CS246 Biquadris Final Design Document

Alex Tse

Souradeep Saha

Raymond (Yi Zhen) Zhu

August 13, 2021

1. Introduction

We picked Biquadris because we all enjoyed playing Tetris when we were young. Our goal was to implement all the required features (without the bonouses) while keeping in mind how we can best structure our code so that it is not resistant to sudden changes (i.e, changes in rules, how blocks can be altered, etc..). As with any game, we sought to make our project appealing and user-friendly. We accomplished this by making many modifications to our graphics display, while keeping in mind all the features that it had to properly display. As 2A CS students, we thought that this was the perfect opportunity to exercise our creativity and work on a team before going into our internships.

2. Overview

Overall we have implemented several design patterns in the game and separated our code into individual classes. Here is a description of the classes we have:

- **Main:** This is responsible for starting the game, taking input via the command line, and handling any error which may result from the command line arguments. It is also responsible for parsing the command line arguments, and setting up the Game class. Main is strongly exception safe, in the sense that if the user enters invalid arguments through the command line, the game will throw an exception and terminate, without breaking class invariants or leaking memory.

- **Game:** The game class is responsible for playing the game and necessary helper methods needed for playing the game: such as reading in commands from the user, parsing the commands in a method such that the user only needs to type in the characters needed to distinguish the command from other commands, keeping track of the current user, their current scores, switching the user, and the highscore.

- **Level:** The level class is responsible for creating blocks in different levels of the game according to the rules. This class reads in from a file or generates random blocks based on the given probabilities and returns a block of the appropriate type. This class has a no-throw guarantee as we take care of preprocessing the input in higher level classes and this class assumes that the input is valid.

- **Board Class:** The Board class is responsible for creating the blocks, handling the levels, keeping track of the player's board and its blocks, removing them when needed, and maintaining/updating user scores. There is an AbstractBoard class according to the UML diagram and a concrete Board class which implements the methods found in Board. Decorators can be applied to the board, which is the reason for creating the AbstractBoard class. The possible decorators are heavy and blind, and we have created

them such that we could easily add more possible special actions without making major changes to the code.

- **Blind and Heavy:** As described above, these are decorators for the Board and are applied when two or more rows are clear simultaneously.

- **Block Class:** The block class is responsible for keeping track of information about individual block objects that appear on our board. Using the concept of inheritance, we make all the different types of blocks inherit from a parent block class. With the exception of the starblock, the blocks only differ in terms of their char type and their shape. Thus, their constructors differ in that different types of blocks are initialized with different coordinates. Each block carries with it its coordinates, the level on which it was created, as well as its point of rotation (which is called the bottomLeft). Whenever the board wants to rotate or move a block, it must get a copy of the would-be new coordinates from the block object. The board then checks if this move is valid. If it is, it feeds the coordinates back to the block object, and the block will update accordingly.

Changes from Due Date 1: The overall structure, design patterns and public interface remained broadly the same, however we changed some of the implementation details while working on the code. Some of the places where we made changes to the UML diagram is in the display class: we initially planned on having a display class which then manages both the textDisplay and graphicDisplay, but later we implemented them as a method in the Game class to print the Board. We made minor changes to the block, game and level classes as well to reflect the way it is currently implemented.

Test files: We have several test files named as testxxx.in, testxxx.args, and sequence files to facilitate running of test cases. These are described in greater detail in the demo plan

3. Design

Design was a large part of the project and we spent a significant amount of time designing our classes such that they can easily be modified and are responsible for only one type of functionality. We utilised design patterns and OOP principles in our project. Specifically, we implemented the following design patterns in our project:

- **Decorator Pattern:**

To allow for multiple effects to be added simultaneously, we used the decorator pattern to implement the special actions class. We note that one of the use cases of the decorator pattern is “stacking”, and dividing the current task into subtasks which are then solved by an inner object. In this specific case, we anticipated the fact that the blind, heavy and force effects might stack on top of each other; furthermore, if we invented more effects, we would definitely need to

make an else-if branch for every type of combination of actions possible. This is one of the main reasons we have used a decorator pattern for this part of the project.

- **Template Method:**

In the Level and Block classes, we realized that there was an opportunity to use the template method to implement the common behaviour and functionality of all the blocks and levels respectively in the abstract classes and thus we utilised the Template method as one of our design patterns. Then we implemented the functionality specific to each class in their individual implementation files, while leaving the methods in the abstract class as pure virtual.

