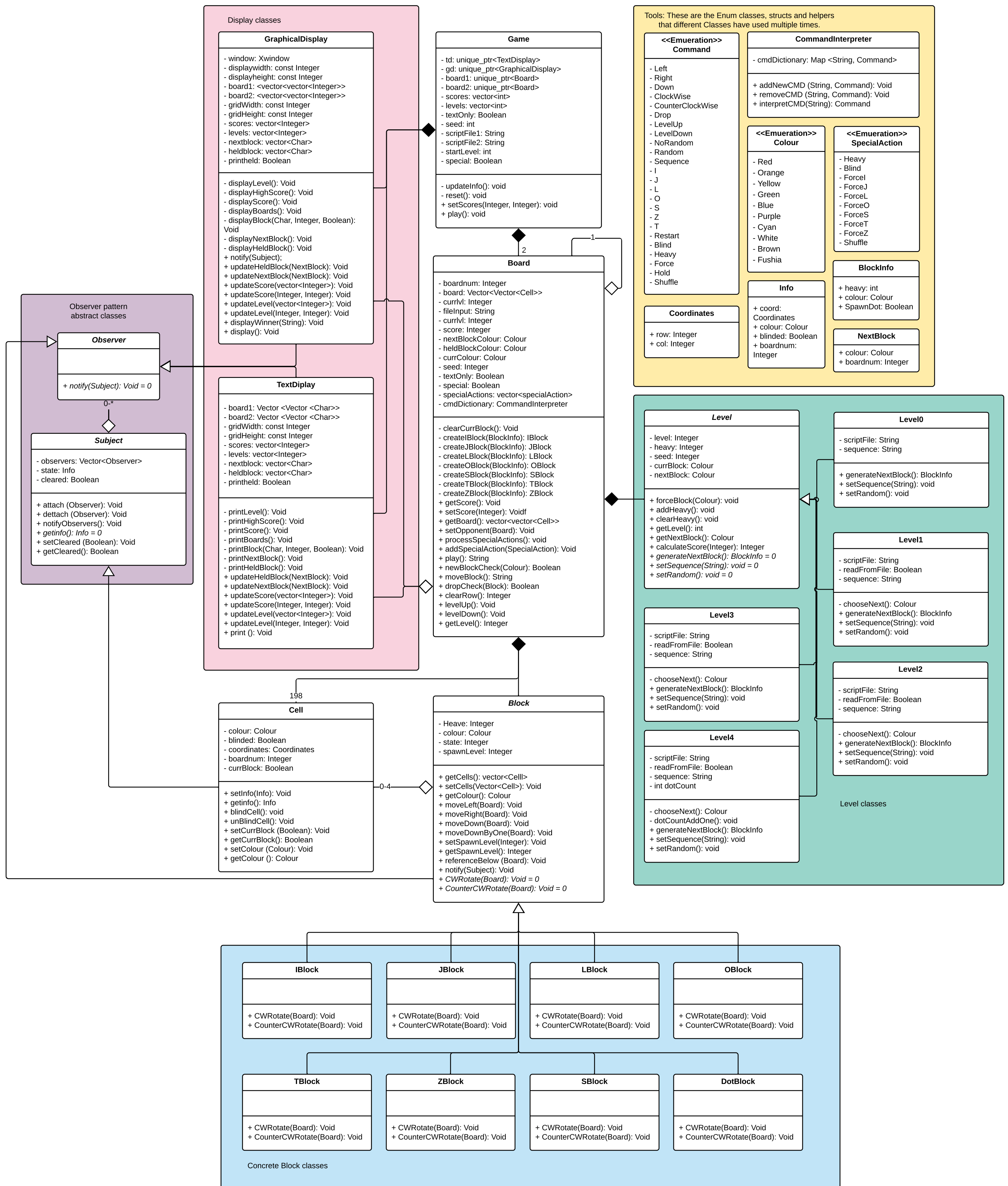


Biquadris

12.03.2019

Shida Yang (s368yang), Chengjie Huang (c267huan)



Overview

We designed and implemented Biquadris, a Latinization of the game Tetris. In the overview section of the report, we will introduce the overall structure of the program. We will introduce three major sections of the game, as well as their relationships with the minor sections.

The program starts running from the main function. The main function is used to read the arguments for most of its parts. Near the end of the main function, it creates a Game object, passes the parameter it reads to the Game object. That being said, the responsibility of the main functions is limited to only read in the parameters. The control flow of the game is handled by the Game object it owns. After the construction of the Game object, it calls the play() method of the Game object.

