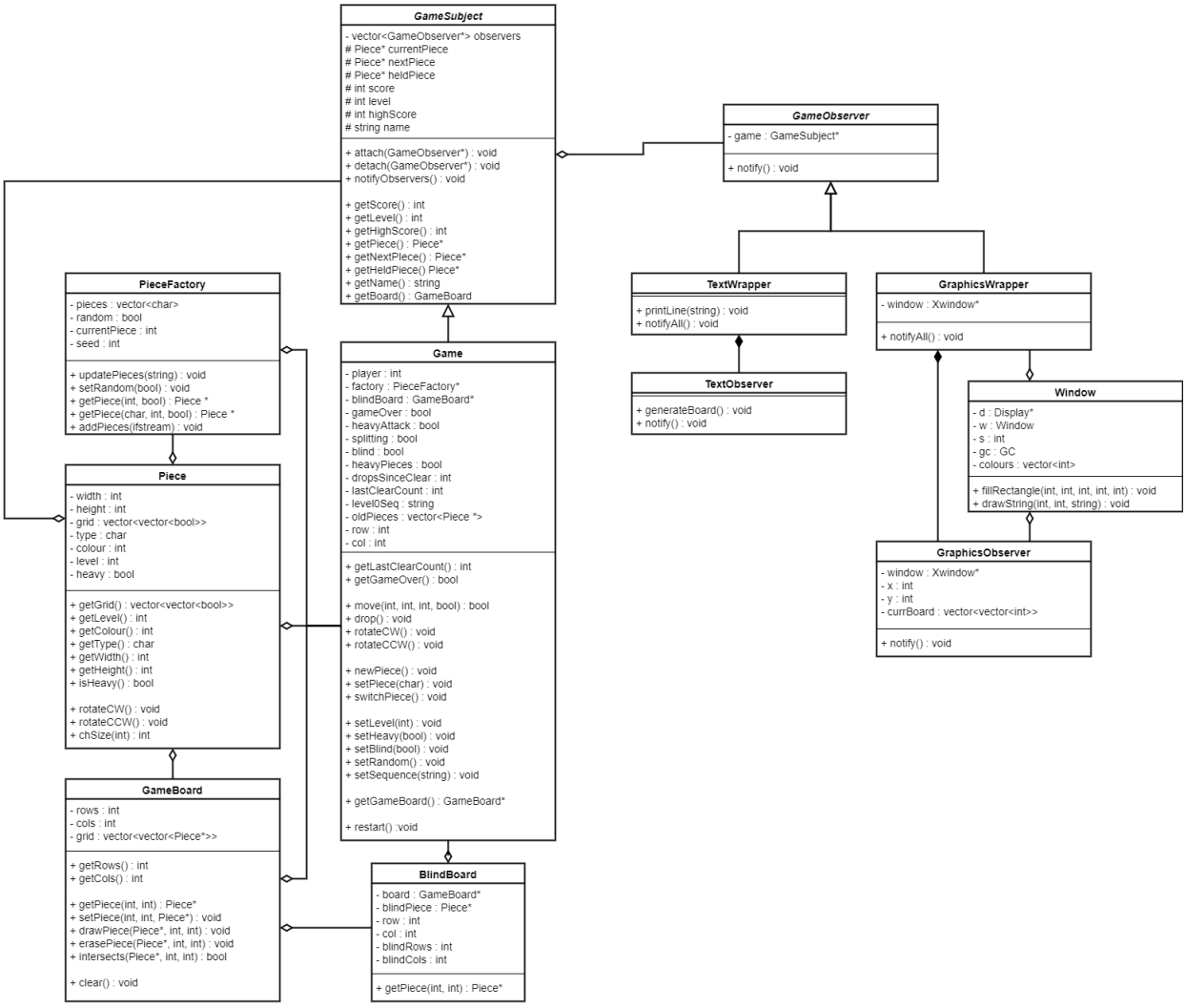
CS246: Biquadris DD2

Andrey Karmanov (akarmano) John Dong (j92dong) Nolan Zurek (nzurek)

Updated UML



Overview

The program and the classes within can be split into three main modules according to their functions.

Display

This is the most straightforward category and contains an **observer design pattern**, using the abstract classes **GameSubject** and **GameObserver**. There are concrete implementations of the GameObserver, including **TextWrapper** and **GraphicsWrapper**. These two classes create nested classes, **TextObserver** and **GraphicsObserver** which are responsible for displaying the game

state, including the scores, board, and upcoming pieces. The graphical components rely on **Window** to display using the X window system.

Logic

This portion of the program includes the Main.cc file and the Game class. It is responsible for accepting commands and modifying the state of the game. The Main.cc is responsible for taking in command line arguments, and commands during the game, converting shortforms into the full commands as well as multipliers. The Game class is responsible for creating, deleting, moving, and updating pieces on the board. It also keeps track of the score, active effects, and the level.

