Introduction

This report outlines the design of Biquadris, a two-player competitive version of the classic Tetris game, developed by Isabella Xu, Qiguang Chen, and Jeffery Zhang. The project extensively employs Object-Oriented Programming principles, including Encapsulation, Data Abstraction, Polymorphism, and Inheritance. Additionally, the implementation makes use of various design patterns, such as the model-view-controller architectural pattern, decorator pattern, and factory method pattern. The game is thoughtfully modularized, boasting low coupling and high cohesion, resulting in a well-organized and maintainable codebase. In the following sections, we present an overview of the game's design, features, and functionality to provide a comprehensive understanding of Biquadris.

Overview

Main

Our game begins in the main class, where we handle any input from the command-line, and instantiate a Game object.

Game

The Game class plays the role of the “controller” in the model-view-controller design pattern. It utilizes two pointers to AbstractBoard objects, serving as the abstract base class for the game board implementation, one for each player. Additionally, the Game class holds a pointer to the View class responsible for managing the visual representation, and plays the role of “view” in our design pattern. Within the Game class, various data fields store each player's initial game options. It further includes a command interpreter method to process player inputs and a rendering method to display the game board visually.

AbstractBoard

The AbstractBoard class serves as the abstract base class for the game board in our implementation. It employs a 2D vector of Cell objects to represent the grid, where each cell corresponds to a specific location on the board. For instance, an 18 x 11 grid is represented as a vector of size 18, containing vectors of Cells of size 11.

Additionally, the AbstractBoard class maintains a connection to the Block class, which is responsible for generating, modifying, and managing the "blocks" in the game. The class includes a pointer to the current Block object on the screen, allowing for easy tracking and manipulation of the active block.

The AbstractBoard class features a range of methods to handle various aspects of blocks and general board information. Some of these methods include moveBlock() for moving blocks within the grid, rotateBlock() for rotating the active block, createNext() for generating the next block to be displayed, and more.

Board

The Board class is a concrete implementation that inherits from the AbstractBoard class. This inheritance allows each player (board) to have distinct game interactions based on their initial game options. As the model in the model-view-controller pattern, the Board class holds all player-specific information, such as playerID, score, and level.

By overriding methods inherited from the AbstractBoard, the Board class customizes the behavior of these methods to suit the specific player's gameplay requirements. This enables each player to have individualized interactions with the game board. The Board class serves as the center-piece of the game, where most of the data visible to the player is stored and managed.

Block

The Block class is responsible for storing information about a block and managing its movements, including rotations. It tracks essential block information, such as type, creation level, special effects, and position on the grid. The class handles the on-screen representation of the block by filling out Cells on the board based on its coordinates and provides functions for movement management, as well as getters and setters for block information. The Block class also has a vector member that holds the positions corresponding to where the block's cells would be initially placed on the game board. The Block class can further utilize this vector to keep track of the block's current position and adjust it as the block moves or undergoes rotations.

Level

The types and special features of Blocks generated are determined by the player's current level. The Level class plays a crucial role as the "next block generator." It is responsible for generating the type of the next block based on the player's current level. Once the next block type is determined, the Level class also assigns a random initial orientation to the block and passes this information to the NextBlock class for the block to be generated with the specified type and orientation on the game grid.

Decorator

The Decorator class implements the decorator design pattern to efficiently apply "special effects" to players. It inherits from the AbstractBoard class and acts as an interface for special action classes like BlindBoard and HeavyBoard. By serving as an intermediary, the Decorator class allows these classes to modify or extend the functionality of the AbstractBoard without altering its core implementation.

BlindBoard

The BlindBoard class is a component of the decorator design pattern, inheriting from the Decorator class. Its purpose is to activate the "blind" special action effect on the opponent's board.

HeavyBoard

The HeavyBoard class is a component of the decorator design pattern, inheriting from the Decorator class. Its purpose is to activate the "heavy" special action effect on the opponent's block movements.

Cell

The Cell class serves as the building block for the in-game grid. It represents individual positions on the game board and contains essential information about each cell, including its x and y position on the grid and the pattern occupying that specific space (if any). With methods to access and modify the information about the cell, it allows other components of the game to retrieve and update data related to individual cells as needed during gameplay.

