Praxis der Softwareentwicklung

Entwurf der Schach-App

Rukiye Devran, Tim Groß, Daniel Helmig, Orkhan Aliev

Inhaltsverzeichnis

1 Einleitung

bli bla blub

2 Klassendiagramme

2.1 Spiel

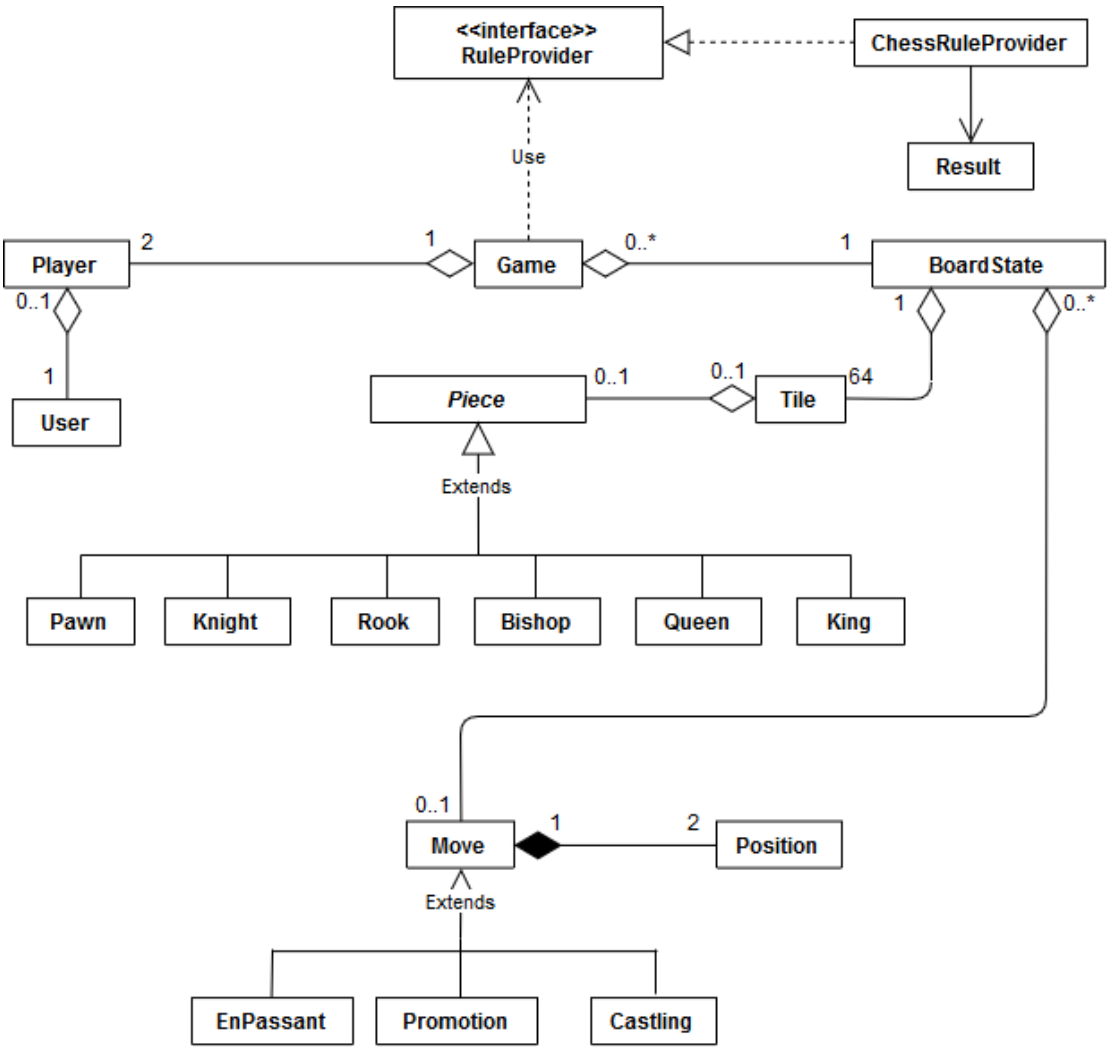

Abbildung 1: TotalGame

- ***Game***

  Game stellt ein Spiel dar. Es besteht aus zwei Spielern ***Player*** und einem Spielfeld ***BoardState***. Außerdem benutzt es eine Instanziierung des Interface ***RuleProvider***.

  In diesem Fall wird das Interface nur von ***ChessRuleProvider*** implementiert. Dieser benutzt als Rückgabewert einer Methode die Klasse ***Result***.

  Ein ***Player*** besteht aus einem ***User***. Umgekehrt muss ein ***User*** aber nicht zu jedem Zeitpunkt einen ***Player*** haben.

  BoardState enthält 64 ***Tiles*** und einen ***Move***, welcher hier den letzten ausgeführten Zug darstellt.

  Ein ***Tile*** kann ein ***Piece*** enthalten, muss es aber nicht.

  Piece ist eine abstrakte Klasse und stellt eine Figur dar. Sie wird von den sechs konkreten Figurenklassen beerbt.

  Ein ***Move*** besteht im Allgemeinen nur aus zwei ***Positionen***. Es gibt drei Spezialfälle eines Zuges, welche eine gesonderte Implementierung benötigen, da sie aus mehr als nur zwei ***Positionen*** bestehen.

  Position Stellt eine Position auf einem Schachbrett dar und besteht aus zwei Zahlen, welche die Koordinaten darstellen.

