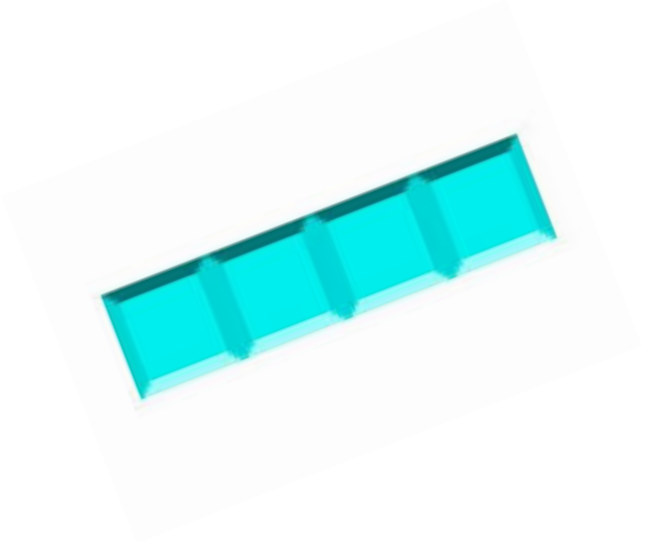
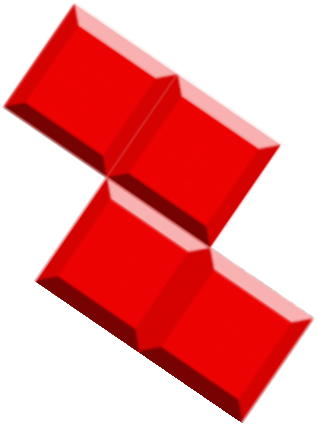
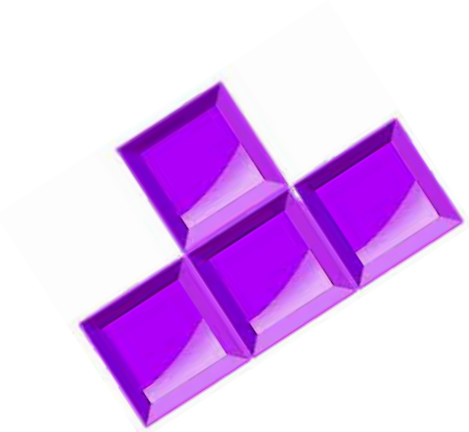


Final Project: Biquadris

CS 246 Assignment 5



Sharuga Suthakaran
Tanisha Gottemukula



Introduction

Biquadris is a unique spin on the well-known, classic game of Tetris, where two players play simultaneously with 8 unique blocks. The game supports multiple difficulty levels, special actions, scripted sequence files, a block-holding feature, and both graphical and text displays.

The objective of the game is to clear as many rows as possible by stacking these blocks on top of one another. Our implementation focuses not only on reproducing classical mechanics but also on designing our codebase around strong object-oriented principles. Our design has evolved significantly from our initial submission into a more modular, stronger architecture.

Overview

From a top-down view, the project is structured in the following major components:

1. The Model
2. The Controller
3. The View

The Model encapsulates all persistent game state and gameplay logic. `Grid` and `Cell` are the basis of the data structures that represent a board's contents. The `Block` hierarchy models the geometry and rotation behaviour of all block types, while the `Level` hierarchy determines how blocks are generated and how difficulty evolves. Each `Player` maintains its own grid, current block, next block, held block, level logic, and scoring information.

The Controller is implemented through the `GameEngine` and the `CommandInterpreter` classes. The engine manages turn-taking, interacts with players and coordinates the game flow, whereas the interpreter receives user input and expands it into fully *normalized* commands. This includes resolving abbreviations, validating prefixes, handling numeric multipliers and more.

The View is comprised of both `TextDisplay` and `GraphicsDisplay`, with the main purpose of responding only to updates issued by the engine; they do not know gameplay rules. Each display verifies the current state from the engine, interprets it and renders the appropriate visual representation.

The Model maintains state, the Controller instructs the Model, and the View observes any changes in the Controller. Together, this design supports modularity, reduces circular dependencies and allows for easy extensibility.

