Kabeelan Sivamanoharan (20924150)
William Lin (20897955)

Biquadris - Final Design Document

Introduction

In this document we will be detailing at a high level how we implemented all aspects of our biquadris project. We begin by discussing our due date 1 plan and UML diagram and how they compare to our actual project.

For the overall program, we did not deviate from the overall structure of our UML-diagram from due date 1 significantly. In the beginning we had missed some dependencies that we had noticed during the actual coding of the project. The Xwindow class had a dependency with the Board class and Coords class. Also the Blocks class had a dependency with the Cell class. Further, we added new functions during the actual coding process that we discovered were needed and we had made several variables and functions public in the UML as we required. Overall, our overall project was very accurate to what we had planned for during due date 1, which made the coding process less confusing and more organized as we could test our code by implementing the classes in order of dependencies.

Overview

Through the entire program we greatly relied on the concept of inheritance, and implemented the factory method in our project. We split the program into the following files; Window, Biquadris, Board, Cell, Coords, Blocks, and Levels. Within the files we had implemented the concrete classes Xwindow, Biquadris, Board, Cell, and Coords. Also, we implemented the abstract class Blocks which was inherited by the concrete classes IBlock, JBlock, LBlock, OBlock, ZBlock, SBlock, and TBlock. Further we implemented the abstract class Level which had been inherited by the concrete classes, Level0, Level1, Level2, Level3, and Level4.

The root of all the other classes is the Biquadris class. It stored the current block and next block for each turn with unique pointers. The Biquadris class was also responsible for the graphical display created from the Xwindow class, storing each board, for player 1 and player 2, along with a vector of the two board pointers, high score, and the sequence files which would be optionally given by the player when the biquadris was called. Within its playGame() method, the Biquadris class was responsible for applying the appropriate commands read in by the players, while alternating between player 1 and player 2. From this we see that Biquadris has a composition relationship with the Board class with associated multiplicity of 2. The Biquadris class also owns a Xwindow, so that is also a composition relationship.

The graphical interface for the program was made using X11 libraries. It contained functions that printed rectangles and strings onto the display. These functions depended on the Board Class to get the Coords of the board and use those to calculate and print the desired colours to visualize the pieces. Because it also used Coords, Xwindow depended on the Coords class as well.

For the basics, the Board class just maintained the board and its current state. The playable board was a vector of vectors of Cells, which is a composition relationship. We had implemented the Board class which was responsible for printing out the tetris board for player 1 and player 2, keeping track of the score, and applying the special actions which included blind, heavy, and force. The Board class also kept track of the current level the player was on since that affects the generation of the blocks, and the scoring system. The Board also has an aggregation relationship since the level then determines which Level class pointer the board has.

The Level class was an abstract class that contained the pure virtual makeBlock() method, which indicated and returned the appropriate block to be constructed and displayed depending on the current level the player was on. It was a superclass that had the following subclasses: Level0, Level1, Level2, Level3, and Level4. This is where we implemented the factory method design pattern. Level consisted of a counter to keep track of how many blocks were placed without clearing a row, a sequence file specifically for level 3 and 4, a boolean in_file specifically for level 3 and 4.

The Cell class was used to keep track of the information of blocks in each index of the board which included the type of block, the turn the block was placed, and the level the block was constructed. This way we could set each Cell in the board to the type of block it consisted of to print out the boards with the correct block type in each index.

The Block class was a superclass that implemented the general movement functions for all the subclass block types (I, J, L, O, Z, S, T). The methods included move, drop, clockwise, and counter-clockwise. We had stored a pointer to the board that the specific block belonged to (player 1 or player 2) in order to put the blocks in the appropriate board which was accomplished by the placeBlock() function and to also clear the blocks in the appropriate board which was accomplished by the clearBlock(). Finally, the Block class had an important variable which was a list of Coords which represented the positions in the board of the 4 respective blocks that make up each block type.

The Coords class consisted of two integers x, y which used to keep track of the position in the board. The Concrete classes, IBlock, JBlock, LBlock, OBlock, ZBlock, SBlock, and TBlock, inherited from the Block class and was used to set the list of Coords according to the type of block and information which included turn, level, type, and the board that the block belonged to (either player 1's board or player 2's board). Then, the movement functions stated above, could be performed on a specific block's vector of Coords in order to determine the final position of the block in the board.