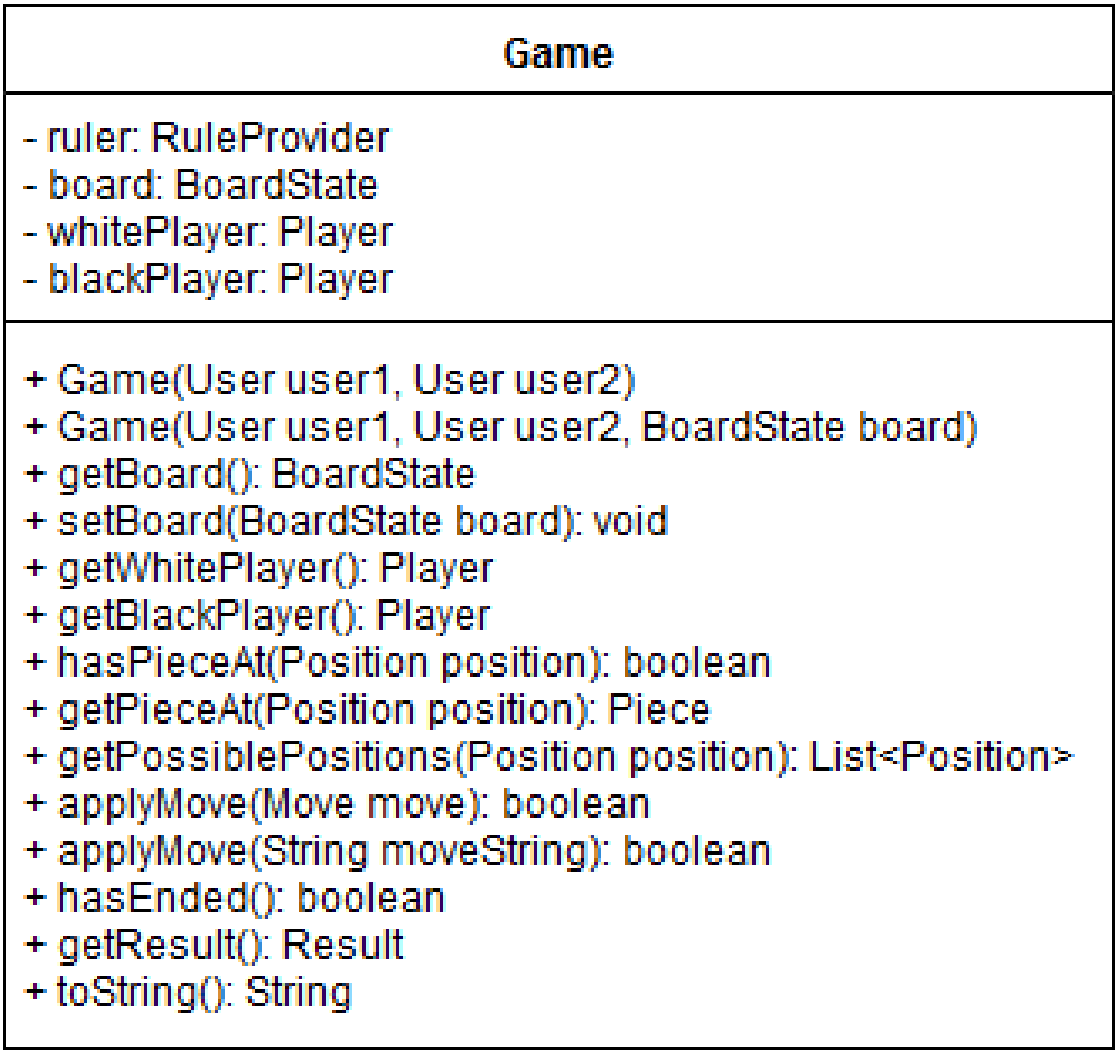

Abbildung 2: Game

- **Game**

  Stellt ein Spiel dar, verwaltet teilnehmende Spieler, Regelwerk sowie Brettzustand

  ruler: Objekt, welches die Spielregeln zur Verfügung stellt.

  board: Hier wird der aktuelle Spielstatus gespeichert.

  whitePlayer: Der Spieler mit den weißen Figuren.

  blackPlayer: Der Spieler mit den schwarzen Figuren.

Game(User user1, User user2) Konstruktor, welcher ein Spiel mit den Standard Schachregeln und einem Brett auf Anfangsposition erzeugt. Der zuerst übergebene User erhält die weißen Figuren, der zweite die schwarzen.

Game(User user1, User user2, BoardState board) Konstruktor, welcher ein Spiel mit den Standard Schachregeln und dem übergebenen Brettstatus erzeugt. Der zuerst übergebene User erhält die weißen Figuren, der zweite die schwarzen.

getBoard(): Gibt den Brettstatus des akutellen Spiels als Objekt zurück.

getBoard(BoardState board): Setzt den Brettstatus auf das übergebene Objekt.

getWhitePlayer(): Gibt den weißen Spieler zurück.

getBlackPlayer(): Gibt den schwarzen Spieler zurück.

hasPieceAt(Position position): Gibt zurück, ob sich an der übergebenen Position auf dem Brett eine Figur befindet.

getPieceAt(Position position): Gibt die Figur zurück, welche sich an der übergebenen Position befindet. Ist die Position nicht besetzt, wird null zurückgegeben.

getPossiblePositions(Position position): Gibt alle möglichen Positionen als Liste zurück, auf welche eine Figur, welche sich auf der übergebenen Position befindet, ziehen kann. Zum Berechnen dieser wird das ***ruler*** Objekt benutzt.

applyMove(Move): Führt einen übergebenen Zug auf dem Brett aus.

applyMove(String):Führt einen als String übergebenen Zug auf dem Brett aus. Dazu muss der String erst in ein Zugobjekt umgewandelt werden.

hasEnded(): Gibt zurück, ob das Spiel gemäß den Schachregeln beendet ist. Dazu wird das ***ruler*** Objekt benutzt.

getResult(): Gibt das Ergebnis eines Spiels zurück. Ist das Spiel noch nicht beendet, wird null zurückgegeben.

toString(): Wandelt den gesamten Spielstatus in eine Zeichenkette um. Dazu werden die Benutzernamen sowie die String-Repräsentation des ***board*** Objekts genutzt.