Three major Sections

The Game object is the first major section of the program. It is responsible for switching between the two players, and tracks information that is in a higher scope of what a single board could track, such as the highest score of the two players. It creates and owns two Board objects (one for each player), a GraphicalDisplay object, and a TextDisplay object. It notifies the two displays to update scores and levels if any change happens. It also passes the GraphicalDisplay object and the TextDisplay object to the two boards it owns, in order for them to notify and update the displays. In the play() method of the Game object, it only controls the turn switching between the two players, and leaves the rest to its boards.

The Board objects are the second major section of the program. Each Board object represents the board of a player. It is the core of the program that processes most of the functionalities. It also contains a `vector<vector<Cell>>`. It spawns the block with the corresponding level (details introduced later), takes in commands when it's the player's turn, and moves the block accordingly. When a block is dropped (i.e. end of turn), the Board object will clear the rows and calculate the scores accordingly. Lastly, it will notify the displays to update.

The last major section is the two display classes. They are simply responsible for displaying the changes on and off the boards, such as row clears, score updates, and level changes. They are notified (not as in an observer pattern) by the other major sections of the program all the time. With the display sections, our program achieves the separation of modules for "View" purpose and modules for "Model" and "Control" modules, which ensures that each module only processes one functionality.

Other Minor Sections

The rest of the program consists of the Block classes, the Level classes, the Cell class, and multiple enumeration classes, information structures, and helper classes. These are the minor sections that are controlled and used by the three major sections.

Design

Observer Pattern


We used two design patterns in our program. The first one is the Observer pattern. The only Subject we have in the program are the Cell objects. These are the individual cells on the board (11x18=198 cells in total). Upon a change, they will notify the three observers in the program: the GraphicalDisplay object, the TextDisplay object, and the Block objects. The reason why the first two are observers of each individual Cell on the board are self-evident - when the state of a cell is changed, the displays should reflect the change. The block is made to be an observer since it needs to track the states of the cells it contains after it drops. When all of the four cells are cleared, the block is considered completed "removed" from the Board, and will get a score incrementation accordingly.

RAII, unique_ptr, vector and exception safety

In addition to the 4% bonus, we chose to use smart pointers (namely unique_ptr, as we didn't use any other) and vectors over pointers and arrays to control our memory because of another reason: it achieves the RAII idiom. By achieving the RAII idiom, we essentially wrapped pointers to heap-allocated objects and heap-allocated objects, and stored them on the stack. This also ensures a basic guarantee in terms of exception safety levels, since when any exception occurs, the RAII idiom ensures that when the function stacks are popped, the destructors of these wrapper objects will be called and heap-allocated memories will be properly deleted as if no memory was ever allocated in the heap. This is also the reason why our program leaks no memory during testing.

Changes since Deadline 1: Decorator Pattern & Merge of Player and Board

There are two major changes we made in terms of program design since the first deadline. Originally we planned to implement a Decorator Pattern for Special Actions, that is, to decorate a special action with other special actions. For example, we can apply a Force special action on a Heavy special action. However, we eventually removed the feature, as we found that there is indeed no point in implementing the special actions in this way. Firstly, the three existing special actions do not decorate themselves or each other. It



makes no sense to decorate something with a layer of question mark (Blind) with the Heavy property. It also does not make any sense to Force a block twice. The processing of special actions can be achieved in simplicity by using an enumeration class and a stack of actions.

The second change we made is to merge the Player and the Board classes. There are two reasons for this decision. Firstly, these two classes are not distinct enough to be made into two classes in terms of functionalities. Secondly, it is very redundant to pass the same information through these two levels, when they can be passed in only one vertical level. For example, let's say Board1 clears a row, then it needs to update its own score, and pass the score to Player1. Player1 then passes the score to the Game class. We merged these two classes, again, to make this program less redundant and achieve simplicity.

Others

We also implemented the program with Factory Method & Dynamic Dispatch, which fits more in the "Resilience to Change" section. Please proceed to the next section.

Resilience to Change

Factory Method & Dynamic Dispatch / Abstract Class & Inheritance

We implemented the Block generations by using a factory method. Firstly, we implemented the abstract class "Level", and its concrete subclasses "Level0", "Level1", ... "Level4". These Level classes are set in a way that they each spawn blocks differently, according to the rules of Biquadris. For example, a Level3 class will spawn a block with the "heavy" property, while a Level1 class won't. On the other side, the Level classes also tracks whether the next block is "heavied" by the "heavy" special action. By doing so, the Board class outsourced the generation of blocks to the Level classes, which ensures a high cohesion of the modules.

