Praxis der Softwareentwicklung

# Entwurf der Schach-App

Rukiye Devran, Tim Groß, Daniel Helmig, Orkhan Aliev

# Inhaltsverzeichnis

1	Ent	vurfsentscheidungen	3
	1.1	Paketdiagramm	3
	1.2	GUI	3
	1.3	Server	3
	1.4	Spiellogik	4
2	Klas	sendiagramme	5
	2.1	Spiel	5
		2.1.1 Game	8
		2.1.2 RuleProvider	11
		2.1.3 BoardState	15
		2.1.4 Player	19
		2.1.5 Result	22
		2.1.6 Move	24
		2.1.7 Pieces	28
	2.2	GUI	34
		2.2.1 Activities	34
		2.2.2 Dialoge	38
	2.3	ClientSocket	41
	2.4	Server	43
3	Akti	vitätsdiagramm	48
4	Seg	uenzdiagramm	49

# 1 Entwurfsentscheidungen

## 1.1 Paketdiagramm

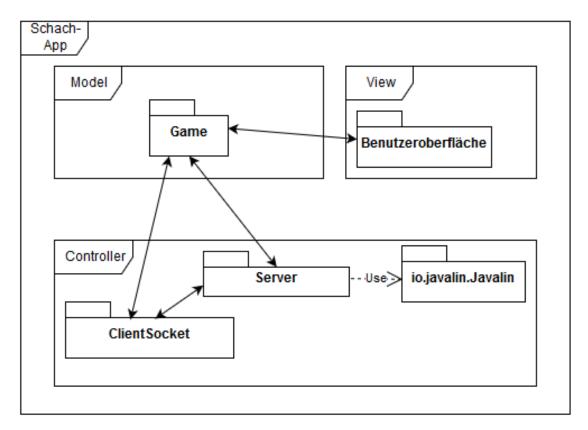


Abbildung 1: Paketdiagramm

#### 1.2 **GUI**

#### 1.3 Server

Der Server wurde nach dem REST Prinzip entworfen. Alle Anfragen werden per URL gesendet. Benachrichtigungen über neue Züge erfolgen über eine WebSocket-Verbindung. Da Züge abwechselnd und meist nicht direkt nacheinander ausgeführt werden, wurde sich gegen eine konstante Verbindung zum Zugaustausch entschieden.

## 1.4 Spiellogik

Die Spiellogik wird komplett selbst implementiert ohne externe Bibliotheken zu benutzen. Zur Spielmodellierung werden verschiedene Konzepte als Klassen implementiert. Die Spiellogik muss zum einen über einen Zug/Brettverwalter mit der Benutzeroberfläche kommunizieren können, um das Spiel in Echtzeit visualisieren zu können, als auch dem Server zur Verfügung stehen, damit dieser die Spielregeln überwachen und über den Spielzustand Bescheid wissen kann.

