Block (virtual superclass)	March 27	March 27	Will	March 27	N/A
Block .h files	March 27	March 28	Jacob	March 27	N/A
iBlock	March 28	March 29	Will	N/A	N/A
jBlock	March 28	March 29	Will	N/A	N/A
lBlock	March 28	March 29	Will	N/A	N/A
oBlock	March 28	March 29	Jacob	N/A	N/A
sBlock	March 28	March 29	Jacob	N/A	N/A
zBlock	March 28	March 29	Jacob	N/A	N/A
tBlock	March 28	March 29	Will	N/A	N/A
grid.h	March 29	March 30	Will	N/A	N/A
grid.cc	March 30	March 31	Jacob	N/A	N/A
Score	March 30	March 30	Will	N/A	N/A
NextBlock	March 30	March 31	Jacob	N/A	N/A
display.h	March 30	March 31	Jacob	N/A	N/A
display.cc	March 31	April 1	Will	N/A	N/A
xWindow.h	April 2	April 2	Jacob	N/A	N/A
xWindow.cc	April 3	April 3	Will	N/A	N/A

Note that the above plan does not include the Interpreter class in the UML. This is because the Interpreter will likely be completed as the other classes are, just like the Quadris class has a very early start date, but late finish as it will likely be constantly updated as the other files are created to test that each is working individually.

As well as this plan our group has done a number of other things in preparation for this assignment. We have decided that the easiest way to collaborate for this assignment would be to use a shared folder on Github, as it not only allows a shared online folder like Dropbox, but will also allow us to give a description of every update we make (essentially adding comments for each update for clarity). This will also be a useful way for us to make sure that we are actually on top of the deadlines as the finish date should not be entered until the files have actually been updated to GitHub.

We also plan to meet every 2-3 days, either in person or over Skype or Facebook to update on another, in case we run into errors, bugs, or any other sort of issue. As well as giving each other input on the completed work, whether it need be changed or improved, or anything along those lines.

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:

Our Block class will be coded such that it is a pure virtual class. There are multiple reasons for this and the answer to the above question is one of them. With a pure virtual Block, a Block cannot be created individually, instead on of the subclasses of Block must be called. Thus we have created 7 Block subclasses, representing each of the seven pre-determined blocks. This will mean that no other Block objects can be created, and that no non-standard configurations will be able to be made, unless a class for the non-valid Block is added to the project.

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:

We think that a quite simple to understand and elegant solution to have old (10+ turn) blocks disappear is to assign each block a “lives” argument of type integer. This argument will be initiated as soon as the block is dropped into place with an initial value of 10 (10 turns). Furthermore, when dropping the block, we go through the rest of the previously dropped blocks in the quadric board and we decrement the “lives” argument in each one. Simply put, if the “lives” argument of a given block is less than 0, we clear it, that is to say, fill the square with a space.

With levels, this does not become a problem at all. Since the primary difference in levels is the score and the generation patterns of the blocks, we have little to worry about with respect to our “lives” argument since each of the generated blocks will have a “lives” argument and by default it will always be set to 10. Decrementing is the same on all blocks, as we will essentially just be decrementing by cell, not by block object. The only difficult part would be to properly assign the score when clearing the block, to do this however, we need only pass an argument to the clearing method letting it know which level we’re in and to pass the proper score to the score object.

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

Answer:

Levels affect multiple classes in our code, these being Cell, Grid, Score, Block and Interpreter. The reasons these all must be affected is that the Cells for certain levels may have, like above, the 'disappearing' aspect at a certain level for example. The Grid will also have this aspect at certain levels, as well as a possible change in size or other possible feature, depending on the level (it would also need to be recompiled if Cell changed). Next the Score needs to know the level as it will calculate the score differently based on which level the user is playing on. NextBlock also uses level for how to call the Blocks, with files, randomly ,or with certain ratios. Finally, the Interpreter needs levels as different classes may be called based on the level. By this we mean that effective coupling will be used such that the Interpreter will handle the current level and affect the other classes as necessary. For example, a theoretical level where controls are reverse, i.e. left is right, down is drop, and clockwise is counterclockwise. All those changes would be handled by the Interpreter.

Both the Makefile and the design of this program will allow for minimal recompilation for the addition of levels (provided extra classes aren't added for the level). The program will be designed such that only the necessary classes will be updated when changes are made. In this case it is only those mentioned previously. The Block classes will not have to be recompiled (unless a Block is added for an extra level) and the displays will also remain 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:

If a new command needed to be added to the program, most likely it can be handled by just the Interpreter, provided the existing code will accommodate the new command. Similarly, the Interpret can handle the changes to an existing command. For changes in

a command name, this is easy handled by the Interpreter with exceeding minimal changes. The command will just now be recognized in a new way, but called identically. Then only the Interpreter and the main Quadris files would need updating. Essentially all situations posed by this question are answered by our Interpreter class, allowing all these situations to be handled with ease.