Biquadris Plan of Attack
CS246 Fall 2022

Alex Zhang / William Zhang / Jason Yu

Project Breakdown

Deadline 1

Before going over the whole thing and distributing tasks to each member, we each read through the project guidelines and descriptions individually and noted down vital points that we each consider essential for our project. While meeting together to gather information, we integrated our thoughts and made a general mind map to meet the basic expectations of our implementation target - Biquadris.

After understanding the requirements and the overall structure of the program several times, we started drafting the UML as requested in the outline and simultaneously polish our design by brainstorming the implementation demands along with the specifications. Subsequently, we proceed to the plan of attack, involving the four answers to questions asked in the specification, by which we completed the requirements of submission by the deadline 1 - Nov 25.

Deadline 2

Facing the next stage of our project, we are going to break down the implementation based on the classes we have designed along with the design patterns we would like to incorporate.

We are planning to ask each group member to choose classes one wants to work on for the sake of cohesion and efficiency. To achieve a strategic manner, we will carefully assign corresponding tasks for accuracy, which will be explained thoroughly in the section on roles and responsibilities later on. With regard to the core classes and methods (e.g., the player in our UML), we should be doing unit testing to prevent debugging frustration to a maximum extent.

After the tasks are completed on an individual basis, we are going to push our code to GitHub and examine the implementation together as a group. There would be union debugging, specific modifications, and potential additions of extra features for polishment if all underlying things go well as desired. Both black box and white box testing will be performed to fully test the game. Finally, we submit our implementation and final design document by the deadline 2 - Dec 6.

Demo Presentation
We are also going to book a time slot for a demo with the TA. Two to three rehearsals will be scheduled to ensure that our demonstration would be clear, concise, and cohesive. More specifically, we are going to prepare a thorough walkthrough of the game as well as the answers of the potential questions the TA might have.

Estimated Completion Dates

The following is a list of estimated completion dates in which we will try to adhere to and should make reasonable adjustments if necessary.

- Dec 6 Deadline 2
  - Submit final program and design document.
  - Start preparing for the demo presentation.
- Dec 4 Absolute Internal Deadline 2
  - Account for any unexpected circumstances.
- Dec 3 Internal Deadline
  - Finalize the project (should be functioning properly).
  - Perform both black and white box testings to ensure the correctness of the functionalities.
- Dec 2 Bug Fixings and Improvements
  - Make improvements if necessary.
- Dec 1 Bug Fixings
  - Refining individual coding parts and push code to GitHub.
  - Debug the program.
- Nov 26-30 Coding
  - Complete individual coding.
  - Self-debugging the code by writing basic tests.
  - Perform unit testing for important methods.
- Nov 25 Deadline 1
  - Submit plan of attack and UML after final confirm.
- Nov 23 Ahead Deadline 1
  - Finalize the plan of attack and UML (should be ready to submit).
- Nov 22 Draft UML
  - Decide the general structure of the project.

Project Member Roles & Responsibilities

We will be using private GitHub repository to collaborate. Roles and responsibilities are allocated such that talents can be maximized and process loss can be minimized.

Deadline 1
Alex Zhang
- Write UML.

William Xhang
- Design UML.

Jason Yu
- Write the plan of attack.

Together:
- Draft UML
- Decide corresponding tasks
- Discuss program structure (classes, design patterns, methods).

Deadline 2 (Subject to change)
Alex Zhang
- Implement the commandLine interpreter, command interpreter, display (Text and Graphics) classes individually.
- Mainly in charge of implementing the outer framework, merging all classes, and polishing.

William Zhang
- Implement the object class and its corresponding subclasses (totalling 8) individually.
- Mainly in charge of implementing the composition relationships and the factory design pattern.

Jason Yu
- Implement the level class, its corresponding subclasses (totalling 5) individually.
- Mainly in charge of implementing the observer design pattern.

Together
- Implement the abstract classes, remaining methods, and the overall flow.
- Finalize, merge, and debug the entire program.
- Perform white box testing to identify and fix persisting bugs.
- Ask friends to partake in black box testing.

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?

Answer:

Have the abstract class of Objects be the parent class of two more subclasses: RegularObjects and DisappearingObjects. This allows DisappearingObjects to have additional properties. For example, having a counter for the number of objects placed after the current block was placed allows the block to delete itself once the counter reaches zero. Further extension of these two subclasses with the seven possible shapes of the tetris blocks. This allows the implementation of the factory method as well as allows the ratio of RegularObjects and DisappearingObjects to fluctuate based on the difficulty, or can generate specific blocks due to special actions. Using this method allows the RegularObjects and DisappearingObjects to have their own unique library of possible shapes, and it can be relatively easily implemented by adding more subclasses to RegularObjects and/or DisappearingObjects. This method also allows the properties of DisappearingObjects to be easily changed (blocks fallen before disappearance, unique letter/color representation, unique score calculations, etc.)

Question 2:

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

Answer:

By implementing the levels as subclasses of an abstract level superclass, we can easily add levels by building subclasses and can alter certain aspects such as block generation, special properties, etc. This way, only the new subclasses and the main function needs to be compiled and the other parts of the program can remain the same. By using the factory method, each level can generate their own objects.

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?

Answer:

Although the Decorator design pattern could be used, since the special actions affect different aspects of the gameplay (display, game mechanics, etc.), it can be difficult to implement. Another possible solution to this is to add boolean fields to the Player class for each type of special effects (isHeavy, isBlind). When the commands that are affected by these special actions are triggered, a single if-statement can check for each type of effect, and it eliminates the need for an if-else statement for every possible combination. If more special actions need to be implemented, we could simply add more boolean fields into the Player class's private fields.

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.

Answer:

Firstly, to implement the features required (rig -> right), a vector of strings is needed as a reference for all possible commands. By looping through the vector, we can see if any substrings of commands of length n (where n is the length of the user input) matches the user input. Using this method, adding new commands can be easy as we could easily add more commands into the vector of strings. To execute these commands, however, can be more difficult. Since we don't know the classes/functions needed by the new class, it can be troublesome. A possible solution is to have the command class implement an aggregation relationship with all the classes, and set up accessors and mutator commands to allow for the commands to be more easily implemented.

To allow users to use a macro language, a map could be implemented, where the key would be the macro for the command and the value would be a vector of strings, with each string being a valid command. Since the keys must be unique, it prevents the users from using the same macro for two or more sequences of commands. This way, the command interpreter could simply check the map to see if the input is a macro, and execute commands accordingly. To allow users to rename commands, we could implement another map, where the key would be a valid command, and the values would be the names given by the user. When the user enters a command, the command interpreter would first check to see if the input is a macro, and if so, it would execute those commands. If it is not a macro, we could use a for loop to compare the input with valid commands as well as renamed commands. If the input does match, then that command would be executed.