Overview

The three largest classes in our design were Grid, Block and Level, most other classes were subclasses of one of these three classes, with the rest of our functionality being implemented in main or being classes to implement the text and graphics displays.

The game begins with main, which reads in any command line arguments and then calls the grid constructor. Throughout the game, all input is read in through main which passes the appropriate command to grid.

Grid

The Grid class is the actual grid on which the game is played. It also acts as the central class in which all other classes interact with, since every other aspect of the game has to interact with the current grid. When the game commences, the grid is initialized through its constructor to create the default 11 x 18 board. It does this by creating a 2-dimensional vector of Cells. Cells is a separate class which stores its position (in Pos, a defined structure), whether it is empty and if not, which block is currently occupying it. Grid also automatically creates the first block in the top left corner by calling the block constructor (more on this later), which allows the game to commence. Throughout gameplay, the Grid class clears rows as they fill.

Grid also stores basic game functions, such as restart and hint, and works with other classes, GameData and Level, to increase/decrease the current level and keep track of the score. It also checks if the game has ended every time a new piece is generated.

Furthermore, two other classes, TextDisplay and GraphicsDisplay read from Grid in order to configure a working text and graphics display. Every time a change is made to the display, Grid calls our ‘update displays’ function to allow for the text display and graphics display to change accordingly with the Grid.

The Grid class also contains the implementation of the drop command. The reason why the drop command was included in the Grid and not the Block (where the rest of our block commands are), is because drop is dependent on the location of other blocks on the grid, not just the block itself. Similarly, while Block carries out the remainder of our block commands, Grid checks for any collisions. Essentially, Block defines the behavior of a single block in isolation, whereas Grid allows for blocks to interact with each other. Throughout our program Grid continuously acts as a central point of interaction for other classes, such as TextDisplay, GraphicsDisplay, and Level.

Block

The Block class is an abstract class which contains the attributes of a block. Each different type of block (I, J, etc.) and the class 'SingleBlock' inherits from Block, using the inheritance relationship. 'SingleBlock' is the block dropped by the game into the middle of the board during level 4 as an added difficulty.

Block keeps track of the position of each of the four components of itself. It also holds a boolean field which stores whether or not the block has bottomed out. A block is bottomed out when it is at the bottom of the grid or cannot be moved any lower without collision. Block also stores a field that holds the level that it was created in in order for the score to be properly recorded. For example, if a block is created in level 3, but cleared in level 4, the score is still able to record the score according to level 3 because of this field.

Block has methods implemented for the block commands such as left, right, clockwise, etc. Thus, the game commences with the construction of a grid, which initializes the empty grid, calls the construction of the first block (and the next block, since the display must show the next block) and places that first block in the top left corner. Any command called on the block will call one of the methods from the block class. The resulting new block location will be passed to the grid which will ensure that this will not cause a collision. If it will cause a collision the game displays "Invalid Move". Otherwise, Grid will take the new block location provided by the block function and change the appropriate cells to reflect the new position (clearing out the old cells and filling in the new cells). The next block will be called onto the board when the current block has bottomed out.

Level

Level is another abstract class, which has five subclasses (levels 0 – 4). Every subclass contains functions which allow for one to level up or level down (by calling the constructor of a previous level or the next level respectively) and contains an integer referring to the level that it is (i.e level4 class stores integer 4 as its field). Every time the player wishes to level up/down, the grid calls level up/down on the current level, which calls the constructor for the next/previous level and the current level field in the grid holds the newly constructed level.

The most significant method in level is the ‘generateBlock’ method. Since the game chooses which block to generate based on the level that it’s in, the ‘generateBlock’ method is different for each level. When the game starts, the grid calls the level constructor (by default 0, or if provided a field, the provided level), and keeps the current level as a field. Every time a new block is required (when the current block bottoms out), Grid calls the ‘generateBlock’ from the current level field.

Updated UML

One of the major changes that was made to the UML is the elimination of the class Game. Instead of having Game hold functions such as restart, gameover, etc. we implemented these functions in the Grid class. We changed these functions since all of them used the functionality provided by Grid. That is, when looking to restart the game, we would have to call several functions in Grid.

Furthermore, we created smaller structures to increase the readability of our code. For example, instead of using fields (x, y) every time, we created a position structure.

Finally, we also eliminated the utilization of the Observer pattern in our code. We noticed that only the TextDisplay and GraphicsDisplay acted as observers and were not updated every time a change to the grid was made. Thus, instead of using a standard Observer pattern, we used a pseudo-Observer pattern (more on this in Design).

Class Relationships

The largest difference we made in the relationships between classes was turing the Grid class into a controller for the entire game. Now the Grid is in charge of overseeing every object instantiated in the game, so the Grid owns all Blocks, Cells, and both Displays. None of the other classes exists without the Grid, so when the Grid is destroyed everything else in the game is destroyed also. This change allowed us to remove many of the ‘chained’ relationships in the initial UML, permitting a simple design more centralized around the Grid class.

