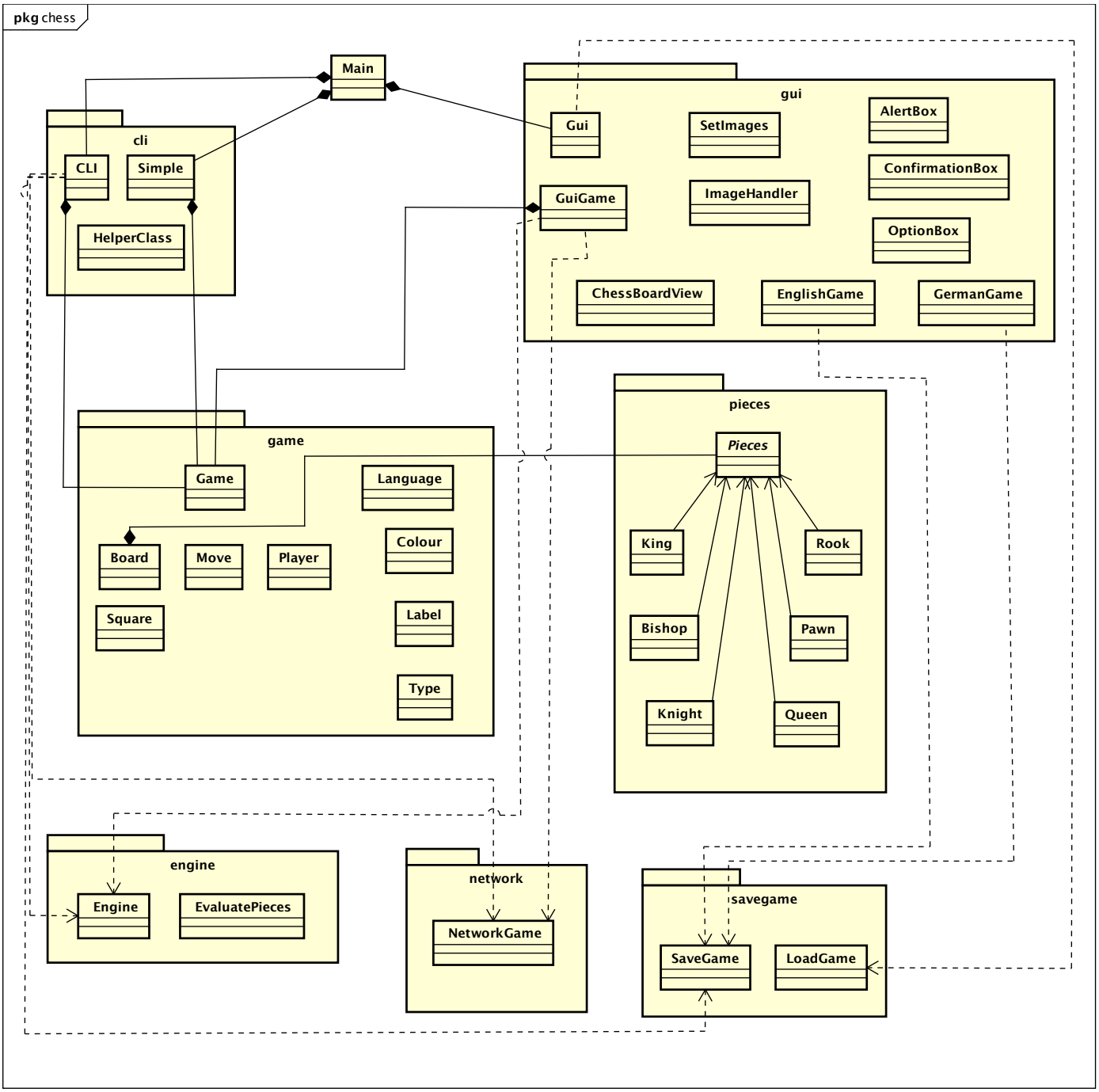
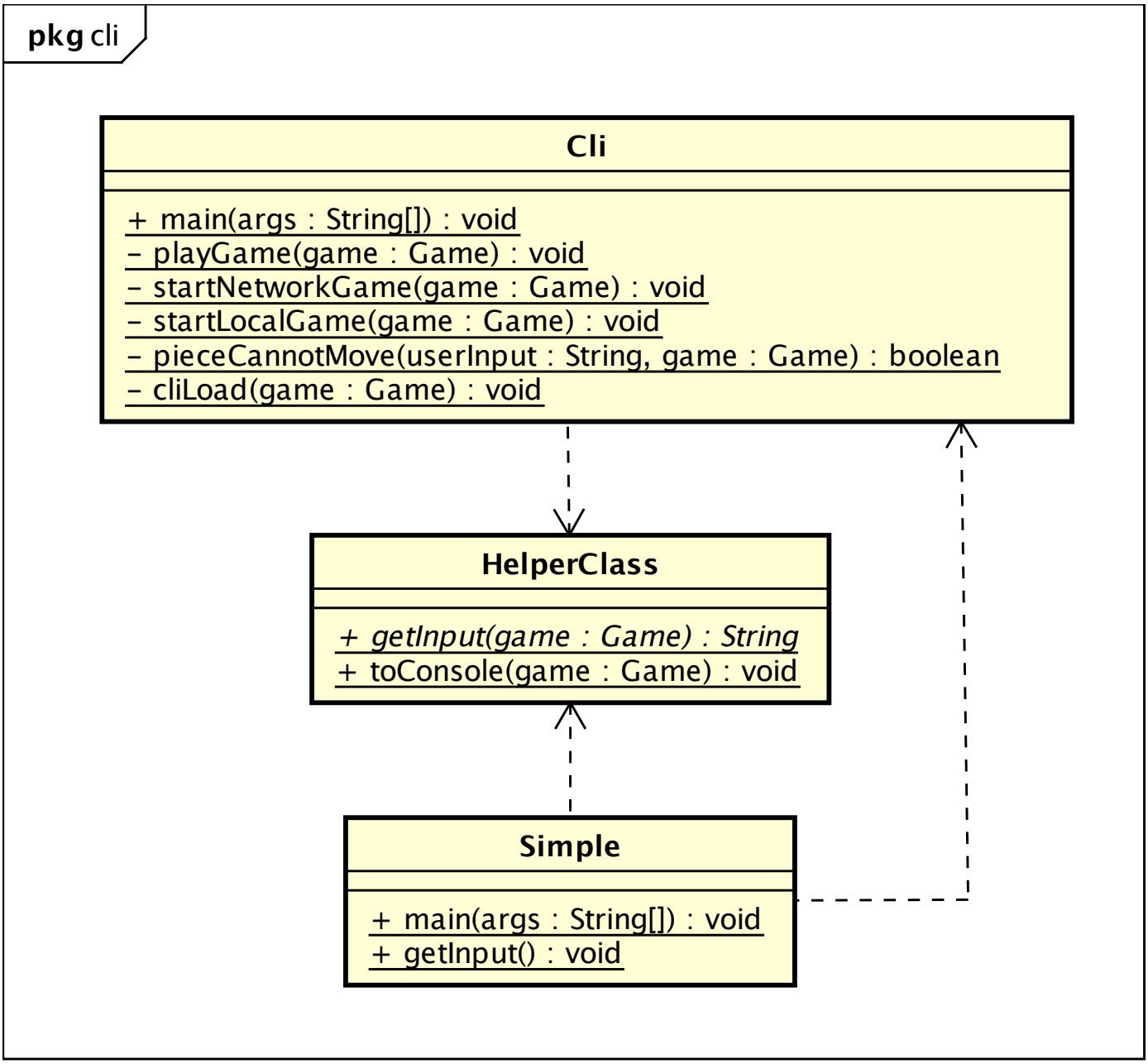
