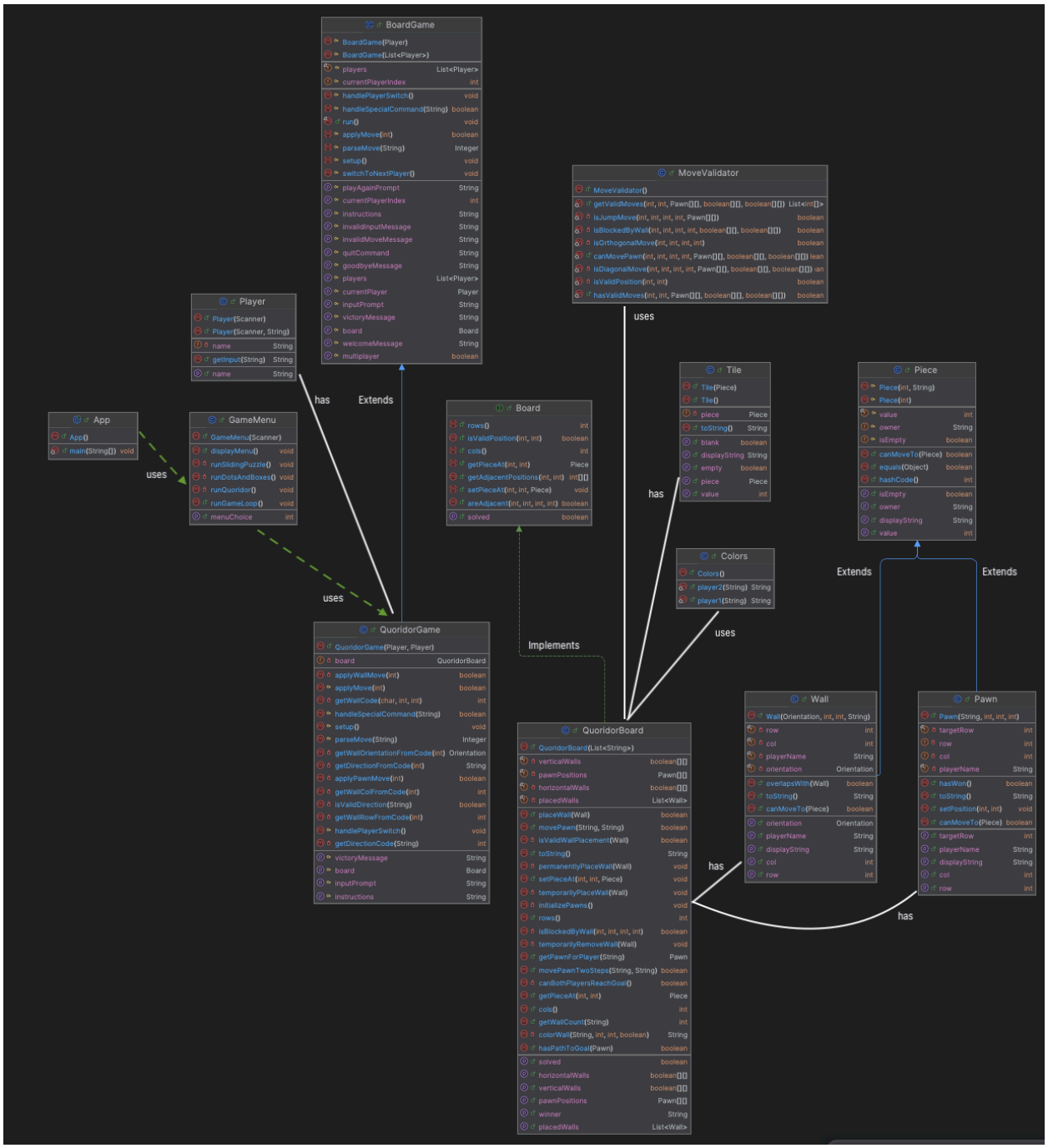
Saksham Goel & Edaad Azman
Github: https://github.com/Saksham106/quoridor

UML



Scalability

Configurable Board Sizes: The BOARD_SIZE constant in QuoridorBoard allows easy modification to different grid dimensions. Similar constants exist in other game boards, demonstrating pattern consistency.

