CSE 211 - Data Structures

Term Project

c++ 20

Table of Contents

Core Information

Game Elements

Data Structures

Implementation

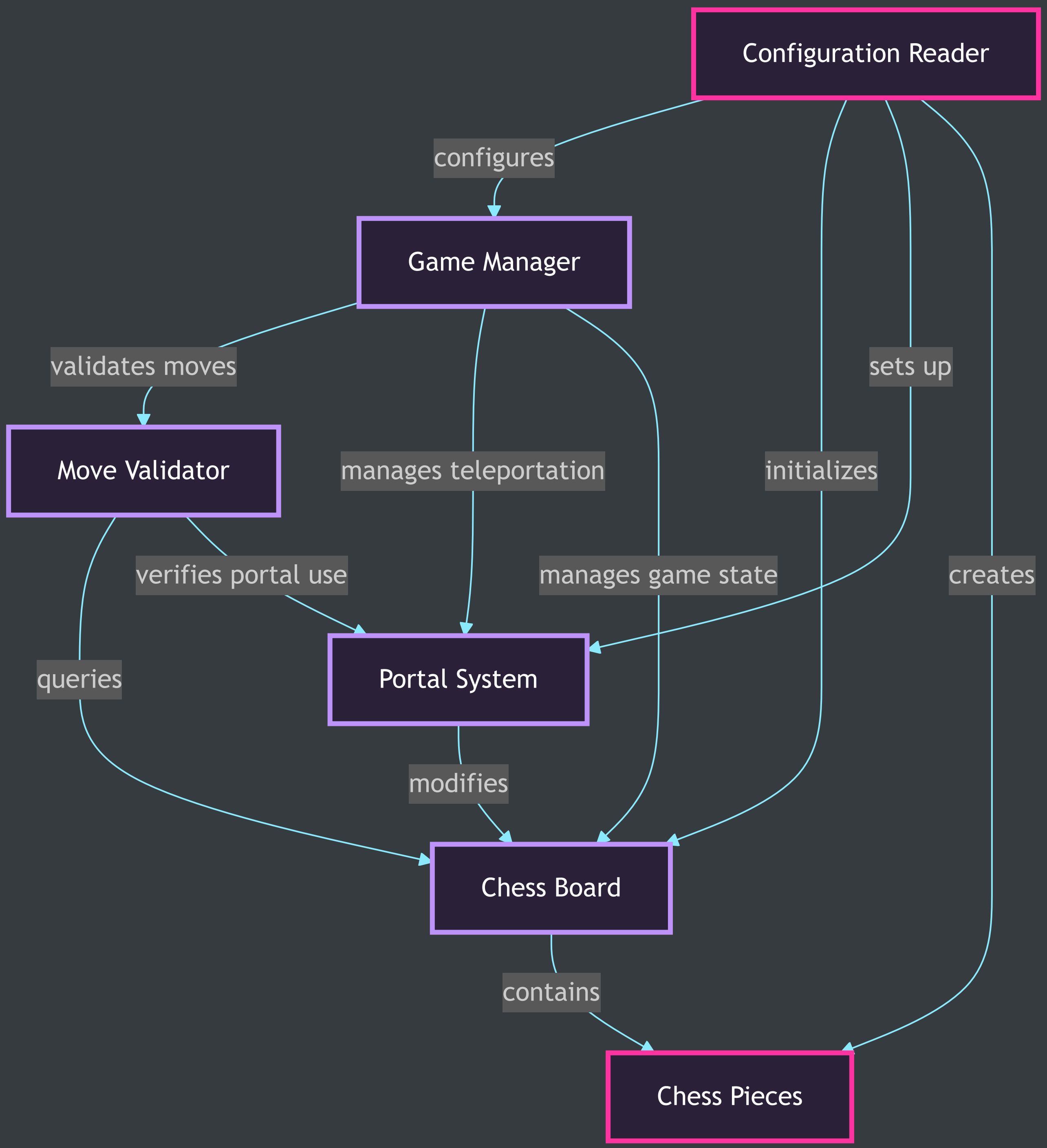
Project Overview

Description

This project implements a custom chess game with configurable pieces and portals. The game extends traditional chess with fantasy elements like teleportation portals and custom movement patterns.