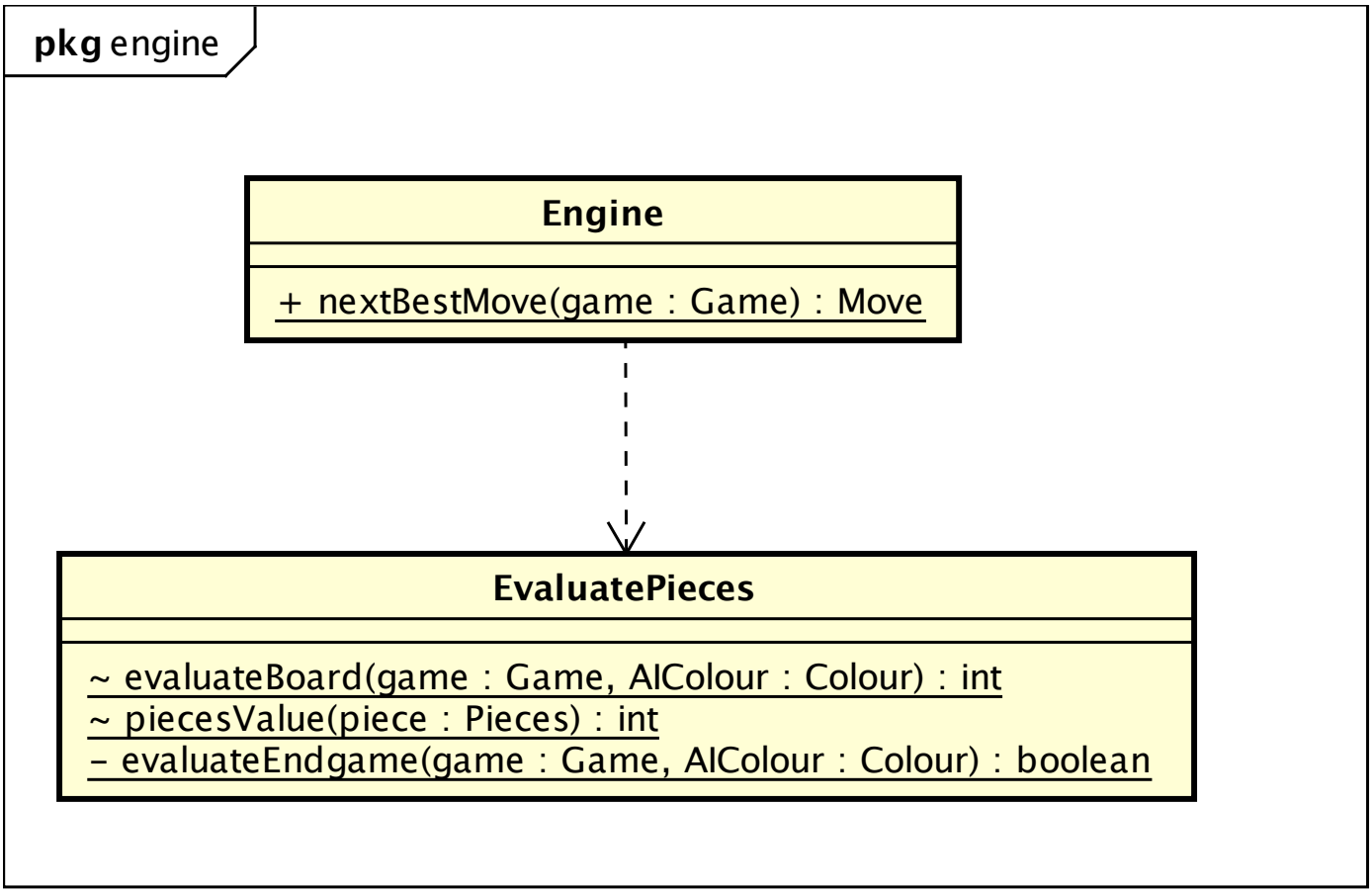
