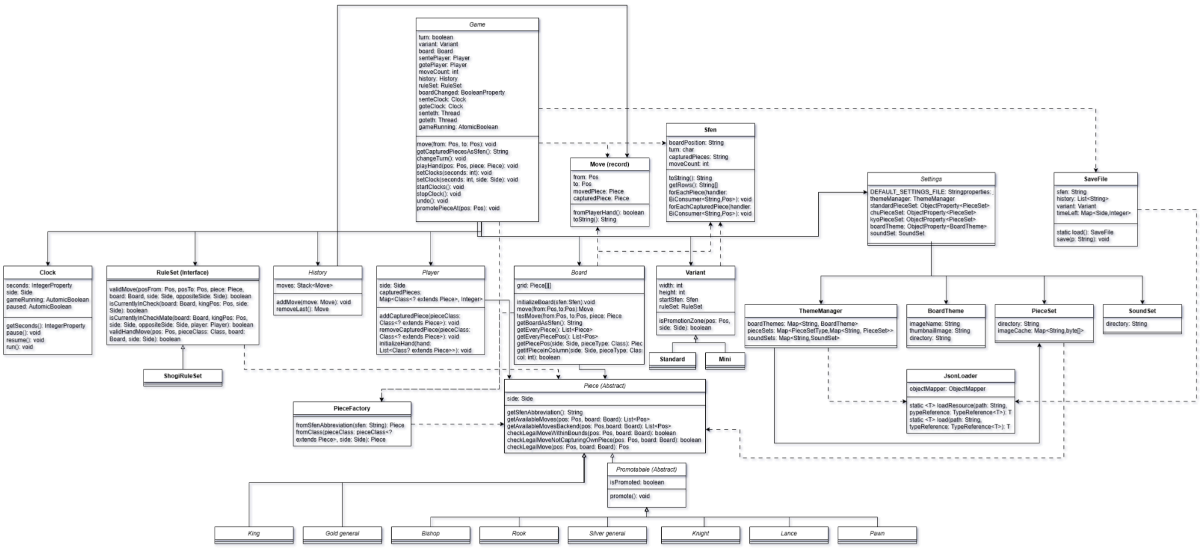
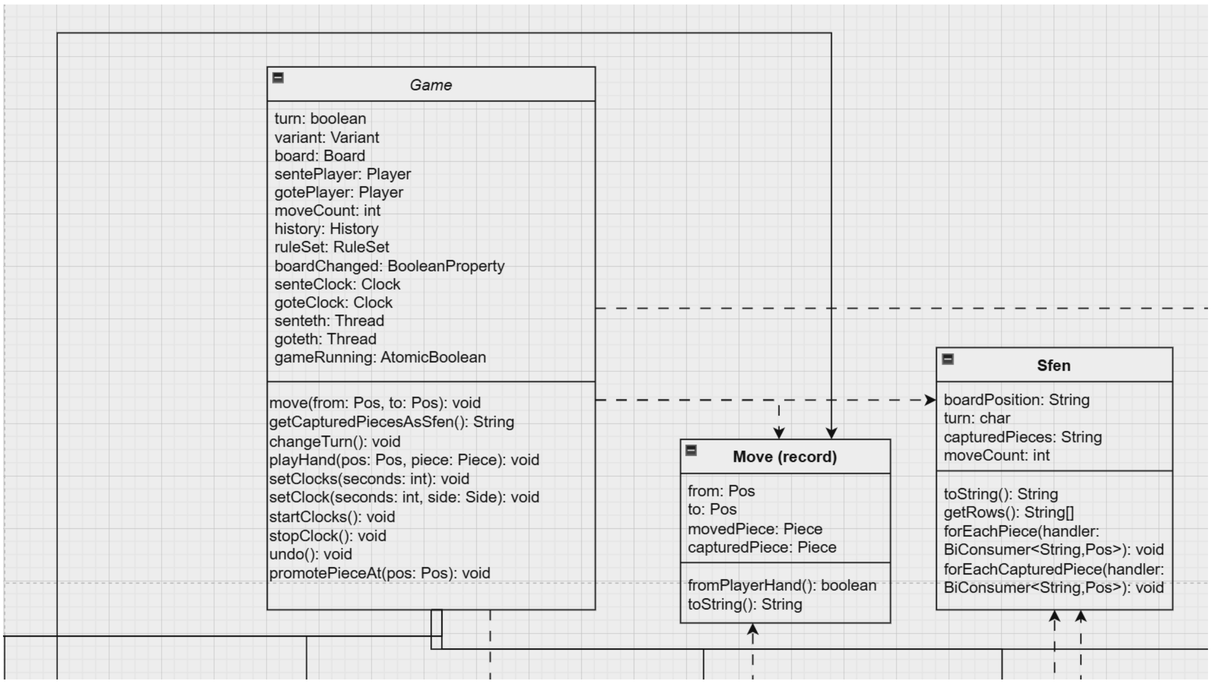
Our project follows the MVC pattern and SOLID principles. These design choices aim to make the system software more maintainable, scalable, and easier to understand. This allows other users to extend our algorithm and add additional content.

Model


Game
turn: boolean
variant: Variant
board: Board
sentePlayer: Player
gotePlayer: Player
moveCount: int
history: History
ruleSet: RuleSet
boardChanged: BooleanProperty
senteClock: Clock
goteClock: Clock
senteth: Thread
goteth: Thread
gameRunning: AtomicBoolean
move(from: Pos, to: Pos): void
getCapturedPiecesAsSfen(): String
changeTurn(): void
playHand(pos: Pos, piece: Piece): void
setClocks(seconds: int): void
setClock(seconds: int, side: Side): void
startClocks(): void
stopClock(): void
undo(): void
promotePieceAt(pos: Pos): void
Move (record)
from: Pos
to: Pos
movedPiece: Piece
capturedPiece: Piece
fromPlayerHand(): boolean
toString(): String
Sfen
boardPosition: String
turn: char
capturedPieces: String
moveCount: int
toString(): String
getRows(): String[]
forEachPiece(handler: BiConsumer<String,Pos>): void
forEachCapturedPiece(handler: BiConsumer<String,Pos>): void
Settings
DEFAULT_SETTINGS_FILE: Stringproperties:
themeManager: ThemeManager
standardPieceSet: ObjectProperty<PieceSet>
chuPieceSet: ObjectProperty<PieceSet>
kyoPieceSet: ObjectProperty<PieceSet>
boardTheme: ObjectProperty<BoardTheme>
soundSet: SoundSet
SaveFile
sfen: String
history: List<String>
variant: Variant
timeLeft: Map<Side,Integer>
static load(): SaveFile
save(p: String): void
Clock
seconds: IntegerProperty
side: Side
gameRunning: AutomicBoolean
paused: AutomicBoolean
getSeconds(): IntegerProperty
pause(): void
resume(): void
run(): void
RuleSet (Interface)
validMove(posFrom: Pos, posTo: Pos, piece: Piece, board: Board, side: Side, oppositeSide: Side): boolean
isCurrentlyInCheck(board: Board, kingPos: Pos, side: Side): boolean
isCurrentlyInCheckMate(board: Board, kingPos: Pos, side: Side, oppositeSide: Side, player: Player): boolean
validHandMove(pos: Pos, pieceClass: Class, board: Board, side: Side): boolean
ShogiRuleSet
History
moves: Stack<Move>
addMove(move: Move): void
removeLast(): Move
Player
side: Side
capturedPieces: Map<Class<? extends Piece>, Integer>
addCapturedPiece(pieceClass: Class<? extends Piece>): void
removeCapturedPiece(pieceClass: Class<? extends Piece>): void
initializeHand(hand: List<Class? extends Piece>>): void
Board
grid: Piece[][]
initializeBoard(sfen:Sfen):void
move(from:Pos,to:Pos):Move
testMove(from:Pos, to:Pos, piece: Piece
getBoardAsSfen(): String
getEveryPiece(): List<Piece>
getEveryPiecePos(): List<Pos>
getPiecePos(side: Side, pieceType: Class): Piec
getIfPieceInColumn(side: Side, pieceType: Clas
col: int): boolean
Variant
width: int
height: int
startSfen: Sfen
ruleSet: RuleSet
isPromotionZone(pos: Pos, side: Side): boolean
Standard
Mini
ThemeManager
boardThemes: Map<String, BoardTheme>
pieceSets: Map<PieceSetType,Map<String, PieceSet>>
soundSets: Map<String,SoundSet>
BoardTheme
imageName: String
thumbnailImage: String
directory: String
PieceSet
directory: String
imageCache: Map<String,byte[]>
SoundSet
directory: String
JsonLoader
objectMapper: ObjectMapper
static <T> loadResource(path: String, pypeReference: TypeReference<T>): T
static <T> load(path: String, typeReference: TypeReference<T>): T
PieceFactory
fromSfenAbbreviation(sfen: String): Piece
fromClass(pieceClass: pieceClass<? extends Piece>, side: Side): Piece
Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos
Promotabale (Abstract)
isPromoted: boolean
promote(): void
King
Gold general
Bishop
Rook
Silver general
Knight
Lance
Pawn

For the model, we focused on having all data and functionality in this part of the code and making it independent from the other parts. We created multiple classes to represent different parts of the software that handles specific things.

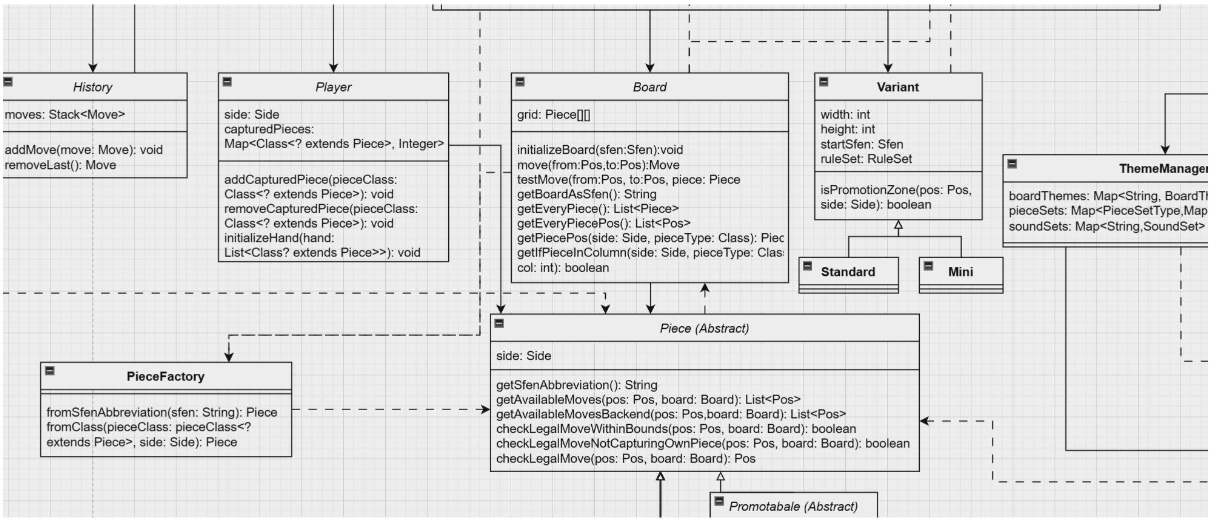
Game class:



The Game class represents a Shogi game, managing its state, players, board, rules, clocks, and move history. It oversees the gameplay by handling moves, turn changes, and win conditions, providing the logic for validating moves according to a rule set defined by a

Variant. The class also supports game persistence, enabling games to be initialized from save files and returning the state in SFEN notation. The Game class depends on other components such as Board for piece placement, Player for player-specific data, Clock for time tracking, History for move recording, and RuleSet for move validation and game logic. Additionally, it relies on utilities like Pos for board positions and Side for player identification. The class uses threads for managing player clocks and integrates a shutdown hook to ensure proper cleanup. It incorporates several key functionalities, such as tracking captured pieces, promoting pieces, undoing moves, and determining valid hand placements. The class leverages the observer design pattern by using a BooleanProperty (boardChanged) to notify listeners when the board state changes. Additionally, it uses a thread-safe mechanism (AtomicBoolean) to control the game's active state, ensuring concurrency safety. This cohesive and modular design encapsulates the core game logic, providing a foundation for higher-level user interface or AI integration, while adhering to object-oriented principles like encapsulation, single responsibility, and separation of concerns. It embodies the façade design pattern by centralizing complex interactions among dependent components, presenting a unified interface to manage and interact with the game. This approach ensures extensibility, making it easy to adapt to different Shogi variants or integrate additional features like multiplayer support or advanced rule configurations.