In an effort to not have unnecessary classes in the project, we also removed some of the small classes in the initial UML. We turned the classes “NextBlock” and “Score” into simple fields contained in their most related classes. In doing this, NextBlock was stored in the GameData class causing GameData to own a Block.

Design

In order to maintain efficiency through our text display and graphics display, we used a relationship similar to the observer pattern for the relations between the grid and both the text display and the graphics display. This allowed us to update only the parts of the display that needed to be changed, for example when a block is placed, only the 4 cells that it occupies are updated, rather than updating every cell on the display. This is done through an overloaded updateDisplay() function which when passed to 2 int (x, y) parameters it will update the displays to contain the value in row x and column y. But when no parameters are passed it means there was a update in the gameData thus the score, level, and nextBlock are updated.

Resilience to Change

In order to support the possibility of various changes to the program specification, we implemented several of our large classes as abstract classes. For example, Block and Level are both abstract classes, meaning the implementation of any new block types or levels can all be added with minimum recompilation and change to functions, other than the new subclasses being added. Furthermore, the methods in the abstract class specify the necessary functionality required in order for these pieces to work. For example, all levels must be able to level up, level down and generate a block according to its level rules. If implementing a new level, it must have these functions. This ensures that the rest of the program will continue to work. For instance if level 6 is added, level 5 will still be accessible since level down *had* to be implemented. Moreover, the high cohesion of these two classes ensures that the methods imposed on any new subclasses are vital to the implementation of those respective objects. Thus, the new subclasses will have the flexibility to be implemented according to a large variety of program specifications, but will also allow for the rest of the program to continue to work.

Another aspect of our design that supports change is low coupling. In order to minimize coupling, most of our functions interact through Grid. Thus, other major classes such as Block and Level have minimal interaction with each other. This supports change, since when implementing a new design feature or input syntax, only Grid and the newly implemented class must be changed. Grid will be able to control those changes and communicate those to the existing classes accordingly. If modifying existing classes, again, only the classes being modified will have to change. A change to a single class will not prevent the rest of the program from executing correctly.

To build on the the topic of high cohesion and low coupling, the fact that the graphical and text displays are implemented by only interacting with Grid, again allows for modification. Any changes to the program interface are very easily implemented with minimal change to the other existing classes.

Questions in Project Specifications

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?

In our design, grid holds a field ‘boardBlocks’, which is a 2d vector of Block pointers. This contains all of the blocks that are held within the entire grid. 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, and use the ‘remove_block’ function in Grid, which would pop the front block and remove the generated blocks.

How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?
Our design has the abstract class ‘Level’. Each level is an instance of level, so to introduce additional levels into the system, we can just add another subclass of Level. Every method required for a Level is actually already declared in the class ‘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.

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 our main function, we interpret commands. Our ‘matchCommand’ function (implemented in main), we pass a different integer parameter for each command. For example, the minimum number of characters needed to match the command ‘left’ is three, since ‘lef’ is enough to distinguish this command from these others. Thus, we pass the parameter three to matchCommand.

In order to rename an existing command, we would only have to change the parameter passed to matchCommand and what the command is called. For example, instead of having ‘counterclockwise’ being the name of the command we would change this to ‘cc’ and pass a parameter of two to matchCommand. After matchCommand has determined whether the input matches a parameter (and which one it does) it passes this to grid which calls the appropriate function. When renaming an existing command, the behaviour will remain identical, since Grid will call the same functions (but only doing so when “cc” is provided instead of “counterclockwise”.

In order to support a macro language which supports a sequence of commands, we could create a queue of commands (strings) based on the specific sequence. When the macrolanguage is provided, the queue of commands would run as usual through our main function.

<u>Final Questions</u>

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

A major lesson that we learned about developing software in teams is the significance of the planning stage, especially with large programs. Originally, we did create a UML (as per the project specifications) and almost immediately after, we began coding. However, it is our contention that insufficient time was spent on the planning stage. Although we had a general idea of what classes we had to create and how they interacted with each other, minor details, such as variable names constantly impeded our ability to complete the assignment efficiently. For example, while two of us would be working on the same file, both of us would create a variable for the same purpose, but under different names. Resultantly, we would have two of the same variables. Minor errors such as this hindered our ability to complete the assignment within a timely manner.

A second lesson that we learned, was the importance of being familiar with minute implementation details. Often in class, and even when coding our first implementation, we felt as though we had a concrete grasp on the concepts/patterns being taught. However, throughout debugging our code, it became increasingly apparent that it is incredibly important to completely understand the patterns, otherwise it becomes impossible to achieve compilation.

What would you have done differently if you had the chance to start over?
As a result of the issues we experienced (as described in the first question), we believe that it would have been exceedingly more efficient to be more detail-oriented during the planning stage. We should have planned out exactly what variables, methods and minor subclasses we were planning to use.