On the other side, the Block classes are implemented in a similar way. We made an abstract Block class, and made eight concrete Block subclasses "IBlock", "JBlock", ... , "TBlock", and the special "DotBlock" that only occurs in level 4. By doing so, we can use dynamic dispatch to alleviate the complexity of controlling and tracking the blocks on the board. Now, we only need a `unique_ptr<Block>` to point to and control the moves of the current Block, and a `vector<unique_ptr<Block>>` to track the blocks on the boards.

These two features are essential to the extent of resilience to change of the program, as if there were more blocks and levels to be added into the program, we can easily achieve so by implementing additional Level classes and Block classes without modifying the rest of the programs.

Command Interpreter & Enumeration classes

We implemented a command interpreter to interpret given commands. The interpreter translates abbreviations into the Command enumerations. Were there any new commands to be added, we simply need to add a few lines of code in the command interpreter class.

High Cohesion, Low Coupling

Our program achieves a high cohesion and low coupling state, which is suitable for adding new features and modules.

We achieve high cohesion by ensuring that the program maintains the single responsibility principle, that is, each module is only responsible for one task. For example, we clearly separately GraphicalDisplay class and TextDisplay class from the rest of the program, so that the only responsibilities of these classes are to display the board. We used the factory method to ensure that Level class now is in charge of managing and generating new blocks, instead of as another task for the Board class.

On the other side, we achieved low coupling as we use the technique of pre-declaring classes in the header files, and placing the include statement in the .cc files as many times as possible. This ensures that there are no dependency loops in our program. In addition, we declared no “Friend” among classes, to ensure the individuality of the classes. Lastly, as you may notice on our UML, the program is structured in a way that there are several major sections as the skeletons of the program, and several minor sections attached to these major sections. Other than the necessary connections among the major sections, the minor sections themselves do not connect to each other, or to other major sections. There is no situation where two major sections share minor sections. This also ensures the low coupling of the program.


Answers to Questions

(including how your answers differ from the answers you gave on Due Date 1, if they did)

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?

In our design, we used the factory design pattern where a Block is created by an instance of Level (an abstract class), which is owned by a Board.

We can allow for Blocks to be cleared from the screen if not cleared before 10 more blocks have fallen by modifying the Block class. In the Block class, we can add a field that



tracks the number of Blocks that have been dropped after this Block. Since the Blocks are generated by an instance of Level, in the implementation of the generatedNextBlock() method by the concrete Level class, the Levels that are considered more advanced will generate a Block with the field initiated to 10; otherwise, it will be initialized to -1.

When the Block gets dropped into place on the Board, the Boards will check if any rows have been completed to clear them by going through the board row by row in our clearRows() method. During this process, all the Blocks that are non-negative will have the tracking field be decreased by one and if the field becomes 0, the Block will be removed from the Board.

With this design, only the Blocks generated by the more advanced fields will self-destruct, and steps that are used to implement this feature already exist within the game such as having the Level generate the block and cycling through each block after each block is placed.

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

We designed the factory method to spawn blocks. If additional levels were to be introduced into the system, we can simply add new concrete level classes that inherit from the Level superclass. Methods in all other modules need only minimal modification.

Keep in mind that the only class that relates to the Level class is the Board class. In order to have minimum recompilation, we can use a Level pointer as a private field in board.h, and thus only declare the Level class in board.h instead of directly including the class. By doing so, modification of the Level class will not require the recompilation of other classes.

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

Currently we have four special actions: Heavy, Force, Blind, and Shuffle. These are implemented as enumerations. At the end of a player's turn, if they clear two or more rows they are able to choose a SpecialActions. These special actions are placed in a vector<SpecialActions>.

If any special actions are used, we will process these actions at the beginning of the player's turn by directly applying them on corresponding objects (Blind on TextDisplay/GraphicDisplay, Force and Heavy on nextBlock field of the Level class, and Shuffle on the Board). This can be done in a loop depending on the number of elements in

the vector<SpecialActions>. Each SpecialAction that is processed will be popped from the vector.

We acknowledged the possibility that multiple heavy effects may be applied simultaneously on a new block. Therefore we added a private integer field of “Heavy” within the Level class and Block class. Whenever a heavy command is added to a player, the “Heavy” field in the Level class will be incremented by one. When a player’s turn starts, a new block will be generated by the Level class as part of the factory method. The Level class will read the “Heavy” field, and create a block with the same “Heavy” field. The new block’s moves methods will read from the “Heavy” field and decide what to do.

There are two advantages to implementing special actions in this way.