Board class:


History
moves: Stack<Move>
addMove(move: Move): void
removeLast(): Move
Player
side: Side
capturedPieces:
Map<Class<? extends Piece>, Integer>
addCapturedPiece(pieceClass:
Class<? extends Piece>): void
removeCapturedPiece(pieceClass:
Class<? extends Piece>): void
initializeHand(hand:
List<Class? extends Piece>>): void
Board
grid: Piece[][]
initializeBoard(sfen:Sfen):void
move(from:Pos,to:Pos):Move
testMove(from:Pos, to:Pos, piece: Piece
getBoardAsSfen(): String
getEveryPiece(): List<Piece>
getEveryPiecePos(): List<Pos>
getPiecePos(side: Side, pieceType: Class): Piec
getIfPieceInColumn(side: Side, pieceType: Clas
col: int): boolean
Variant
width: int
height: int
startSfen: Sfen
ruleSet: RuleSet
isPromotionZone(pos: Pos,
side: Side): boolean
Standard
Mini
ThemeManager
boardThemes: Map<String, BoardTh
pieceSets: Map<PieceSetType,Map
soundSets: Map<String,SoundSet>
PieceFactory
fromSfenAbbreviation(sfen: String): Piece
fromClass(pieceClass: pieceClass<?
extends Piece>, side: Side): Piece
Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos
Promotabale (Abstract)

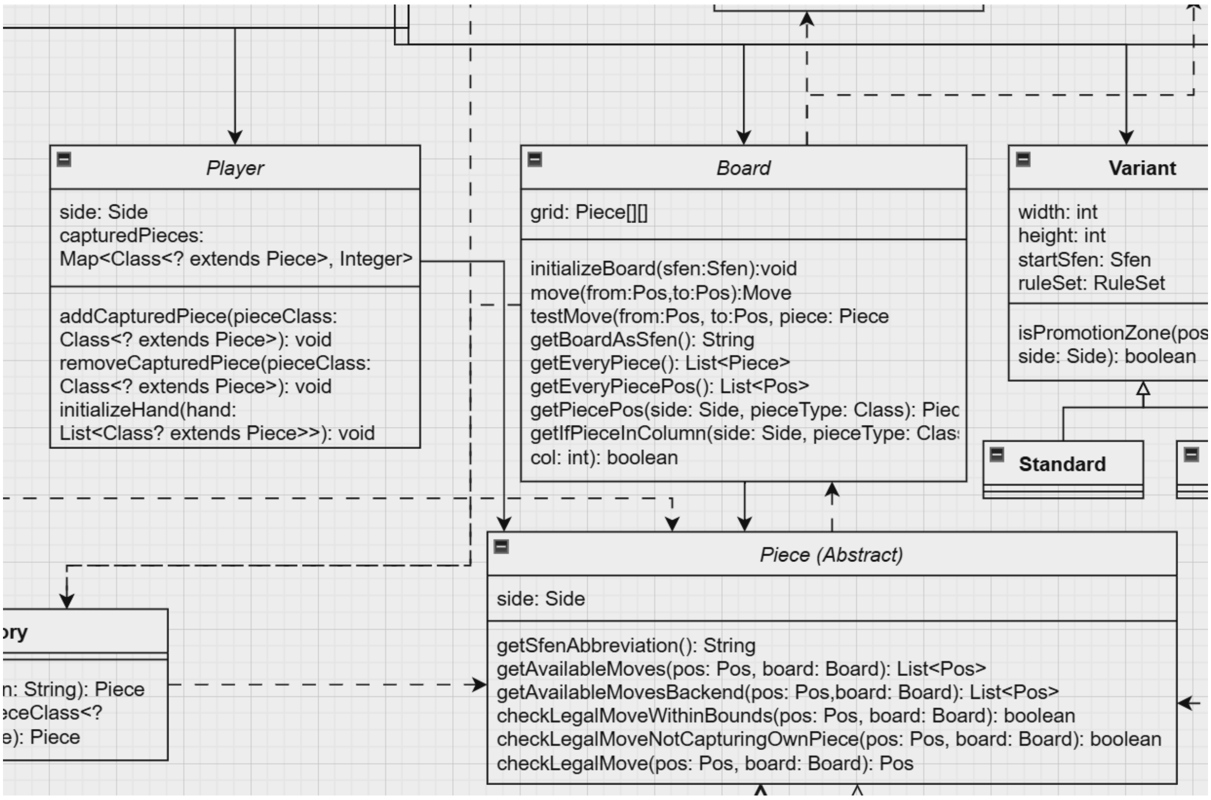
The `Board` class is the core of a Shogi game, acting as the central hub for managing the pieces and their movements. It uses a 2D grid to track where each piece is placed or where the empty spots are. This class provides several useful methods for moving pieces, retrieving pieces, and handling the board's overall state. For example, the `move` method updates the board by moving a piece from one position to another, while `testMove` allows you to simulate a move temporarily to check if it's valid.

The class also handles setting and retrieving pieces at specific positions with methods like `setAtPosition` and `getPieceAt`. It works with SFEN notation, which is a shorthand way of representing the board's state. Using methods like `setSfen`, the board can be initialized from an SFEN string, and `getBoardAsSfen` allows the current state to be converted back into SFEN format.

When it comes to placing pieces on the board, the `initializeBoard` method uses a `PieceFactory` to create pieces from SFEN abbreviations and then places them on the grid. The class also has some advanced features for querying the board, such as `getEveryPiece` and `getEveryPiecePos`, which return lists of all the pieces or their positions. If you need to find a specific piece, `getPiecePos` can pinpoint it based on its type and side. There's even a method, `ifPieceInColum`, to check if a certain piece exists in a given column.

With its ability to handle customizable board sizes and its focus on object-oriented design, the `Board` class makes it easy to manage the Shogi game logic. It keeps everything organized and helps integrate the board with other parts of the game, such as the pieces and the overall game flow.

Player class:


Player
side: Side
capturedPieces:
Map<Class<? extends Piece>, Integer>
addCapturedPiece(pieceClass:
Class<? extends Piece>): void
removeCapturedPiece(pieceClass:
Class<? extends Piece>): void
initializeHand(hand:
List<Class? extends Piece>>): void
Board
grid: Piece[][]
initializeBoard(sfen:Sfen):void
move(from:Pos,to:Pos):Move
testMove(from:Pos, to:Pos, piece: Piece
getBoardAsSfen(): String
getEveryPiece(): List<Piece>
getEveryPiecePos(): List<Pos>
getPiecePos(side: Side, pieceType: Class): Piec
getIfPieceInColumn(side: Side, pieceType: Clas
col: int): boolean
Variant
width: int
height: int
startSfen: Sfen
ruleSet: RuleSet
isPromotionZone(pos
side: Side): boolean
Standard
ory
n: String): Piece
eceClass<?
e): Piece
Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos

The Player class represents a player in the Shogi game, focusing on tracking the player's side and the pieces they have captured during the game. Each player is associated with a Side, and their captured pieces are stored in a map, where the key is the piece class and the value is the number of captured pieces of that type.

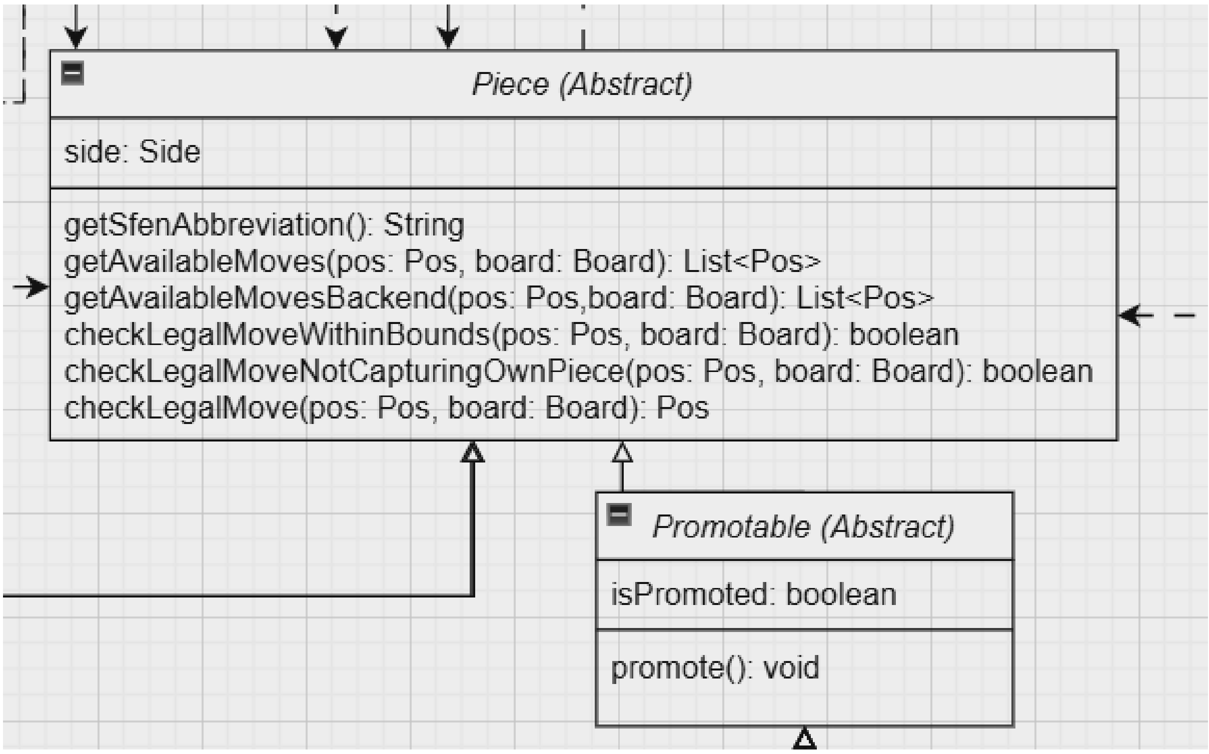
The class provides several methods for managing captured pieces. The `addCapturedPiece` method allows pieces to be added to the player's hand, either by specifying the number of pieces or just adding one. If a player needs to remove a captured piece, the `removeCapturedPiece` method decreases the count of that piece. This makes it easy to track the changes in captured pieces throughout the game.

