Biquadris Outline 1

Divjot B
Bilal Nasar
Khizar Ali Chaudhry

Check UML for relations and more detail

Plan

We started on **July 15th** to piece together a rough outline as step one for the UML. This means, starting off with the main function, and adding classes/functionalities in order of their importance for the project. The command line arguments will first be processed and interpreted in main. Then the relevant functions/class will be accessed in regard to the desired input.

Implementation-wise, the two main pieces of the game are the board and the blocks. So the Board and Blocks classes are added first. Next, the immediate functionalities of the Board include the players, so a separate Player class is added. On the other hand, the blocks also have different (7) types of blocks, each one of which has a class of its own that inherits from the Blocks class. After that, each Player can have special actions so a class is added for Special Actions. Lastly, the display works via a Display class that is in composition with the Board class. Now that there's a Display class, the text-based output class TestDisplay and the graphical-based output class GraphicDisplay both inherit from the Display class.

The roles have been roughly defined. Divjot will focus on the board and blocks aspect of the project. Khizar will focus on the levels and special effects implementation. Bilal will focus on the text and graphical aspects of the project. We will first implement the basic classes needed to make a functional first version of the game. This includes the Board, Blocks, and Display classes. When these are working properly, then we can further expand with the rest of the implementation of Players, Special Actions, seven different block subclasses, and the text/graphical-based displays.

By **July 18th** the purpose and definitions of classes are to be finalized. This will give us time to think about how to implement the class methods and how they'll interact with other classes. It will also complete the UML in the process.

The class definitions and how they interact with other classes will be defined. The Board class will represent a board as a 2-D array (rows and columns) as a field. It will also have two virtual public methods GetBoard() and setBoard(int int). This now determines how the rest of the project will be implemented. Since the board is a 2-D array, the Blocks class also has fields containing a 2-D array (representing a block), a bool to determine the blind functionality, and a bool to determine the heavy functionality. Blocks also have two public methods Rotate() and CounterRotate() for rotating the blocks in each direction. Next, the Display class also has two virtual public methods printBoard() and printScore(), which print the board and score, respectively. These classes will be implemented first when the implementation stage begins in a few days.

Moving onto the subclasses of Board, the Player class will have two private fields, an int to keep track of the score, and a pointer to the Board. Moreover, the class will also have two public methods GetBoard() and setBoard(int int) each of which can override the functions from the Board class. After this, the Special Actions class has a Player pointer in its field and three public methods, Blind(), Force(), and Heavy(), for the three special actions.

The subclasses of Blocks will be similar in their design and implementation as they represent the different kinds of blocks available. Each of these subclasses will have a bool to determine if the block is dropped, and a 2-D integer array to represent the block's unique shape. These subclasses will also have two public methods Rotate() and CounterRotate() from the Blocks parent class.

The Display subclasses TextDisplay and GraphicDisplay will have access to the two virtual methods printBoard() and printScore() from the Display parent class. This will allow us to change the definition of printBoard() and printScore() depending on the interface (text vs graphical). Both TextDisplay and GraphicDisplay will also have a pointer to Board, but GraphicDisplay will have an additional pointer to Window as its fields.

At this point, the UML will be finalized and ready to submit, by **July 18th.**

Now the goal is to finish implementation based on the UML. The aim will be to stick to the UML wherever possible, and if there is a change in the process, then it will be documented. This part of the process is expected to take the longest as it'll require good communication and forming the right schedules to work together as a group. Same as with the UML design, the most important parent classes are to be implemented first. Then the subclasses. In the case of the Block subclasses, one implementation will first be tested to see if it runs smoothly. Then the rest can be implemented. This is to avoid wastage of time in case the code doesn't work or a better approach is discovered in the process. One of the last priorities at this stage will be the implementation of the graphical interface if the program itself doesn't function properly first, a graphical interface will not help.

The project code will be finalized by **July 25th.**

The next 3 days are to be used to add minor changes and extra-credit functionality. This is the most volatile phase in regard to code breaking and/or debugging. This is why after the 3 days are up, there will be no more changes to the code to add extra-credit aspects to it. Instead, the focus will shift to testing and improving the code at hand.

By **July 28th** the code will be finalized after adding as much extra-credit functionality as possible. This is to start and have enough time for the testing and debugging process.

Now, more testing will take place in order to prepare for the live demo and iron out any gaps in optimization that can be improved. This will continue for 3 more days, which the finalized version of the project will then be. All members of the group can test the code in their own unique ways. This way the testing is more in-depth and any edge cases or faults will be caught.

The group is expected to be ready for the live demo by **July 30th.**

Summarized Timeline

July 15th - Created UML
July 18th - The purpose and definitions of classes are finalized
July 25th - Finish implementation and/or add any changes needed that may differ from the UML
July 28th - Add any more necessary classes/last-minute relations and extra-credits
July 30th - Test thoroughly and optimize; finish project, and practice demo presentation

Questions

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?

We would need to add a static counter for how many blocks have fallen in the class "blocks". This would increase every time a player places a block subsequent to the current one. Every time a block is added, it would call a method that updates the counter field for each block on the grid. There can be another check for each existing block, whenever a block is added, that checks if the counter field is equal to 10, which results in the block being deleted. We would then rely on the game mechanics to move every block down until it cannot move, this deletion would not count towards the score.

Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

It could be possible that we have an abstract Level class, for which each of the actual levels inherits as subclasses. The level classes would be implemented as concrete classes, which would have the implementations for selecting blocks, special effects, and scoring methodologies. Additionally, over here we could implement the factory template method, the abstract factory class would be responsible for creating the actual instances of the levels, whereas the actual concrete levels would generate the blocks. This allows us to easily diagnose any issues with the new levels created since all bugs would be contained to that class, an advantage of the factory design pattern. Additionally, to lower recompilation, since whenever we create a new level, we just derive a concrete class (with the implementation) from the abstract class, all we need to do is recompile the new level and link it with the rest of the code. We don't need to do recompiling for the remaining dependencies of the abstract class and other derived level classes, since we aren't touching that.

Question 3: How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one *else-* branch for every possible combination?

We believe this would be best for the decorator pattern since we can add these effects dynamically, similar to how we implemented the ASCII Art program in A3. First, we would have an abstract decorator class, in which we have a method that applies the actual effect. We then implement an inheritance architecture where we create different derived classes for the actual effects (one for heavy, force, etc.). We would also need a base concrete that acts as a wrapper and actually delegates the effect calls. By implementing a decorator pattern, we prevent having singular else- branches for every possible combination, since by using this pattern we can employ as many effects as we want dynamically. If we weren't using the decorator pattern, we would need to create every single permutation of effects the user could call.

Question 4: 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 would need to create a new command_handler class, that would basically handle the reading of user input, processing them, and then calling the correct call for the Block class. For both the renaming, and macro capability, we would need to create a map <string, string>. For renaming, anytime rename is called, the first argument would be the key, and the second a value. The rename command would then map the new name to the old original command name. Thus, whenever we read in from the command line, we first check against our map to see if the user input is part of any value. A similar logic would be needed for the macro capability. The first argument would be the macro name, and the rest of the input would be stored as the value of the macro name key. Thus, whenever the command_handler recognizes a macro (through the use of the map), it would create an issue object and parses through the macro command, and delegate the relevant calls to the Block class.

TO DO:

- Rotations Bilal Nasar
- Level Class Khizar Chaudhry
- Implementations of random blocks (spawn block) - Will aggregate from levels class (use) Divjot 101
- Special Actions, Scoring
- Observer Pattern (Display Class)
- Graphical Display
- Converting Int Locations for X and Y into 2d Arrays
- Adding extras