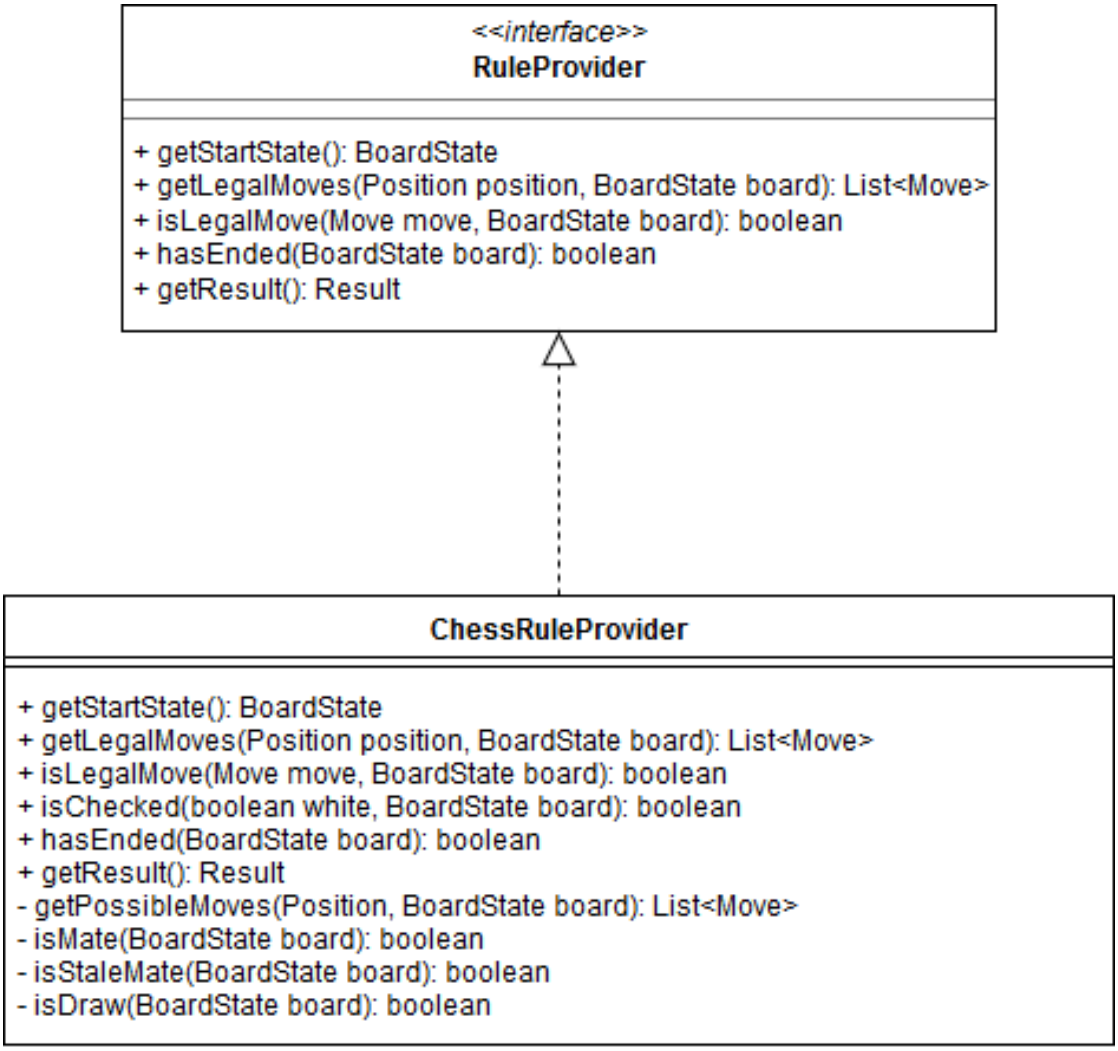

Abbildung 3: RuleProvider

- **RuleProvider**

  Interface, welches alle nötigen Regeln eines Spiels auf einem Schachbrett bereitstellt. Die zu implementierenden Methoden sind:

  getStartState(): Soll die Anfangskonfiguration eines Brettes für das jeweilige Spiel zurückgeben.

  getLegalMoves(Position position, BoardState board): Soll auf einem Brett ausgehend von einer Position die Zugmöglichkeiten der sich darauf befindenden Figur berechnen, entsprechend der implementierten Regeln.

isLegalMove(Move move, BoardState board): Soll überprüfen, ob ein Zug gemäß der jeweiligen Regeln auf einem bestimmten Brett erlaubt ist.

hasEnded(BoardState board): Soll prüfen, ob das Spiel auf einem bestimmten Brett gemäß den jeweiligen Regeln beendet ist.

getResult(BoardState board): Soll das Ergebnis eines Spiels zurückgeben. Ist das Spiel nicht beendet soll null zurückgegeben werden.

- **ChessRuleProvider**

  Konkreter Regellieferer, welcher die genauen Schachregeln zur Verfügung stellt.

  getStartState(): Gibt die Standard Anfangsstellung eines Schachspiels als ***BoardState*** zurück.

  getLegalMoves(Position position, BoardState board): Gibt eine Liste an erlaubten Zügen ausgehend von einer ausgewählten Position und einem Brett zurück. Dazu werden zunächst mit ***getPossibleMoves(Position, BoardState)*** alle möglichen Züge berechnet. Anschließen wird jeder Zug auf einer Kopie des Brettes simuliert, und mithilfe von ***isChecked(boolean, BoardState)*** überprüft, ob der selbe Spieler danach im Schach stünde (was den Zug ungültig machen würde).

  isLegalMove(Move move, BoardState board): Prüft, ob ein Zug gemäß den Schachregeln auf dem übergebenen Brett erlaubt ist.

  isChecked(boolean white, BoardState board): Überprüft ob ein Spieler auf dem gegebenen Brett im Schach steht. Ist der übergebene boolean true, wird Weiß überprüft, bei false Schwarz. Dazu wird geschaut, ob es eine gegnerische Figur gibt, welche durch ***getPosssibleMoves(Position position, BoardState board)*** einen Zug erhält, mit welchem der gegnerische König erreicht werden könnte.

  hasEnded(BoardState board): Prüft, ob das Schachspiel zu Ende ist. Dazu wird geprüft, ob der zu ziehende Spieler laut ***isMate(BoardState board)*** Matt gesetzt, laut ***isStaleMate(BoardState board)*** Patt gesetzt oder ob laut ***isDraw(BoardState board)*** andersweitig ein Unentschieden erreicht wurde.

  getResult() Ruft ***hasEnded(BoardState board)*** auf, gibt aber bei beendetem Spiel das jeweilige Ergebnis als Result mit Begründung zurück. Ist das Spiel nicht beendet wird null zurückgegeben.

getPossibleMoves(Position position, BoardState board): Gibt alle möglichen Züge einer Figur auf dem Brett zurück, ohne dabei zu berücksichtigen, ob der ziehende Spieler nach diesem Zug im Schach stehen würde. Dazu wird die jeweilige getMovement()-Methode der Figur an der angegebenen Position ausgeführt.

isMate(BoardState board): Prüft, ob der zu ziehende Spieler Matt gesetzt wurde. Das ist der Fall, wenn der Spieler laut ***isChecked(boolean, BoardState)*** im Schach steht, und es für keine Figur einen nach ***getLegalMoves(Position, BoardState)***erlaubten Zug gibt.

isStaleMate(BoardState board): Prüft, ob der zu ziehende Spieler Patt gesetzt wurde. Das ist der Fall, wenn der Spieler laut ***isChecked(boolean, BoardState)*** nicht im Schach steht, und es für keine Figur einen nach ***getLegalMoves(Position, BoardState)***erlaubten Zug gibt.

isDraw(BoardState board): Prüft, ob ein anderweitiges Unentschieden erreicht wurde. Dies ist der Fall, wenn zu wenig Figuren auf dem Brett vorhanden sind um Matt zu setzen, oder wenn 50 Züge lang keine Figur geschlagen und kein Bauer gezogen wurde.