To represent the captured pieces in SFEN format, the `getHandAsSfen` method generates a string that matches the SFEN notation, which is a compact way of showing the state of the captured pieces. It uses reflection to create instances of captured piece classes and retrieve their SFEN abbreviations. This string is then returned, allowing the captured pieces to be represented in a format compatible with other parts of the game system.

Additionally, the `intializeHand` method is used to set up the player's hand with a list of piece classes, initializing the count of each piece to zero. This ensures that the player starts with an empty hand, ready to begin capturing pieces as the game progresses.

Overall, the Player class provides an intuitive way to manage a player's captured pieces and integrates seamlessly with the rest of the game, offering methods to track, modify, and display captured pieces as the game unfolds.

Piece class:


Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos
Promotable (Abstract)
isPromoted: boolean
promote(): void

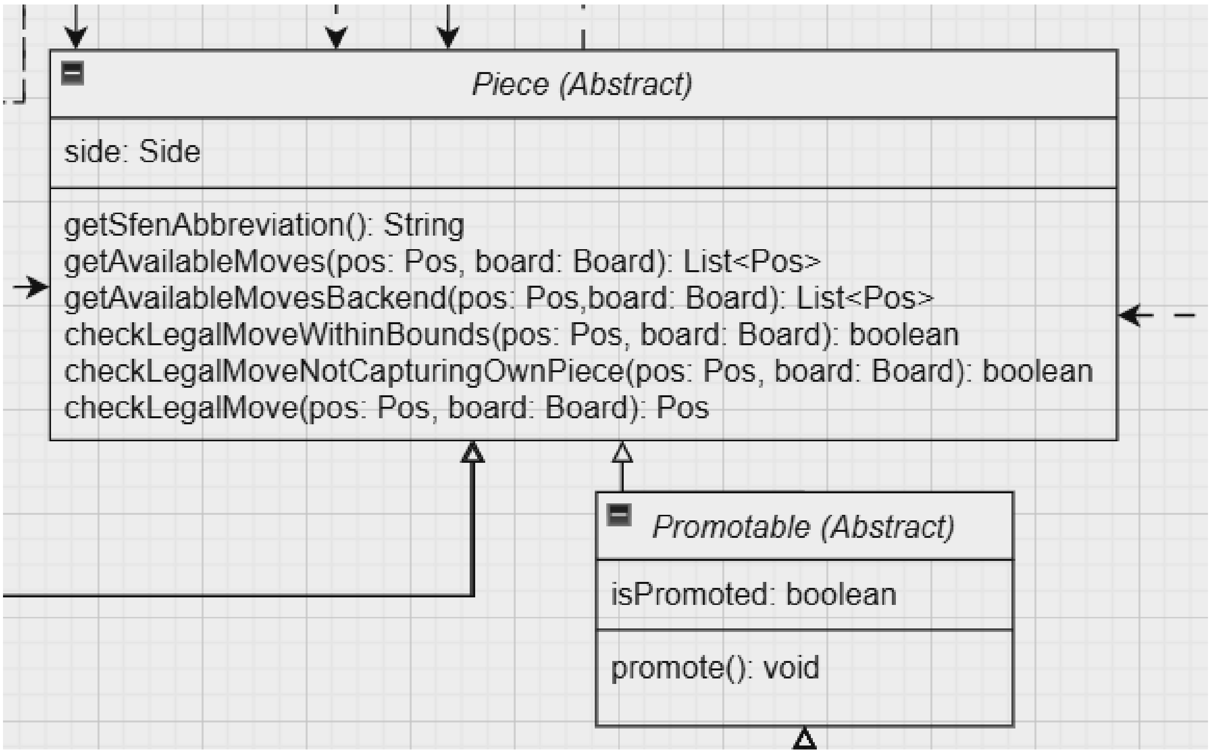
The Piece class serves as the base class for all Shogi pieces, encapsulating common functionality for handling pieces. It has a `side` property representing whether the piece belongs to the SENTE (black) or GOTE (white) player. The constructor takes a `Side` argument to initialize the piece's side.

The class includes a method to get the SFEN abbreviation of a piece, which is used to represent the piece in a compact form that is compatible with the Shogi notation system. The abbreviation is generated based on the first letter of the piece's class name, and it is converted to lowercase for the GOTE side to distinguish between the two players.

Abstract methods `getAvailableMoves` and `getAvailableMovesBackend` are defined, which are intended to be implemented by subclasses to specify the available movement rules for each specific piece.

There are also utility methods to check if a move is legal within the board's bounds (checkLegalMoveWithinBounds) and if the move would involve capturing a piece of the same side (checkLegalMoveNotCapturingOwnPiece). The `checkLegalMove` method combines these checks and returns a valid position if the move is legal, or `null` if it's not.

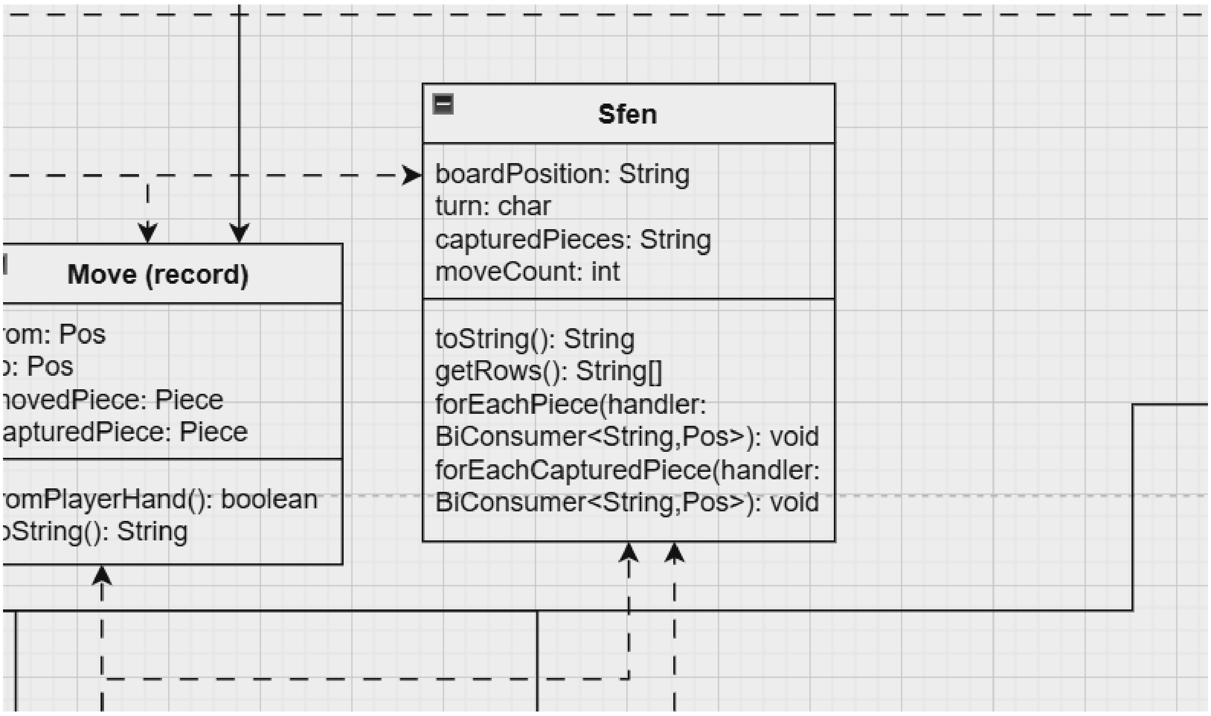
Promotable class:


Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos
Promotable (Abstract)
isPromoted: boolean
promote(): void

The Promotable class extends `Piece`, adding functionality for pieces that can be promoted. The class introduces a `boolean` field `isPromoted` to track whether the piece has been promoted. The `getSfenAbbreviation` method is overridden to append a "+" to the abbreviation when the piece is promoted, adhering to Shogi's promotion notation. A `promote` method is provided to mark the piece as promoted, and the `getIsPromoted` method allows querying the promotion status.

Overall, these two classes form the core behavior for pieces in the Shogi game. The `Piece` class handles basic movement rules and legal checks, while the `Promotable` class extends this functionality by allowing specific pieces to be promoted, altering their behavior and representation in the game.

Sfen class:

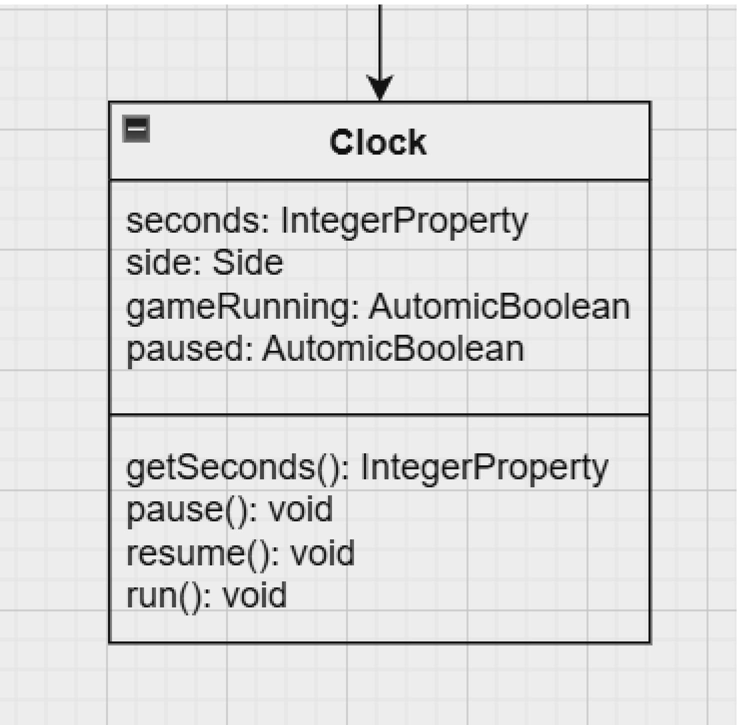


The Sfen class is essential for representing and manipulating the state of a Shogi game using the SFEN (Shogi Forsyth-Edwards Notation) format. This class encapsulates four key elements of the game state: the board position, the current turn, the captured pieces, and the move count. The `boardPosition` field stores the configuration of the game board, where each character or group of characters represents a row of the board. The `turn` field indicates whose turn it is, using 'b' for Black (Sente) and 'w' for White (Gote). The `capturedPieces` field represents pieces captured during the game, where characters indicate piece types, and numbers specify how many pieces of that type have been captured. The `moveCount` field keeps track of the number of moves made, starting from 1.

The class provides a constructor to initialize these fields either through individual parameters or by parsing an SFEN string. This string can be split into its components to set the board state, turn, captured pieces, and move count. To represent the game state as an SFEN string, the `toString()` method formats the internal fields into the correct SFEN format, which can be used for storage or transfer.

Additionally, the class offers methods for processing the board and captured pieces in the SFEN format. The `getRows()` method splits the board position into rows, while `forEachPiece()` iterates over each piece on the board, calling a handler with the piece's abbreviation and position. Similarly, `forEachCapturedPiece()` processes the captured pieces, calling a handler for each captured piece type and its count. These methods enable easy interaction with the game state, allowing for efficient processing and manipulation of board positions and captured pieces. The Sfen class plays a pivotal role in the game, ensuring that the game state is accurately represented, stored, and modified.