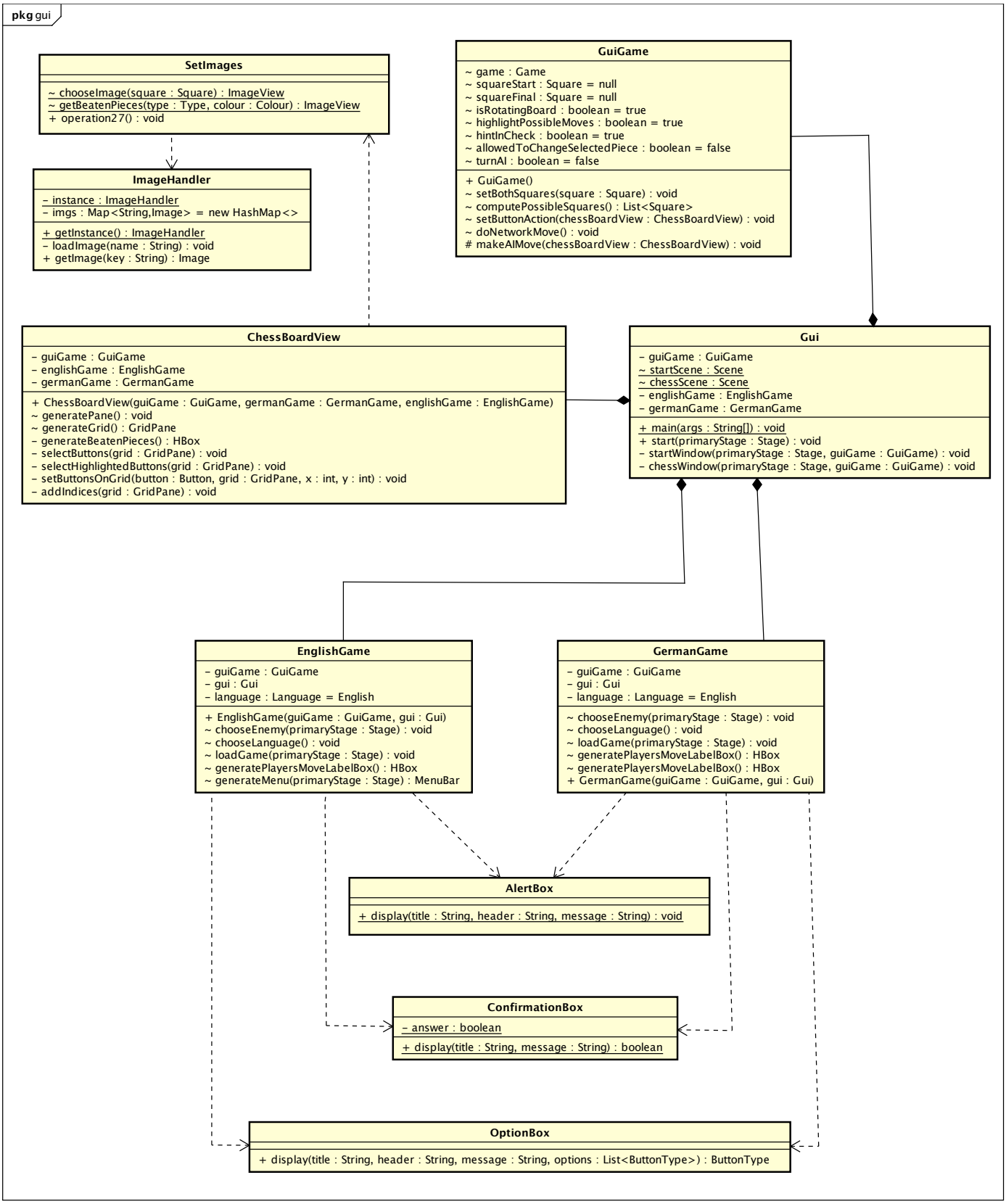
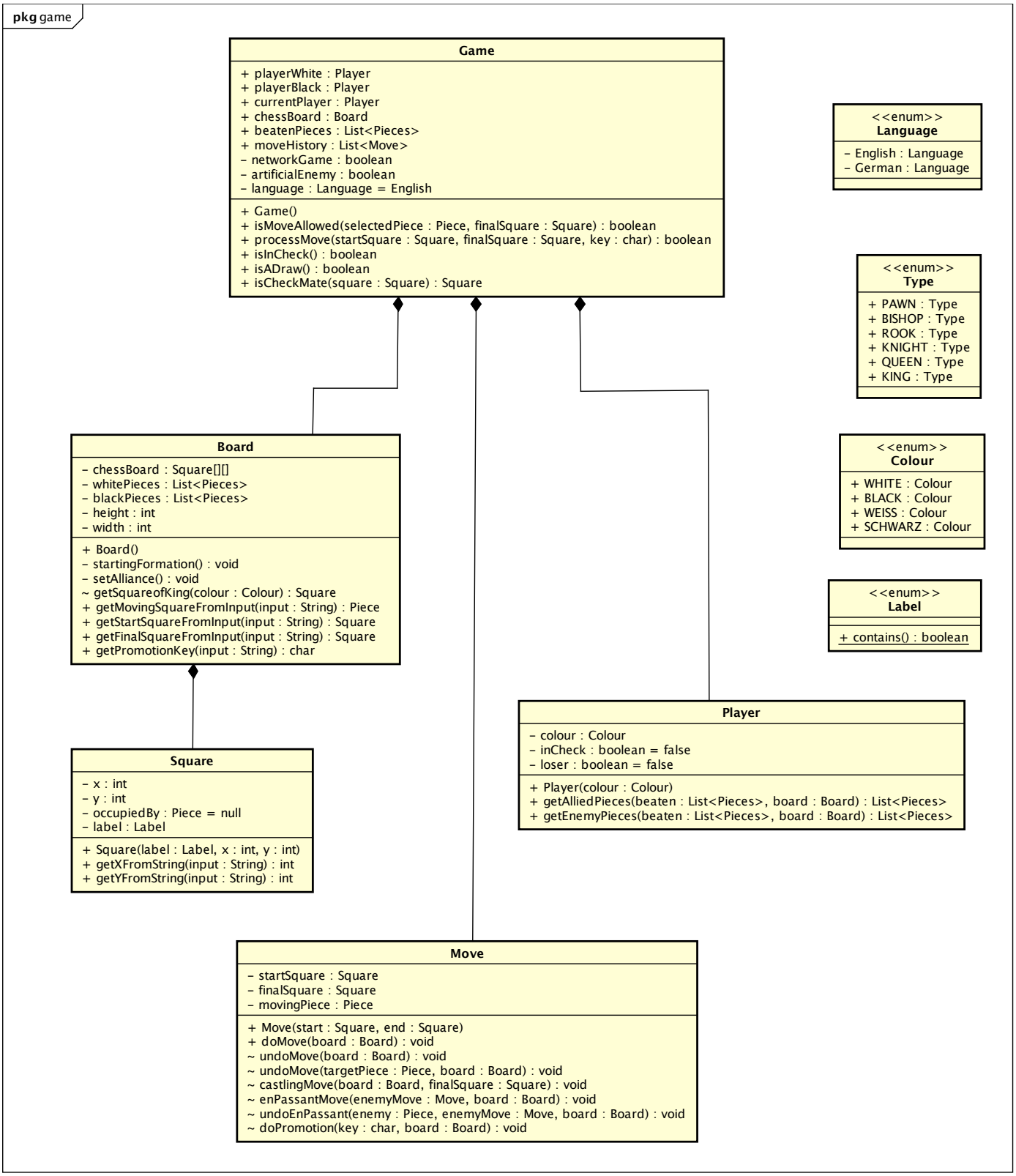