High-Level Architecture

This project uses several data structures working together to create a flexible chess system:



> **Color Scheme**: Purple borders (Game Manager, Move Validator, Portal System, Chess Board) represent core game components that interact directly with game state. Pink borders (Configuration Reader, Chess Pieces) represent supporting components that are configured or contained by core components. Cyan connection lines show the flow of data and control between components.

Key Features

- **Standard Chess Rules**: All traditional chess pieces implemented with standard movements
- **Portal System**: Pieces can teleport across the board using portals
- **Configurable Game**: Game settings, pieces, and portals are configurable via JSON
- **Advanced Data Structures**:
  - **Graph-based Move Validation**: Uses a graph representation for validating moves
  - **HashMap for Board Representation**: Fast O(1) lookups for pieces on the board
  - **Queue for Portal Cooldowns**: Efficient management of portal cooldown times
  - **Stack for Move History**: Complete move history with undo functionality

---

Class Architecture

Core Classes and Interactions

1. GameManager

Responsibilities:

- Game state management
- Turn processing
- Win/loss condition checking
- Player management

Implementation Considerations:

- Central controller for game flow

- Maintains references to other components
- Processes player commands

2. ChessBoard

Responsibilities:

- Board state representation using hashmap
- Piece placement and movement
- Capture handling

Implementation Considerations:

- Choose efficient data structure for position-to-piece mapping
- Consider immutable vs mutable design for state changes
- Implement bounds checking and position validation

3. MoveValidator

Responsibilities:

- Move validation using graph representation
- Path checking
- Portal usage validation

Implementation Considerations:

- Build an efficient graph structure to represent possible moves
- Implement search algorithms (BFS/DFS) for path validation
- Consider caching valid moves for performance

4. PortalSystem

Responsibilities:

- Teleportation logic
- Cooldown management using queue

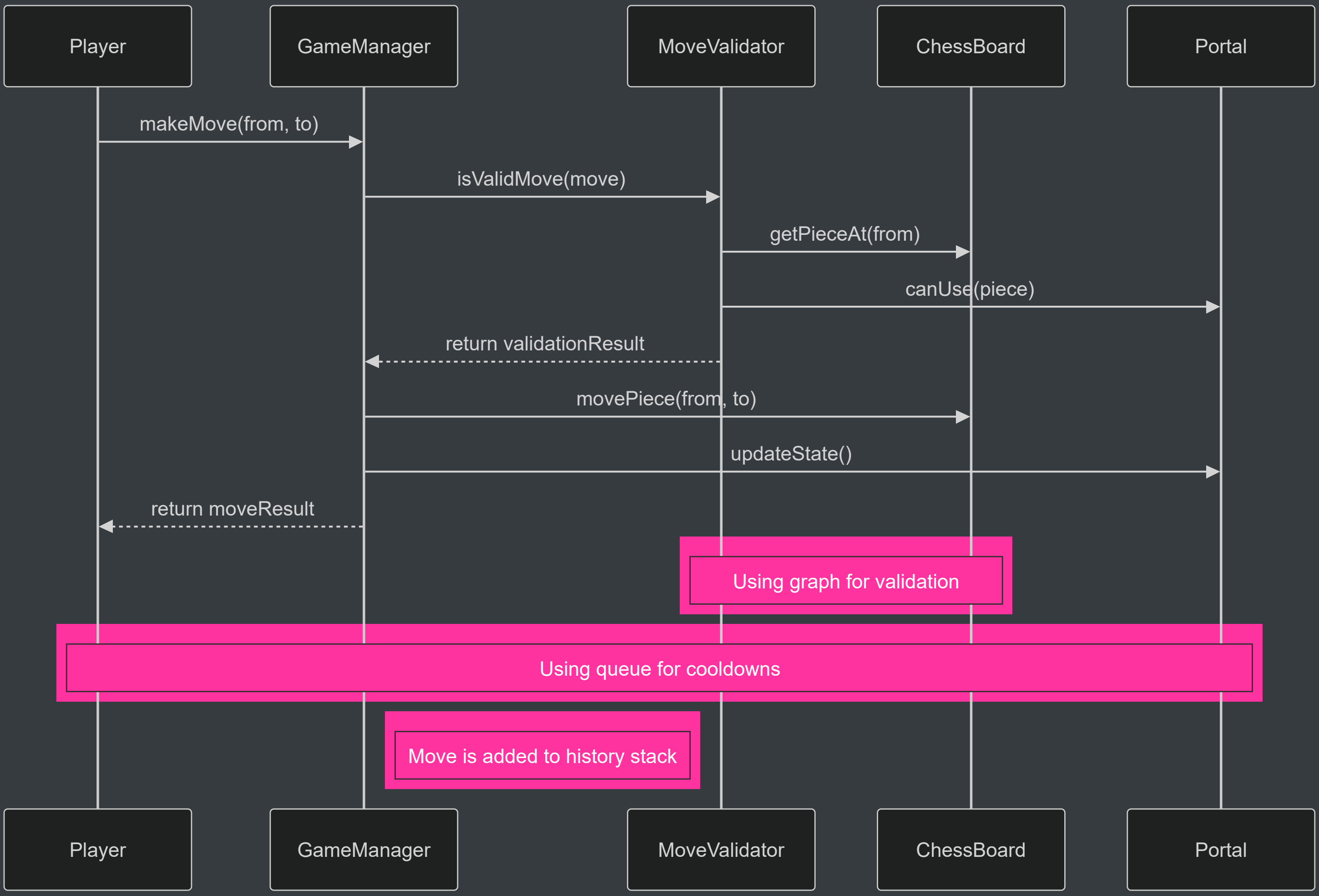
- Direction preservation

Implementation Considerations:

- Design for extensibility to allow different portal types
- Implement efficient cooldown tracking
- Consider observer pattern for portal state changes