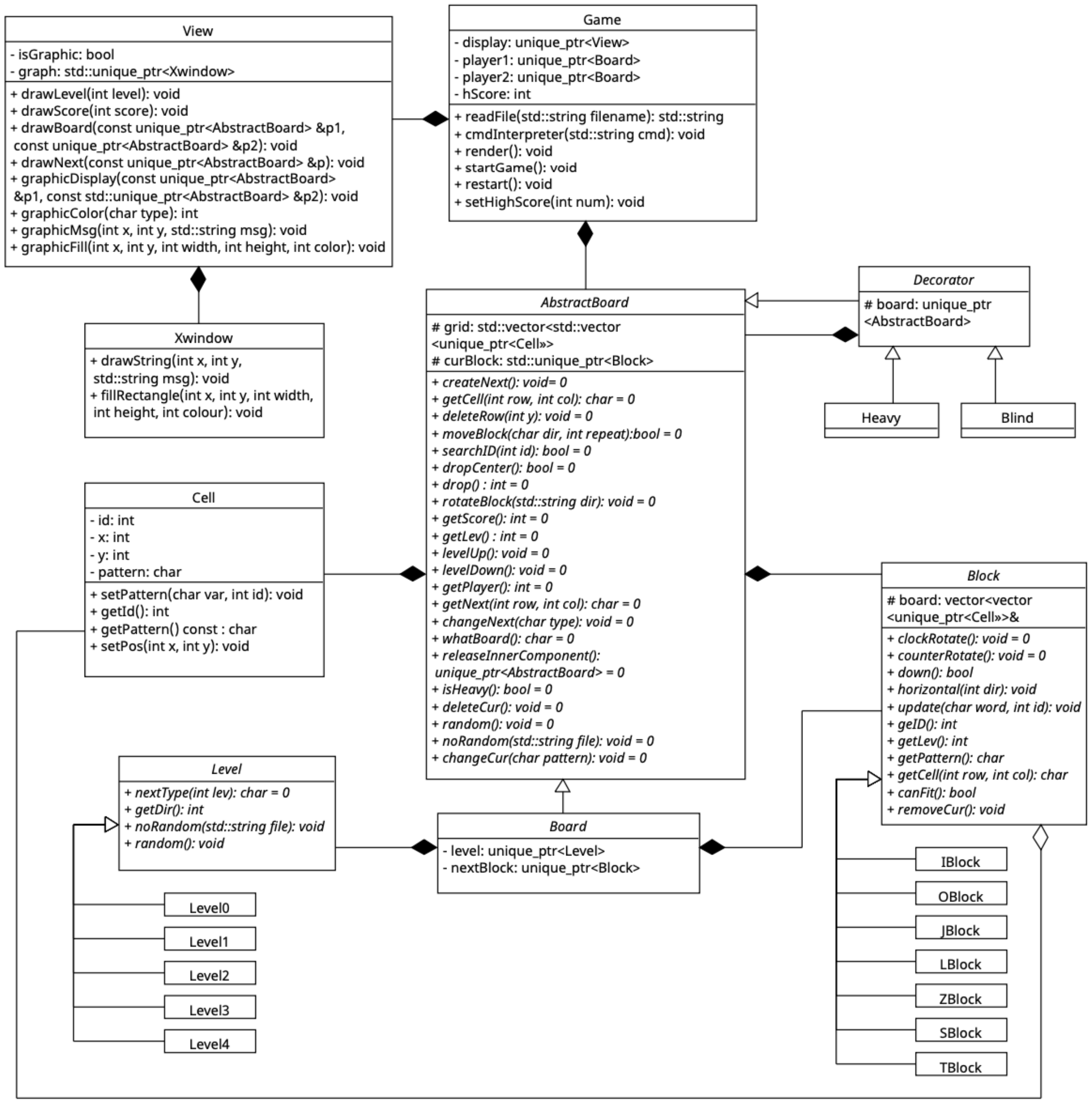
View

The View class is responsible for managing the both the text display and graphical representation of the game. It utilizes the Xwindow library to handle graphics and rendering. The class contains a pointer to an Xwindow object that we constructed in class Window, and has methods to update and draw the graphics. It also contains method for Controller to call for displaying special commands or messages.

Window

The Window class provides a graphical window interface for the game, allowing for visual rendering and user interaction. This class is defined in the "window.h" header file and includes several functionalities to draw shapes and display text in the window.

