Patrick Huynh, Muhammad Rabee, Adrian Chung
CS246 | Spring 2021

CS246 Spring 2021 Project – Biquadris

Group Members: a63chung, mrabee, p23huynh

Introduction

Our group plans to build a fully-capable program running the Biquadris program utilizing design patterns of strategy, factory, and observer.

UML

- Board
- Block (Abstract) -> I, J, L, O, S, Z, T, Dot
- TextDisplay
- GraphicalDisplay
- Score
- BlockGame
- Level (Abstract) -> 0, 1, 2, 3, 4
- Cell
- CommandInterpreter

Plan of Attack

Block (Abstract)

- Has subclasses of differing blocks (I, J, L, O, S, Z, T, and Dot)

Board

- To store the blocks, we can use a 2D vector of size 3x3 of Cell objects. (Makes it easy to rotate the blocks accordingly) We can store the state of the board with 2D vector of size: 11 columns and 15 rows (3 extra rows at the top of the board to rotate, so 18 in total)

TextDisplay

- Print out the board by using the state of the 2D vector.

GraphicalDisplay

- Utilizing a similar structure to XWindow class from A4Q4.

BlockGame

- Controls the game flow, such as starting, restarting, initializing, switching turns, etc. Has a composition relationship with two Board objects.

Level (Abstract)

- We will use the factory method to produce blocks according to the level.

Level 0

- File names will be taken in by arguments, using ifstream to read from text files for both players.

Level 1

- Use a random number generator and modified probability distributor to select a block.

Level 2

- Use a random number generator and an equal-probability distributor to select a block.

Level 3

- Use a random number generator and modified probability distributor to select a block, which will be initialized with the "heavy" trait set as true.

Level 4

- Use a random number generator and a modified probability distributor to select a block, which will be initialized with the "heavy" trait set as true. Level four will have a unique field which stores the number of blocks placed. This will be incremented each time a block is placed and check every turn if the number of blocks placed is a multiple of five, upon which will create a Dot object in the centre of the board.

Cell

- Stores information such as if it is being inhabited, what block type is occupying it, etc.

CommandInterpreter

- Parses all input/output which is passed to the BoardGame class.

Input/Output

Command-Line Interfaces

- Since command line arguments don't change during run-time, there is no need to create a class to handle them. Instead, we will parse the command-line arguments in the main function, and pass the information to the BlockGame class when initializing it.

Command Interpreter

- During gameplay, when there is a need for additional command-line input such as rotating/moving blocks, the function within the Board class will handle the arguments as needed.

Reading Files

- To read files, they will be parsed by ifstream, which will be parsed as input regarding blocks to be created.

Timeline Breakdown

- July 28 - July 30: Develop initial classes such as the abstract Cell and Block classes and all derived subclasses to a functionable level.

    - Adrian: I, J, L.
    - Patrick: O, S, Z
    - Muhammad: T, Dot
- July 30 - Aug 1: Develop Board class to a functionable level with at least level 1 and/or 2. Implement 1-2 special actions.
    - Adrian: Board, Level 1.
    - Patrick: Level 2
    - Muhammad: Level 0, Level 4
- Aug 1 - Aug 4: Develop remaining levels to the Board function. Finish implementing special actions.
    - Adrian: Level 4, finish additional special action if necessary.
    - Patrick:
    - Muhammad:
- Aug 4 - Aug 6: Develop BlockGame and CommandInterpreter classes.
    - Adrian: CommandIntepreter.
    - Patrick: BlockGame
    - Muhammad: BlockGame
- Aug 6 - Aug 9: Develop TextDisplay and GraphicalDisplay class.
    - Adrian: TextDisplay.
    - Patrick: TextDisplay
    - Muhammad: Graphical Display
- Aug 9 - Aug 12: Debug, contribute additional documentation, and develop the design report.
    - Adrian: Complete additional documentation.
    - Patrick: Design report
    - Muhammad: Design report
- Aug 12 - Aug 13: Submit the program and report to Marmoset for marking, ensuring to submit early to avoid issues.

Collaboration

We plan to leverage GitLab to collaborate with one another to be able to actively collaborate and share code with one another to be able to remotely work. All communication can be done via social media.

Questions

Question: 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?

There are two aspects that we must consider, the first being the ability for blocks to inhibit certain traits, and the second being the ability for these blocks to only generate at advanced levels. In regards to the first feature, the decorator design pattern would allow us to decorate the different types of blocks with the disappearing characteristic. The decorator pattern would allow us to generate the base blocks and add different characteristics that we can choose to implement. Additionally, with respect to the advanced levels feature, we could utilize the factory method. This design pattern allows us to generate different types of blocks based on different conditions, which in this case would be checking if the game is at an advanced level. The combination of the decorator pattern and the factory method can help us implement blocks with a disappearing characteristic that can be generated at advanced levels. It is also important to note that we can use this same concept for any characteristic that we may want the blocks to possess and we can do this under any specified game condition.

Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?
We note that at different levels, the game changes accordingly and the algorithm used to play the game differs. The best way to do this is through the strategy method. The Strategy design will allow us to add additional levels which changes aspects of the game depending on the level. The algorithm can be changed dynamically which makes it very useful to add levels, minimizing the need to recompile. The Strategy method encapsulates each algorithm, allowing high cohesion and low coupling. Low coupling reduces the interdependence of one module on another and this is the key to minimizing recompilation as the addition of a new level will not require the recompilation of other modules.

Question: 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?
In order to apply multiple effects, we would need to implement a Decorator pattern. This would allow us, while the program is running, to modify the effects that are applied onto the block. More effects can be easily accounted for by adding more subclasses. By doing so, when multiple effects are applied simultaneously, we can account for permutations in terms of their order of application and combination of choice. This prevents unnecessary else

branches from being needed to be considered for each combination. Furthermore, the Decorator pattern easily allows the blocks to be affected by multiple effects, even effects that we choose to implement after production. The low coupling associated with the Decorator pattern is what makes it the ideal method for this scenario.

Question: 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 accommodate the addition of new command names, or changes to existing command names, we will create a class called CommandInterpreter, that will have fields for each command. Potentially accounting for such a change would be as easy as creating fields which store each individual command. To implement the rename command, we will simply change the field to whatever the user supplies. For example, if the user wants to change clockwise to cw, we will change the string clockwise variable to equal "cw" within the class. When reading input, we would need to check if the input passed matches any of the fields within the class. This requires minimal alterations, as it involves only changing the structure of one class. This implies that for renaming command names, only the CommandInterpreter class is recompiled, while supporting new commands only requires recompiling this class and other files which have seen alterations to the code for such new commands in terms of implementation. To execute a sequence of commands, we could create a vector which stores strings which can be iterated through. The strings in such a vector can be read individually as commands, while the vector itself would serve as a macro efficiently.