Clock class:



Clock

seconds: IntegerProperty
side: Side
gameRunning: AutomicBoolean
paused: AutomicBoolean

getSeconds(): IntegerProperty
pause(): void
resume(): void
run(): void

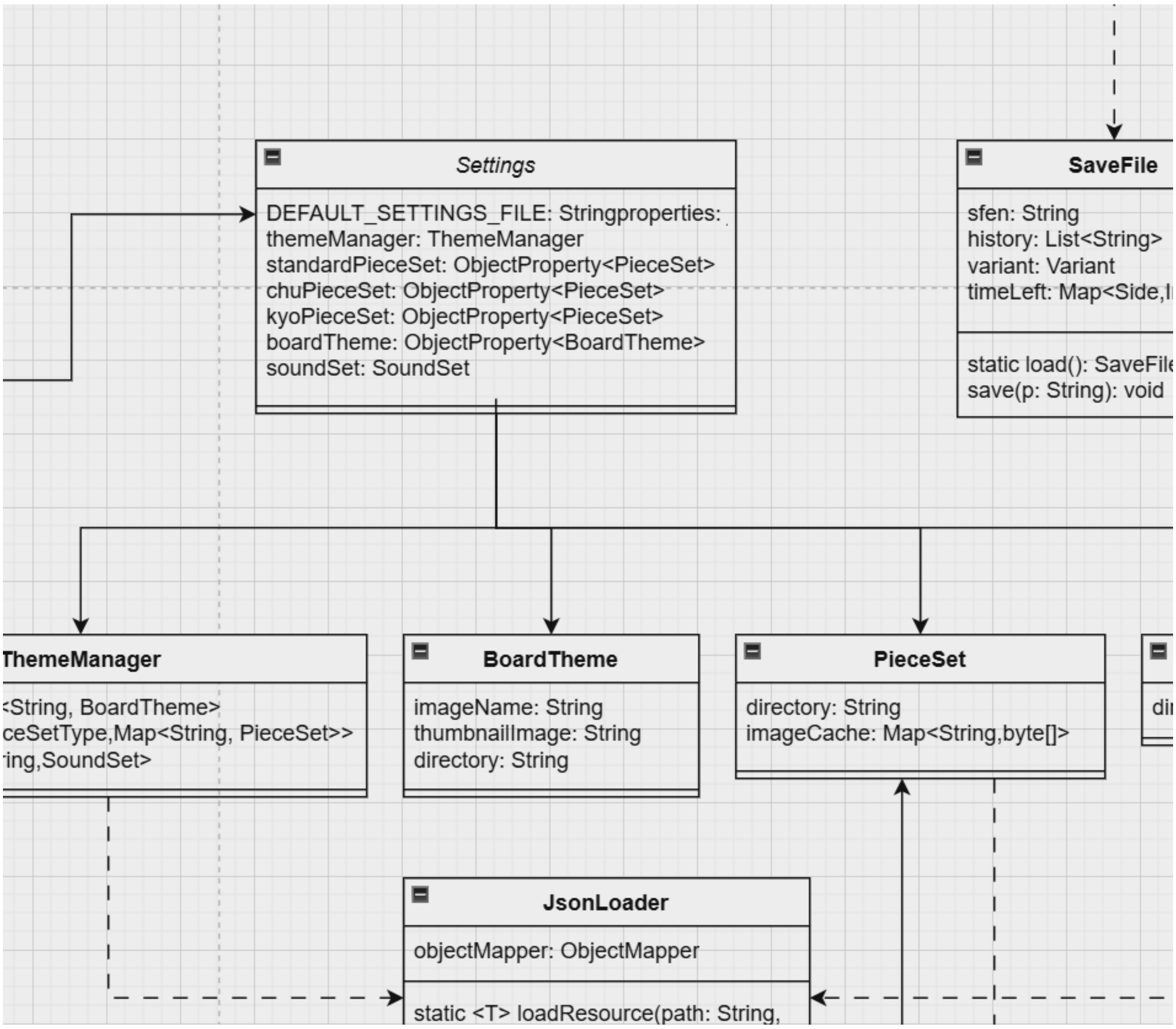
The Clock class in this Shogi game is responsible for managing the time for each player, ensuring that the game progresses within a controlled time frame. The class encapsulates several key features: the remaining time (in seconds) for the player, the associated side, and the state of the game (running or paused). The time left for a player is stored in an `IntegerProperty`, which is part of the JavaFX property system, allowing for easy binding to a user interface if necessary.

The Clock class also uses an `AtomicBoolean` to manage the game's running state and whether the clock is paused. The clock can be paused and resumed using the `pause()` and `resume()` methods, which manipulate the `paused` state and manage synchronization with the game thread. This ensures that the clock halts and resumes as needed, allowing for greater control over the game's flow.

The primary function of the clock is in the `run()` method, which operates on a separate thread to decrement the remaining time every second. The thread continuously checks if the game is running and if the clock is paused, using synchronization to wait when paused. If the time reaches zero, the clock will announce that the respective player has lost due to time expiration and stop the game. Additionally, if the thread is interrupted or if an error occurs during its operation, it handles these interruptions gracefully, ensuring the game's state is appropriately updated.

By separating the clock's operation into a dedicated thread, the Clock class allows for seamless time management without blocking the main game logic. This ensures a smooth user experience, where players are aware of their time limits, and the game adheres to a strict time constraint. The class offers flexibility by being easily extendable or modifiable to suit different types of time controls, such as adding additional features like incrementing time per move or handling sudden death scenarios.

Settings class:


Settings
DEFAULT_SETTINGS_FILE: Stringproperties:
themeManager: ThemeManager
standardPieceSet: ObjectProperty<PieceSet>
chuPieceSet: ObjectProperty<PieceSet>
kyoPieceSet: ObjectProperty<PieceSet>
boardTheme: ObjectProperty<BoardTheme>
soundSet: SoundSet
SaveFile
sfen: String
history: List<String>
variant: Variant
timeLeft: Map<Side,I
static load(): SaveFile
save(p: String): void
ThemeManager
<String, BoardTheme>
ceSetType,Map<String, PieceSet>>
ing,SoundSet>
BoardTheme
imageName: String
thumbnailImage: String
directory: String
PieceSet
directory: String
imageCache: Map<String,byte[]>
di
JsonLoader
objectMapper: ObjectMapper
static <T> loadResource(path: String,

The Settings class manages user preferences for the application, providing a robust system to handle themes, piece sets, and sound configurations. This class ensures that user settings are persisted across sessions by loading and saving them to a file, offering seamless integration between the application's resources and user preferences. It uses a `Properties` object to store key-value pairs representing settings and integrates a `ResourceManager` to resolve available resources like board themes and piece sets dynamically.

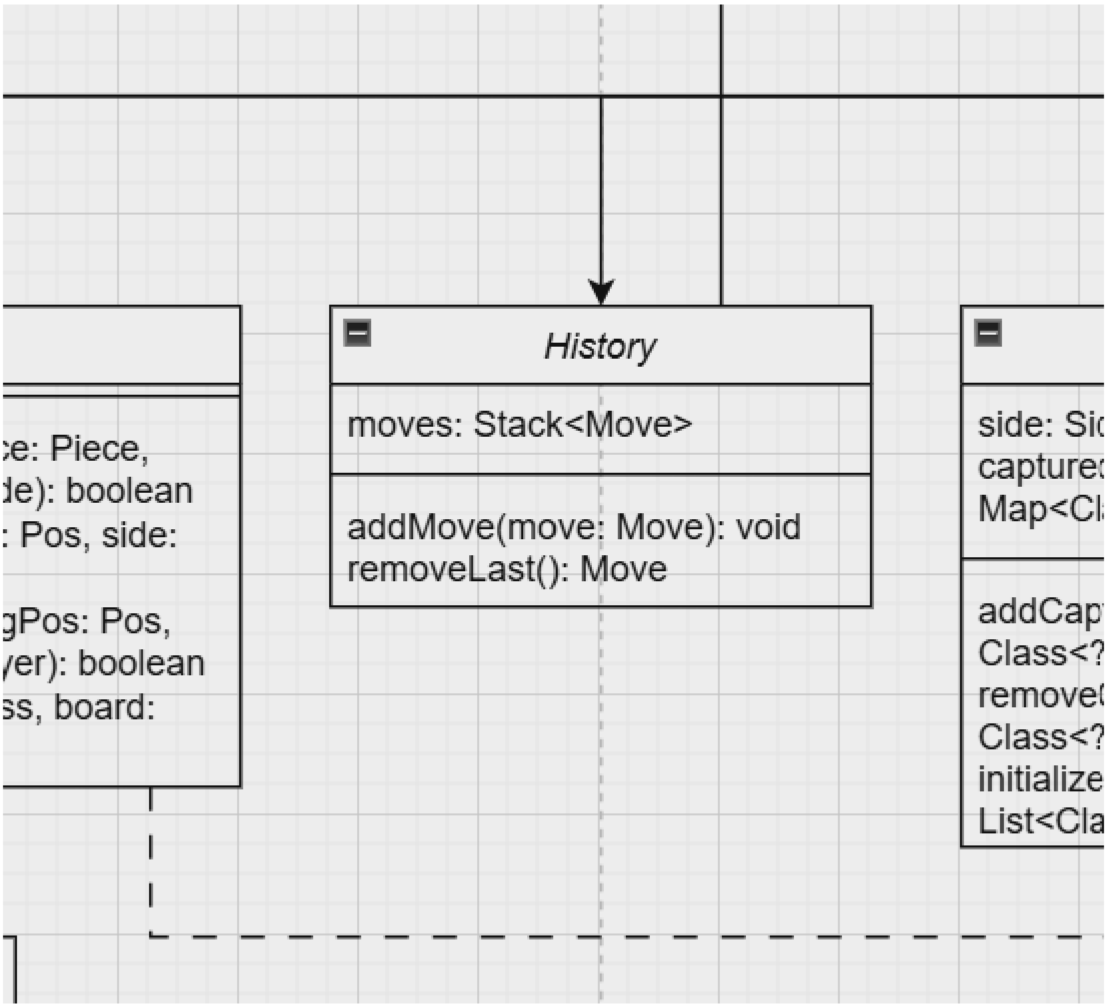
At initialization, the Settings class attempts to load a user-specific settings file. If the file does not exist, it creates one based on default settings embedded in the application resources. This approach ensures that users always start with a functional configuration while maintaining the flexibility to customize settings. The `load()` method reads the settings into JavaFX `ObjectProperty` objects, enabling reactive updates in the UI when these settings change.

The `save()` method allows modifications to persist by writing the current settings back to the user-specific file. This functionality updates properties like the selected board theme or sound set name to ensure consistency between the user's choices and the stored data.

Additionally, the class provides methods to access available resources, such as retrieving all board themes or piece sets. It facilitates setting specific configurations through methods like `setBoardTheme` and `setPieceSet`, which streamline customization. The inclusion of categorized piece sets (Standard, Chu, and Kyo) caters to specific game variations, highlighting its adaptability.

Overall, the Settings class acts as a centralized hub for managing user preferences and resources, promoting a smooth and user-friendly application experience. Its modular design makes it easy to extend with new settings or resource types, enhancing its utility for developers and end-users alike.

History class:



The History class provides a structured way to track and manage a sequence of moves in a game. Designed with versatility and efficiency in mind, it uses a `Stack` data structure to store moves, enabling quick additions and removals. This implementation supports typical operations required for managing game states, such as adding, retrieving, iterating, and removing moves, making it an integral part of game logic.