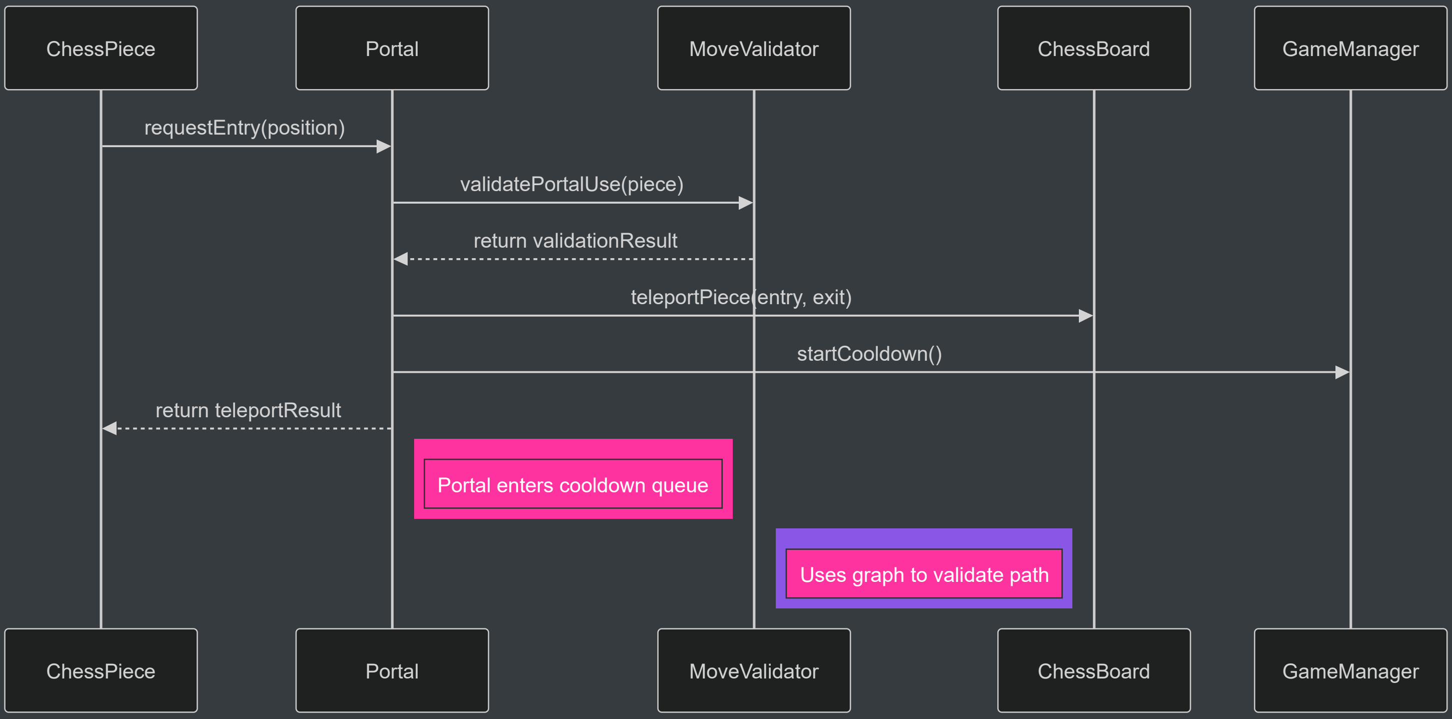
Class Interaction Flow

1. **Move Processing:**



> **Note**: Pink highlights indicate where specific data structures are being utilized in the process.

2. **Portal Usage:**



> **Note**: Pink highlight shows queue data structure usage, while purple highlight shows graph data structure usage.

Game Mechanics

Movement Rules

1. **Standard Pieces**
   - Follow traditional chess rules
   - Additional portal interaction capabilities
2. **Portal Mechanics**
   - Entry and exit points
   - Direction preservation options
   - Color-specific restrictions
   - Cooldown periods

Game Commands

Once running, the game accepts the following commands:

- `move <from> <to>` : Move a piece from one position to another (e.g., `move e2 e4` or `move 4,1 4,3`)
- `undo` : Undo the last move
- `quit` or `exit` : End the game

Configuration System

File Structure

```
{
  "game_settings": {
    "name": "Custom Chess",
    "board_size": 8,
    "turn_limit": 100
  },
  "pieces": [
    {
      "type": "King",
      "positions": {
        "white": [{ "x": 4, "y": 0 }],
        "black": [{ "x": 4, "y": 7 }]
      },
      "movement": {
        "forward": 1,
        "sideways": 1,
        "diagonal": 1
      },
      "special_abilities": {
        "castling": true,
        "royal": true
      }
    }
```

```json
  ],
  "custom_pieces": [
    {
      "type": "Teleporter",
      "positions": {
        "white": [{ "x": 1, "y": 2 }],
        "black": [{ "x": 6, "y": 5 }]
      },
      "movement": {
        "forward": 2,
        "sideways": 2
      },
      "special_abilities": {
        "portal_master": true
      }
    }
  ],
  "portals": [
    {
      "type": "Portal",
      "id": "portal1",
      "positions": {
        "entry": { "x": 2, "y": 3 },
        "exit": { "x": 5, "y": 4 }
      },
      "properties": {
        "preserve_direction": true,
        "allowed_colors": ["white", "black"],
        "cooldown": 1
      }
    }
  ]
}
```