Wall Array Flexibility: horizontalWalls and verticalWalls arrays are dimensioned based on BOARD_SIZE, making the system work for any grid size without algorithmic changes.

Move Validation Modularity: MoveValidator operates independently of board size, using helper functions like isValidPosition() that dynamically check boundaries.

Extendibility
Adding New Variants: Adding a new game requires minimal code: create game-specific piece classes (extending Piece), implement Board interface, extend BoardGame, and add menu option in GameMenu. We wouldn't need any core framework modifications.

Example - Adding a New Variant: To add a 4x4 mini-Quoridor variant:

1. Create MiniQuoridorBoard implementing Board with BOARD_SIZE = 4
2. Reuse Pawn, Wall, and MoveValidator (no changes needed)
3. Create MiniQuoridorGame extending BoardGame
4. Add menu option in GameMenu

The pathfinding, move validation, and piece logic are completely reusable.

Modification to Support Another Turn-Based Variant
Modification: Separation of Turn Management: The original BoardGame assumed each player alternates with a single action per turn. Quoridor requires two types of turns (pawn move or wall placement per turn), which necessitated changes:

- Added nextMoveType field in QuoridorGame to track whether current turn is for moving or wall placement
- Added handleSpecialCommand() hook in BoardGame for game-specific commands like "switch" between move/wall modes
- Modified applyMove() to route to either applyPawnMove() or applyWallMove() based on nextMoveType
- Added movePawnTwoSteps() method to support automatic jump detection when single-step moves are blocked

This architectural change allows any future turn-based game (chess, checkers, etc.) to override applyMove() and handle turn-specific logic without modifying the core framework.

Deciding on Design Infrastructures

Although the main classes and high level design of the Quoridor implementation was easy to come up with due to our scalable and extendable design from the prior assignments, we initially struggled with figuring out how to have good design for our validation logic.

These are the designs we had initially considered:

1. Integrated Validation
    a. Validation logic is inside QuoridorBoard
    b. There is very tight coupling and hard to test without initiating the Board.
    c. Also lower reusability
2. Inheritance based validations
    a. Abstract MoveValidator base with subclasses
    b. There is a unnecessary hierarchy for utility functions only
    c. Over engineered
3. Static Utility (This is what we chose)
    a. MoveValidator as a static utility
    b. Independent and easily testable
    c. Also reusable across games if other logic needs to be added
    d. Clear separation of concerns

Workload Split
Saksham Goel:
Saksham did the core game architecture and logic design. He implemented the Piece hierarchy (Pawn, Wall) and resolved major structural issues involving tile and piece interactions early in development. He created the standalone MoveValidator module that manages all movement logic, including orthogonal movement, jumping, and wall collision detection. Saksham also developed the main QuoridorGame class, handling turn management, input parsing, and game state transitions. His later work focused on refining gameplay like improving jump mechanics, fixing indexing bugs, and ensuring walls correctly block two tiles as in the official rules.

Edaad Azman:
Edaad did the board representation, visual interface, and gameplay validation features. He built the QuoridorBoard class with a 9×9 grid structure and implemented BFS-based pathfinding to verify that wall placements never block all possible player paths. He also enhanced the terminal interface, adding color-coded elements, replay support, and improved wall visualization for better readability. Beyond visual polish, Edaad fixed several critical issues related to wall overlap and crossing logic, ensuring stable and consistent board behavior across different scenarios. His contributions made the game experience smoother and more user-friendly.

Collaboration Summary:
Saksham focused on the internal game mechanics and logic, while Edaad concentrated on the board systems and player interface. This division of work allowed both of us to work in parallel with minimal conflicts.