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	March 28
quadris.cc	March 26	April 3	Jacob	March 27	April 4
Cell	March 27	March 28	Jacob	March 27	March 28
Block (virtual superclass)	March 27	March 27	Will	March 27	March 27
Block .h files	March 27	March 28	Jacob	March 27	March 29
iBlock	March 28	March 29	Will	March 28	April 2
jBlock	March 28	March 29	Will	March 28	April 2
lBlock	March 28	March 29	Will	March 28	April 2
oBlock	March 28	March 29	Jacob	March 28	April 2
sBlock	March 28	March 29	Jacob	March 28	April 2
zBlock	March 28	March 29	Jacob	March 28	April 2
tBlock	March 28	March 29	Will	March 28	April 2
grid.h	March 29	March 30	Will	March 28	April 3
grid.cc	March 30	March 31	Jacob	March 30	April 3
Score	March 30	March 30	Will	March 30	April 3
NextBlock	March 30	March 31	Jacob	March 30	April 3
display.h	March 30	March 31	Jacob	March 30	April 3
display.cc	March 31	April 1	Will	March 30	April 3
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 private repository 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 and it also gives us a detailed submission history so we can see who did what and when.

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 implemented 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 one 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.

This also has the added benefit of not being able to generate a block with different methods that are not applicable or useful. It keeps everything nicely in order and makes generating based off of probabilities for higher levels quite easy and manageable.

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 is needed to be added to the program, most likely it can be handled by just adding a new checking method to 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 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.

Changes made to the initial Plan

While coding this project, it was inevitable that we reached some tough situations and needed to deviate from the original plan. Some of these situations included requiring to make some public functions private and vice versa, adding a new class called coordinates and removing the xWindow class because we couldn't get it to work. All of these changes will now be discussed in detail.

Most of the change between public and private methods occurred in the interpreter class. The main reason is that our main in quadris.cc had to call specific methods (like isDrop(), isRestart()) based on edge cases into the application. These methods, previously private, had to be changed to public so that the quadris main could access them. The rest of the methods (e.g. isRight(), isLeft(), etc.) were moved to private, as to protect the implementation.

Initially when going through the planning, we didn't feel it necessary to create a coordinate class. We thought it would be easy enough to pass an array of integers, 8 long with (x,y) pairs. However, when creating our program it quickly became apparent that this was not a very elegant solution. It made our code look terrible, even unreadable in some parts and as a result, we gave in and created the coordinate class. We also removed the xWindow class. We did our best to try and implement the graphics for the program however every time we tried it, it would leak memory. Even just running the graphics test, we leaked memory. As a result of this, in the end we ditched it preferring a leak free program over a "frilly" leaky one.

Final questions and reflections

Question:

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large problems?

Answer:

Developing software in a team is challenging, to say the least. Everyone has their own development philosophies, their own strengths and their own weaknesses. For me, I love to design programs, but when it comes down to the actual hard coding, I tend to procrastinate it off. William, was the opposite, preferring coding over planning. We started off the project by delegating tasks and working separate from each other. It became quickly visible that this was not the way to go. Not only were we building parts of a program with no understanding of how they fit together, but we were not

communicating how we were doing so. This lead to having a bunch of completed, working parts… that didn't work together because well… we didn't work together.

What we did next was a much better idea. We had code, but nothing tangible, or a main file. So we opted to build the main first and the rest of the files (dependencies) around that. Within a few hours we had a board with initiated cells. A few hours later, a block. A few hours later, movement! And then collision, dropping, multiple commands, line clearing, no more memory leaks, leveling, edge case coding… we built it from the ground up, using some new code, some old. This is now obvious to us, what we should have been doing all along. That just belting out code without a worry of how it'll all fit together is not only bad teamwork, communication and planning… but would actually never ever lead to a finished product. Instead, communicating and figuring out how to put all the parts together and working from the ground up, implementing extra features as the project progressed, is the only way to make a program like this in a team.