Utility

This portion of the program is a utility to the logical components and enables them to generate and move pieces. It contains the **PieceFactory**, **Piece, GameBoard**, and **BlindBoard** classes. These classes are designed to store the state of the game and provide pieces for play. There is a **factory design pattern**, with the PieceFactory and Piece class, where the PieceFactory generates random or sequential blocks. There is also a **decorator pattern** with BlindBoard and GameBoard. BlindBoard overrides GameBoard's getPiece(int, int) method to change the displayed board.

Changes in Design from DD1

The key differences between the UMLs and our design, is the more correct application of design patterns and the splitting of GameBoard into Game and GameBoard. We found that having the Board and Game be the same class, it would be difficult to add new features and effects which affected the display of the game. We did not correctly apply the observer patterns in the initial design, as the GameObserver had a pointer at the GameBoard class, instead of the abstract parent GameSubject. We also encapsulated the classes better, hiding the underlying implementations of classes. One example of this is creating mutators for the Game class to edit the PieceFactory it owns, instead of exposing the pointer to the factory directly. This helped with memory management, as in the first plan, there were many questions about who was responsible for deletion.

Design

Difficulties with strong coupling

We quickly found that every class seemingly needed access to data from every other class. With one such example being the GraphicsObserver needing access to the next piece that PieceFactory will generate. In our initial plan, we did not have any way to get that information without directly accessing PieceFactory. We found that the best way to solve this issue, was to have GameSubject to make better use of the already existing **observer design pattern**. We then included any and information that might be needed by an observer into the Subject, such as the

next coming piece. This was very successful, and completely decoupled the display module from the rest of the program.

Difficulties with adding on features

When working on the program, we began with the simplest features first. As we kept adding on more advanced features which required the use of multiple classes, such as piece rotation, we found that it was oftentimes confusing where it should be implemented. For the rotation, we were initially getting the piece's underlying representation (a grid) and rotating it within the Game class. This was slowing down work in addition to increasing coupling between the logical and utility modules. We then applied the **Single Responsibility Principle**, which led us to ensure that every class has a single purpose. This simplified development, as each feature could be broken down into steps for multiple classes to perform, as apposed to a single class holding full responsibility. We moved the piece rotation into the Piece class, and when moving forward ensured that every class relied on the absolute minimum of other classes' internal implementations.

Resilience to Changes

We believe that our program is highly resilient to future changes in the program specifications. This comes from very strong cohesion and minimal coupling between the three modules, display, logic, and utility. The display portion (GameSubject, GameObserver) has absolutely no knowledge on how the logic of the game is implemented. This means that if the program changes to include new commands such as the rename or macro command, or how pieces move, the display portion does not need any modifications. Additionally, the use of abstract classes such as in the **observer design pattern** enable new methods of output simple to implement, as they only have to override the notify() method to work. Also, due to the **Single Responsibility Principle,** each class has minimal coupling, while staying cohesive because each class within the three modules only has one overarching task. This means that changes to the functions within individual classes have minimal impact on other aspects of the program. This further increases the resilience of our system design. An example of the strong cohesion and low coupling is the Piece and PieceFactory classes. The Piece class can be used independently from PieceFactory, so a class like BlindBoard can use it for a novel purpose, without having to create the Piece through PieceFactory. There are also smaller effects in play, such as the lack of global variables used between classes which decreases coupling.

Biquadris Questions

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?

Nearly the entirety of the responsibility for this feature falls onto the Game class. The game class keeps a vector of pieces that are in play, and a variable can be added to the piece class

which tracks at which point they fell. Additionally, a flag can be added that is only active at higher levels, like the “heavy” flag which enables heavy piece generation. After every drop, for every active piece on the board that is above a certain age, a random number can be generated to see if it gets deleted from the board, with increasing probability the older a piece is. The selected pieces can then be deleted by checking adjacent squares on the board based on the maximum width and height of the pieces, removing them from the board if found.

Our answer has changed a small amount, as we changed classes to follow the Single Responsibility Principle more closely. This means that the Game class would have to work with the interface of the GameBoard class to make this feature happen, as opposed to implementing all aspects of the feature.

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

Levels and the effects that arise from them are entirely controlled by the Game class. As the rest of the classes are separate and no class relies on Game, only the executable and Game.cc would have to be recompiled. This of course changes if new levels have new effects, however if it is possible to achieve them with existing interfaces, still only Game.cc and the executable would have to be recompiled.

The answer changed to say there is less recompilation, due to misunderstanding separate compilation.

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?

The ideal method for solving the problem of multiple effects is the **decorator design pattern.** This can allow effects on the display to be built on top of each other, like how the BlindBoard is used. New effects can be applied on top of the existing effects, which would allow for them to act simultaneously. This would prevent the need for branches, as only if an effect is not applied, it will not be in the stack of decorators, and as such the effect will not be enabled.