One of the key features of this class is the ability to add moves to the history using the `addMove` method. Each move represents a meaningful change in the game's state, and this history ensures that all actions can be tracked and reviewed. The `getNumberOfMoves` method provides the total number of moves in the stack, offering insights into the game's progression.

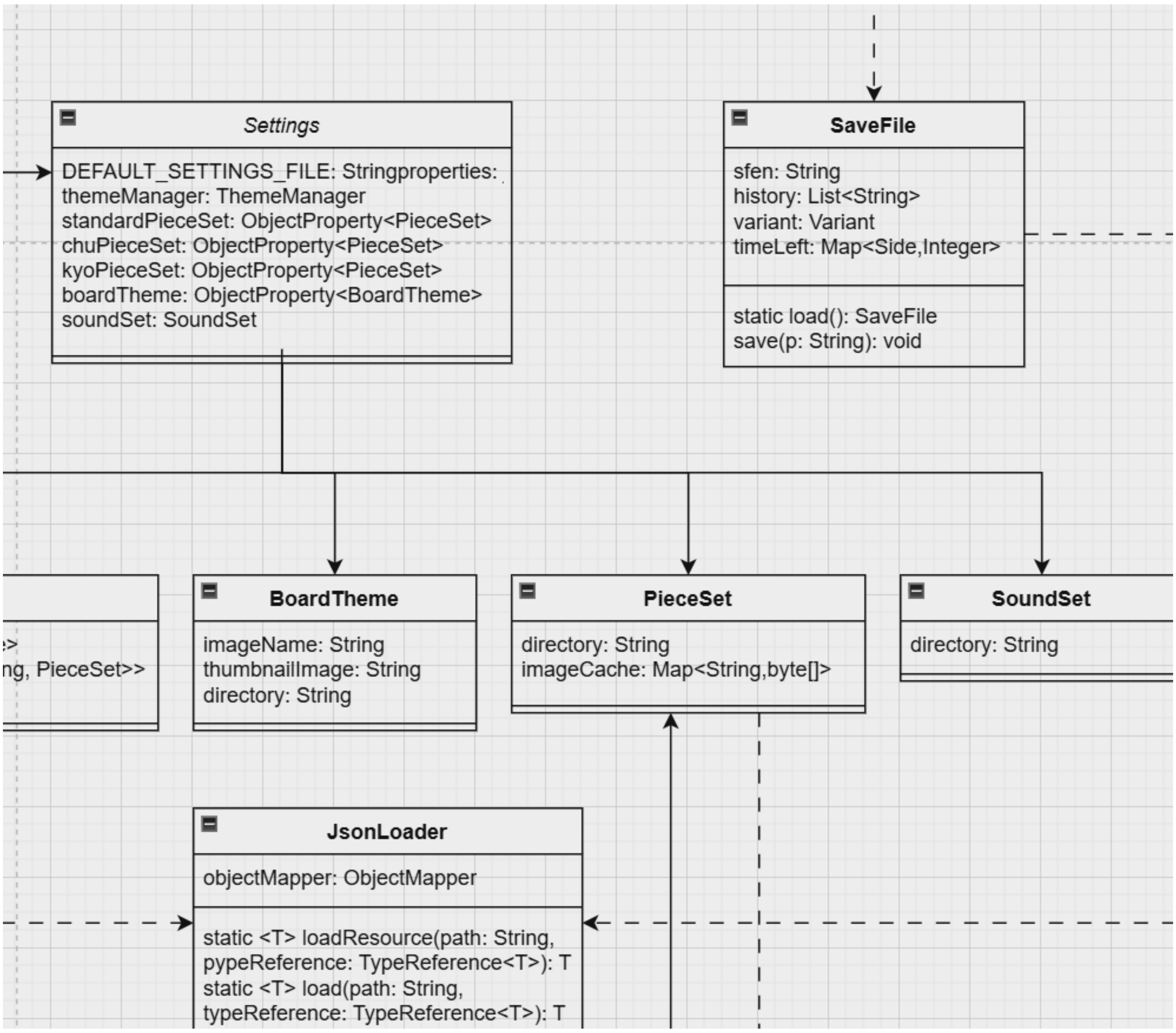
The `getMoves` method allows flexible iteration over a specified range of moves, either forward or backward, depending on the given indices. If no indices are provided, the method defaults to returning an iterator for the entire history. This dynamic range-based iteration makes it easy to analyze or replay specific segments of the game's history, supporting use cases like reviewing gameplay or implementing undo/redo features.

For modification, the `removeLast` method removes and returns the most recently added move, providing a straightforward way to undo actions. Similarly, the `getLast` method retrieves the latest move without altering the history, ensuring the integrity of the stack when only inspection is required.

The `serialize` method leverages Jackson annotations to convert the history into a serializable list of moves, facilitating saving and loading game states. This ensures that game progress can be persisted across sessions or transmitted between systems.

In summary, the History class combines robust functionality and modular design to handle move management effectively, supporting both gameplay mechanics and game state persistence.

SaveFile class:

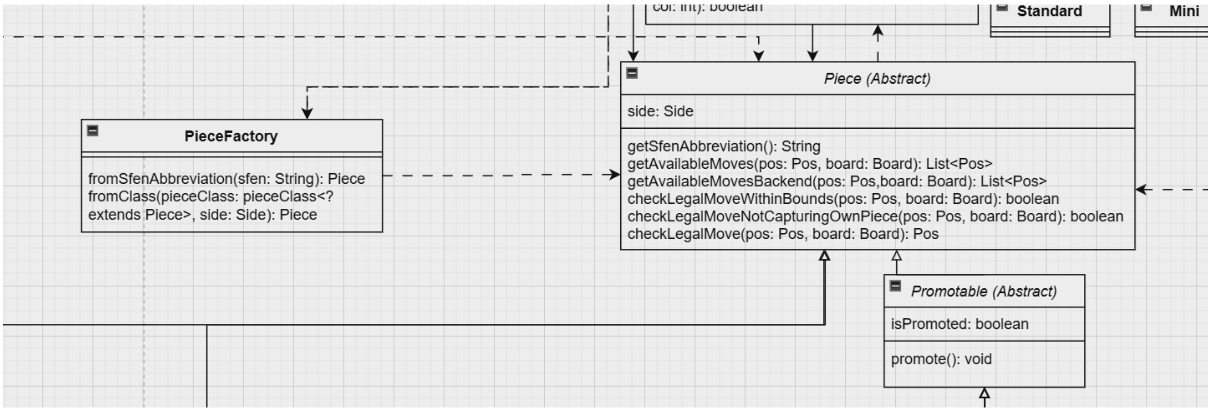

Settings
DEFAULT_SETTINGS_FILE: Stringproperties:
themeManager: ThemeManager
standardPieceSet: ObjectProperty<PieceSet>
chuPieceSet: ObjectProperty<PieceSet>
kyoPieceSet: ObjectProperty<PieceSet>
boardTheme: ObjectProperty<BoardTheme>
soundSet: SoundSet
SaveFile
sfen: String
history: List<String>
variant: Variant
timeLeft: Map<Side,Integer>
static load(): SaveFile
save(p: String): void
>
ng, PieceSet>>
BoardTheme
imageName: String
thumbnailImage: String
directory: String
PieceSet
directory: String
imageCache: Map<String,byte[]>
SoundSet
directory: String
JsonLoader
objectMapper: ObjectMapper
static <T> loadResource(path: String,
pypeReference: TypeReference<T>): T
static <T> load(path: String,
typeReference: TypeReference<T>): T

The SaveFile class serves as a comprehensive representation of a Shogi game's state, encapsulating critical details like the current board configuration, move history, game variant, and player timers. Leveraging JSON annotations, this class seamlessly integrates serialization and deserialization capabilities, making it ideal for saving and loading game progress.

Key attributes include the `sfen` string for representing the board state in Shogi Forsyth-Edwards Notation, a `history` list to track moves, a `variant` string identifying the game type, and a `timeLeft` map to store each player's remaining time. The class provides constructors for creating a SaveFile from raw data or directly from a Game instance, ensuring flexibility in usage.

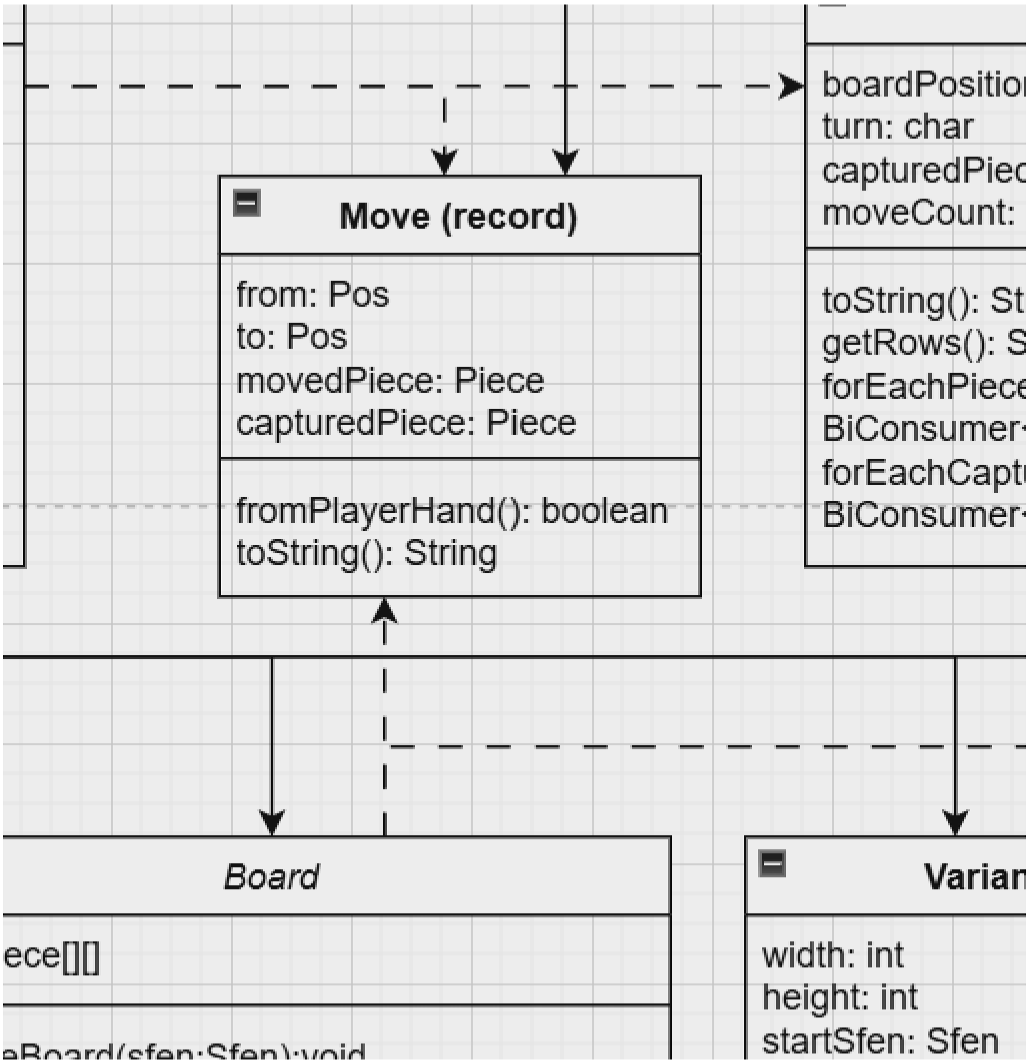
Its functionality extends to loading and saving game data, with default and custom file path options, ensuring robustness in handling persistence. Methods like `getSfen`, `getHistory`, and `getVariant` allow retrieval of deserialized objects for gameplay logic, while corresponding serialized getters ensure compatibility with JSON-based workflows. Overall, the SaveFile class is a vital component for managing and preserving Shogi game states effectively.