Question:

What would you have done differently if you had the chance to start over?

Answer:

As mentioned above, the number one thing we'd change if we started over would be to focus on building on completed parts and not do everything at once. Also, to communicate better through the design and implementation processes. Had we done that, we would have certainly had much much more time left over to implement some of the things we missed, like block clearing for score and gotten graphics to work.

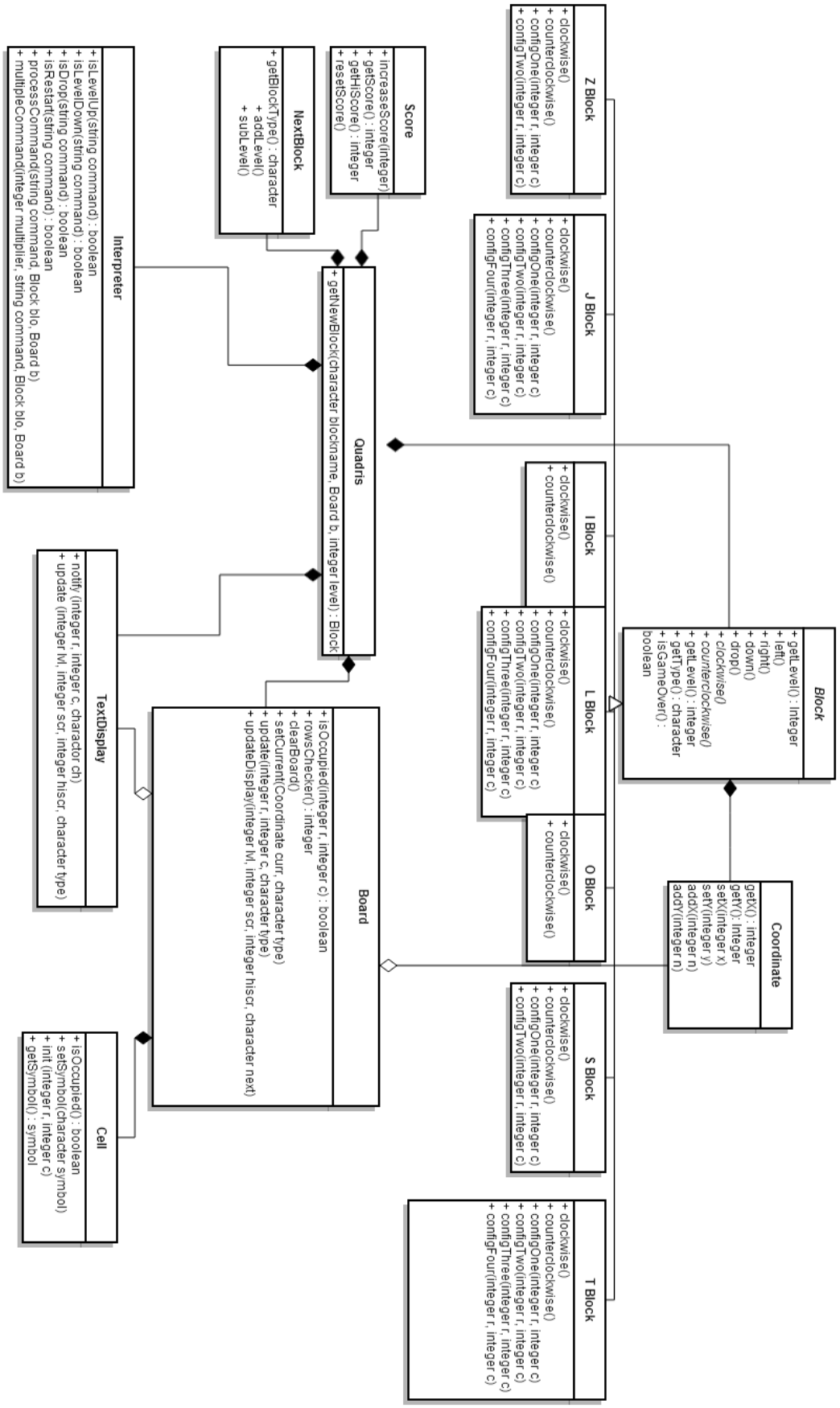
With respect to implementation, we would change the board to follow a singleton pattern with the board as the singleton. Since no matter what, only one board is ever required, the singleton pattern serves to isolate issues with multiple boards being instantiated and unpredictability. Also, had we implemented it as a singleton, our program may have run more efficiently, or at least been implemented more clearly.

