Biquadris

CS246 Final Project

Adrian Chung, Muhammad Rabee, Patrick Huynh

Introduction

The Biquadris Project is a functional 2-player game based on the popular game Tetris. The game is equipped with up to 4 levels and a fully-fledged scoring system. Two players go head to head, earning points as they clear rows. The game ends when the player's blocks can no longer be successfully loaded on the board.

Overview

We noticed that this project would have many different parts that would need to work together. Having everything in one class would be a disaster, so we had to apply object-orientated principles to maximize cohesion and minimize coupling. We separated our program into several major classes in order to enforce the single responsibility principle. The major classes were: **Block**, **BlockGame**, **Board**, **Cell**, and **Level**. The other classes include **GraphicDisplay**, **Interpreter**, **Score**, **TextDisplay**, and **Window**. This separation of classes allows each class to have one common goal which evidently demonstrates high cohesion.

Block

The Block class was mainly designed to be responsible for moving the blocks and rotating them around the board. It also holds fields for special characteristics that the block may have, for example, the heavy attribute.

BlockGame

The BlockGame class was responsible for storing the boards of the two players and managing when the game would start and finish.

Board

The Board class consisted of a vector of Cells that would store the state of the board. It also had the job to interact with the other classes to accomplish many board functions that included checking if a row is cleared and storing the current block in play.

Cell

The Cell class has an aggregation relationship with the Board class as the Board has many cells. Each Cell represents a spot on the board. A Cell holds many important fields that include the type of block that is inhabiting the Cell, and the unique ID associated with each block.

Level

The Level abstract base class uses the template method to store the algorithm used to generate blocks. There are five child classes currently implemented, levels 0-4, but the Level class was designed to support an arbitrary number of classes.

Design

Factory Method

Both the Level class and the Block class work together in order to implement the Factory method. Examining the game principles, we noticed that different blocks are generated differently at different levels. In essence, the blocks are dependent on the level. In our Level class, we have a purely virtual method called makeBlock() that returns the block type to be created. This is an especially useful design pattern because the Level class determines how blocks are generated.

Template Method

The abstract Level class also implements the Template Method such that the subclasses redefine the algorithm used. Since the blocks need to be generated in a certain way, the makeBlock() function is purely virtual and the subclasses define the algorithm which varies from subclass to subclass. In this case, the probability of the blocks are different as the Level changes, so each level implements different probabilities for the different blocks.

RAII Principles

Smart pointers were used instead of explicitly managing memory with new and delete. Unique pointers were used in places where there was only one owner. Additionally, smart pointers minimize the potential for memory leaks in our code.

Single Responsibility Principle

Each class was designed to be responsible for one functionality. For instance, the Level class was designed to handle the game rules based on the level. This helps us maximize cohesion for our classes. Consequently, this was also the reason why we decided to add a new class.

Resilience to Change

Abstract Base Classes

Level

The Level abstract class holds the foundational fields and methods for all level classes. We sectioned levels 0-4 into their own unique classes that change how the game works. For instance, we noticed that at each different level, the block generation was different. So to effectively implement this, we chose to use a pure virtual function that would generate the different blocks. This allows us to easily add levels that change the way the blocks are generated at different levels.

Block

The Block base class was designed with the idea that there could potentially be additions of new blocks or slight modifications to already existing blocks. That is why almost every function in the Block class was generalized to work with all the given block patterns and other ones if they were to be added. For example, the rotate algorithm was slightly difficult at first but we managed to come up with a solution that works with any general block. We decided to have all the specific blocks inherit from the Blocks superclass as a generalization relationship.

High Cohesion & Low Coupling

Our design was built mainly to support potential changes or upgrades to the classes. As mentioned prior, our Level class supports up to 4 levels. By minimizing coupling and maximizing cohesion, it allows us to easily add a new level without needing to change a lot of code from other classes. We wanted to separate the game into different components, so we created dedicated classes, as mentioned in the design, to handle each aspect, thus maximizing cohesion.

Interpreter Class

Additionally, our Interpreter class helps interpret the different commands the game supports. This is very useful because for instance, in the scenario where a command name is changed so that it is shorter, all we have to do is change the interpretation for the command line input and that is all. This makes the program very resilient to changes from a user-input aspect and reduces compilation.

Answers to 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?*

Answer: 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?*

Answer: 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 Template Method. The Template design will allow us to add additional levels which change aspects of the game depending on the level. This was a change from our previous response where we stated the Strategy Method would work here. The reason we switched is that we realized the Template Method would apply more here because there is a default way a level works, but with increasing levels, the algorithm used is redefined by the subclasses.

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?*

Answer: 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.*

Answer: To accommodate the addition of new command names, or changes to existing command names, we will create a class called Interpreter, which 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 Interpreter 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 that stores strings that 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.

Extra Credit Features

Our project was constructed without explicitly managing the memory using new or delete. This was especially challenging because we are so accustomed to explicitly managing memory. Even in CS136, we were taught to use malloc and free to manage the memory we were allocating. In order to not explicitly manage the memory, we had to use various STL's, containers and smart pointers. Smart pointers are still relatively new to us as opposed to raw pointers, so it was difficult to work with them and implement them into our program. We had difficulties with some of the STL's and smart pointers. We solved this by researching our issues and consulting the C++ references given to us at the beginning of the course. In the end, we feel that taking on this

enhancement was very beneficial to us in the long run as explicitly managing memory can be very tedious and the memory leaks that may arise can be incredibly difficult to debug.

Final Questions

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 biggest issue with developing software as a team is being able to effectively communicate with team members and ensuring everyone is on the same page. When there's a project with a lot of flexibility, we have different ways of implementing designs, and as a result, it can get really confusing. On the topic of confusion, oftentimes, a group member has done a lot of work and when another group member wants to take over, it takes time to read and understand the code the other member has implemented. Sometimes it can be too confusing to the point where we need the person to explain to us how a section of code works. This would not be an issue when working on the project solo, as every piece of code was written by yourself and it only takes a brief moment to understand how a piece of code works. From this project, we learned that in a group setting, it's very important to write clear and readable code. Even if a certain way is slightly more efficient, readability is a priority, especially in the early stages of development. Documentation is also essential in order to effectively communicate how a certain function is used or what an intricate section of code does. This makes it so that others that are picking up where you left off will know exactly how to work with that piece of code.

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

If we had the chance to start over, we would definitely put more emphasis on the planning stage of the project. We initially had a plan of what we were going to do, but upon executing the plan, we noticed that there were many things that were not specified in the plan. So we coded the program in a certain way, only to find out later that we missed a certain feature. Consequently, we had to go back and change the implementation to support the different features. This took up a lot of time and caused a slight delay in our schedule.

Changes to UML

We had many changes to the UML because, during development, we realized that we needed to modify a couple of things in our initial design. The first being a new class called GraphicalDisplay, we needed this class to handle the graphical component of the project. We realized that it would not be wise to implement it using the current classes we had. So in order to practice good object-orientated principles, we created a separate class called GraphicalDisplay. Furthermore, we added different fields and member functions to the classes to handle the

different game principles we forgot to consider. Lastly, we changed a couple of the names of member functions because we felt that the current names were not concise enough.