Updated UML

[insert image]

The updated UML (shown above) reflects a more structured and interconnected design than the one we had originally proposed.

In the Model layer, `Block`, `Level`, `Display`, `SpecialAction` and `Subject` are all *abstract base classes* with *concrete subclasses* that inherit from them. Each block (ie, `IBlock`) extends the common `Block` interface, and each level, from 0 to 4, implements the logic required by the abstract `Level` interface.

The Player class composes a `Grid`, a `Level` object, and several block pointers, but does not inherit from any of them. `GameEngine` inherits from `Subject`, making its role as the *central* component evident. This reflects the View component; `Display` is only called when the game state has changed.

We have introduced a new utility for constructing and reconstructing `Level` objects, a `LevelFactory` which implements the factory pattern. Although this wasn't a part of our initial UML, it became evident as we were implementing the solution that this would be an essential component of our final design.

After all the refactors were made to our original plan, we were able to create a clean organizational structure with a clear flow of information between the components. This system was implemented through many of the C++ idioms we have learned throughout the course, such as abstract interfaces, smart pointers and observer and factory patterns.

Design

We centred our implementation on one key idea: program to interfaces, not implementations.

We were also able to maintain high cohesion by ensuring that each class had narrowly focused responsibilities. These responsibilities are detailed below, accompanied by a design rationale.

Block and the Block Hierarchy

The Block class is an abstract representation of a Tetris game piece. It defines the core geometric behaviours that are common to all blocks and maintains relative cell offsets, applies rotations, and communicates both relative and absolute positions.

Block's responsibilities include:

→ Storing a block's shape as a set of relative coordinates

→ Supporting rotation logic (both CW and CCW)
→ Reporting absolute grid positions through getAbsoluteCells() given an origin
→ Tracking the block's current orientation and origin location

Blocks are owned exclusively by the Player, and their lifetimes correspond exactly to gameplay states (current, next, held). Using std::unique_ptr<Block> enforces single ownership and ensures automatic cleanup when blocks are replaced (ie, during Force or after placement).

This prevents:
→ Memory leaks
→ Dangling pointers after block replacement
→ Accidental shared ownership or aliasing

Originally, we planned for movement logic to be partially inside Grid. The final design corrected this: Grid performs only validation and placement, while Player performs all movement. This eliminates circular dependencies and reduces coupling substantially.

Separating the geometry in Block from the placement validation in Grid was crucial to our implementation; blocks only know what they look like, and not where they go. This allows new block types to be added easily by simply creating more subclasses with a new set of relative coordinates.

Grid and Cell

Grid encapsulates the playable area of each Player's board and is responsible for maintaining occupancy, handling row clearing and managing cell-level state.

The Cell class represents a single unit in the grid; each cell stores its character representation and provides basic setters/getters that the grid can manipulate.

Grid's responsibilities include:
→ Storing the state of all board cells
→ Validating potential block placements with isValidPosition()
→ Permanently placing blocks once dropped
→ Clearing full rows and updating score-related data
→ Applying temporary effects such as Blind

We intentionally designed Grid to be highly cohesive: it only deals with the board. Movement and user commands all occur at a higher level inside of Player, which allows Grid to be queried for validity. This allowed our grid implementation to be simple and independent of game progress logic.

Level and the Level Hierarchy

The Level abstraction isolates block generation logic and level-specific rules. It has the following responsibilities:

→ Generating the next block type according to level rules
→ Tracking state related to sequence files (for Level 0 and sequence-based modes)
→ Handing level-specific mechanics (ie, Level 4 star blocks, Level ¾ heavy effect)

Each level implements its own strategy for producing block types, managing sequence-file input, and applying difficulty mechanics such as Level 4's star blocks or the heavy effect in Levels 3 and 4. This design applies the Factory Method pattern: LevelFactory constructs the appropriate level object, allowing Player to depend only on the abstract Level interface. As a result, difficulty logic remains completely isolated from gameplay mechanics, and introducing new levels or modifying probability distributions requires no changes outside the level hierarchy. Compared to our DD1 plan, which instantiated concrete levels directly in Player, the final design provides cleaner separation, lower coupling, and greater extensibility.

