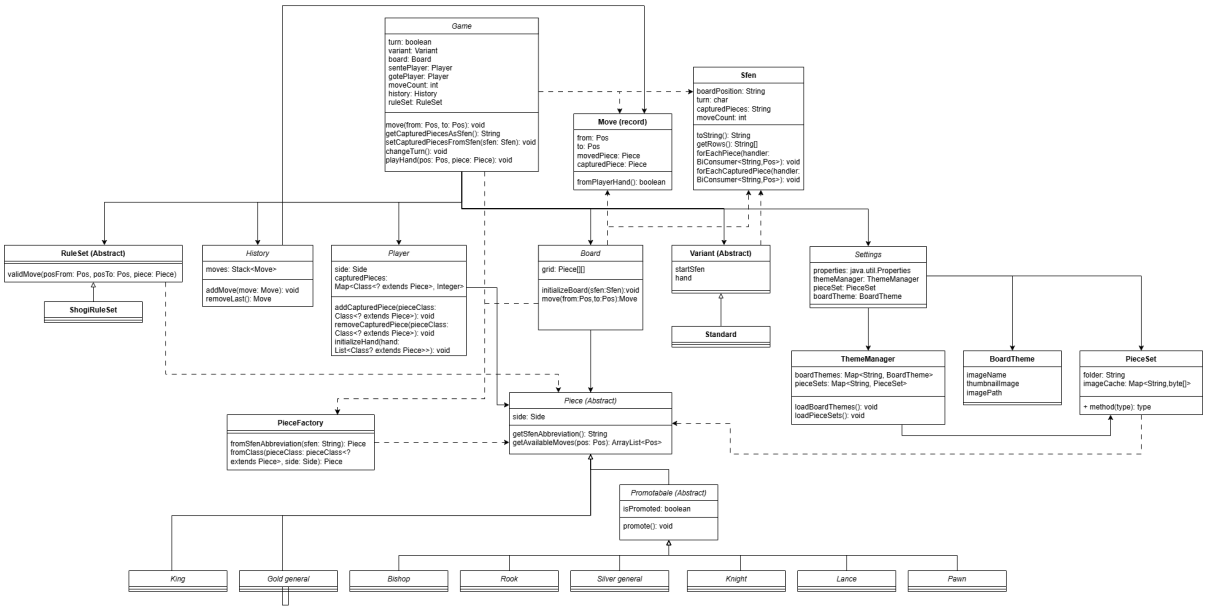
Our project follows the MVC pattern and SOLID principles. These design choices aim to make the system software more maintainable, scalable, and easier to understand. This allows other users to extend our algorithm and add additional content.

Model



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 is dependent on everything in the model and takes care of the gamestate. It takes in the variant of Shogi that is chosen, a New initialized board, new History instance and initializes two players, SENTE and GOTE. The ruleset is also fetched from the Variant class. The Game class has a few methods that affect the gamestate. It has move() which takes in two positions, from and to. The method checks if there is a piece at from-pos and also checks if the to-pos is a valid move for the piece that is moved. If the move is valid, we create a Move instance that holds the pos variables and two pieces, the one that is moving and the other is a Maybe Piece that is either null or a captured piece. The method handles the captured piece and changes turn when done.

The Game class has a few getters that are used to get important information about the gamestate, for instance, getBoard and getVariant are two important getters because other classes may need this information to create a working and correct Shogi game.

Board class:

The Board class takes care of everything related to the board, which is the grid of the board, the pieces positions in the board, and of course takes care of piece movement. This class is

dependent on Piece class and the utility class Pos, which is used for position. It is also dependent on the Sfen class which is the universal Shogi notation used for describing board positions of shogi games. This allows other users to extend our project with ease by following the same notation system. Sfen is used in the initializeBoard-function, which is used to place pieces on the board in the game's starting position. It may also be used when saving and loading games, but that is not implemented yet.

Player class:

This class handles player information that is which side the player is and the captured pieces that the player has. Board takes care of which pieces on the board that the player can use. The Player class is dependent on multiple utility classes, that is List, Map and LinkedHashMap. It is also dependent on the Piece class because the Player class holds information about the captured pieces. The Player class has a few methods that are used to access the captured pieces. These methods are getHandAsSfen, addCapturedPiece which has two implementations if the amount of captured pieces is not in the argument, removeCapturedPiece, and initializeHand which creates a key for every subclass of Piece and gives it value 0 for amount.

Pieces (Piece class and Promotable class):

There are multiple classes for piece but they all follow the abstract classes Piece and Promotable.