PieceFactory class:


col: int): boolean
Standard
Mini
Piece (Abstract)
side: Side
getSfenAbbreviation(): String
getAvailableMoves(pos: Pos, board: Board): List<Pos>
getAvailableMovesBackend(pos: Pos,board: Board): List<Pos>
checkLegalMoveWithinBounds(pos: Pos, board: Board): boolean
checkLegalMoveNotCapturingOwnPiece(pos: Pos, board: Board): boolean
checkLegalMove(pos: Pos, board: Board): Pos
PieceFactory
fromSfenAbbreviation(sfen: String): Piece
fromClass(pieceClass: pieceClass<? extends Piece>, side: Side): Piece
Promotable (Abstract)
isPromoted: boolean
promote(): void

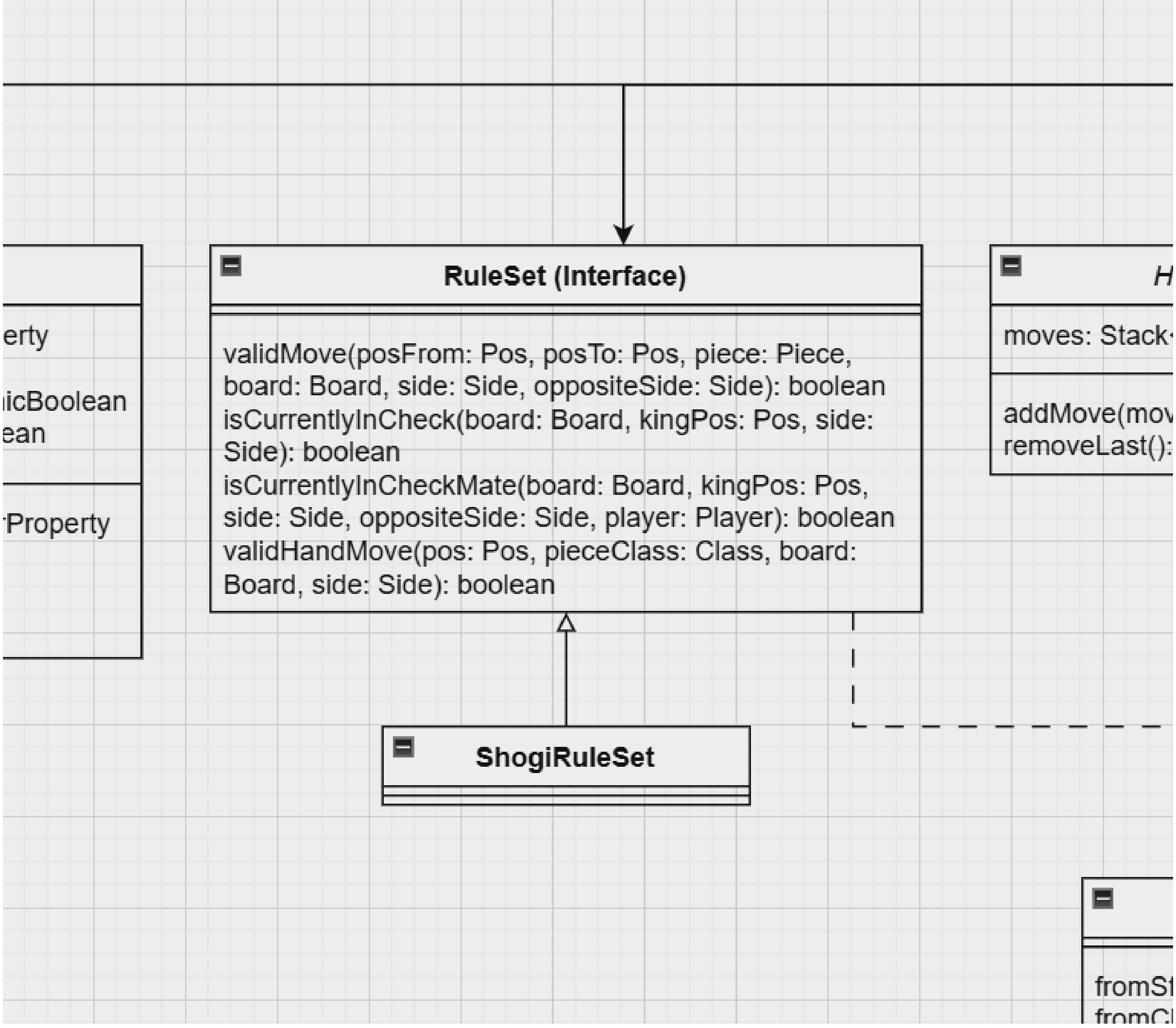
The PieceFactory class simplifies the creation of Shogi game pieces by providing methods to instantiate them from either SFEN abbreviations or their class types. The `fromSfenAbbreviation` method parses an SFEN string, identifies the piece type, its promotion status, and player side, and returns the corresponding Piece object, applying promotion if applicable. The `fromClass` method uses Java reflection to dynamically create a piece based on its class and side, allowing for flexible instantiation during gameplay or testing. By centralizing the logic for piece creation and initialization, PieceFactory ensures consistency, supports promotion mechanics, and handles errors gracefully when invalid input or instantiation issues occur.

Move class:



The Move class represents a single action in a Shogi game, encapsulating details about the starting position, ending position, the piece moved, and any piece captured during the move. It provides a constructor with JSON annotations for easy serialization and deserialization, supporting integration with external data formats. The class includes a method to determine if the move originated from a player's hand rather than the board and a `toString` method that generates a concise, SFEN-like representation of the move, indicating its type (normal move, capture, or drop). This design ensures clarity, efficiency, and compatibility for handling game moves.

ShogiRuleSet/RuleSet implementation:


erty
icBoolean
ean
Property
RuleSet (Interface)
validMove(posFrom: Pos, posTo: Pos, piece: Piece, board: Board, side: Side, oppositeSide: Side): boolean
isCurrentlyInCheck(board: Board, kingPos: Pos, side: Side): boolean
isCurrentlyInCheckMate(board: Board, kingPos: Pos, side: Side, oppositeSide: Side, player: Player): boolean
validHandMove(pos: Pos, pieceClass: Class, board: Board, side: Side): boolean
H
moves: Stack
addMove(mov
removeLast():
ShogiRuleSet
fromSt
fromCl

The ShogiRuleSet class implements the RuleSet interface and defines the rules for validating moves, check, and checkmate conditions in a game of Shogi. This class encapsulates the logic to enforce Shogi's gameplay constraints, ensuring valid piece movements, handling edge cases, and managing complex interactions between game elements.

The `validMove` method checks if a move is valid for a given piece by ensuring that the piece's movement logic, board state, and specific rules like avoiding self-check are respected. This method also considers special rules for the `King`, ensuring that a move doesn't place it in check.

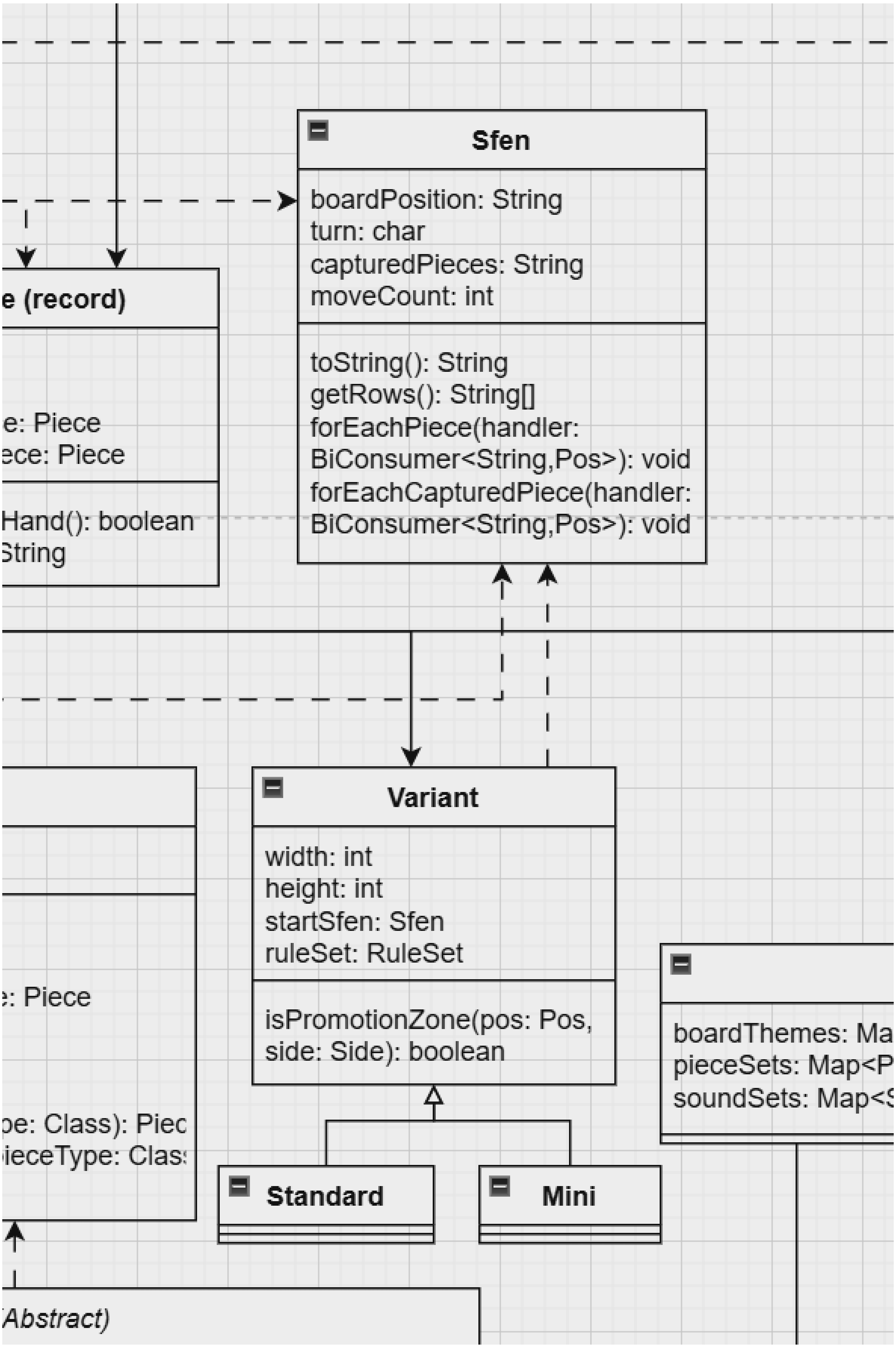
The `validHandMove` method enforces rules specific to placing a captured piece back on the board, such as ensuring pawns aren't placed in a column already containing another pawn of the same side and restricting certain pieces (e.g., `Knight`, `Lance`) from being placed in unreachable positions.