Player

The Player class is the central coordinator of gameplay for each player. It integrates block generation, movement logic, grid interactions, scoring and special effects.

Player has the following responsibilities:

→ Managing the current block, the next block and the held block
→ Delegating block creation to the Level
→ Moving and rotating blocks using grid validation
→ Scoring, row clearing and applying bonuses
→ Handling special actions
→ Enforcing rules and the behaviour of Force

Placing the core gameplay logic in Player rather than Grid or GameEngine allowed us to clearly separate the local game state from the global turn logic. Each player manages their own grid and blocks while the engine sequences these interactions. This supports extendability by keeping heavy logic within a single class.

GameEngine

GameEngine orchestrates the entire game: it coordinates players, enforces game structure, receives commands and notifies observers.

GameEngine's responsibilities include:

→ Managing turn order and game progression
→ Mapping commands to player controls
→ Applying special actions across players

→ Acting as the subject in the observer pattern for displays
→ Exposes immutable views of board state for displays

Having the engine inherit from Subject centralized all display updates. Any state change within the game triggers a call to notifyObservers(), allowing displays to redraw themselves without engine–display coupling. This also improved testability, as game logic no longer makes assumptions about its output environment.

Display, TextDisplay and GraphicsDisplay

Displays implement the visual interface and observe the game engine for updates. Its responsibilities include:
→ Pulling state from the engine and rendering the current board, scores and data
→ Providing both ASCII and graphical interfaces

The class uses the Observer pattern to decouple rendering from game logic. GameEngine acts as the Subject, and both TextDisplay and GraphicsDisplay register as observers. After every state-changing operation, the engine issues a notifyObservers() call, prompting each display to pull the updated state through public, read-only accessors. This design keeps displays entirely independent from the model and controller, prevents any rendering logic from leaking into gameplay code, and makes the system easily extensible; new display types can be introduced without modifying existing logic.

SpecialAction, Blind and Heavy

The SpecialAction abstraction was refined significantly compared to our DD1 design. Blind and Heavy remain as clean, self-contained effects applied to a player, while Force was moved out of the hierarchy to avoid excessive coupling.

SpecialAction's responsibilities include:
→ Applying a temporary/immediate effect to a player
→ Encapsulating the behaviour associated with each effect

By ensuring that special actions operate exclusively through Player's public interface, the design avoids mixing special-effect logic with unrelated modules. The decision to remove Force from this hierarchy was an important simplification that significantly improved modularity!

CommandInterpreter

The command interpreter isolates all parsing logic from the game's core. Its responsibilities include:
→ Normalizing user input into known commands
→ Handling multipliers (ie, 3Left)
→ Resolving command prefixes and abbreviations

→ Expanding command sequences loaded from files

This separation ensures that the game engine only deals with valid commands, keeping gameplay code concise and error-free. It also makes command semantics easy to update without modifying the engine itself.

Resilience to Change

A major design goal for us was to ensure that the final implementation would be adaptable to future extensions with minimal recompilation. Several aspects of the design directly support this.

Introducing a new block type only requires defining a new subclass of Block and adding a mapping entry in Player::createBlockFromType(). Because block geometry is encapsulated and grid interactions depend solely on the abstract interface, no further modifications are necessary.

Adding a new level is similarly straightforward. A new subclass implementing the Level interface can be integrated into the game with minimal changes, restricted to the level factory. No gameplay logic needs adjustment because the engine and players do not depend on specific level types.

Special actions are also easily extendable. Because each effect modifies independent aspects of gameplay and the hierarchy is limited to actions that do not influence block creation, additional effects can be added without affecting existing behaviour.

Finally, the CommandInterpreter allows significant flexibility in handling user input. New commands, renaming of commands, or introduction of command macros can be handled within the interpreter without requiring modifications to the engine. This separation of concerns also improves testability and reduces opportunities for input-related bugs.

Overall, the final design provides strong resilience to change. Responsibilities are well-isolated, interfaces are stable, and extensions can occur with minimal code modification.