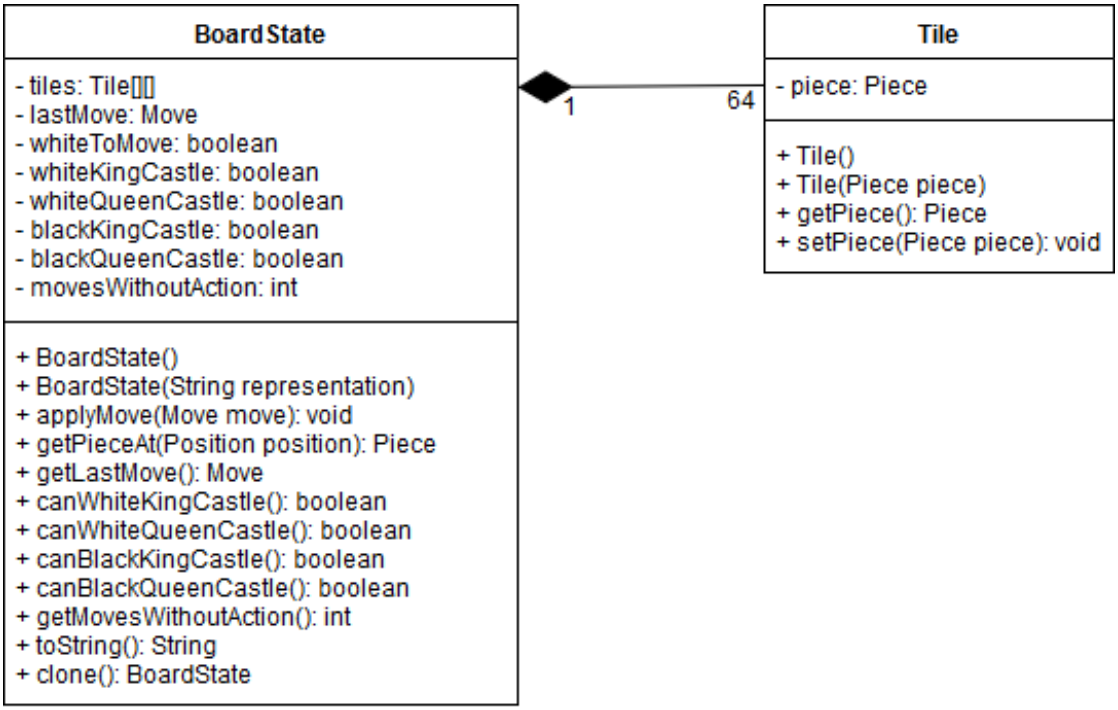

Abbildung 4: BoardState

- ***BoardState***

  Der gesamte Status eines Schachspiels der notwendig ist, um ein Spiel rekonstriuren und alle Regeln überprüfen zu können.

  tiles: Ein Array von Tiles, welches den Aufbau des Spielbretts darstellt

  lastMove: Der letzte gespielte Zug.

  whiteToMove: Speichert, ob Weiß am Zug ist.

  whiteKingCastle: Speichert ab, ob Weiß noch auf der Königsseite rochieren kann.

  whiteQueenCastle: Speichert ab, ob Weiß noch auf der Damenseite rochieren kann.

  blackKingCastle: Speichert ab, ob Schwarz noch auf der Königsseite rochieren kann.

  blackQueenCastle: Speichert ab, ob Schwarz noch auf der Damenseite rochieren

kann.

movesWithoutAction: Speichert die Anzahl der Züge in Folge, in welcher kein Bauer gezogen und keine Figur geschlagen wurde.

BaordState(): Erzeugt ein leeres Schachbrett und setzt alle Variablen auf ihre Standardwerte.

BoardState(String representation): Erzeugt ein Brett, ausgehen von einem String. In diesem müssen alle notwenidigen Informationen in einem bestimmten Format gespeichert sein.

applyMove(Move move): Führt einen Zug auf dem Brett aus, indem es die Figur(en) wie im Move-Objekt vorgegeben bewegt und möglicherweise andere Figuren schlägt(überschreibt). Außerdem wird die zu ziehende Farbe, der letzte Zug, sowie die Anzahl der Züge ohne Aktion aktualisiert. Bei entsprechender Verletzung werden möglich Rochaden auf false gesetzt.

getPieceAt(Position position): Gibt die Figur an der übergebenen Position zurück. Ist die Position nicht besetzt, wird null zurückgegeben.

getLastMove(): Gibt ***lastMove*** zurück.

canWhiteKingCastle(): Gibt zurück, ob Weiß noch auf der Königsseite rochieren kann.

canWhiteQueenCastle(): Gibt zurück, ob Weiß noch auf der Damenseite rochieren kann.

canBlackKingCastle(): Gibt zurück, ob Schwarz noch auf der Königsseite rochieren kann.

canBlackQueenCastle(): Gibt zurück, ob Schwarz noch auf der Damenseite rochieren kann.

getMovesWithoutAction): Gibt ***movesWithoutAction*** zurück.

toString() Gibt den gesamten Brettzustand als String codiert zurück. Aus diesem String muss mithilfe des entsprechenden Konstruktors ein identisches Brett erzeugt werden können.

clone(): Erzeugt und übergibt ein identisches Spielbrett, indem die String-Repräsentation des aktuellen Brettes als Parameter des Konstruktors verwendet wird.

- ***Tile***

  Stellt eine Feld eines Schachbretts dar.

  piece Die Figur, welche sich auf dem Feld befindet. Befindet sich keine Figur auf dem Feld, steht hier null.

  Tile(): Konstruktor, welcher ein leeres Feld erzeugt.

  Tile(Piece): Konstruktor, welcher ein Feld mit der übergebenen Figur darauf erzeugt.

  getPiece(): Gibt ***piece*** zurück.

  setPiece(Piece piece): Setzt ***piece*** auf die übergebene Figur.

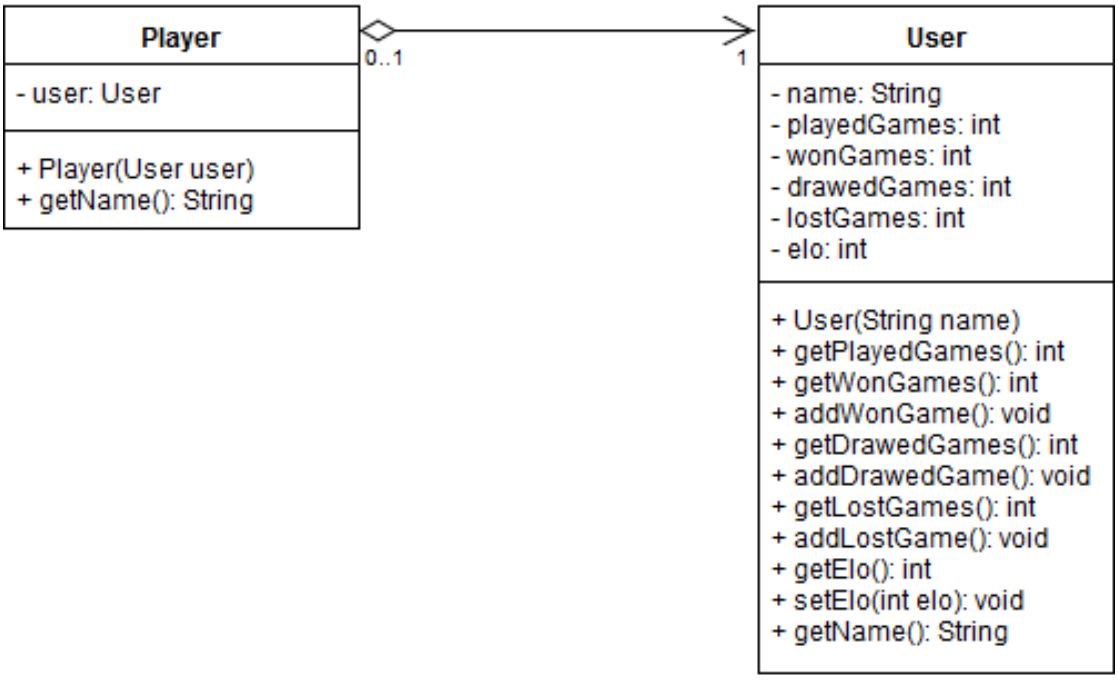

Abbildung 5: Player

- ***Player***

  Stellt einen Spieler eines konkretes Spiels dar. Existiert nur während das jeweilige Spiel existiert.

  user: Der mit diesem Spielerobjekt verknüpfte Benutzer

  Player(User user): Konstruktor, welcher ***user*** auf den übergebenen User setzt.

  getName(): Gibt den Namen des verknüften Benutzers als String zurück.