The `isCurrentlyInCheck` method evaluates if a player's `King` is in a threatened position by checking all possible moves from the opponent's pieces. It iterates through the board to gather the potential moves of all pieces and determines if any of them target the `King`.

The `isCurrentlyInCheckMate` method determines if a checkmate condition exists. It checks whether the `King` can escape the check and evaluates if other pieces can block the attack or capture the threatening piece. It also considers the possibility of using captured pieces from a player's hand to prevent checkmate.

The class adheres to object-oriented principles such as encapsulation by delegating piece-specific behavior to their respective classes and following the interface-driven design via `RuleSet`. This structure enables flexibility, as new rule sets or variations of Shogi can extend this class or implement the interface for customization, implementing a strategy pattern. The class provides clear and modular methods, ensuring maintainability and scalability in complex rule enforcement scenarios.

Variant class:



The Variant class serves as an abstract base class for defining different variants of Shogi. It encapsulates the common properties and methods shared by all Shogi variants, while leaving specific rules and configurations to be defined by concrete subclasses. This class implements some core features related to the board size, initial piece placement (Sfen), rule set, and promotion zones, which are essential for any variant of the game.

The `width` and `height` fields represent the dimensions of the board, with methods `getWidth()` and `getHeight()` providing access to them. These dimensions may vary depending on the specific variant (e.g., a larger board for certain variant types).

The `startSfen` field holds the initial setup of pieces in the game in the SFEN format, which is a standard Shogi notation. The method `getStartSfen()` allows retrieval of this initial configuration.

The `ruleSet` field is an instance of a `RuleSet`, which defines the gameplay rules for the specific variant. The `getRuleSet()` method provides access to the `RuleSet` for evaluating moves, check, checkmate, and other rules in the game.

The class includes abstract methods such as `getPieceSetType()`, `getHand()`, and `getPromotionZones()`. These methods must be implemented by subclasses to specify the type of pieces, available pieces that can be placed back on the board (the "hand"), and the promotion zones for each side in the variant.

The `inPromotionZone()` method checks if a given position is in the promotion zone for a particular side, based on the variant's rules.

The `serialize()` method uses the `@JsonValue` annotation, indicating that the class can be serialized to JSON using its simple class name, which can be useful for persisting or transferring variant data.

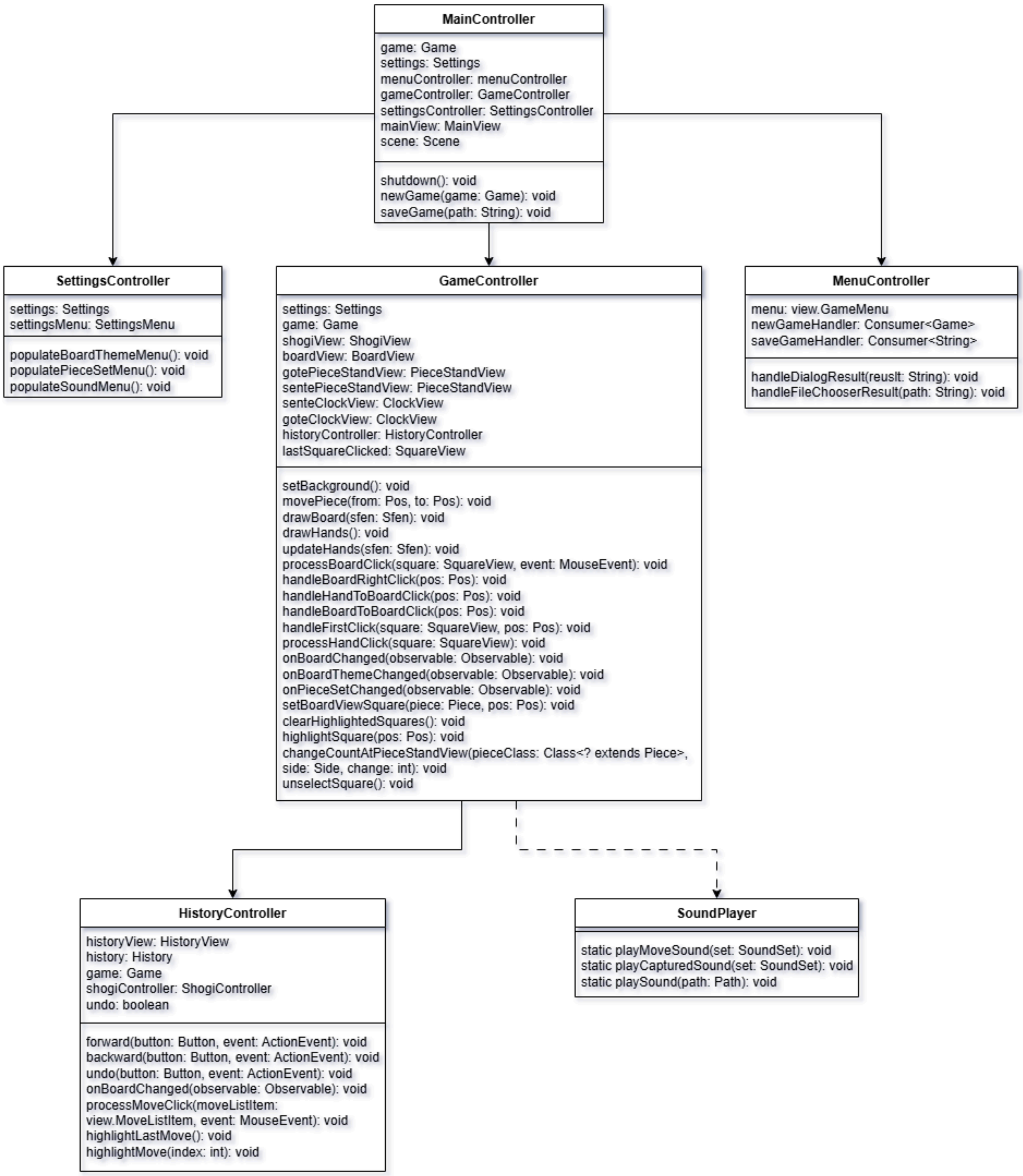
Overall, the Variant class provides a flexible and extendable framework for creating Shogi variants by defining the core properties and behaviors common across all variants, while allowing for the customization of specific rules and settings in subclasses.

ResourceManager:

The ResourceManager class is responsible for managing and providing access to resources such as board themes, piece sets, and sound sets used in the Shogi game. It stores these resources in maps: `boardThemes` (maps theme names to `BoardTheme` objects), `pieceSets` (categorizes piece sets by `PieceSetType`), and `soundSets` (maps sound set names to `SoundSet` objects). The constructor of ResourceManager initializes these maps by loading resource data from configuration files using the `JsonLoader`. The class provides methods for retrieving specific resources, such as `getBoardTheme()`, `getPieceSet()`, and `getSoundSet()`, which throw exceptions if the requested resource is not found. It also includes methods for retrieving the names of resources based on their objects, such as `getBoardThemeName()` and `getPieceSetName()`. A helper method, `getKeyFromValue()`, is used to search for a key based on a given value in the maps. Overall, the

ResourceManager class centralizes resource access, ensuring the game can load and use customizable resources like themes, piece sets, and sounds efficiently.

Controller:


MainController
game: Game
settings: Settings
menuController: menuController
gameController: GameController
settingsController: SettingsController
mainView: MainView
scene: Scene
shutdown(): void
newGame(game: Game): void
saveGame(path: String): void
SettingsController
settings: Settings
settingsMenu: SettingsMenu
populateBoardThemeMenu(): void
populatePieceSetMenu(): void
populateSoundMenu(): void
GameController
settings: Settings
game: Game
shogiView: ShogiView
boardView: BoardView
gotePieceStandView: PieceStandView
sentePieceStandView: PieceStandView
senteClockView: ClockView
goteClockView: ClockView
historyController: HistoryController
lastSquareClicked: SquareView
setBackground(): void
movePiece(from: Pos, to: Pos): void
drawBoard(sfen: Sfen): void
drawHands(): void
updateHands(sfen: Sfen): void
processBoardClick(square: SquareView, event: MouseEvent): void
handleBoardRightClick(pos: Pos): void
handleHandToBoardClick(pos: Pos): void
handleBoardToBoardClick(pos: Pos): void
handleFirstClick(square: SquareView, pos: Pos): void
processHandClick(square: SquareView): void
onBoardChanged(observable: Observable): void
onBoardThemeChanged(observable: Observable): void
onPieceSetChanged(observable: Observable): void
setBoardViewSquare(piece: Piece, pos: Pos): void
clearHighlightedSquares(): void
highlightSquare(pos: Pos): void
changeCountAtPieceStandView(pieceClass: Class<? extends Piece>,
side: Side, change: int): void
unselectSquare(): void
MenuController
menu: view.GameMenu
newGameHandler: Consumer<Game>
saveGameHandler: Consumer<String>
handleDialogResult(reuslt: String): void
handleFileChooserResult(path: String): void
HistoryController
historyView: HistoryView
history: History
game: Game
shogiController: ShogiController
undo: boolean
forward(button: Button, event: ActionEvent): void
backward(button: Button, event: ActionEvent): void
undo(button: Button, event: ActionEvent): void
onBoardChanged(observable: Observable): void
processMoveClick(moveListItem:
view.MoveListItem, event: MouseEvent): void
highlightLastMove(): void
highlightMove(index: int): void
SoundPlayer
static playMoveSound(set: SoundSet): void
static playCapturedSound(set: SoundSet): void
static playSound(path: Path): void

MainController:

MainController is initialized in App and it is the class from which Game, MainView, MenuController, SettingsController and GameController are instantiated. It is also where the Javafx Scene is created. It contains the method run when the program is closed, a method for adjusting the controller and view when a new game is created, and a method that creates a SaveFile instance when the game is saved.

GameController class:

This class handles the functionality of the project by allowing the user to interact with the view and actually changing the model data. The controller is dependent on the view, a few Javafx classes, the util classes, and many classes from the model. The class creates variables needed to save instances of other classes. The constructor takes settings, game, and shogiView as arguments and adds them to their corresponding variable. The PieceStandView:s, the ClockView:s, BoardView and HistoryView are fetched from shogiView and the constructor sets ClickHandlers on them to allow the controller to update the views when the handler informs it. At last, the constructor sets the background, draws hands, and updates the screen with everything. There are a few methods in this class for handling change and updating stuff in the model. The redraw method clears the boardView from pieces, or images, and draws new ones with the new data. The movePiece method calls game.move and removes the highlights from selecting a piece, and at last calls redraw. The drawBoard method creates all the pieces for the game and draws their image on the boardView by calling boardView.drawImageAt. The drawHands method gets a hand structure from game.variant and calls PieceFactory for each pieceClass that hand has, and draws their image on the views. The updateHands method calls setCountAt from PieceStandView on SENTE and GOTE. The processBoardClick method gets the position of the square that got pressed and checks the mouseEvent for different cases and handles it based on what input. If the square has a piece on it, the square is highlighted and saved so that next time a square is clicked, it can make a move. processHandClick handles clicks on the player hands. If the square clicked contains at least one captured piece, the square is saved. Next time processBoardClick is called, the captured piece is placed on the board.