- **Abstract Factory Pattern:**

We used the abstract factory pattern to dynamically generate the blocks based on the Level of the current player. Since there are 7 different types of blocks which can be potentially generated in each level, we created the Levels classes which take in parameters such as seed and filename (when necessary in the constructor), and initializes that specific level, and then generates the appropriate block based on the rules of that specific level.

- **Decorator Pattern:**

We have also used the decorator pattern on the board class to apply the negative effects, such as heavy and blind. The pattern allows us to stack these negative effects, which proves to be useful during our implementation (when we stack level 3 and 4's heaviness together with other effects), and can provide flexibility to future design changes, for example adding other negative effects.

We also followed the following OOP principles:

- **Inheritance/Abstraction:**

We realize that inheritance is a core part of OOP and thus we made parent classes for the Level and Block objects, and then made children classes which then specialized that subclass based on the rules and design of the game. We also made several abstract classes such as the AbstractBoard class, Level(Abstract level) and Block (Abstract block) classes, from which its children classes inherit. We only exposed the necessary methods for each class, and hid the details of the class using private and protected labels.

- **Single Responsibility Principle:**

We designed our classes to follow the single responsibility principle as much as possible -- that is, we separated our classes in a way such that one class is only responsible for a particular feature. For example, we made the main class responsible for taking input via the command line and then initialized a Game class to handle and play the game. In a similar fashion, we made

the Game class responsible for parsing the command, and using the board to handle playing the game. All the other classes were implemented similarly.

- **RAII:**

We have implemented some of the concepts of RAII throughout the code using smart (unique and shared pointers). However, we did use some new and delete statements as well.

- **Low Coupling and High Cohesion:**

We programmed our classes such that it reduced the dependency of one class on another (partly using the Single Responsibility Principle), and made our code cohesive such that one class contained all the necessary attributes and methods needed for its complete implementation. Furthermore, this enabled our code to be changed or modified very easily since changing one class would not interfere with another as we hid the implementation details from the end user using OOP principles (such as encapsulation and abstraction).

- **Error Handling and Exception Safety:**

We designed and implemented our classes such that they were strongly exception safe: for example, if the user enters any invalid command either while playing the game or passes them through the command line, the state of the game remains unchanged and we did not let any memory leak. Furthermore, we designed several Exception classes such as InvalidCmd and InvalidArgs to catch these errors which might be generated from code inside the try blocks.

- **Good documentation and comments:**

We have made clear and concise documentation of our code such that end-users can easily understand what a particular line/block of code does without having to read them explicitly.

4. Resilience to Change

Using the principles discussed above, we designed our code so that it would not break other classes when one class was changed or modified. For example, by implementing the decorator pattern for the Special Actions, we have made our code more adaptable to change since now we can add as many more special actions we want without hard-coding every possible combination of these actions; furthermore, these actions have the ability to stack on top of one another in such a way that they can also be removed during run time, taking care of the order they were added and removed.

Another way we made our code resilient to change is by having low coupling and high cohesion. We have created classes for every time of functionality such that our code is divided into small modules with submodules within them such that we have a lot of files, but each of them is oftentimes small in itself but they only handle one specific function. Furthermore, we have made functions whenever necessary to avoid code repetition such that if we needed to repeat a part of

the code, we could just call that function instead of repeating that block of code. This made our code more maintainable in the long run. We also have classes and abstract classes such that we can easily add children classes which override the behaviour of its parent class without having to change other implementation files.

For example, during Due Date 1 submission, we did not realize that we would need a special type of block which would be dropped in Level 4 at the center of the board; however, since we designed the abstract Block class as well, adding one more block was essentially painless as we did not need to make any changes at all to any other types of blocks or the board, or the game. This is one example of how we made our code resilient to change.

5. Answers to Questions

a. 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?

Answer: To achieve this, we can just introduce an attribute to count the number of blocks generated and one attribute to hold the current level in the Block class. Then, we would update the counter every time a block has fallen. If by the time 10 blocks have fallen and a row has not been cleared yet, we can make the new block disappear and set the counter to 0. Similarly, if we want to only have this effect in advanced levels, then we can increment the attribute for the number of blocks fallen when the level of the block equals a pre-specified level.

Due Date 2: We did not implement this feature in our code.