We could have also used an abstract factory pattern to generate the blocks for us. With a factory pattern, we could have easily shown that only the 7 specified types of blocks could be produced and initiated and it would have helped with the random generation of blocks.

Our main function is quite cluttered with all the different edge case calls to specific interpreter methods. Having a "game" class could have come in handy with

“startgame()” or “endgame()” methods that would leave our main edge case and clutter free and more clearly implemented.

Finally, our interpreter could have employed a trie class from assignment 3 question 3 to figure out commands based off of half entered data. The reason that this would have been better than our current implementation is that adding, editing and removing commands would have been much easier to do and much more flexible for the programmer. Also, it could handle the addition of a large number of commands without having to change implementation very much, whereas ours is completely dependent on the basic commands detailed in the quadris pdf.


Block
+ getLevel() : Integer
+ left()
+ right()
+ down()
+ drop()
+ clockwise()
+ counterclockwise()
+ getLevel() : integer
+ getType() : character
+ isGameOver() : boolean
Coordinate
getX() : integer
getY() : integer
setX(integer x)
setY(integer y)
addX(integer n)
addY(integer n)
Z Block
+ clockwise()
+ counterclockwise()
+ configOne(integer r, integer c)
+ configTwo(integer r, integer c)
J Block
+ clockwise()
+ counterclockwise()
+ configOne(integer r, integer c)
+ configTwo(integer r, integer c)
+ configThree(integer r, integer c)
+ configFour(integer r, integer c)
I Block
+ clockwise()
+ counterclockwise()
L Block
+ clockwise()
+ counterclockwise()
+ configOne(integer r, integer c)
+ configTwo(integer r, integer c)
+ configThree(integer r, integer c)
+ configFour(integer r, integer c)
O Block
+ clockwise()
+ counterclockwise()
S Block
+ clockwise()
+ counterclockwise()
+ configOne(integer r, integer c)
+ configTwo(integer r, integer c)
T Block
+ clockwise()
+ counterclockwise()
+ configOne(integer r, integer c)
+ configTwo(integer r, integer c)
+ configThree(integer r, integer c)
+ configFour(integer r, integer c)
Score
+ increaseScore(integer)
+ getScore() : integer
+ getHiScore() : integer
+ resetScore()
NextBlock
+ getBlockType() : character
+ addLevel()
+ subLevel()
Quadris
+ getNewBlock(character blockname, Board b, integer level) : Block
Interpreter
+ isLevelUp(string command) : boolean
+ isLevelDown(string command) : boolean
+ isDrop(string command) : boolean
+ isRestart(string command) : boolean
+ processCommand(string command, Block blo, Board b)
+ multipleCommand(integer multiplier, string command, Block blo, Board b)
TextDisplay
+ notify (integer r, integer c, character ch)
+ update (integer lvl, integer scr, integer hiscr, character type)
Board
+ isOccupied(integer r, integer c) : boolean
+ rowsChecker() : integer
+ clearBoard()
+ setCurrent(Coordinate curr, character type)
+ update(integer r, integer c, character type)
+ updateDisplay(integer lvl, integer scr, integer hiscr, character next)
Cell
+ isOccupied() : boolean
+ setSymbol(character symbol)
+ init (integer r, integer c)
+ getSymbol() : symbol