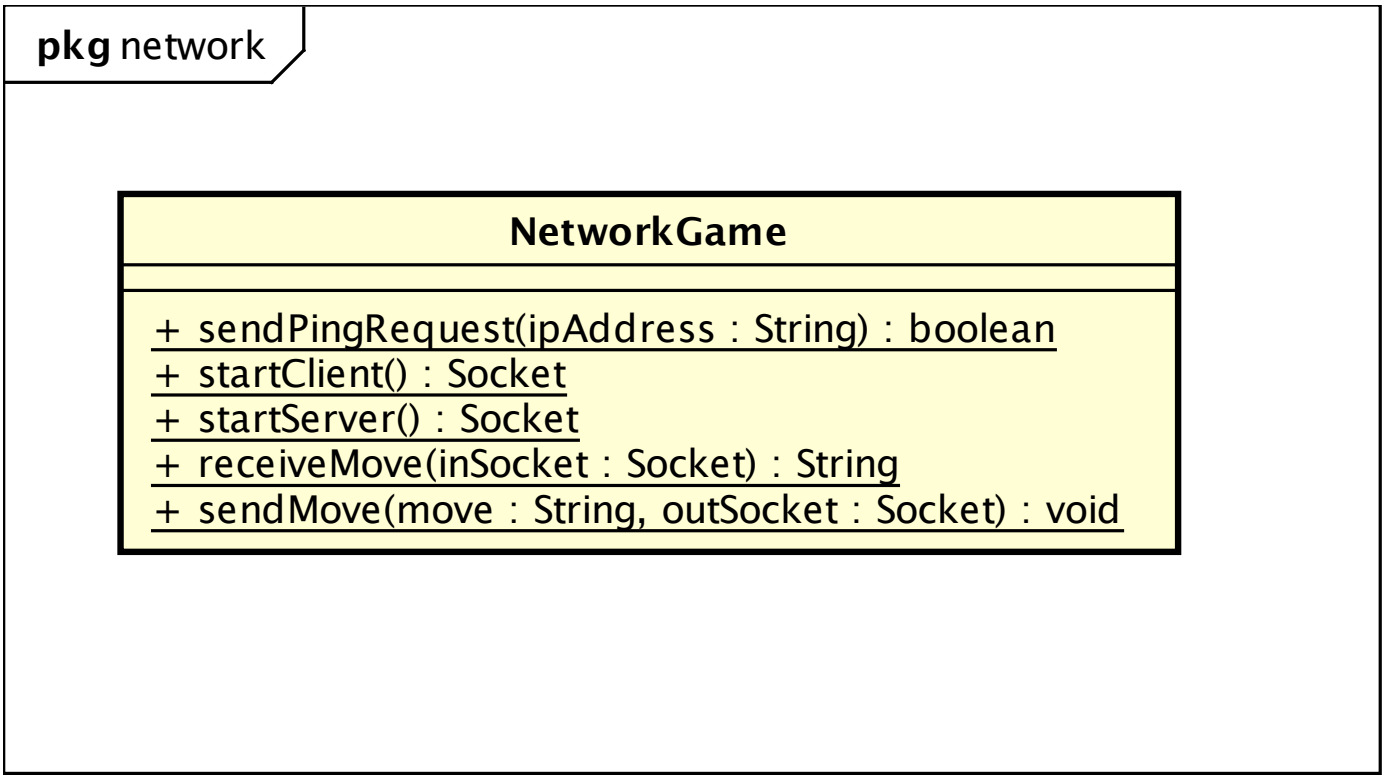