Validation Rules

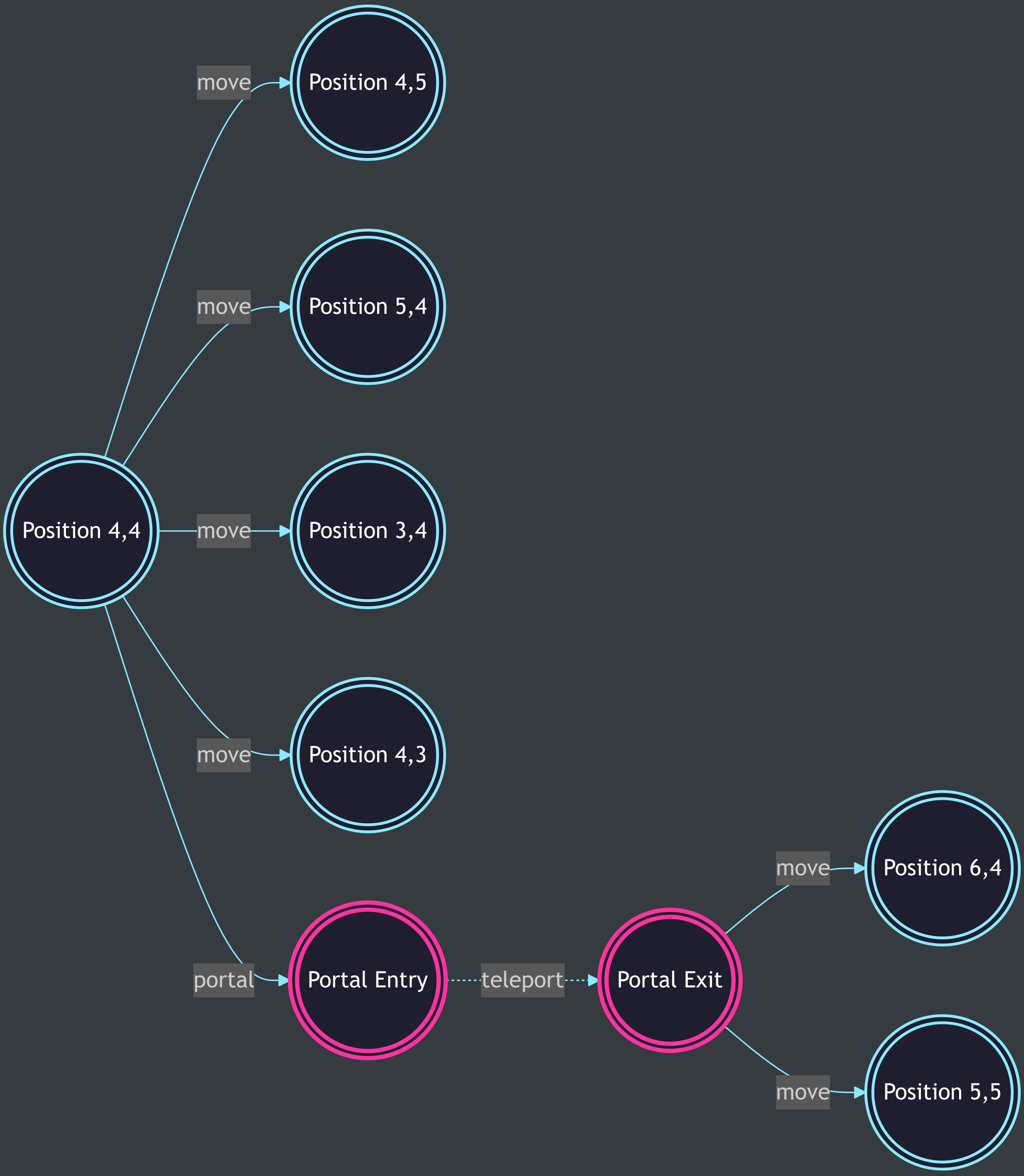
1. Board boundaries
2. Piece counts
3. Portal placement
4. Movement patterns

Key Data Structures

Graph for Move Validation

The move validator builds a graph representation of the board where:

- Nodes are positions on the board
- Edges represent possible moves between positions
- Special edges represent portal traversal
- BFS is used to find valid move paths, especially with portals



> **Color Scheme**: Position nodes (cyan borders) represent standard board positions. Portal nodes (pink borders) represent special teleportation points. Solid lines show normal moves, while dotted lines show teleportation.