UML


View
- isGraphic: bool
- graph: std::unique_ptr<Xwindow>
+ drawLevel(int level): void
+ drawScore(int score): void
+ drawBoard(const unique_ptr<AbstractBoard> &p1, const unique_ptr<AbstractBoard> &p2): void
+ drawNext(const unique_ptr<AbstractBoard> &p): void
+ graphicDisplay(const unique_ptr<AbstractBoard> &p1, const std::unique_ptr<AbstractBoard> &p2): void
+ graphicColor(char type): int
+ graphicMsg(int x, int y, std::string msg): void
+ graphicFill(int x, int y, int width, int height, int color): void
Game
- display: unique_ptr<View>
- player1: unique_ptr<Board>
- player2: unique_ptr<Board>
- hScore: int
+ readFile(std::string filename): std::string
+ cmdInterpreter(std::string cmd): void
+ render(): void
+ startGame(): void
+ restart(): void
+ setHighScore(int num): void
Xwindow
+ drawString(int x, int y, std::string msg): void
+ fillRectangle(int x, int y, int width, int height, int colour): void
AbstractBoard
# grid: std::vector<std::vector<unique_ptr<Cell»>
# curBlock: std::unique_ptr<Block>
+ createNext(): void= 0
+ getCell(int row, int col): char = 0
+ deleteRow(int y): void = 0
+ moveBlock(char dir, int repeat):bool = 0
+ searchID(int id): bool = 0
+ dropCenter(): bool = 0
+ drop() : int = 0
+ rotateBlock(std::string dir): void = 0
+ getScore(): int = 0
+ getLev() : int = 0
+ levelUp(): void = 0
+ levelDown(): void = 0
+ getPlayer(): int = 0
+ getNext(int row, int col): char = 0
+ changeNext(char type): void = 0
+ whatBoard(): char = 0
+ releaseInnerComponent(): unique_ptr<AbstractBoard> = 0
+ isHeavy(): bool = 0
+ deleteCur(): void = 0
+ random(): void = 0
+ noRandom(std::string file): void = 0
+ changeCur(char pattern): void = 0
Decorator
# board: unique_ptr<AbstractBoard>
Heavy
Blind
Cell
- id: int
- x: int
- y: int
- pattern: char
+ setPattern(char var, int id): void
+ getId(): int
+ getPattern() const : char
+ setPos(int x, int y): void
Block
# board: vector<vector<unique_ptr<Cell»>&
+ clockRotate(): void = 0
+ counterRotate(): void = 0
+ down(): bool
+ horizontal(int dir): void
+ update(char word, int id): void
+ geID(): int
+ getLev(): int
+ getPattern(): char
+ getCell(int row, int col): char
+ canFit(): bool
+ removeCur(): void
Level
+ nextType(int lev): char = 0
+ getDir(): int
+ noRandom(std::string file): void
+ random(): void
Board
- level: unique_ptr<Level>
- nextBlock: unique_ptr<Block>
Level0
Level1
Level2
Level3
Level4
IBlock
OBlock
JBlock
LBlock
ZBlock
SBlock
TBlock

UML is mainly updated from DD1 by changing the observer pattern to MVC model newly introduced. The polymorphism is also further strengthened by implementing more virtual methods. The Block class that was originally part of the decorator pattern is moved out as composition due to individual functionality and easier implementation.

Design

Game

In the MVC design pattern, the Game class assumes the role of the controller, responsible for managing in-game data and facilitating communication between the model and view components. To ensure proper ownership and sharing of resources, we employed unique pointers for the two AbstractBoard objects representing each player's board and the View object, indicating that the Game object has ownership of the player's boards while they share a display.

By using unique pointers, we effectively manage ownership and lifetime of the AbstractBoard and View objects within the Game class. This ensures proper cleanup and prevents memory leaks. Because Game is our controller, It coordinates the flow of information, allowing data from the AbstractBoard to be displayed in the View, and user input from the View to be interpreted by the AbstractBoard. It also encapsulates the in-game data and provides a high-level interface to interact with the model and view components. This abstraction enables easy maintenance and modification without affecting other parts of the system.

With access to both player boards, the Game object can easily apply special effects, such as the "blind", "heavy" or “force” effects, to the opponent's board. The Game object also allows easy updating of global data (e.g. if a player has lost, highscores, etc.)

AbstractBoard & Board

The AbstractBoard class serves as the abstract base class for our boards. We employ a 2D vector of Cell objects to represent the grid. The AbstractBoard class features a range of virtual methods to handle various aspects of blocks and general board information to be overridden in the boards.

As a concrete implementation of the AbstractBoard class, the Board class plays a pivotal role in the Biquadris game. By inheriting from the AbstractBoard, it benefits from the common functionalities and interfaces while allowing for customization and distinct game interactions for each player (board) based on their initial game options. By utilizing polymorphism and inheritance, the Board class can inherit essential functionalities from the AbstractBoard class while providing specific implementations tailored to each player's requirements. This approach enables code reuse and promotes a more organized and extensible design.