pkg chess
Main
cli
CLI
Simple
HelperClass
gui
Gui
SetImages
AlertBox
ConfirmationBox
GuiGame
ImageHandler
OptionBox
ChessBoardView
EnglishGame
GermanGame
game
Game
Language
Board
Move
Player
Colour
Square
Label
Type
pieces
Pieces
King
Rook
Bishop
Pawn
Knight
Queen
engine
Engine
EvaluatePieces
network
NetworkGame
savegame
SaveGame
LoadGame

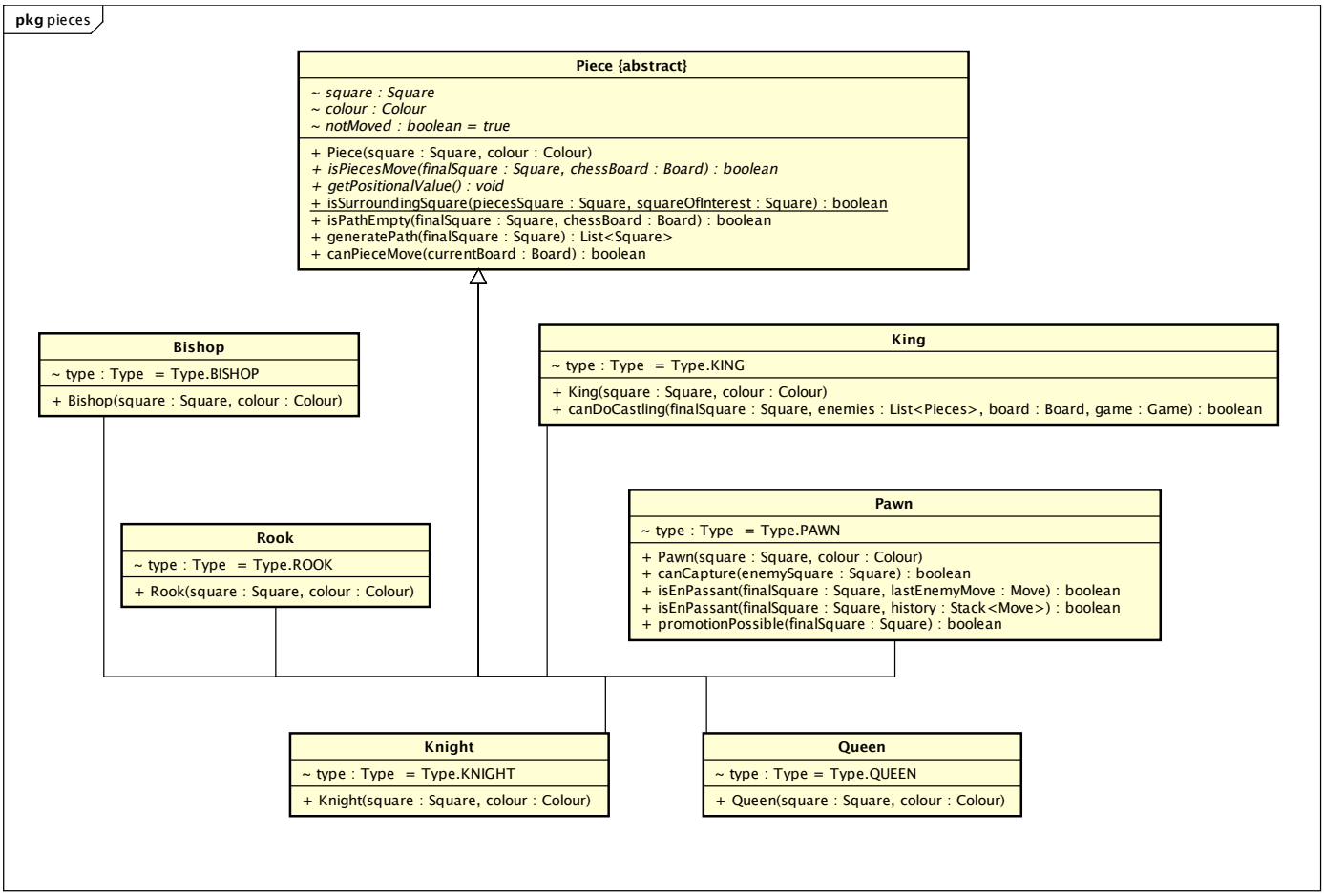

pkg cli
Cli
+ main(args : String[]) : void
- playGame(game : Game) : void
- startNetworkGame(game : Game) : void
- startLocalGame(game : Game) : void
- pieceCannotMove(userInput : String, game : Game) : boolean
- cliLoad(game : Game) : void
HelperClass
+ getInput(game : Game) : String
+ toConsole(game : Game) : void
Simple
+ main(args : String[]) : void
+ getInput() : void


pkg engine
Engine
+ nextBestMove(game : Game) : Move
EvaluatePieces
~ evaluateBoard(game : Game, AIColour : Colour) : int
~ piecesValue(piece : Pieces) : int
- evaluateEndgame(game : Game, AIColour : Colour) : boolean


pkg gui
SetImages
~ chooseImage(square : Square) : ImageView
~ getBeatenPieces(type : Type, colour : Colour) : ImageView
+ operation27() : void
ImageHandler
- instance : ImageHandler
- imgs : Map<String,Image> = new HashMap<>
+ getInstance() : ImageHandler
- loadImage(name : String) : void
+ getImage(key : String) : Image
GuiGame
~ game : Game
~ squareStart : Square = null
~ squareFinal : Square = null
~ isRotatingBoard : boolean = true
~ highlightPossibleMoves : boolean = true
~ hintInCheck : boolean = true
~ allowedToChangeSelectedPiece : boolean = false
~ turnAI : boolean = false
+ GuiGame()
~ setBothSquares(square : Square) : void
~ computePossibleSquares() : List<Square>
~ setButtonAction(chessBoardView : ChessBoardView) : void
~ doNetworkMove() : void