If the effects are not graphical, such as a combination of numbers, they can be done in an arithmetic operation. For example, score for a line clear is calculated with $(Level + 1)^2$, but if multiplier effects are added such as combos and an effect from the opponent then the equation for score can be $(Level + 1 + combo)^2 * mult$, where $mult$ is an integer that the opponent can decrease, allowing for both the $combo$ effect and the $mult$ effect to act simultaneously. While this is a simplistic example, this can be expanded to other numbers such as the random number generator for pieces.

The answer to this question changed to expand on how effects can be combined arithmetically with an example of a new effect that wasn't previously mentioned. We also gave an example of how BlindBoard and the decorator pattern was used to enable an effect with minimal branching.

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 (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 these features would have on the available shortcuts for existing command names

This feature takes advantage of the existing structure for auto-complete of commands. The commands are stored in key-value map, such as the basic command "left" being stored as {"left", "left"}. A command like "rename counterclockwise cc" can store the pair {"cc", "counterclockwise"} into the map. Then, when a command "cc" is read in, it will be checked by the existing function **findSimilarCommand() in main.cc**, which finds the nearest (unique) match among the keys in map, and replaces it with the command in the value field. These commands are then read into a stream, and when input is needed from the user, it will be taken from the stream instead of standard input.

A macro language is similar to rename but would need some character to signify the end of the commands, possibly like "newMacro MyMacro cmd1 cmd2 … endMacro", which would get added to the map as {"myMacro", "cmd1 cmd2 …"}. As stated before, the value of the pair would be read into a stream, so the macro would be treated as any other sequence of commands when used, as string stream uses space as a delimiter. One issue that can arise is recursion in the macros, so care would have to be taken. A simplistic approach is to limit the user to creating one macro, and not allowing to call the macro within itself.

The answer did not change significantly, however more detail was added on how existing frameworks in the program can be used to enable the macro and rename feature. We also implemented the rename feature, so the answer was backed up by evidence.

Extra Features

We have added several extra features for the user to experience. They can

1. Play with up to 4 people at a time, using arg [-players n]
2. Change the number of rows and columns that they play on using [-rows n] and [-cols n]
3. Rename commands during play, without losing their turn, using [rename command newname]
4. Hold pieces, allowing to save them for a better turn. [hold]
5. And best of all, create their own pieces of any size to play with!

See customExample.txt and use command [-custom f] in combo with [-scriptfile f]

The first four features were a challenge because of the planning associated with scaling. It would have been very simple to hardcode all values, including graphics, board display, piece spawning, and command loops. To facilitate scaling the players and board size, everything we did had to be relative to the number of players and board size. This brought in bugs and other considerations we had to account for. We took great care to decrease the coupling required, by passing arguments during construction instead of carrying global variables for number of players and board size.

The last feature, creating your own pieces was more challenging than the other three combined. We added this feature by allowing the user to pass in a text file, that's formatted like the example is. The users can modify the scriptfiles to create a custom sequence of pieces, or clear lines in game and attack other players with them through the force command. To implement this feature, we had to work around the already completed PieceFactory, where we added file reading and a static array to hold the custom pieces. We were having performance issues when using large pieces and large grids, so we had to add optimizations to rotations and save graphic redraws. Additionally, debugging was difficult due to the difficulty of seeing how the arrays were being read into the program. We learned to use gbd to help with this issue.

Final Questions

What lessons did this project teach you about developing software in teams?

The first issue came from the early design stages, from the first UML. We did our best to design the best plan for the project that we could, so we could split up tasks and begin working quickly. We very quickly ran into issues where there were fundamental flaws in the interfaces between the classes, meaning that we could not finish our classes independently. An example of this is how we initially the GameSubject didn't have access to the score, high score, and the upcoming piece, as described in the issues section. Since we were working independently, it was time consuming and difficult to plan how to proceed in our own portion of the project, as we had not reached a consensus on what different classes should handle. We learned a valuable lesson about working together closely during the early stages of the project. Specifically, doing preliminary sanity checks, running through step-by-step how a program would perform basic tasks like displaying pieces helped us effectively re-design and update our class structure.

The second issue that we learned from, was the difference between member's understanding of the requirements of the program. It was common to run into a feature that one or more members simply did not know we had to support. While each feature was relatively simple, such as displaying the next piece that is coming up, it distracts from other tasks and requires a context switch from other work. We also had our own interpretation of what the instructions were asking for, such as how pieces were meant to rotate which led to long deliberation. While

these individual issues were small, in our eyes we could have easily solved them by planning ahead and ensuring that each member understands every aspect of the program.

We also learned some good lessons on how to effectively use version control, as had a lot of success separating individual development with branches, resulting in each person having a clean environment to work in, without the chance of someone inadvertently causing issues like segmentation faults.

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

We are happy with how our program turned out and enjoyed the project overall. However, there are aspects that we would change. Firstly, we would apply the learnings that we gained, planning more, with run-throughs of every function and what it's purpose would be. Secondly, we would use more standard library components, as it would give us a chance to see what's out there and how we can use it effectively. The standard library is very powerful, and when we used it, it quickly simplified or even eliminated functions we made. Using it from the very beginning, we think we could save time and effort, while also creating better code.