CS 246 ASSIGNMENT 05

Quadris

Plan of Attack

Timeline

Friday, November 25
Finish writing this document and submit it, as well as our UML class diagram, to Marmoset. Set up a Git repository on git.uwaterloo.ca which we will use for version control and collaboration.

Saturday, November 26 - Sunday, November 27
Each group member will begin writing the interface files for each class as shown in the UML diagram.
Stephen will write the interface files for Subject, Observer, TextDisplay and GraphicsDisplay.
Harsh will write the interface files for Board and Block.
Mufaddal will write the interface files for BlockGenerator, Level, and its inherited subclasses.

Monday, November 28
Implementation of game begins. The group collectively begins to implement Block.cc which involves creation of a mapping between character inputs and their respective blocks as well as implementation of other methods included in the class. Through this project, we will attempt to collaborate as much as possible, reviewing any code that was written independently, so that every group member understands how the pieces work and fit together.

Tuesday, November 29 - Friday, December 2
Implementation of Board.cc, BlockGenerator.cc, Level.cc and its subclasses, as well as Observer.cc, Subject.cc, GraphicsDisplay.cc, TextDisplay.cc and main will be completed.
Stephen will work on implementing GraphicsDisplay and TextDisplay while Harsh writes up the implementations for main as well as Board and Mufaddal implements BlockGenerator, Level and its subclasses.

Saturday, December 3
All implementation code should be merged and pushed to the git repository (if not done already) and debugging and testing of the game will begin. This will be an individual as well as a group effort. Each of us will attempt to debug and test our own implementations, then each other's, and then any issues that continue to arise will be addressed by the group as whole.

Sunday, December 4
Implementation of extra features begins (in separate git branches, of course). Features that we are considering include:

i) Adding the ability to rename commands as described in the project document

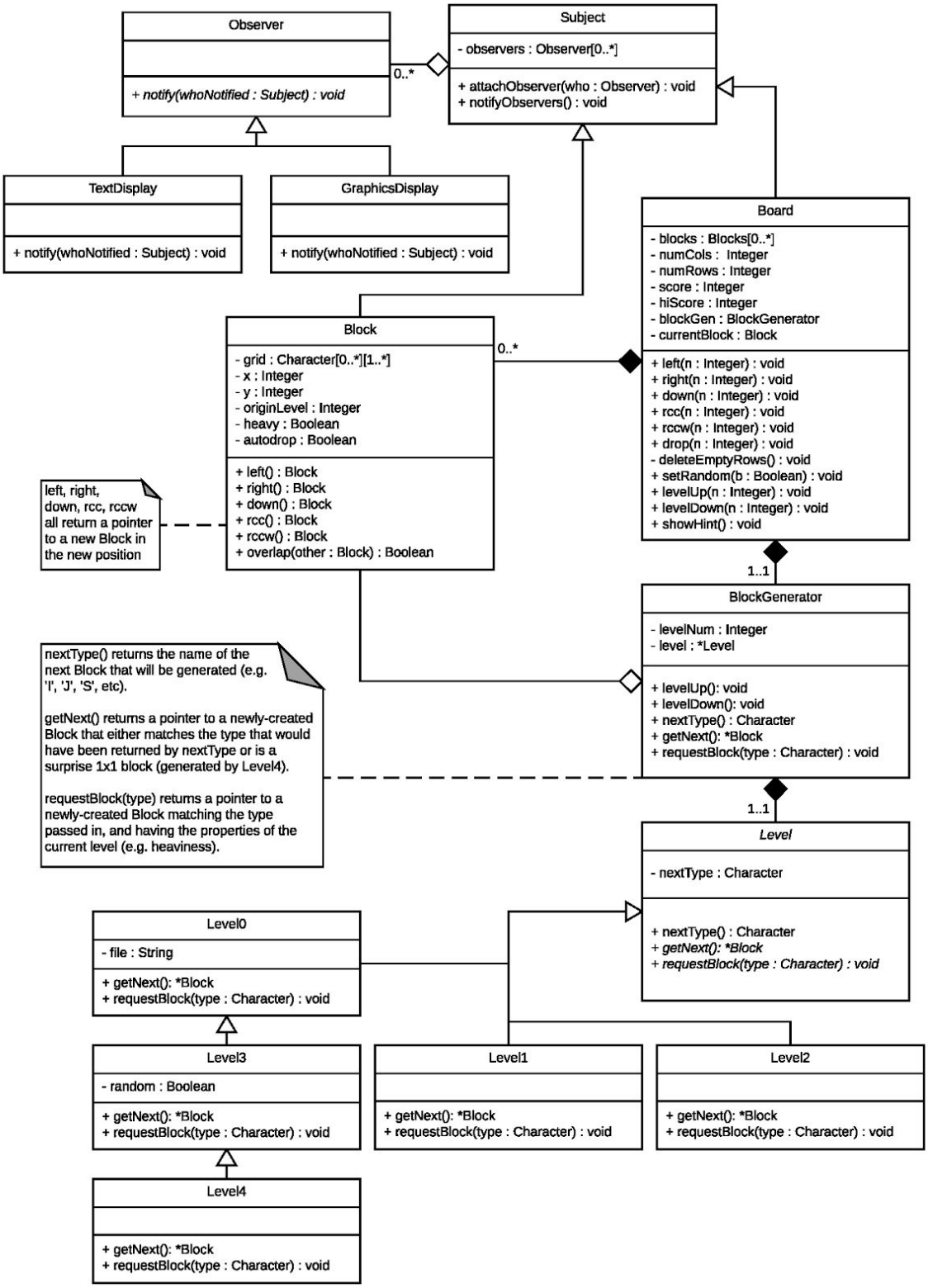
ii) A command to allow the user to skip a block
iii) Real-time gameplay (for the graphical interface, at least)
iv) Disappearance of blocks after 10 (or *n*) moves
v) Add new advance levels which could include preset patterns