The Piece class is the structure for every subclass of it. It tells them which side they are on and gives them three getters, one for getting which side the piece is on, one for getting the Sfen abbreviation for the piece, and the third is for getting the Image abbreviation so we can get the correct image from our resources. The Piece class also has 2 abstract methods that the subclasses need to implement, getImageAbbreviationLetters and getAvailableMoves. The Piece class is dependent on the utility classes Pos, Side, and ArrayList.

The Promotable class extends Piece class and overrides the abbreviation getters to instead get the promoted version of the piece. It also adds a boolean isPromoted that is defaulted to false, followed by a getter for it. There is also a method that "promotes" the piece by changing isPromoted to true.

With these classes, multiple subclasses of Piece can be implemented with promotable aspects or not. In a subclass of promotable, we have two lists, moves and promotedMoves that are arraylists containing lists of ints. These lists are {column,row} which tells us how many grid slots the piece can move. The implementation of getAvailableMoves returns an ArrayList with the available positions that the piece can go from its current position.
With this implementation, a user can extend the amount of pieces by extending Piece or Promotable.

Sfen class:

This class implements the Universal Shogi notation Sfen which as explained in the Board class part is used for describing board positions in shogi games. This class depends on the model and the utility classes Pos and function.BiConsumer. This class is used for parsing

and creating Sfen strings which makes our project compatible with other Shogi implementations because it follows the same notations. The Sfen class contains a few getters and setters and methods that are used to handle these Sfen notations in different uses.

Settings class:

This class handles the properties and themes for the game. It depends on two io classes, IOException and InputStream, and one util class Properties. The class has two getters for pieceSet and boardTheme. Right now, everything happens in the constructor but this will be moved out to implement settings change. The constructor tries to fetch from a resource path as an inputStream and throws IllegalArgumentException if it fails. If it succeeds, it then loads the properties and adds the values to the corresponding variables.

History class:

The history class stores all moves made in a stack. It will be used to create a log where the players can revisit past moves and game states. It contains methods to access, add and remove from the stack. An instance of the History class is contained in the Game class.

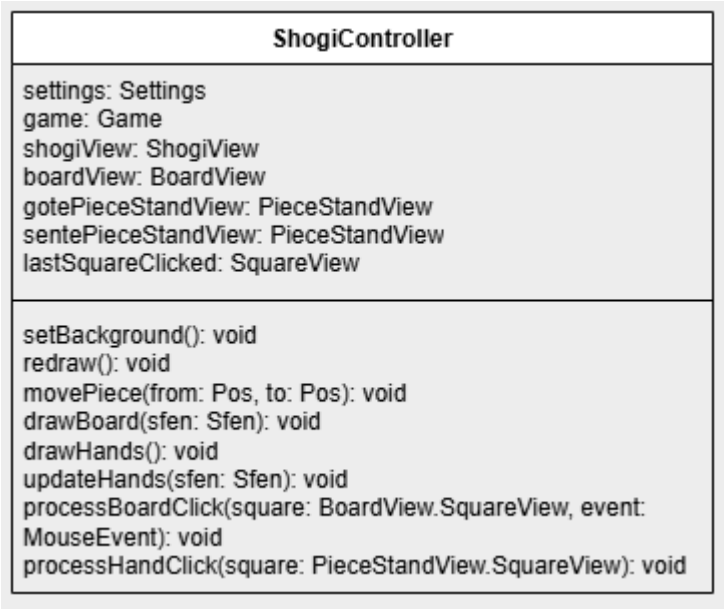
RuleSet:

RuleSet is an abstract class that contains a method to check if a move is valid within the rule set specified by the concrete implementation. We will also implement methods to check if a player has won and if there is a draw. Currently the only concrete implementation is ShogiRuleSet, which specifies the rules of standard shogi. Which rule set is used is determined by the variant.

Variant:

Variant is an abstract class representing different game variants (currently we only have standard shogi). Each concrete class inheriting from Variant contains information about the size of the board, the starting position as a Sfen string and the ruleset as a concrete subclass to RuleSet. It also has the method getHand, which returns a list of piece-types which should be contained in the player’s hand. (The player’s hand is where the captured pieces are stored).