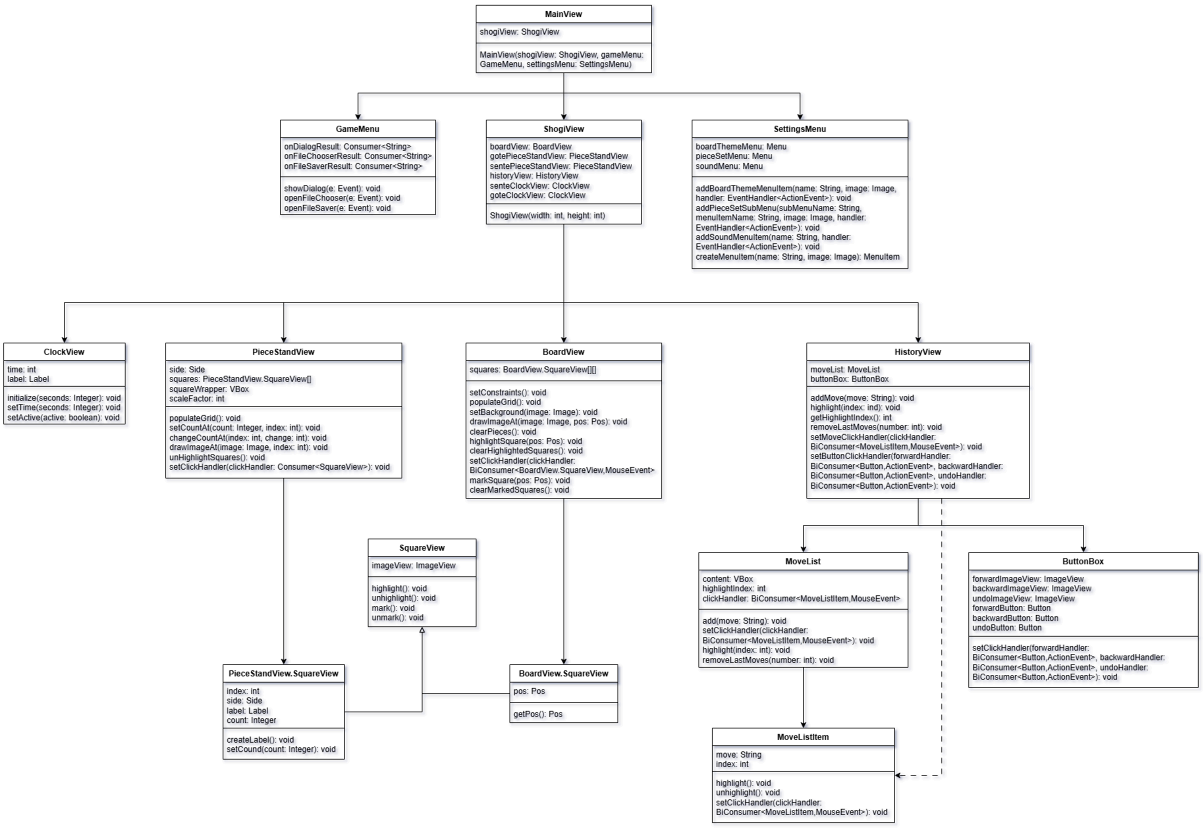
HistoryController class:

HistoryController contains instances of HistoryView, History and Game (from model), and GameController. It sets click handlers for HistoryView and contains methods that update HistoryView and ShogiView when the buttons and list items of HistoryView are clicked. In order to follow the Law of Demeter, ShogiView is not accessed directly, instead it is updated through methods in GameController. When a new move is highlighted, HistoryController uses the methods forwardMove and reverseMove in order to update the BoardView based on the changes made by a certain move. By running those methods for every move between the previously highlighted move and the new highlighted move, the BoardView is made to display the state of the board after the newly highlighted move had been made.

MenuController:

MenuController handles user input from the menus for creating a new game, saving and loading. It is dependent on several classes from model (Game, SaveFile, Variant and concrete implementations of Variant) as well as Side from util and GameMenu from view.

View:


MainView
shogiView: ShogiView
MainView(shogiView: ShogiView, gameMenu: GameMenu, settingsMenu: SettingsMenu)
GameMenu
onDialogResult: Consumer<String>
onFileChooserResult: Consumer<String>
onFileSaverResult: Consumer<String>
showDialog(e: Event): void
openFileChooser(e: Event): void
openFileSaver(e: Event): void
ShogiView
boardView: BoardView
gotePieceStandView: PieceStandView
sentePieceStandView: PieceStandView
historyView: HistoryView
senteClockView: ClockView
goteClockView: ClockView
ShogiView(width: int, height: int)
SettingsMenu
boardThemeMenu: Menu
pieceSetMenu: Menu
soundMenu: Menu
addBoardThemeMenuItem(name: String, image: Image, handler: EventHandler<ActionEvent>): void
addPieceSetSubMenu(subMenuName: String, menuItemName: String, image: Image, handler: EventHandler<ActionEvent>): void
addSoundMenuItem(name: String, handler: EventHandler<ActionEvent>): void
createMenuItem(name: String, image: Image): MenuItem
ClockView
time: int
label: Label
initialize(seconds: Integer): void
setTime(seconds: Integer): void
setActive(active: boolean): void
PieceStandView
side: Side
squares: PieceStandView.SquareView[]
squareWrapper: VBox
scaleFactor: int
populateGrid(): void
setCountAt(count: Integer, index: int): void
changeCountAt(index: int, change: int): void
drawImageAt(image: Image, index: int): void
unHighlightSquares(): void
setClickHandler(clickHandler: Consumer<SquareView>): void
BoardView
squares: BoardView.SquareView[][]
setConstraints(): void
populateGrid(): void
setBackground(image: Image): void
drawImageAt(image: Image, pos: Pos): void
clearPieces(): void
highlightSquare(pos: Pos): void
clearHighlightedSquares(): void
setClickHandler(clickHandler: BiConsumer<BoardView.SquareView,MouseEvent>
markSquare(pos: Pos): void
clearMarkedSquares(): void
HistoryView
moveList: MoveList
buttonBox: ButtonBox
addMove(move: String): void
highlight(index: ind): void
getHighlightIndex(): int
removeLastMoves(number: int): void
setMoveClickHandler(clickHandler: BiConsumer<MoveListItem,MouseEvent>): void
setButtonClickHandler(forwardHandler: BiConsumer<Button,ActionEvent>, backwardHandler: BiConsumer<Button,ActionEvent>, undoHandler: BiConsumer<Button,ActionEvent>): void
SquareView
imageView: ImageView
highlight(): void
unhighlight(): void
mark(): void
unmark(): void
MoveList
content: VBox
highlightIndex: int
clickHandler: BiConsumer<MoveListItem,MouseEvent>
add(move: String): void
setClickHandler(clickHandler: BiConsumer<MoveListItem,MouseEvent>): void
highlight(index: int): void
removeLastMoves(number: int): void
ButtonBox
forwardImageView: ImageView
backwardImageView: ImageView
undoImageView: ImageView
forwardButton: Button
backwardButton: Button
undoButton: Button
setClickHandler(forwardHandler: BiConsumer<Button,ActionEvent>, backwardHandler: BiConsumer<Button,ActionEvent>, undoHandler: BiConsumer<Button,ActionEvent>): void
PieceStandView.SquareView
index: int
side: Side
label: Label
count: Integer
createLabel(): void
setCound(count: Integer): void
BoardView.SquareView
pos: Pos
getPos(): Pos
MoveListItem
move: String
index: int
highlight(): void
unhighlight(): void
setClickHandler(clickHandler: BiConsumer<MoveListItem,MouseEvent>): void

MainView:

MainView contains a shogiView and initiates a MenuBar (Javafx) with a GameMenu and a SettingsMenu. It contains a method setShogiView, which can be used to replace the current ShogiView instance with a new one when a new game is created. It extends BorderPane from Javafx. It is only dependent on Javafx and ShogiView, GameMenu and SettingsMenu.

ShogiView class:

This class controls the layout of the game. It depends on BoardView, HistoryView, PieceStandView and ClockView and creates instance variables for each of them, creating a centralized class for all the views. ShogiView extends the HBox class from Javafx.

BoardView class:

This class handles the visual parts for the board. It extends GridPane from Javafx and is dependent on multiple Javafx classes. The constructor for this class creates an instance of BoardView.SquareView[size][size] and adds property, constraints and an id for the view. The last thing it does is populate the grid with BoardView.SquareView-objects. The class contains methods for updating the view with images and highlights and other things that are needed for the view to display the game correctly.

PieceStandView class:

PieceStandView extends the class VBox from Javafx. It is used to display the captured pieces. It contains an array of an internal SquareView-class, which inherits from the SquareView-class in view. Each square is a spot for one type of piece and contains a label denoting the number of that piece that has been captured. The SquareView also contains information about the player it belongs to and its index in the piece stand. That information is accessed by the controller when a square is clicked.

HistoryView class:

HistoryView extends VBox and is used to display the move history. It contains an object of the ButtonHBox class, which is an HBox containing the buttons used to navigate the move log. The list of moves itself is handled by an instance of the MoveListClass, which inherits from ScrollPane in Javafx. MoveList contains functionality to add and remove list items and select which item should be highlighted. Each list item is represented by an instance of the MoveListItem class, which extends HBox and specifies the appearance of a list item. The HistoryView class itself handles layout and contains methods to access functionality in MoveList and ButtonBox so that other classes don't have to access them directly, thus better abiding by the Law of Demeter.

Improvements:

To improve the code based on Object-Oriented Programming (OOP) principles and the Model-View-Controller (MVC) structure, several enhancements can be made. Firstly, adhering to the Single Responsibility Principle (SRP) can be achieved further by splitting large classes like ShogiRuleSet, GameController, and ResourceManager into smaller, more focused classes that handle specific tasks. For instance, a 'CheckmateValidator' can be made for checkmate logic and a ´ThemeLoader´ for resource loading. This promotes modularity and maintainability. Furthermore, encapsulation can be improved by making internal states more private and exposing only the necessary methods, limiting the direct access to fields like ´BoardThemes´ or ´PieceSets´.

There are a few improvements that could be made to the controller. Currently, several classes in the controller are dependent on many classes in the model. In order to avoid coupling and better follow the open-closed principle, it would be better if the controller interacted with the model via a single class or an interface implementing a facade pattern.

Further improvements can be made to the controller. Right now, the controller handles the orientation for the captured pieces, making the maintainability and extensibility worse. By changing this logic in the PieceStandView and moving the logic from controller to the view for orientation, the maintainability and extensibility would improve and would make the controller thinner, which follows the MVC structure.

During the implementation of the view, we planned to follow the Facade pattern by hiding the other views behind ShogiView and only letting the controller handle user inputs with ShogiView methods. However, this pattern was not implemented well enough and

GameController now has direct access to the other views by using getters. By hiding the views behind abstract methods and only letting the controller access these, the Facade pattern would be better implemented, following a more abstract and extensible View structure.

There are some things about the output of the app we would have improved with more time. One example is that pieces should only be able to promote on the same move that they enter the promotion zone. Currently pieces can promote at any time while inside the zone. Additionally, promotions are not stored in the Move class, meaning they can’t be reversed when using the undo-feature or viewing old states of the board.

Also, we have not implemented any visual indicator of when a player has won the game in the view (it only prints a message in the terminal). Implementing that would be an improvement.

The design would be more intuitive if there was an opening menu where the user could choose whether to create a new game, load, etc. There could also be an option to view a manual inside the game, which would improve the experience of new users.