As the model in the model-view-controller pattern, the Board class effectively encapsulates all player-specific information, including playerID, score, and level. By centralizing this data in the Board class, the game's model can easily manage and update player-related details. The Board class abstracts away player-specific information, allowing the rest of the game components to interact with players without needing to know the underlying details. This abstraction enhances code modularity and reduces dependencies between different parts of the game. By having the

Board class act as the model, it decouples the data representation from the graphical representation provided by the View class. This clear separation of concerns promotes maintainability and code readability.

Block & Level

The Block class takes charge of storing and managing essential information about a block during gameplay. This includes the block's type (e.g., 'J' or 'I'), the level at which it was created, any special effects associated with the block, and its current position on the game grid. By encapsulating block-related data and movement logic within the Block class, it achieves high cohesion, focusing solely on block-specific functionalities. The createNext() method in the Board class acts as a block generator for the game. It is responsible for producing the next block type, and initial orientation based on the player's current level. The Level class employs the Factory design pattern. Given a level, the nextType() method in the Level class calls the corresponding factory method that randomly generates the information for the next block (based on level). By using the Factory pattern, the Level class can be easily extended to support new levels, block types or variations without modifying existing code. It allows for a modular and flexible approach to block generation. This separation of block generation logic from the Block class keeps the responsibilities well-organized, promoting low coupling between the different aspects of block management.

Decorator, BlindBoard & HeavyBoard

We utilized the Decorator Design Pattern to efficiently apply "special effects" to players during gameplay. The Decorator class inherits from the AbstractBoard class and acts as an intermediary between the AbstractBoard and the special action classes. It provides a common interface that special action classes can use to modify the behavior of the player board. The BlindBoard and HeavyBoard classes are concrete decorators that inherit from the Decorator class. They represent the specific special actions in the game, i.e., "blind" and "heavy" effects. These classes extend the behavior of the AbstractBoard by adding new functionality, such as blocking visibility or imposing block weight, respectively. Instead of creating a separate class for the "force" special action, we decided to implement it as a command directly in the Game class. This command calls certain methods that allow players to change the opponent’s current block, effectively activating the "force" special action.

Using the Decorator Pattern promotes code reusability, flexibility, and maintainability. It enables us to add new special actions or modify existing ones without changing the structure of the AbstractBoard or introducing complex conditional logic. It also allows us to apply “stacking effects” efficiently which improves the gameplay.

Window & View

The View class is responsible for handling the user interface and rendering text in standard output, and graphics using the Xwindow library. The View class contains a pointer to an

Xwindow object, which allows it to interact with the graphical window. The View class has methods to draw various game elements on the graphical window, including the game board, player scores, levels, and next blocks. It utilizes the Xwindow library to efficiently render graphics and update the visual representation of the game. As the “view” role of the MVC design pattern, the View class helps maintain a clear separation between the user interface and the game's underlying logic. This separation enhances the code's modularity, making it easier to modify and maintain the different components independently. This division of responsibilities promotes code reusability, scalability, and overall code organization.

The Window class provides a graphical window interface for the game. It allows for drawing basic shapes such as rectangles and lines on the graphical window. This capability is essential for representing game elements like the game board and blocks. The Window class supports multiple colors, enabling the View class to render colorful graphics and create an engaging visual experience for players.

By combining the View class with the functionalities provided by the Window class, the Biquadris game achieves an appealing and interactive graphical user interface. The View class serves as the bridge between the game logic (AbstractBoard) and the graphical representation (Window), ensuring that the game's state is accurately reflected in the visual output presented to the players. Through the Window class's capabilities, the View class can efficiently draw and update graphics, enhancing the overall gaming experience.

Resilience to Change

The design of the Biquadris game demonstrates resilience to change by adhering to principles of modularity, encapsulation, and abstraction. These design principles allow for various changes to the program specification with minimal impact on other components, making the codebase more flexible and maintainable.

Modularity: The program is divided into distinct modules, each responsible for a specific aspect of the game. For example, we have separate classes for Block management, Board behavior, and View for graphical representation. Each module operates independently and communicates through well-defined interfaces, making it easier to modify or replace one module without affecting the others.

Encapsulation: Each class encapsulates its functionality and hides internal implementation details from other classes. For instance, the AbstractBoard class encapsulates the core logic of the game board, and the View class encapsulates graphical rendering. This encapsulation ensures that changes made to the internal implementation of a class do not ripple through the entire codebase, limiting the scope of potential changes.

Abstraction: The use of abstract classes, interfaces, and design patterns (e.g., Factory, Decorator) allows for a high level of abstraction. For instance, the Decorator pattern provides a flexible way to add or modify special effects without altering the core behavior of the AbstractBoard. Similarly, the Level class abstracts the next block generation process, making it easy to extend or customize block types and block spawn rates.