HashMap for Board Representation

The chessboard uses a hashmap for O(1) lookups:

- **Purpose**: Efficiently store and retrieve chess pieces by their position
- **Key Components**:
  - Keys: String representation of position ("x,y")
  - Values: Chess piece objects with their type and color
- **Operations**:
  - Get a piece at a position - O(1)
  - Place a piece at a position - O(1)
  - Remove a piece from a position - O(1)
- **Benefits**:
  - Constant-time access regardless of board size
  - Easily handle sparse boards (not all positions filled)
  - Quick position-based lookups for move validation

Queue for Portal Cooldowns

Portal cooldowns are managed using a queue:

- **Purpose**: Track and update portals that are in cooldown
- **Key Components**:
  - Elements: Portal references with their cooldown times
  - Front: Oldest portal in cooldown
  - Rear: Most recently used portal
- **Operations**:
  - Enqueue portal when used (enters cooldown)
  - Dequeue portal when processing cooldowns
  - Requeue portal if cooldown remains
- **Process Flow**:
  1. When a portal is used, it enters the cooldown queue

2. At the end of each turn, cooldowns are decremented
3. Portals exit the queue when their cooldown reaches zero
4. Ensures portals cannot be used repeatedly without waiting

Stack for Move History

A stack maintains the complete move history:

- **Purpose**: Track all moves for undo functionality and game state traversal
- **Key Components**:
  - Elements: Move records containing source, destination, captured pieces, used portals
  - Top: Most recent move
- **Operations**:
  - Push move record after successful move
  - Pop move record when undoing
  - Peek to view last move without removal
- **Benefits**:
  - Perfect for undo/redo functionality (LIFO)
  - Enables move verification and game replay
  - Supports special rules like en passant that depend on previous moves

---

Creating Custom Pieces

To create custom chess pieces, add entries to the `custom_pieces` array in your configuration file:

```
  "custom_pieces": [
    {
      "type": "Archer",
      "positions": {
        "white": [{ "x": 2, "y": 1 }],
        "black": [{ "x": 5, "y": 6 }]
      },
      "movement": {
        "forward": 1,
        "sideways": 1
      },
      "special_abilities": {
        "ranged_attack": true,
        "attack_range": 3
      }
    }
  ]
```

Data Structure Design Considerations

When implementing custom pieces, consider these data structure design aspects:

1. **Inheritance vs Composition**: Decide whether to use inheritance hierarchy or composition for piece behaviors
2. **Movement Pattern Representation**: Choose between lookup tables, rule functions, or graph-based movement patterns
3. **Special Ability Implementation**: Design a flexible system that can be extended with new abilities
4. **Performance Optimization**: Balance memory usage with computational efficiency for move validation

Supported Movement Properties

Property	Description	Example Usage	Visual Representation
`forward`	Number of squares forward	Rook: 8, Pawn: 1	↑ vertical movement
`sideways`	Number of squares sideways	Rook: 8, King: 1	← → horizontal movement
`diagonal`	Number of squares diagonally	Bishop: 8, Queen: 8	↖️ ↗️ ↘️ ↙️ diagonal movement
`l_shape`	Knight-like L-shaped movement	Knight: true	⎰ ⎱ L-shaped pattern
`diagonal_capture`	Diagonal capture distance	Pawn: 1	↖️ ↗️ only for captures

Property	Description	Example Usage	Visual Representation
`first_move_forward`	Extra forward distance on first move	Pawn: 2	↑↑ extra range initially

Supported Special Abilities

Ability	Description	Effect on Gameplay	Data Structure Impact
`castling`	Allows king-rook castling	Special two-piece move	Requires history tracking
`royal`	Piece is royal (losing it loses the game)	Defines win condition	Requires game state monitoring
`jump_over`	Can jump over other pieces	Ignores pieces in path	Modifies graph pathfinding
`promotion`	Can be promoted when reaching last rank	Transforms into another piece	Requires piece transformation logic
`en_passant`	Can be captured en passant	Special capture condition	Requires move history in stack

---

Getting Started

Prerequisites

- C++20 compatible compiler (gcc, clang, MSVC)
- Make build system

The project will automatically download and set up the required third-party dependencies (nlohmann/json).

