Architektur

Ahmad Mohammad
Mounir Darwish
Seyedbehnam Mirhashemi

May 2021

Architecture

We decided to divide our program into three different components:

- Model
- View
- Controller

in order to isolate the model from the user interface (View), with also a third component (Controller) that coordinates both the models and views.
The ‘main‘ method creates a new console view when specifying the –no-gui parameter, and transfers it to a new console controller.
The Attribute class holds all the attributes that the game needs, for example, the color of pieces that the player choose to play with, and their respective methods. It holds also the attributes which specifies the status of the game, for example, if the game has ended in a draw or a win, and when one of the player's King is in danger.

Model

The Model, in this case, the Game, is responsible for managing the data of the program. It receives user input from the controller.
The game has multiple fields, for example, the board of the game and the players, that are playing and interacting with game. The players choose to play with color either the white or the black.
The model has different methods, for example methods that load the player's pieces, add or remove pieces to the beaten list of the player, the game also checks if the player inputted a legal move.
After each round the game checks the status of itself, if the player's king is in danger or the game ended with a win or a draw, it tells the controller, which then the controller tells the view to display the message.
Each piece has it's own class that extends the overall piece class. In each piece

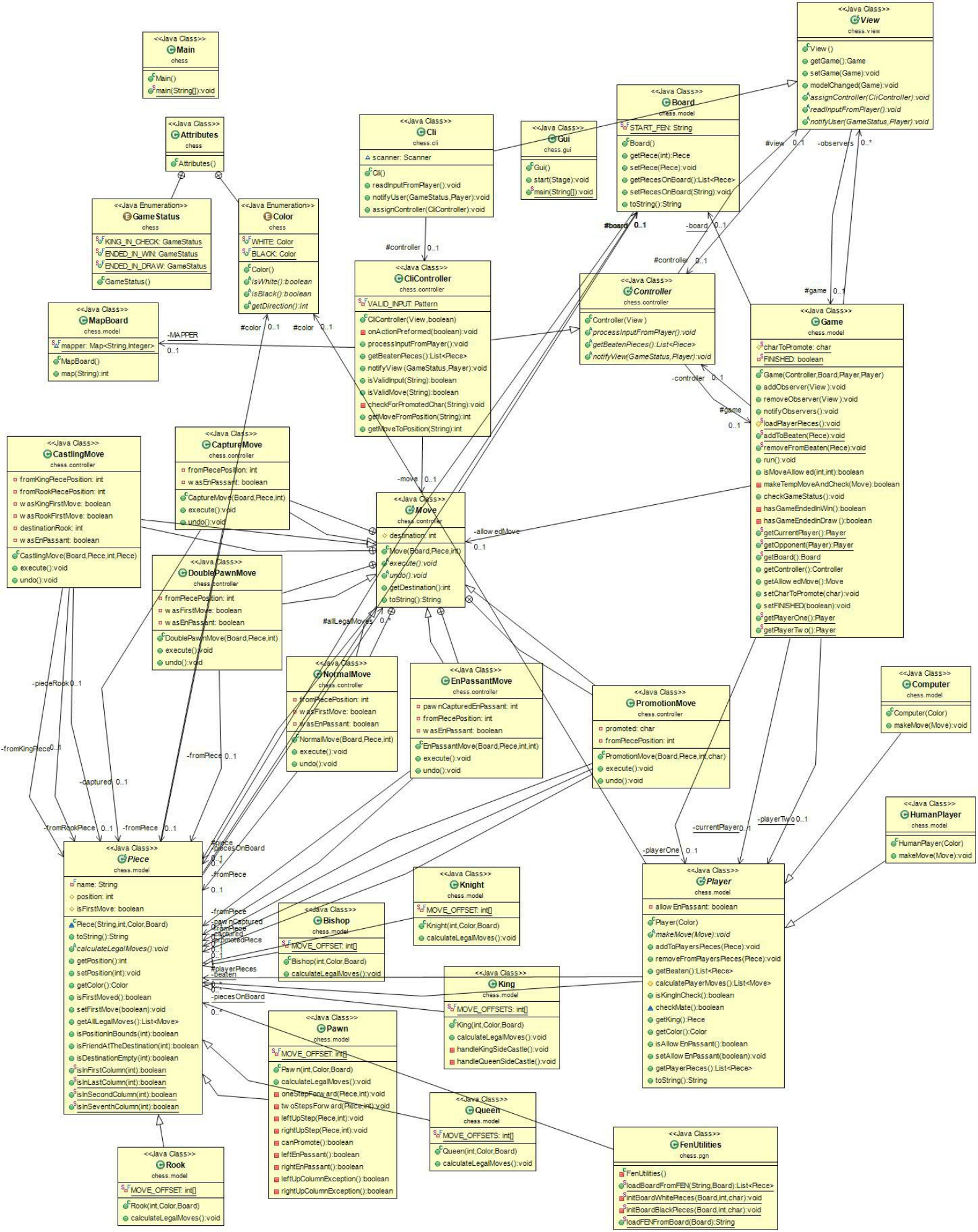
class, the respective logic of the legal moves of the piece is implemented based on the current position of the piece and the move offsets of the piece, with regard to some exception that are added to each piece individually.
In the class player, each player has it's own list of pieces and king. The beaten pieces can be called from the respective player. The class has a field 'allowEnPassant' that tells the game if the player can preform an En Passant move or if he missed his chance in doing so.