Overall, the design of the Biquadris game demonstrates flexibility and adaptability to changes in the program specification. By following good OOP design principles and patterns, the codebase is well-structured, making it easier to add new features, improve existing functionalities, and maintain the codebase over time. As the program evolves, the design's resilience to change ensures that modifications can be made efficiently, reducing the risk of introducing bugs and improving the overall development process.

Answers to Questions

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?

- To allow blocks to disappear if not cleared before 10 more blocks have fallen, we can implement a block drop counter that resets upon block clearance. Once the counter reaches 10, specific blocks can be removed from the screen. To confine this feature to advanced levels, we use the features of our Block class. Specifically, our block’s id and level field. We can then remove blocks in order of their id and based on their level.

Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

- To accommodate the possibility of introducing additional levels into the system with minimum recompilation, we can extend the Factory Design Pattern utilized in the Level class. We can add a new factory method specifically tailored to generate the next block for the newly introduced level. This new method will handle the block generation logic based on the characteristics of the new level without affecting the other existing classes. Other classes, such as the Game, Block, and Board classes, will remain unchanged, and no recompilation is required for them. This design approach enhances the flexibility and extensibility of the program, allowing us to easily introduce new levels without disrupting the existing codebase.

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?

- To allow for multiple effects to be applied simultaneously, we can leverage the Decorator Design Pattern for the special effects. When a player activates a specific effect or multiple effects, we can decorate the player's board with all the corresponding decorator classes. Each decorator will introduce the specific behavior of the corresponding effect without modifying the existing classes. If we invent more kinds of special effects, we can simply implement new decorator classes following the design pattern. This design approach ensures that the addition of new effects does not affect other classes and keeps the codebase modular and maintainable.

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.

Using our command interpreter in the Game class, we create a key-value dictionary, to store the command names (strings) as keys, and store an array of methods as values. This way, we can add / remove commands, and also create macros with a sequence of commands (more than 1 method in the array).

Extra Credit Features

The project is without leaks, and does not explicitly manage any memory. That is, all memory management is via vectors and smart pointers.

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?

- This project taught us that effective communication is crucial when developing software in teams. Miscommunication can lead to inefficiencies, such as implementing redundant code or working on tasks that are not needed. Regular team meetings and clear documentation can help ensure that everyone is on the same page and working towards common goals. Additionally, finding ways to split the work fairly among team members is essential for maintaining a balanced workload and promoting collaboration.

- Collaboration and version control are key lessons learned in this project. When working in teams, utilizing version control systems like GitHub allows for easier collaboration, better tracking of changes, and the ability to roll back if needed. Consistent code formatting and adhering to coding standards help in maintaining a consistent codebase and understanding each other's code. Moreover, frequent testing and debugging are essential to catch issues early and ensure the program's stability and reliability.

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

- If we had the chance to start over, we would allocate more time to thoroughly plan the overall structure of the program before diving into individual class implementations. By developing the interfaces and defining the main structure first, we could ensure better cohesion and consistency among classes when they are linked together later on. This approach would minimize potential issues and save time on refactoring code to fit the desired program architecture.

- Additionally, we would focus on documenting the code and maintaining clear documentation throughout the project. Comprehensive documentation, including comments within the code and external documentation, helps other developers understand the program's functionality and design choices. This documentation would facilitate easier maintenance, teamwork, troubleshooting, and potential future enhancements to the program.

Conclusion

In conclusion, our project involved designing and implementing the Biquadris game, a modified version of the popular game Tetris, with expanded features for two-player competition. Throughout the development process, we applied major Object-Oriented Programming concepts, such as Encapsulation, Data Abstraction, Polymorphism, and Inheritance. Additionally, we utilized several design patterns, including the model-view-controller architectural pattern, decorator pattern, and factory pattern, to achieve a well-structured and flexible program.

Our team focused on modularity, low coupling, and high cohesion, resulting in a codebase that is easier to maintain, extend, and understand. Effective communication and collaboration were key to the success of the project, as we worked together to split tasks and ensure a fair distribution of workload.

Looking back, we recognized the importance of thorough planning and clear documentation in maintaining consistency and avoiding potential issues during development. Testing was emphasized to catch bugs early and validate the correctness of our implementation.

Overall, the project provided valuable insights into software development in a team setting, teaching us the significance of effective communication, planning, and collaboration. It also strengthened our understanding of design patterns and Object-Oriented Programming principles, preparing us for future projects and challenges in the world of software development.