- ***User***

  name: Der eindeutige Name eines Benutzers.

  playedGames: Die Anzahl der gespielten Spiele des Benuters.

  wonGames: Die Anzahl der gewonnenen Spiele des Benuters.

  drawedGames: Die Anzahl der remisierten Spiele des Benuters.

  lostGames: Die Anzahl der verlorenen Spiele des Benuters.

elo: Der Elo-Wert des Benutzers.

User(String name): Erstellt einen neuen Benutzer mit dem übergebenen Namen. Setzt alle Attribute auf 0, ***elo*** auf 1000.

getPlayedGames(): Gibt ***playedGames*** zurück.

getWonGames(): Gibt ***wonGames*** zurück.

addWonGame(): Erhöht ***wonGames*** und ***playedGames*** um eins.

getDrawedGames(): Gibt ***drawedGames*** zurück.

addDrawedGame(): Erhöht ***drawedGames*** und ***playedGames*** um eins.

getLostGames(): Gibt ***lostGames*** zurück.

addLostGame(): Erhöht ***lostGames*** und ***playedGames*** um eins.

getElo(): Gibt ***elo*** zurück.

setElo(int Elo): Setzt die Elo des Benutzers auf den übergebenen Wert.

getName(): Gibt ***name*** zurück.

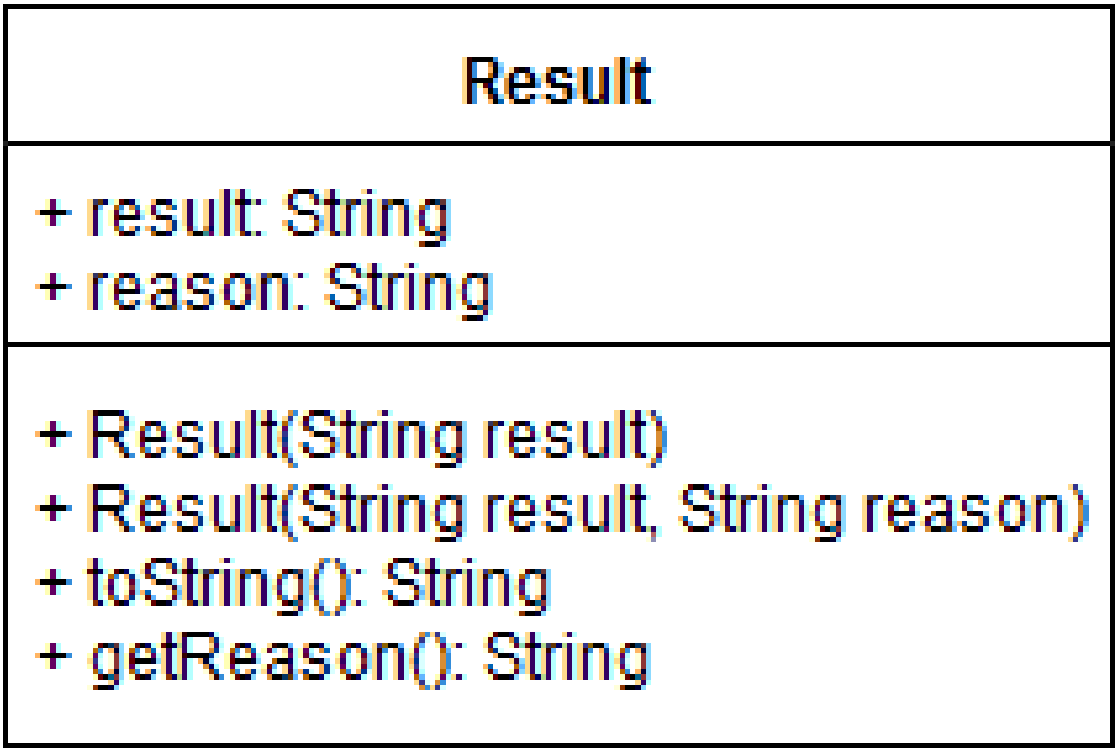

Abbildung 6: Result

- ***Result***

  Stellt ein Ergebnis eines Spiels dar.

  result: Das Ergebnis eines Spiels als String codiert.

  reason: Eine Begründung zum Ergebnis als String.

  Result(String result): Erzeugt ein Objekt mit dem übergebenen String als ***result*** und einer leeren ***reason***.

  Result(String result, String reason): Erzeugt ein Objekt mit dem ersten übergebenen String als ***result*** und dem zweiten übergebenen String als ***reason***.

  toString(): Gibt ***result*** zurück.

  getReson(): Gibt ***reason*** zurück.

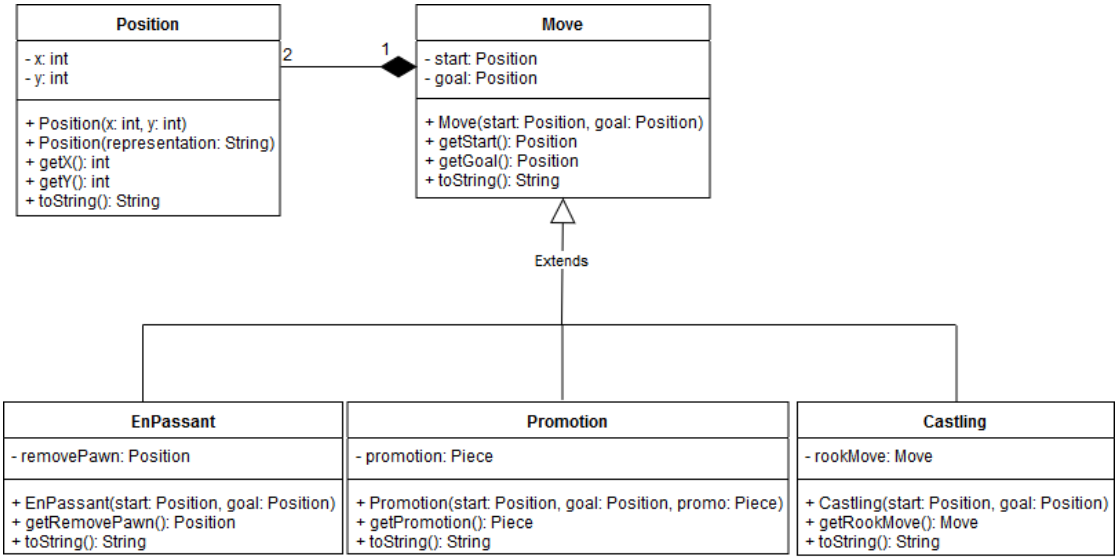

Abbildung 7: Move

- ***Move***

  Stellt einen Zug dar, ohne über die zu ziehende Figur bescheid zu wissen.

  start: Position, von welcher aus der Zug ausgeführt wird.

  goal: Position, zu welcher der Zug hinführt.