Updated Answers to Project Questions

The final implementation supports the extension scenarios described in our DD1 plan with even clearer modular boundaries. Although we did not implement timed blocks, additional levels, or new effects, the architecture remains fully compatible with those designs. In each case, the separation between interface-based components, the use of factories, and the isolation of parsing logic reinforce the extensibility originally envisioned.

Q1. 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?

Our DD1 proposal suggested extending the Block class with metadata such as timed, lifespan, and spawnTurn, and using the grid's turn counter to detect expiration. This approach remains feasible in the final design. Although we did not implement timed blocks, the system's separation of responsibilities would support them cleanly: Level would generate timed blocks in greater difficulties, while Grid (already responsible for row clearing and occupancy) could check for block expiration after each placement. Because geometry, grid state, and gameplay rules are modularized in the final architecture, timed blocks could be added with minimal disruption to the rest of the system.

Q2. How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

Our final design fully realizes the approach outlined in DD1. The game interacts with levels exclusively through the abstract Level interface, and all construction is centralized in LevelFactory. As a result, introducing a new level requires implementing a new subclass (e.g., Level5) and registering it in the factory. No existing gameplay code, display code, or engine logic needs to change. This improves upon the DD1 design by eliminating direct instantiation of levels from Player and by preventing level-specific logic from leaking into other modules.

Q3. How could you design your program to allow for multiple effects to be applied simultaneously? New effects? Avoid large if-else?

In DD1, we proposed representing simultaneous effects using separate flags or a collection of active SpecialAction objects, with each effect modifying the player via apply(). The final design remains consistent with this vision. Blind and Heavy operate independently by updating distinct fields within Player, and Player's movement and rendering operations refer to these fields when applying effects. Force was moved out of the SpecialAction hierarchy to avoid excessive coupling, but its relocation simplifies rather than hinders extensibility. Adding a new effect would involve implementing a new SpecialAction subclass and invoking it through existing entry points, without requiring large conditional structures or modifications to the game engine.

Q4. How easily can new commands, renamed commands, or command macros be supported?

As described in DD1, the CommandInterpreter encapsulates all parsing, abbreviation resolution, and command normalization. This remains true in the final design. Renaming existing commands or adding new ones involves updating the interpreter's command table, and macros can be supported by mapping macro names to sequences of standard engine commands that the interpreter expands before dispatch. Because the engine receives only canonical commands, extending command semantics does not affect gameplay logic. The final system matches and strengthens the flexibility envisioned in DD1 by keeping shortcut logic, name resolution, and macro expansion strictly within the interpreter.

Extra Credit Features

Although we did not implement additional gameplay modifications beyond the project requirements, we did produce fully functional graphical and textual displays, a robust observer mechanism, a flexible command parser, and a refined special-action system. These improvements enhance the design quality and provide a strong foundation for future project extensions.

Final Questions

Completing this project provided several insights into collaborative software development. One key lesson was the importance of establishing clear interfaces early. By doing so, we were able to divide work effectively and minimize integration issues. We also learned the value of refactoring. Several design ideas from DD1 proved insufficient when implemented, and allowing ourselves to revisit and improve them led to a cleaner and more maintainable codebase.

Another important lesson was the need for clear communication. Subtle aspects of gameplay mechanics, such as the difference between level-induced heavy behaviour and special-action heavy behaviour, required careful clarification to ensure consistent implementation. Maintaining alignment in our understanding of rules and design goals helped prevent divergence in components that needed to work together closely.

If we were to restart the project, we would incorporate testing earlier, especially for grid and player logic, which contain the majority of gameplay complexity. We would also formalize and document interface boundaries earlier in the process and keep the UML diagram updated continuously rather than revising it at the end. These practices would streamline development and reduce uncertainty during implementation.

Conclusion

The final implementation of Biquadris meets the functional requirements of the specification while demonstrating strong adherence to object-oriented design principles.

Through consistent use of interfaces, well-defined class boundaries, and thoughtful application of patterns such as factories and observers, the program achieves modularity, clarity, and extensibility. Although the design evolved beyond the initial DD1 plan, these changes resulted in a cleaner and more robust architecture.

This project deepened our understanding of software design and provided valuable experience in translating abstract principles into practical, maintainable solutions.