# makeAIMove(chessBoardView : ChessBoardView) : void
ChessBoardView
- guiGame : GuiGame
- englishGame : EnglishGame
- germanGame : GermanGame
+ ChessBoardView(guiGame : GuiGame, germanGame : GermanGame, englishGame : EnglishGame)
~ generatePane() : void
~ generateGrid() : GridPane
- generateBeatenPieces() : HBox
- selectButtons(grid : GridPane) : void
- selectHighlightedButtons(grid : GridPane) : void
- setButtonsOnGrid(button : Button, grid : GridPane, x : int, y : int) : void
- addIndices(grid : GridPane) : void
Gui
- guiGame : GuiGame
~ startScene : Scene
~ chessScene : Scene
- englishGame : EnglishGame
- germanGame : GermanGame
+ main(args : String[]) : void
+ start(primaryStage : Stage) : void
- startWindow(primaryStage : Stage, guiGame : GuiGame) : void
- chessWindow(primaryStage : Stage, guiGame : GuiGame) : void
EnglishGame
- guiGame : GuiGame
- gui : Gui
- language : Language = English
+ EnglishGame(guiGame : GuiGame, gui : Gui)
~ chooseEnemy(primaryStage : Stage) : void
~ chooseLanguage() : void
~ loadGame(primaryStage : Stage) : void
~ generatePlayersMoveLabelBox() : HBox
~ generateMenu(primaryStage : Stage) : MenuBar
GermanGame
- guiGame : GuiGame
- gui : Gui
- language : Language = English
~ chooseEnemy(primaryStage : Stage) : void
~ chooseLanguage() : void
~ loadGame(primaryStage : Stage) : void
~ generatePlayersMoveLabelBox() : HBox
~ generatePlayersMoveLabelBox() : HBox
+ GermanGame(guiGame : GuiGame, gui : Gui)
AlertBox
+ display(title : String, header : String, message : String) : void
ConfirmationBox
- answer : boolean
+ display(title : String, message : String) : boolean
OptionBox
+ display(title : String, header : String, message : String, options : List<ButtonType>) : ButtonType