Final testing of code should be completed by 23h59.

Monday, December 5
Final runthrough of code. Submission of code on marmoset by 10h00. We will have a group meeting to plan out the project demonstration for the TA's.

Note: Project documentation for the assignment will be written as features are implemented. We will draft the final document after we finish debugging and have a working program.

UML Diagram (just for quick reference, also in uml.pdf)



Question 1

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?

Each block object would include a counter which would be initialized to 10 and would be decremented by one every time a new block is dropped. Once the counter reaches 0, the block would be removed from the array of blocks and the displays would be updated. The generation of such blocks could easily be confined to more advanced levels because each Level provides its own getNext function, which could easily be modified to set a auto-disappear property to true. Any advanced levels added would simply necessitate an adjustment to the block class as described above, adding a new Level subclass, and implementing a function clearBlock() in the Board class which removes a block once its counter reaches 0.

Depending on the level of difficulty, we might also include an adjustBoard() function which, in an easy level, would adjust the board by calling the drop function on each of the blocks until they could not go down any further. It would start from the bottom row and go up the rows. It would then look for any filled rows and clear them. In a medium level, the same would be done except the blocks would only be moved down by one. In a hard level, the block would be removed but the rest of the board would not be adjusted. Note that this last option would be the easiest to implement and is the most consistent with normal Tetris.

Question 2

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

In order to be able to efficiently add new levels, we have a container class called BlockGenerator [it contains an object of class Level] which generates new blocks depending on the current level of the game and levels up or down depending on the input command. To level up or down, BlockGenerator has levelUp() and levelDown() functions which, when called, both create new Level objects according to the new difficulty and replace the existing Level object. Separating the Level object form the Board with BlockGenerator decouples all functionalities to do with block generation and switching levels from Board.cc. When creating a new level, we will create a subclass to Level (which is an abstract class) and include the relevant implementations for the subclass. On compilation, BlockGenerator.cc will be re-compiled in order to inform itself of the new level and the new Level subclass will be compiled as well. This will minimize recompilation when adding new levels to the game, since Board will not need to be recompiled.

Question 3

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.

We have chosen to design our system in a way such that, with minimal re-compilation, names of commands can easily be changed and new commands can easily be added. To support this functionality, we will create an enumeration with unique enumerators representing each function that can actually be called on a board (e.g. Board.left(), Board.right(), Board.down(), etc). We will also create a map which links the command names (strings) that the user can enter to the enumerator for the associated function. When a command name is read in, a switch statement on the enum (which will have been acquired from the map) will handle actually calling the function, and, optionally, reading in additional parameters such as a file name. Adding or renaming a command will only necessitate recompilation of the file containing the main function and the file containing the command-enum map.

With this design, it would be easy to support a command to rename or duplicate an existing command as it would need only to modify the key string in the command-enum map.

Supporting a macro language would also not be too difficult using a variation of this system. One simple way to support macros would be to store the string representing the command sequence, and then interpret it only when the macro is called. This approach raises a few issues, though; if one of the commands included in the macro is renamed, then the macro would need to be redefined. We could get around this by translating the command sequence into its associated enumerators and storing that, instead of storing the original command names. To store macros including commands which require additional input, like filenames, we could do the same, but also store the additional input separately in a string or sstream. For example, a macro for the command sequence "`left down left sequence seq.txt norandom nor.txt right`" could be stored as a list of enumerators `{LEFT, DOWN, LEFT, SEQUENCE, NORANDOM, RIGHT}` and as a string or sstream "`seq.txt nor.txt`" which would be read from as appropriate.