Design

To overcome the basic design challenge of creating different types of blocks we relied heavily on inheritance and the generalization/specialization relationship.

The problem of creating a text-based display and a graphical one was a difficult problem to solve. The text-based display relied on the board and an output operator that printed the character for the type of block stored in each cell of the board. The graphical display similarly used the board. It used the drawRectangle method to draw 20x20 pixel squares indicating the cell each block occupied. It would clear the block at each move then reprint the board with the block at the new location.

Having our program be able to accommodate multiple levels was carried by the Level class. As stated above, the Level class was abstract which allowed the subclasses to modify the probabilities of generating a specific block. To get the random generation we used the rand and srand functions to generate numbers and assigned an interval of numbers to represent a block. The heaviness was mainly an if statement in playGame that checked if level was 3 or 4, if it was it would move it down, which is not as maintainable. The '*' block was created when the counter stored in level was divisible by 5.

One of the hardest design challenges in making biquadris was the presence of special actions. In the Board class, we had implemented the special actions by first implementing the checkRows() function and storing the special actions as booleans, heavy, blind, forced, and a char to store the block is forced to be. If blind had been enforced, this was taken care of in the standard output function which printed the board, if blind was true, the board would be printed out with question marks ('?') in the appropriate rows and columns. The implementation of forced and heavy was implemented in our Biquadris class which was responsible for reading input from each player and performing the appropriate commands.

If heavy was true, whenever the movement functions right or left were called, the function for down would be called twice on the block. If forced were true, we constructed the current block with the appropriate forced block which as stated above, was stored in each player's board if enforced. Further, all of these special actions were automatically set to false, whenever the drop function was called.

Further, as stated above we Biquadris stored the current block and next block using unique pointers. This is where our factory method came in. The factory method consisted of the abstract class Level and the concrete classes which inherit from Level which included Level0, Level1, Level2, Level3, and Level4. Level0, by default, used the sequence1.txt and sequence2.txt respectively, in the makeBlock() function, returning the current block type. Level1 and Level2, we had implemented the makeBlock() function to return the char for the appropriate block type according to the probabilities with the use of rand. In Level3 and Level4, we had implemented the makeBlock() function to return a character for the block type according to the probabilities given with the use of rand. Further, if the norandom command was called, in Biquadris we passed the name of the sequence file to Level and set the boolean in_file to true. This way when in_file was true in Level3 or Level4 we would start reading characters from the appropriate file stored in sequence file from our Level abstract class. In addition, for Level4 specifically, we had updated the counter in Level whenever a block was formed in Level4, in order to keep track whenever a 1 by 1 block was dropped in the middle of the specific player's board. The dropping of the 1 by 1 block was accomplished by the bomb() function in our Board class. With this factory method we had been able to generate characters for the specific block type to be made in Biquadris in order to construct the current and next blocks stored in Biquadris.

Implementing the command interpreter was very unmaintable. It is a series of if-else conditions that check for the multiple spellings to call the appropriate functions. As well, the prefix multiplier was taken as a substring of the input and converted to an integer. This would then loop the command the number of times specified.

Finally, our scoring system is incomplete. We currently only award points equal to (your current level, plus number of lines) squared. We have not implemented awarding scores when an entire block is completely removed from the screen. The high score is kept in Biquadris class and everytime a row is removed and score is updated, the score is checked against hiscore and if it's greater hiscore is set equal to it.

Resilience to Change

When it comes to resilience to change, our design supports the possibility of various changes to program specifications fairly well. Any additional functions for blocks can be easily implemented by implementing them in our Blocks class which will be inherited by our concrete classes IBlock, JBlock, LBlock, OBlock, ZBlock, SBlock, and TBlock. Any additional specifications for the tetris board can be also incorporated efficiently by adding these specifications to our board class. Further, any additional levels could be implemented with ease with the use of the factory method that we had designed for the given levels. The problem arises whenever new effects are added, for example level 4's rule of dropping a 1 by 1 block every time 5 blocks is placed without clearing at least one row or effects such as heavy. The code for these effects is very coupled throughout many classes. This is because we did not design our program to easily implement these effects via the decorator design pattern. As well, when new commands are added, the playGame method will have to change a lot. A new if branch needs to be created along with all the acceptable spellings must be manually checked and added. Therefore our design supports several changes to the program specification with ease unless it has something to do with an effect on the board or block since we did not implement the design pattern.