- a). This allows simultaneous uses of multiple actions without having one else-branch for every possible combination. Each special action is counted on its own.
- b). If we invent more kinds of effects, we only need to create new counters for new actions. There is no need to modify previous actions.

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.

To allow our game to easily accommodate the addition of new commands or changes to existing command names, with minimal changes to the source, we will implement our command interpreter with an enumeration called Command that be easily added to and an if statement can determine which function to execute. By using an enumeration, we can prevent having to changes to every function call whenever a command name is changed. Also, the compiler will output an error if we use an enumerated command that does not exist while it will not when we misspelled a string used in an if statement. To add a new command all we would have to do is add a new Command enumeration and an additional case for the if statement.

To support a command renaming system we have a dictionary of commands using `std::map<string, Command>` where the key is the user input and the value is the `Command` enumeration.

In order to incorporate a macro language that would allow us to give a name to a sequence of commands, we could simply change the previously mentioned implementation to `std::map<string, vector<Command>>`. Here a user can input one value and the command interpreter will execute multiple commands.

Extra Credit Features

(what you did, why they were challenging, how you solved them - if necessary)

Shuffle Special Action

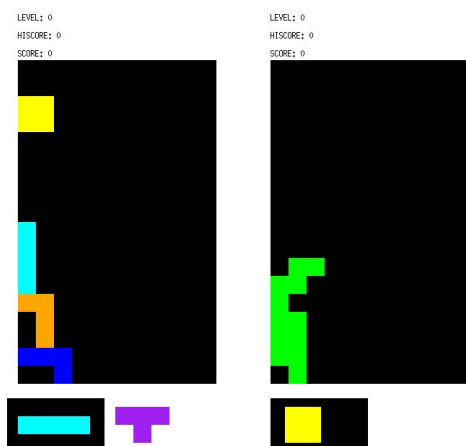
Since we feel that there is not enough fun special actions, we implemented an additional special action "shuffle". When a player clears more than two rows at the same time, they now have the choice of the special action "shuffle". When its opponent starts their turn, their board will be shuffled. All cells on each row will be rearranged in a random order. Cells on each row will not change to a different row. That is, only the column number of a Cell will be rearranged; Row number will not. This is challenging because we have to maintain the trackings of blocks on the boards, that is, the board should add the score when an entire block is cleared as before. This means even though we shuffled the cells, each block should still be tracking the 0-4 cells they have. This is achieved by using the `random_shuffle` function in the `<algorithm>` library, and renumbering the coordinates of the cells on the board. An example of a shuffle is shown below:



Before Shuffle

Clears two rows, selects "shuffle"

Holding Blocks



Player1 with a held TBlock

We implemented a feature that is part of most newer Tetris games, the Hold piece. The player is able to hold the block anytime between when it enters the playing field until it is locked in place. If the player has not previously held a block the held block will be saved until the player enters the hold command. In this case, the block that is currently in the playing field will be saved for later and the previously held block will be placed on the board. The most challenging part of implementing this feature was making sure that the block being held is fully removed from the board and the implementation of the GraphicalDisplay

Memory Management

The program was written without memory leaks and it accomplished without explicitly managing the memory.

The most challenging part of working without explicit memory management was properly planning out a program and deciding which objects are owned by whom. This was especially difficult when implementing the interactions between the Board, Blocks, and Cells.

Final Questions

1. 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?

Planning is Very Important

Developing a project from scratch requires a lot of planning. Unlike the assignments where many of the header files and test harness if given, we had to come p with everything on our own. If a plan was not made at the beginning of the project, it would be very difficult to be productive afterwards. For example, at the beginning of the project, we create a UML and made sure all group members understood why they were there and how they work to be used. This was especially useful because many methods would rely on the implementation of other methods. If one method does not do what it is supposed to do this can affect multiple other methods.

Good Communication is Key

Putting all the code together and integrating all the pieces of the project was very challenging. This took a lot longer than writing the individual pieces of code. Also, many times we changed one of the files at the same time and there would be merge conflicts. So, it is very important to keep all the members posted on any progress being made and to start merging the file early.

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

Read the Requirements More Carefully

Many times during our testing of the code, we went back to read the requirements of the project. We often found that there were features that we did not end up implementing and would have to go back and add it. Sometimes this would take a long time because we would have to change the existing infrastructure to add the feature in addition to just implementing the feature.

Compile Individual Files

At the beginning of the project, we had written multiple file of code in an effort to get a minimum viable product (game with text display, one block, level 0). After writing all this code, we used a single makefile to compile all the code at once. This resulted in many compilation errors that took a long time to debug. We should have compiled each file individually before compiling all of the file at once to make it easier to debug.