  Move(start: Position, goal: Position): Erzeugt einen Zug mit der übergebenen Start- und Zielposition.

  getStart(): Gibt ***start*** zurück.

  getGoal(): Gibt ***goal*** zurück.

  toString(): Gibt eine eindeutige Representation des Zugs als Zeichenkette zurück.

- ***Position***

  Stellt eine Position auf einem Schachbrett dar.

  x: Stellt die horizontale Koordinate auf dem Schachbrett dar.,

  y: Stellt die vertikale Koordinate auf dem Schachbrett dar.

  Position(x: int, y: int): Erzeugt eine Position mit den angegebenen Koordina-

ten. Liegen die Koordinaten außerhalb eines Schachbretts wird eine Exception ausgelöst.

Position(representation: String): Erzeugt eine Position indem der übergebene String in Koordinaten umgewandelt wird. Liegen die Koordinaten außerhalb eines Schachbretts wird eine Exception ausgelöst.

getX(): Gibt x zurück.

getY(): Gibt y zurück.

toString(): Gibt eine Repräsentation der Position als String zurück.

- ***EnPassant***

  Stellt den Bauernzug en passant dar.

  removePawn: Die Position des Bauern, der mit diesem Zug geschlagen wird.

  EnPassant(start: Position, goal: Position): Ruft den Konstruktor von ***Move*** auf, berechnet und setzt die Position***removePawn***.

  getRemovePawn(): Gibt ***removePawn*** zurück.

  toString(): Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

- ***Promotion***

  Stellt eine Bauernumwandlung dar.

  promotion: Die Figur, in welche sich der Bauer verwandeln soll.

  Promotion(start: Position, goal: Position, promo: Piece): Ruft den Konstruktor von ***Move*** auf und setzt ***promotion*** auf die übergebene Figur.

  getPromotion(): Gibt ***promotion*** zurück.

  toString(): Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

- ***Castling***

  Stellt eine Rochade dar.

rookMove: Die Bewegung des Turms, welche zusätzlich zu der des Königs mit diesem Zug ausgeführt wird.

Castling(start: Position, goal: Position): Ruft den Konstruktor von ***Move*** auf, berechnet und setzt den Zug***rookMove***.

getRookMove(): Gibt ***rookMove*** zurück.

toString(): Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

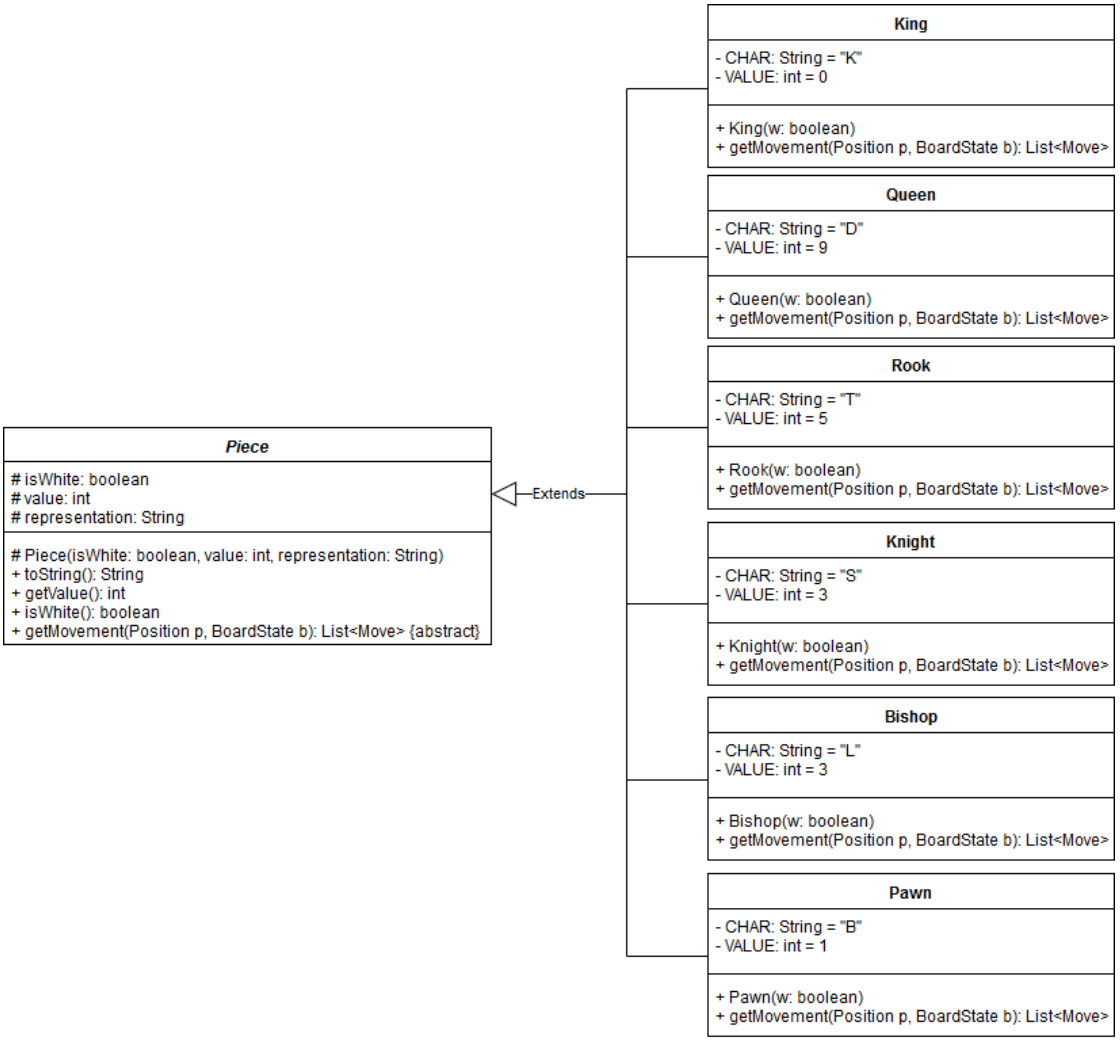

Abbildung 8: Pieces

- ***Piece***

  Stellt das abstrakte Konzept einer Schachfigur dar.

  Attribute

  ---

  isWhite: Speichert, ob die Figur weiß ist.

  value: Speichert den Wert einer Figur ab.

  representation: Speichert das repräsentative Zeichen einer Figur ab.