Installation

Linux/macOS/Windows

```
# Clone the repository
git clone https://github.com/yourusername/chess-project.git
cd chess-project

# Build the project
make
```

Building and Running

Build Commands

```
# Build the project
make

# Normally it automatically downloads dependencies however:
make deps
```

Running

Run the game with the standard chess configuration:

```
make run
```

To use a custom configuration:

```
./bin/chess_game path/to/config.json
```

Troubleshooting

Common Issues

1. **Compilation errors with dependencies**
   - Run `make deps` to reinstall dependencies
   - Ensure you have C++20 support in your compiler
2. **Game crashes when loading configuration**
   - Check JSON syntax
   - Validate piece positions are within board boundaries
3. **Portal doesn't work**
   - Check cooldown settings
   - Verify allowed_colors include your piece's color

Git Workflow Strategies

To effectively collaborate on this project, use the following Git strategies:

Branch Management

1. **Main Branch**: Always keep the `main` branch stable and deployable

   ```bash
   # Only merge tested, reviewed code to main
   git checkout main
   git merge --no-ff feature/my-feature
   ```

2. **Feature Branches**: Create a new branch for each feature

   ```bash
   # Create and switch to a new feature branch
   git checkout -b feature/movement-validation
   ```

3. **Naming Conventions Examples**:
   - `feature/` - For new features
   - `bugfix/` - For bug fixes

    - `refactor/` - For code refactoring
    - `test/` - For adding tests

Working with Files

1. **Staging Specific Files**: Add only the files you want to commit

```
# Add specific files
git add src/MoveValidator.cpp include/MoveValidator.hpp

# Add specific files interactively (choose chunks)
git add -p src/ChessBoard.cpp
```

2. **Checking Status**: Always verify what's being committed

```
# View staged and unstaged changes
git status

# View detailed changes in files
git diff

# View changes staged for commit
git diff --staged
```

3. **Committing Files**: Create focused, atomic commits

```
# Commit staged files with message
git commit -m "[MoveValidator] Implement knight movement pattern"

# Amend the most recent commit (if you forgot something)
git add forgotten-file.cpp
git commit --amend
```

Commit Message Structure

```
[Component] Short summary (50 chars or less)

More detailed explanation if necessary. Keep line length to
about 72 characters. Explain what and why, not how.

- List specific changes if helpful
- Another change detail

Refs #123, #456
```

Example:

```
[MoveValidator] Add BFS algorithm for path finding

Implemented breadth-first search to validate move paths between
positions, accounting for obstacles and board boundaries.

- Used queue data structure for BFS implementation
- Added cycle detection to prevent infinite loops
- Optimized graph traversal for portal connections

Refs #42
```

Managing Changes

1. **Stashing Changes**: Save work-in-progress temporarily

   ```
   # Stash current changes
   git stash save "WIP: Implementing portal cooldown"

   # List stashes
   git stash list

   # Apply most recent stash
   git stash apply
   ```

```
# Apply specific stash
git stash apply stash@{2}

# Apply and remove stash
git stash pop
```

2. **Viewing History**: Understand how the project has evolved

```
# View commit history
git log

# View compact history
git log --oneline

# View history with branch graph
git log --graph --oneline --all

# View history for specific file
git log --follow -- src/ChessBoard.cpp
```