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

Answer: To reduce the number of compilations when adding additional levels to the game, we could have an abstract class “Level”, and add the concrete class under this abstract class. This would enable us to only compile the new Level class we just introduced and skip compiling the levels that have already been implemented. (Note that we would have to use a Makefile to prevent recompilation)

c. How could you design your program 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?

Answer: In this case, we have effects which could be added on the board to change the outcome and/or score. Furthermore, if we had multiple effects which could be simultaneously added on a board, we would need to have all the class combinations of applying these effects. The naive approach would be to create all the classes individually and instantiate the appropriate class during run-time; however, as we have seen in the course content, this is not a good approach as this is not very scalable or maintainable in the long run. A better approach would be to use the Decorator pattern, which would prevent us from creating if/else or switch statements for every possible combination of effects. The Decorator pattern would also be scalable and more maintainable since we could just create classes for individual effects and stack

them on top of each other to prevent code duplication / inefficiencies. Moreover, this would decrease the coupling of the classes as the responsibilities would be shared among the decorators instead of one superclass. Thus we plan to use the decorator pattern in the project to implement the effects.

d. 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? How difficult would it be to adapt your system to support a command whereby a user could rename existing commands? 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.

Answer: We could achieve this by having separate methods for each of the commands, and then calling the appropriate method when a command is entered by the user. In this pattern, to add new commands we would just add either a switch/case statement or an if-else statement, implement the appropriate method in the game/board classes, and call the method. To allow for renaming existing commands, we can use a map to store the commands renamed by the user, and check in the map for the original command.

Due Date 2: We did not implement this feature in our code.

6. 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?

The most important lesson that we learned while working on a team was that of communication. Obviously, with a big project such as biquadris, there are a lot of complex parts as well as many possible implementations. Thus, to save time, we found out that it was more beneficial to discuss amongst ourselves how to divide the tasks and what information would need to be shared between classes rather than how the minute details worked. For example, the updateBlock() method requires the board object to pass back the new coordinates of the block being rotated/moved. Rather than wasting precious time convincing each other why and how the block class would perform the rotation, it was much better to abstract away those details and only have the person in charge of the block class ask the person in charge of the board class to pass the required information when calling the method.

When it comes to communication, we also learned that sometimes it is not enough just to text each other about what needs to get done. Rather, we found out that it was indeed necessary for us all to hop onto a video call all at once so that any and all confusion/ambiguity would be cleared up. We discovered the importance of drawing quick diagrams when giving explanations as well as keeping an open mind to ideas that were brought up. Doing so, we were able to establish trust and we soon realized that it is ok to speak up and ask each other for help when needed.

Working on a team also strengthened our debugging skills because we often had to debug code that we didn’t write ourselves, which is something that is done often in the real world. Sometimes, it was a matter of using gdb. However, in more complex situations, we had to ask each other questions about certain algorithms and once we found the logical flaw, we had to collectively come up with a way to resolve it.

Finally, we learned about how to plan in advance as well as how to be flexible should unexpected things happen. Before starting this project, we all agreed to a tentative schedule that carefully outlined the tasks that we all had to complete by certain dates. This made us accountable to each other and ultimately made us more motivated to get things done. Doing this also allowed us to easily shift tasks around. For example, when one group member had a power outage (actually happened), it was easy for another group member to pick up where he left off that day.

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

If we had a chance to start over, we would do a couple things differently. First off, we would carefully mark certain portions of our initial UML in places where we are unsure about implementation details and in places where we believe trouble can occur. This would have made our lives easier because after coding, we would have a better idea of where all the potential weaknesses lie. As a result, we'd have a better idea of what needs to be fixed and how to fix it. Furthermore, everytime we made a change in our design, we would update the UML immediately rather than update it sporadically throughout the coding process. The reason for this is that group members could have saved time by looking at how the revised UML differs from the old one rather than asking every time a change is made. Lastly, we could have had just one block class without all the subclasses because the only thing that makes blocks different are their type character and their coordinates/shape. These two differences could have more easily been addressed by simply passing them as arguments to the block constructor rather than creating all those block subclasses.

7. Conclusion

While it was demanding work, this project was worth it in the end. We are extremely proud of what we achieved amidst our current online situation and exams in other courses. The lessons that we learned in this course pertaining to OOP and to teamwork are invaluable. Through this project, we learned how to divide up work, how to play to each other's strengths, and how to organize our code so that it's easy to read. Most importantly, we have developed a tight friendship that will stand the test of time.