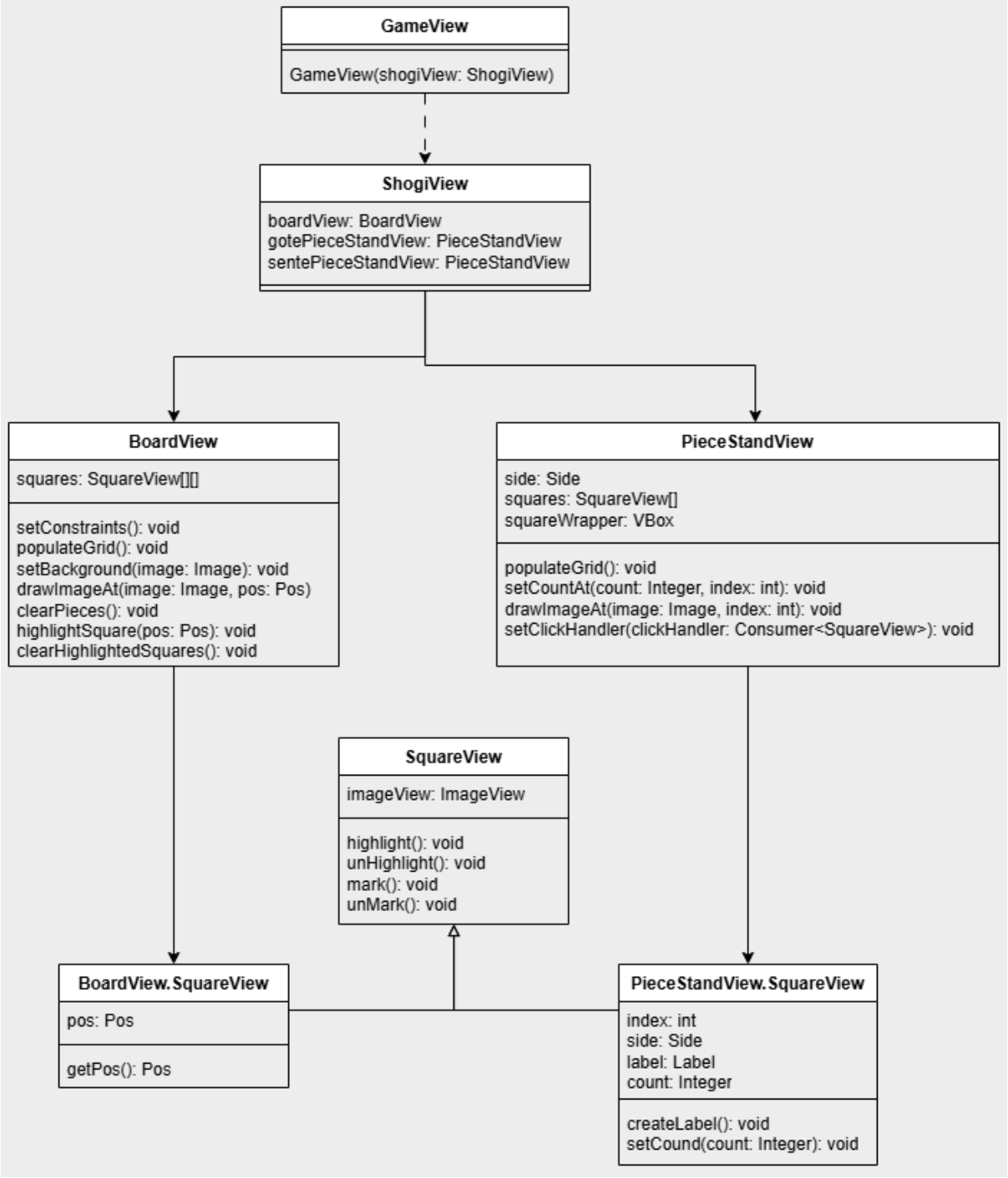
Controller:



ShogiController 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 shogiView, 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. gotePieceStandView, sentePieceStandView, BoardView, and historyView is fetched from shogiView and the constructor sets ClickHandlers on them to allow the constructor 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.

View:



ShogiView class:

This class depends on all the other views and creates instance variables for each of them, creating a centralized class for all the views. The full purpose of this class is for the controller to call redraw/update on specific views without being dependent on them. This makes the controller more independent from view and imitates a “facade” pattern.

BoardView class:

This class handles the visual parts for the board. It extends GridPane from Javafx and is dependent on multiple Javafx classes and other view classes. The constructor for this class creates an instance of SquareView[size][size] and adds property, constraints and an id for the view. The last thing it does is populate the grid with 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.

GameView class:

This class creates a view for the whole game. This view extends and implements BorderPane and is used by the App class to initialize the application with a view containing our program. The GameView contains a VBox with alignments and other properties and adds shogiView as a child for the VBox/GameView.

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.