View

The View means the presentation of the model in a particular format. Here, at the first stage of the program, the view displays the console representation of the program.
The console based interface reads an input from the player using the scanner and passes that input to the controller for validation.
The view receives requests from the controller, which in turns get displayed to the user.

Controller

The Controller creates the game when it gets created itself and responds to the user input from the view and performs interactions on the data model objects. The controller receives the input, validates it and then passes the input to the model.


<<Java Class>>
Main
chess
Main()
main(String[]):void
<<Java Class>>
Attributes
chess
Attributes()
<<Java Enumeration>>
Game Status
chess
KING_IN_CHECK: GameStatus
ENDED_IN_WIN: GameStatus
ENDED_IN_DRAW: GameStatus
GameStatus()
<<Java Enumeration>>
Color
chess
WHITE: Color
BLACK: Color
Color()
isWhite():boolean
isBlack():boolean
getDirection():int
<<Java Class>>
Cli
chess.cli
scanner: Scanner
Cli()
readInputFromPlayer():void
notifyUser(GameStatus,Player):void
assignController(CliController):void
<<Java Class>>
Gui
chess.gui
Gui()
start(Stage):void
main(String[]):void
<<Java Class>>
Board
chess.model
START_FEN: String
Board()
getPiece(int):Piece
setPiece(Piece):void
getPiecesOnBoard():List<Piece>
setPiecesOnBoard(String):void
toString():String
<<Java Class>>
View
chess.view
View()
getGame():Game
setGame(Game):void
modelChanged(Game):void
assignController(CliController):void
readInputFromPlayer():void
notifyUser(GameStatus,Player):void
#view 0..1
-observers 0..*
#board 0..1
-board 0..1
#controller 0..1
<<Java Class>>
CliController
chess.controller
VALID_INPUT: Pattern
CliController(View,boolean)
onActionPreformed(boolean):void
processInputFromPlayer():void
getBeatenPieces():List<Piece>
notifyView(GameStatus,Player):void
isValidInput(String):boolean
isValidMove(String):boolean
checkForPromotedChar(String):void
getMoveFromPosition(String):int
getMoveToPosition(String):int
<<Java Class>>
Controller
chess.controller
Controller(View)
processInputFromPlayer():void
getBeatenPieces():List<Piece>
notifyView(GameStatus,Player):void
-controller 0..1
#game 0..1
<<Java Class>>
Game
chess.model
charToPromote: char
FINISHED: boolean
Game(Controller,Board,Player,Player)
addObserver(View):void
removeObserver(View):void
notifyObservers():void
loadPlayerPieces():void
addToBeaten(Piece):void
removeFromBeaten(Piece):void
run():void
isMoveAllowed(int,int):boolean
makeTempMoveAndCheck(Move):boolean
checkGameStatus():void
hasGameEndedInWin():boolean
hasGameEndedInDraw():boolean
getCurrentPlayer():Player
getOpponent(Player):Player
getBoard():Board
getController():Controller
getAllowedMove():Move
setCharToPromote(char):void
setFINISHED(boolean):void
getPlayerOne():Player
getPlayerTwo():Player
<<Java Class>>
MapBoard
chess.model
mapper: Map<String,Integer>
MapBoard()
map(String):int
-MAPPER
0..1
#color 0..1
#color 0..1
<<Java Class>>
CastlingMove
chess.controller
fromKingPiecePosition: int
fromRookPiecePosition: int
wasKingFirstMove: boolean
wasRookFirstMove: boolean
destinationRook: int
wasEnPassant: boolean
CastlingMove(Board,Piece,int,Piece)
execute():void
undo():void
<<Java Class>>
CaptureMove
chess.controller
fromPiecePosition: int
wasEnPassant: boolean
CaptureMove(Board,Piece,int)
execute():void