Methoden

Piece(isWhite: boolean, value: int, representation: String: Erzeugt eine neue Figur, setzt alle Attribute auf die jeweiligen übergebenen Werte.

toString(): Gibt ***representation*** zurück.

getValue(): Gibt ***value*** zurück.

isWhite(): Gibt ***isWhite*** zurück.

- ***King*** Stellt einen König dar.

  Attribute

  CHAR: Die Repräsentation eines Königs, wird auf K gesetzt.

  VALUE: Der Wert eines Königs, wird auf 0 gesetzt, da dem König an sich kein Wert zugewiesen werden kann.

  Methoden

  King(w: boolean): Erzeugt einen neuen König. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

  getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die ein König auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann, inklusive Rochade. Dabei wird nicht berücksichtigt, ob sich der König nach dem Zug in Schach befinden würde.

- ***Queen***

  Stellt eine Dame dar.

  Attribute

  CHAR: Die Repräsentation einer Dame, wird auf D gesetzt.

  VALUE: Der Wert einer Dame, wird auf 9 gesetzt.

Methoden

Queen(w: boolean): Erzeugt eine neue Dame. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die eine Dame auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann. Dabei wird nicht berücksichtigt, ob sich der König des ziehenden Spielers nach dem Zug in Schach befinden würde.

- ***Rook***

Stellt einen Turm dar.

Attribute

CHAR: Die Repräsentation eines Turms, wird auf T gesetzt.

VALUE: Der Wert eines Turms, wird auf 5 gesetzt.

Methoden

Rook(w: boolean): Erzeugt einen neuen Turm. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die ein Turm auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann. Dabei wird nicht berücksichtigt, ob sich der König des ziehenden Spielers nach dem Zug in Schach befinden würde.

- ***Bishop***

Stellt einen Läufer dar.

Attribute

CHAR: Die Repräsentation eines Läufers, wird auf L gesetzt.

VALUE: Der Wert eines Läufers, wird auf 3 gesetzt.

Methoden

Bishop(w: boolean): Erzeugt einen neuen Läufer. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die ein Läufer auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann. Dabei wird nicht berücksichtigt, ob sich der König des ziehenden Spielers nach dem Zug in Schach befinden würde.

- ***Knight***

Stellt einen Springer dar.

Attribute

CHAR: Die Repräsentation eines Springers, wird auf S gesetzt.

VALUE: Der Wert eines Springers, wird auf 3 gesetzt.

Methoden

Knight(w: boolean): Erzeugt einen neuen Springer. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die ein Springer auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann. Dabei wird nicht berücksichtigt, ob sich der König des ziehenden Spielers nach dem Zug in Schach befinden würde.

- ***Pawn***

Stellt einen Bauern dar.

Attribute

CHAR: Die Repräsentation eines Bauerns, wird auf B gesetzt.

VALUE: Der Wert eines Bauerns, wird auf 1 gesetzt.

Methoden

Pawn(w: boolean): Erzeugt einen neuen Bauern. Dazu wird der Konstruktor von ***Piece*** aufgerufen, als Parameter werden w, ***VALUE***, und ***CHAR*** übergeben.

getMovement(Position p, BoardState b): Gibt eine Liste der Züge zurück, die ein Bauer auf dem gegebenen Schachbrett von der gegebenen Position aus ausführen kann, inklusive en passant, Doppelsprung und Umwandlung. Dabei wird nicht berücksichtigt, ob sich der König des ziehenden Spielers nach dem Zug in Schach befinden würde.