pkg game
Game
+ playerWhite : Player
+ playerBlack : Player
+ currentPlayer : Player
+ chessBoard : Board
+ beatenPieces : List<Pieces>
+ moveHistory : List<Move>
- networkGame : boolean
- artificialEnemy : boolean
- language : Language = English
+ Game()
+ isMoveAllowed(selectedPiece : Piece, finalSquare : Square) : boolean
+ processMove(startSquare : Square, finalSquare : Square, key : char) : boolean
+ isInCheck() : boolean
+ isADraw() : boolean
+ isCheckMate(square : Square) : Square
<<enum>>
Language
- English : Language
- German : Language
<<enum>>
Type
+ PAWN : Type
+ BISHOP : Type
+ ROOK : Type
+ KNIGHT : Type
+ QUEEN : Type
+ KING : Type
Board
- chessBoard : Square[][]
- whitePieces : List<Pieces>
- blackPieces : List<Pieces>
- height : int
- width : int
+ Board()
- startingFormation() : void
- setAlliance() : void
~ getSquareofKing(colour : Colour) : Square
+ getMovingSquareFromInput(input : String) : Piece
+ getStartSquareFromInput(input : String) : Square
+ getFinalSquareFromInput(input : String) : Square
+ getPromotionKey(input : String) : char
<<enum>>
Colour
+ WHITE : Colour
+ BLACK : Colour
+ WEISS : Colour
+ SCHWARZ : Colour
<<enum>>
Label
+ contains() : boolean
Player
- colour : Colour
- inCheck : boolean = false
- loser : boolean = false
+ Player(colour : Colour)
+ getAlliedPieces(beaten : List<Pieces>, board : Board) : List<Pieces>
+ getEnemyPieces(beaten : List<Pieces>, board : Board) : List<Pieces>
Square
- x : int
- y : int
- occupiedBy : Piece = null
- label : Label
+ Square(label : Label, x : int, y : int)
+ getXFromString(input : String) : int
+ getYFromString(input : String) : int
Move
- startSquare : Square
- finalSquare : Square
- movingPiece : Piece
+ Move(start : Square, end : Square)
+ doMove(board : Board) : void
~ undoMove(board : Board) : void
~ undoMove(targetPiece : Piece, board : Board) : void
~ castlingMove(board : Board, finalSquare : Square) : void
~ enPassantMove(enemyMove : Move, board : Board) : void
~ undoEnPassant(enemy : Piece, enemyMove : Move, board : Board) : void
~ doPromotion(key : char, board : Board) : void


pkg network
NetworkGame
+ sendPingRequest(ipAddress : String) : boolean
+ startClient() : Socket
+ startServer() : Socket
+ receiveMove(inSocket : Socket) : String
+ sendMove(move : String, outSocket : Socket) : void


pkg pieces
Piece {abstract}
~ square : Square
~ colour : Colour
~ notMoved : boolean = true
+ Piece(square : Square, colour : Colour)
+ isPiecesMove(finalSquare : Square, chessBoard : Board) : boolean
+ getPositionalValue() : void
+ isSurroundingSquare(piecesSquare : Square, squareOfInterest : Square) : boolean
+ isPathEmpty(finalSquare : Square, chessBoard : Board) : boolean
+ generatePath(finalSquare : Square) : List<Square>
+ canPieceMove(currentBoard : Board) : boolean
Bishop
~ type : Type = Type.BISHOP
+ Bishop(square : Square, colour : Colour)
King
~ type : Type = Type.KING
+ King(square : Square, colour : Colour)
+ canDoCastling(finalSquare : Square, enemies : List<Pieces>, board : Board, game : Game) : boolean
Rook
~ type : Type = Type.ROOK
+ Rook(square : Square, colour : Colour)
Pawn
~ type : Type = Type.PAWN
+ Pawn(square : Square, colour : Colour)
+ canCapture(enemySquare : Square) : boolean
+ isEnPassant(finalSquare : Square, lastEnemyMove : Move) : boolean
+ isEnPassant(finalSquare : Square, history : Stack<Move>) : boolean
+ promotionPossible(finalSquare : Square) : boolean
Knight
~ type : Type = Type.KNIGHT
+ Knight(square : Square, colour : Colour)
Queen
~ type : Type = Type.QUEEN
+ Queen(square : Square, colour : Colour)

Architektur - Schach-Spiel

Cli

Im Cli-Paket werden alle Abfragen und Eingaben des/der Benutzer:in behandelt, um alle anderen Klassen und Methoden sowohl vom CLI als auch vom GUI nutzbar zu machen. Hier wird auch ein neues ‚Game' instanziiert.
Um die eigentliche Cli-Klasse übersichtlicher zu machen, haben wir uns entschieden eine Helfer-Klasse zu erstellen, in die einfache und mehrfach genutzte Methoden ausgelagert werden.

Game

Im Game-Paket ist die Spiellogik verankert. Dabei verwaltet die Game-Methode das gesamte Spiel und verarbeitet die Spiel-Züge. Von hier aus wird auch im Konstruktor ein neues Board mit der Höhe und Breite als Parameter instanziiert (dafür haben wir uns entschieden, da es so möglich ist, das Spielbrett für womöglich andere Spiele weiter zu verwenden und zu erweitern, da auch alle Verweise im Spiel mit dem jeweiligen Getter realisiert wurden). Außerdem wird jeweils ein weißer und schwarzer Spieler initialisiert (was an dieser Stelle auch erweiterbar ist, da für die Farben eine enumeration-Klasse „Colour" verwendet wurde, die leicht erweiterbar ist).
Die Klasse greift auch auf weitere Methoden in der Piece-Klasse zu, um zu berechnen, ob eine Figur sich auf das angegebene Square bewegen darf.