Answers to Questions

As stated in the overview, our project had formed the tetris boards for player 1 and player 2 using a vector of vector of Cells where each Cell held information of what type of block (I, J, L, O, Z, S, T) was in that Cell, the level it was made, and the turn it was placed. Therefore, we could use the turn it was placed in order to carry out the clearing of generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen. We could do this by creating another variable in the Block class which keeps track of the number of blocks that have fallen in total. Then we could create a function that could loop through the board of Cells, and if the Cell's turn dropped is 10 less than the variable that keeps track of total blocks dropped in the board so far, we could reset the Cell to have no type of block, therefore making that specific block to disappear. This would accomplish the goal asked in the question.

We could introduce additional levels into the system by using the factory method for the levels which we had discussed in the overview, which generated characters for the block type to be made according to the specified probabilities, or from the specific input file if norandom was called for levels 3 and 4. Therefore, if more levels were needed we could add more concrete classes that inherit from the abstract Level class in order to implement them. Also, if the levels have anything to do with the board, like for example the rule for level 4 which states a 1x1 block should be dropped every 5 turns blocks are dropped without clearing at least one row, this could be done easily by a function in the Board class.

We could design our program to allow for multiple effects to be applied simultaneously by implementing the decorator pattern. First we would have to create abstract base classes for both the blocks and the board as the effects could affect either the player's board or the block being placed. Then we would have to create two decorator classes, one for the blocks, and one for the board. Next, we would have to create concrete classes which would be the effects that can be applied on the board and blocks. This way we could decorate the board and blocks with effects simultaneously. If more effects were to be invented, we could just create more concrete classes in the decorator pattern in order to decorate the board or block with these new rules if required. This implementation of the decorator pattern would prevent our program from having one else-branch for every possible combination since the decorator would be able to carry out every possible combination of effects.

We could design our system to accommodate the addition of new command names by adding these commands to our function that reads in from standard input and applies the appropriate function (in our case, the Board class). We can add these additional commands as new cases to be considered when reading from standard input. Then we would have to implement these functions in the appropriate classes. We could design our system to accommodate changes to existing command names by implementing string variables in the class responsible for reading from standard input (Board class in our case) that store names for each command. So for example we could store the command counterclockwise as string cmd_ccw and set it to counterclockwise by default. These string variables will be used for each case when reading in from standard input. Then we can make changes to each command string by implementing private methods that change the string for each command. For example, if the user called rename counterclockwise cc, our private method would change the string variable cmd_ccw to cc. Now whenever standard input is read the counterclockwise command would be read in as cc instead. This way we can make changes to existing command names with minimal changes to source and minimal recompilation.

Answers to Final Questions

This project taught us several lessons about developing software in teams. Firstly, planning is very important. As a team you must be able to set realistic deadlines in order to make good progress on the development. Also you must plan in what order to develop each portion of the project so you can smoothly implement the code efficiently while testing along the way. Forming a UML diagram was very helpful in coding the project as we were able to determine the rough idea of what classes were needed and dependencies in advance, allowing us to structure our project accordingly. This also leads to the importance of communication.

You should be able to communicate with your team effectively in order to work together and even plan out everything. Further, you should not be afraid to ask for input or help whenever needed because you may be trying to fix your code for a long time, but your team members may be able to notice a bug or help in a short amount of time. Moreover, whenever coding as a team documentation is very important because not everyone can understand each other's code at the first glance, it may take some time. Therefore documentation helps team members to understand what the code is actually accomplishing, making it easier to understand. Finally, working as a team made us dependent on the git system so we were able to pull, push, and share code. This helped us learn much more about the git system which is very important when working as a team. We believe that this will help us significantly in the future in the workplace environment. Overall, this project has taught us several lessons about developing software as a team.

If we had a chance to start over, we would manage our time better. We had a little deviation from our deadlines for our plan of attack. We would make the deadlines a little sooner for each due date in order to be on schedule in case of random events which may arise. We also should consider the realistic amount of time we would spend on studying for exams on other courses which we had underestimated. We had found all other aspects of developing as a team fine. Therefore, if we were to restart we would better manage our time and push up the due dates for each component.