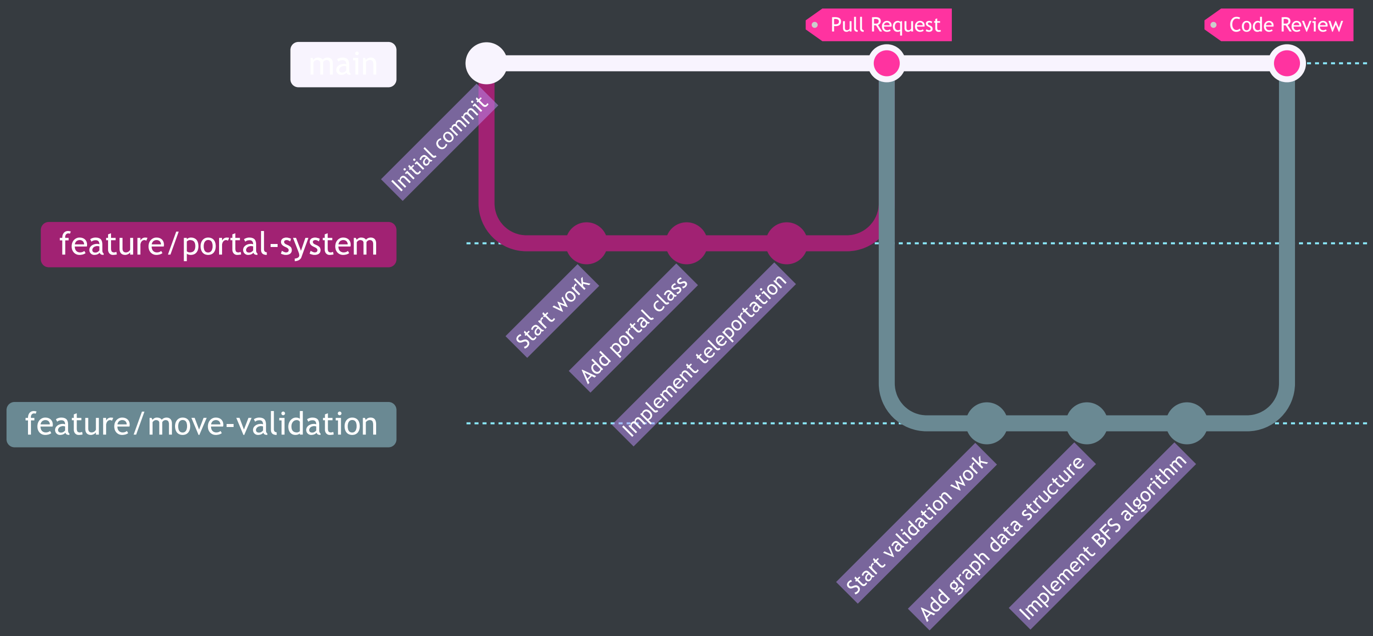
3. **Rewriting History**: Clean up before sharing your work

```
# Interactive rebase to clean up commits
git rebase -i HEAD~5  # Last 5 commits

# Fix up a previous commit (squash without editing message)
# In interactive rebase, change "pick" to "fixup" or "f"

# Reword a commit message
# In interactive rebase, change "pick" to "reword" or "r"
```

Collaboration Workflow



> **Workflow Key Points**:
>
> - Main branch stays stable at all times
> - Feature branches for isolated development
> - Pull requests for code review before merging
> - Regular commits with descriptive messages

Practical Workflows

1. **Starting a New Feature**

```
git checkout main
git pull
git checkout -b feature/portal-system
# Now make your changes
```

2. **Daily Development Routine**

```
# Start the day by getting latest changes
git checkout main
git pull
git checkout feature/your-feature
git rebase main

# Work on your feature
# Commit regularly

# End of day, push your progress
git push -u origin feature/your-feature
```

3. **Preparing for Code Review**

```
# Make sure your branch is up to date
git checkout main
git pull
git checkout feature/your-feature
git rebase main

# Clean up your commit history
git rebase -i HEAD~<number-of-commits>

# Run tests
make test

# Push your changes
git push -f origin feature/your-feature

# Create pull request through GitHub/GitLab UI
```

By following these Git workflow strategies, you'll maintain code quality, minimize conflicts, and create a clear history of project development.