Board

Die Board-Klasse instanziiert für die vorgegebene Höhe und Breite des Schachbretts jeweils einen neuen Square mit x- und y-Koordinate, sodass am Ende in unserem Fall das Schachbrett aus 64 Squares besteht, denen alle über die eumeration-Klasse „Label" das jeweilige Label des Feldes zugewiesen wurden (z.B. a1, g5, etc.). Dies dient dazu, zu kontrollieren, ob eine Benutzer:innen-Eingabe nicht außerhalb des erlaubten Spielfelds liegt. Außerdem wird hierauf zurückgegriffen, um einen AI-Move in einen String zu übersetzen und auszugeben bzw. in der Zug-Historie des Gui-Spiels anzuzeigen.
Nachdem das Board gebaut wurde, werden von hier aus außerdem die einzelnen Figuren mit der jeweiligen Farbe und dem Square auf dem sie stehen instanziiert.

Move

Auf die Move-Klasse wir vom Spiel aus zugegriffen, um hier den Spielzug auf dem Schachbrett auszuführen. Anschließend wird in der Game-Klasse noch einmal geprüft, ob sich der/die Spieler:in selbst ins Schach gesetzt hat. Sollte dem so sein, wird von der Game-Klasse wieder auf die Move-Klasse zugegriffen, um dieser Zug dann wieder rückgängig zu machen.

Language

Die Sprache haben wir in der enumeration-Klasse „Language" hinterlegt, sodass diese auch gut um weitere Sprachen erweiterbar ist.

Piece

Für die Figuren haben wir eine abstrakte Klasse „Piece“ erstellt, von der alle Figuren erben, sodass es auch ein leichtes wäre, weitere Figuren zu erstellen.
Die Figuren wissen jeweils durch die Vererbung, auf welchem Square sie stehen und was für eine Farbe sie sind. Außerdem wird jeder Figur ein „Type“ über die gleichnamige enumeration-Klasse zugewiesen, wodurch auf die z.B. im König oder Bauern implementierten Methoden zum Castling oder enPassant zugegriffen werden kann.
Die Figuren erben die abstrakte Methode „isPiecesMove()“, die in den erbenden Klassen implementiert wird, um von der aktuellen Position und der Spielzug-Eingabe zu berechnen, ob der Zug für die jeweilige Figur erlaubt ist. Eine weitere vererbte abstrakte Methode ist „getPositionalValue()“, die für das Spiel gegen die KI von Bedeutung ist.

Engine

In der Engine-Klasse wird der beste Spielzug für die KI berechnet. Deshalb haben wir die Methode „nextBestMove()“ als static definiert, um auf diese von überall zu jeder Zeit zugreifen zu können.
In der Klasse EvaluatePieces wiederum werden die einzelnen Figuren, die auf dem Schachbrett stehen, in einer Stellungsbewertung bewertet. Zusätzlich wird auf die in der Pieces-Klasse erwähnte Methode „getPositionalValue()“ in der Figuren-Klasse zugegriffen, um eine verbesserte Bewertung für das Schachbrett zu haben.

Network

Das Netzwerk-Spiel verwaltet allein die Netzwerk-Verbindung, damit diese Klasse sowohl aus der Cli- als auch der Gui-Klasse angesteuert werden kann.
Außerdem wird hier der einkommende Zug direkt ausgeführt, da dieser vorher schon vom Gegner abgeprüft wurde und somit valide ist.

Gui-Paket

Im Gui-Paket wird das Spiel über die GuiGame-Klasse als Controller verwaltet und hier wird auch ein neues Game instanziiert, nachdem das GuiGame in der Gui-Klasse instanziiert wurde.
Der View wird über die ChessBoardView-Klasse realisiert, wo die einzelnen Squares als Buttons erstellt werden. Dabei wird bei einem Klick auf ein Square zuerst dieser gesetzt und dann auf den zweiten Klick gewartet. Erfolgt dieser, wird über die GuiGame-Klasse in der Spiellogik berechnet, ob der Zug ausführbar ist und wenn ja, durchgeführt. Anschließend wird das Spielfeld mit der neuen Figurenanordnung wieder im ChessBoardView generiert und ausgegeben.
Die verschiedenen Sprachausgaben werden hier über zwei Klassen für die zwei Sprachen Englisch und Deutsch realisiert, sodass das Spiel auch hier schnell um weitere Sprachen erweiterbar ist. Diese werden auch in der Gui-Klasse ganz zum Anfang des Spiels instanziiert.
Außerdem haben wir verschiedene Alert-Klassen definiert, auf die aus den Klassen GermanGame und EnglishGame zugegriffen wird, um für den/die Spieler:in Nachrichten generieren zu können.
Die Bilder der verschiedenen Figuren wird über die ImageHandler-Klasse geladen, worauf aus ChessBoardView über die SetImages-Klasse zugegriffen wird.