2.2 GUI

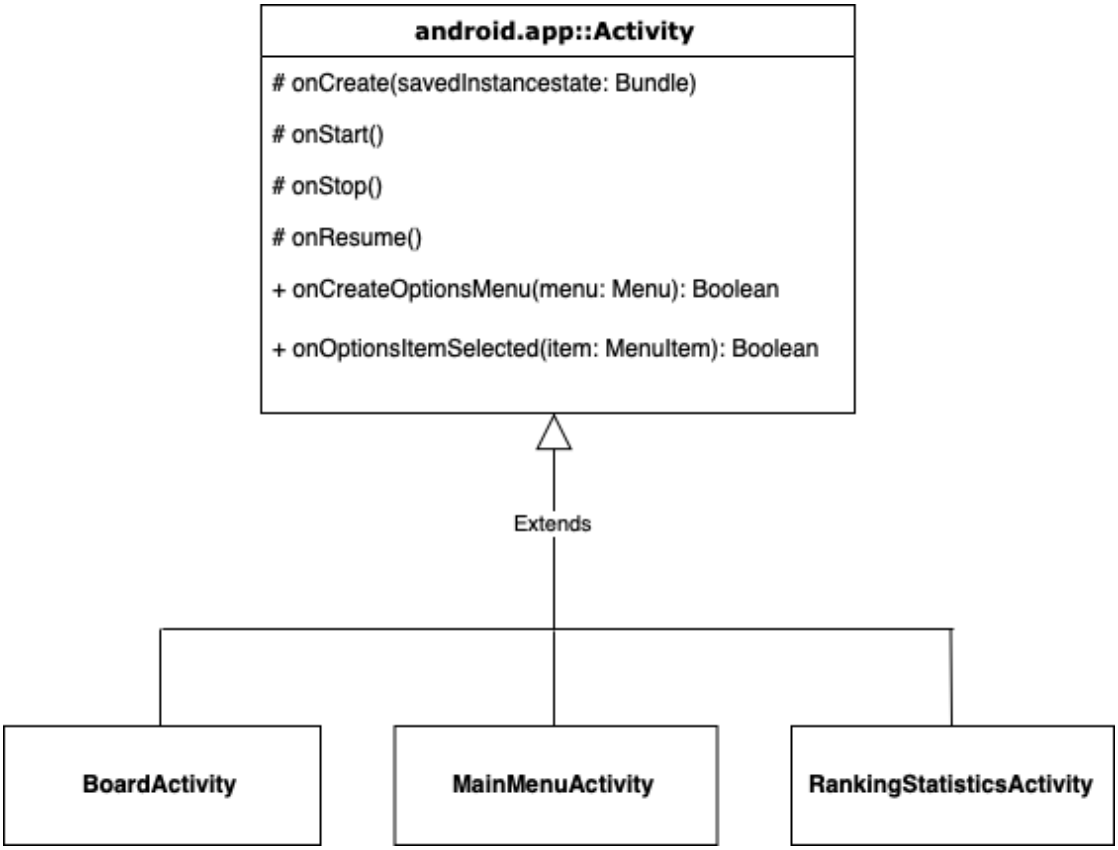

Abbildung 9: Activities

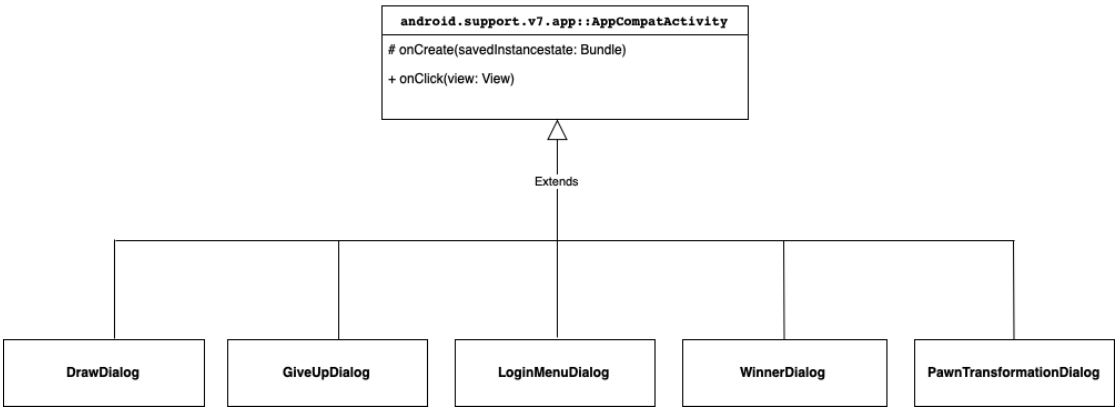

Abbildung 10: Dialoge

2.3 Server

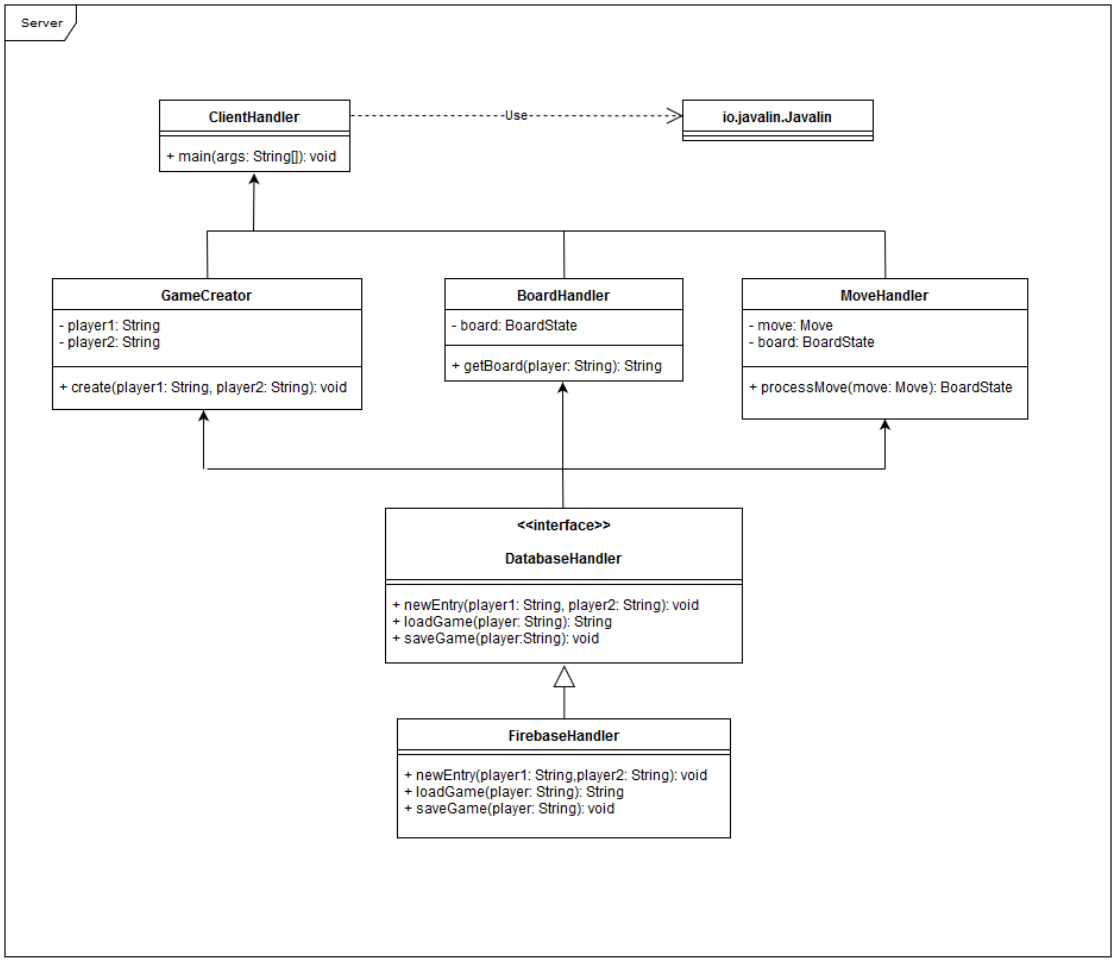

Abbildung 11: Server

- **ClientHandler**

  Verwaltet eingehende Anfragen an den Server.

  Methoden

  ---

  main(String[] args): Die main Methode des Servers. Hier wird der Javalin Server gestartet. Je nach Anfrage wird ein GameCreator, ein BoardHandler oder ein MoveHandler erstellt.

- **GameCreator**

Attribute

---

player1: String: Spieler 1 des neuen Spiels.

player2: String: Spieler 2 des neuen Spiels.

Methoden

---

create(player1: String, player2: String): Erstellt einen neuen Spiel-Eintrag in der Datenbank. Benutzt dazu einen DatabaseHandler.

- **BoardHandler**

  Attribute

  ---

  board: BoardState Aktueller Zustand des Bretts.

  Methoden

  ---

  getBoard(player: String): String Lädt das entsprechende Spielbrett aus der Datenbank. Verwendet ebenfalls einen DatabaseHandler.

- **MoveHandler**

  Attribute

  ---

  move: Move Der Zug der überprüft und ausgeführt wird.

  board: BoardState Das Spielbrett auf dem dieser Zug ausgeführt werden soll.

  Methoden

  ---

  processMove(move: Move): BoardState Laden und Speichern erfolgen wieder über einen DatabaseHandler. Überprüft ob die Kombination aus Brett und Zug gültig ist. Wendet den Zug an, falls dieser gütlig ist. Speichert das neue Brett ab und benachrichtigt anschließend den Gegner.

- **DatabaseHandler**

Legt die Methoden fest, die ein DatabaseHandler implementieren muss.

Methoden

---

newEntry(player1: String, player2: String):void Soll einen neuen Datenbankeintrag mit Spieler 1 , Spieler 2 und einem neuen Brett anlegen.

loadGame(player: String): String Soll das zum Spieler zugehörige Brett aus der Datenbank laden.

saveGame(player: String): void Soll ein Brett wieder in die Datenbank speichern.

- **FirebaseHandler**

Verwaltet die Verbindung zur Firebase Datenbank.

Methoden

---

newEntry(player1: String, player2: String):void Erstellt einen neuen Eintrag mit beiden Spielern und einem neuen Brett.

loadGame(player: String): String Lädt das zum Spieler zugehörige Brett aus der Datenbank.

saveGame(player: String): void Speichert das Brett wieder ab.

3 Sequenzdiagramm