undo():void
<<Java Class>>
Move
chess.controller
destination: int
Move(Board,Piece,int)
execute():void
undo():void
getDestination():int
toString():String
-move 0..1
-allowedMove
0..1
<<Java Class>>
DoublePawnMove
chess.controller
fromPiecePosition: int
wasFirstMove: boolean
wasEnPassant: boolean
DoublePawnMove(Board,Piece,int)
execute():void
undo():void
#allLegalMoves 0..*
<<Java Class>>
NormalMove
chess.controller
fromPiecePosition: int
wasFirstMove: boolean
wasEnPassant: boolean
NormalMove(Board,Piece,int)
execute():void
undo():void
<<Java Class>>
EnPassantMove
chess.controller
pawnCapturedEnPassant: int
fromPiecePosition: int
wasEnPassant: boolean
EnPassantMove(Board,Piece,int,int)
execute():void
undo():void
<<Java Class>>
PromotionMove
chess.controller
promoted: char
fromPiecePosition: int
PromotionMove(Board,Piece,int,char)
execute():void
undo():void
<<Java Class>>
Computer
chess.model
Computer(Color)
makeMove(Move):void
<<Java Class>>
HumanPlayer
chess.model
HumanPlayer(Color)
makeMove(Move):void
-pieceRook 0..1
-fromKingPiece 0..1
-captured 0..1
-fromPiece 0..1
-fromRookPiece 0..1
-fromPiece 0..1
-currentPlayer 0..1
-playerTwo 0..1
-playerOne 0..1
<<Java Class>>
Piece
chess.model
name: String
position: int
isFirstMove: boolean
Piece(String,int,Color,Board)
toString():String
calculateLegalMoves():void
getPosition():int
setPosition(int):void
getColor():Color
isFirstMoved():boolean
setFirstMove(boolean):void
getAllLegalMoves():List<Move>
isPositionInBounds(int):boolean
isFriendAtTheDestination(int):boolean
isDestinationEmpty(int):boolean
isInFirstColumn(int):boolean
isInLastColumn(int):boolean
isInSecondColumn(int):boolean
isInSeventhColumn(int):boolean
#piece
-piecesOnBoard
0..1
-fromPiece
0..1
-fromPiece
-pawnCaptured
-fromPiece
-capturedPiece
-promotedPiece
#playerPieces
-beaten
0..*
-piecesOnBoard
0..*
<<Java Class>>
Player
chess.model
allowEnPassant: boolean
Player(Color)
makeMove(Move):void
addToPlayersPieces(Piece):void
removeFromPlayersPieces(Piece):void
getBeaten():List<Piece>
calculatePlayerMoves():List<Move>
isKingInCheck():boolean
checkMate():boolean
getKing():Piece
getColor():Color
isAllowEnPassant():boolean
setAllowEnPassant(boolean):void
getPlayerPieces():List<Piece>
toString():String
<<Java Class>>
Knight
chess.model
MOVE_OFFSET: int[]
Knight(int,Color,Board)
calculateLegalMoves():void
<<Java Class>>
Bishop
chess.model
MOVE_OFFSET: int[]
Bishop(int,Color,Board)
calculateLegalMoves():void
<<Java Class>>
King
chess.model
MOVE_OFFSETS: int[]
King(int,Color,Board)
calculateLegalMoves():void
handleKingSideCastle():void
handleQueenSideCastle():void
<<Java Class>>
Pawn
chess.model
MOVE_OFFSET: int[]
Pawn(int,Color,Board)
calculateLegalMoves():void
oneStepForward(Piece,int):void
twoStepsForward(Piece,int):void
leftUpStep(Piece,int):void
rightUpStep(Piece,int):void
canPromote():boolean
leftEnPassant():boolean
rightEnPassant():boolean
leftUpColumnException():boolean
rightUpColumnException():boolean
<<Java Class>>
Queen
chess.model
MOVE_OFFSETS: int[]
Queen(int,Color,Board)
calculateLegalMoves():void
<<Java Class>>
Rook
chess.model
MOVE_OFFSET: int[]
Rook(int,Color,Board)
calculateLegalMoves():void
<<Java Class>>
FenUtilities
chess.pgn
FenUtilities()
loadBoardFromFEN(String,Board):List<Piece>
initBoardWhitePieces(Board,int,char):void
initBoardBlackPieces(Board,int,char):void
loadFENFromBoard(Board):String