# 2 Klassendiagramme

# 2.1 Spiel

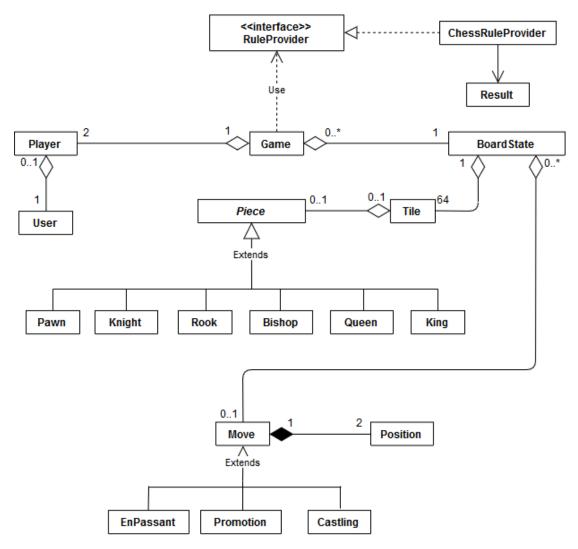


Abbildung 2: TotalGame

## Spiel

- Game stellt ein Spiel dar. Es besteht aus zwei Spielern Player und einem Spielfeld BoardState. Außerdem benutzt es eine Instanziierung des Interface RuleProvider, welches die Regeln für ein beliebiges Spiel auf einem Schachbrett liefert. Die Regeln samt Startkonfiguration des Brettes auf ein Interface auszulagern anstatt sie ins Game direkt einzubauen macht die Spiellogik leichter austauschbar und ermöglicht zum Beispiel das Erstellen weiterer Spielvarianten. Diese könnten dann einfach in dem Konsturktor der Game Klasse mitgegeben werden. Auch könnte man so beispielsweise einen Spieler mit einem einstellbaren Figurendefizit in das Spiel startet lassen, indem man einen RuleProvider mit entsprechender Anfangsstellung einfügt.
- In unserem Fall wird das Interface nur von der Klasse **ChessRuleProvider** implementiert, welche die Standard-Schachregeln liefert. Diese Klasse hat bei einer Methode als Rückgabewert ein Objekt vom Typ **Result**. Ergebnisse als eigenes Objekt zu behandeln anstatt diese einfach nur als String anzugeben hat den Vorteil, dass zum Ergebnis an sich auch eine Begründung mitgeliefert werden kann, welche der Spieler nach Spielende einsehen kann. Außerdem ist ein Ergebnis so auch immer als ein solches erkennbar.
- Ein **Player** ist ein Spieler eines konkretes Spiels und besteht aus einem **User**, welcher einen Benutzer im Allgemeinen darstellt. Umgekehrt muss ein **User** aber nicht zu jedem Zeitpunkt einen **Player** haben. Das Trennen der Konzepte Spieler und Benutzer ermöglicht das Hinzufügen von Spielern zu bestimmten **Game**-Objekten, ohne Benutzer direkt an ein Spiel koppeln zu müssen.
- Das Spielfeld **BoardState** enthält 64 **Tiles** (Schachfelder) und einen **Move**, welcher hier den letzten ausgeführten Zug des Spiels darstellt. Das Speichern des zuletzt ausgeführten Zuges ist für den Zug "en passant" sowie zur Überprüfung gespielter Züge bei Übertragung des Brettes notwendig.
- Ein **Tile** kann ein **Piece** enthalten, muss es aber nicht. Das Brett als Array von **Tiles** mit Figuren zu speichern ist notwendig, um zu jedem Zeitpunkt jedes Feld zur Hand zu haben und auf dieses Figuren setzen und entfernen zu können. Leere Felder können einfach erkannt werden, nämlich wenn sie kein Objekt vom Typ Piece enthalten.
- Piece ist eine abstrakte Klasse und stellt eine Figur dar. Sie wird von den sechs konkreten Figurenklassen beerbt. Diese speichern Informationen über ihren Wert, ihre Darstellung als Zeichen sowie ihre Zugfähigkeiten. Dadurch werden switch-case Abfragen vermieden, und von jeder Figur kann direkt der gewünschte Wert/das Verhalten angefordert werden, ohne über den Typ Bescheid wissen zu müssen.

Ein Move besteht im Allgemeinen nur aus zwei Positionen. Ein Zug muss über die ziehende Figur nicht Bescheid wissen, da zur Ausführung des Zuges ohnehin das gesamte Brett mit allen Positionen aller Figuren vorliegen muss. Es gibt drei Spezialfälle eines Zuges, welche eine gesonderte Implementierung benötigen, da sie neben zwei Positionen noch andere Informationen enthalten müssen. Diese erben alle von Move, da diese trotzdem aus einer Start- und Zielposition bestehen. Das macht das Erstellen und Weitergeben von Listen von Zügen einfacher, und ermöglicht eine einfache Ausführung auf der Benutzeroberfläche.

Position stellt eine Position auf einem Schachbrett dar und besteht aus zwei Zahlen, welche die Koordinaten darstellen. Diese müssen zwischen 0-7 liegen, was beim Erstellen eines solchen Objektes sichergestellt wird. Das hat den Vorteil, dass Positionen nicht bei jeder Nutzung auf Korrektheit überprüft werden müssen um Programmabstürze zu vermeiden.

#### 2.1.1 Game

## Game

ruler: RuleProvider
 board: BoardState
 whitePlayer: Player
 blackPlayer: Player

- + Game(user1: User, user2: User)
- + Game(user1: User, user2: User, board: BoardState)
- + getBoard(): BoardState
- + setBoard(board: BoardState): void
- + getWhitePlayer(): Player
- + getBlackPlayer(): Player
- + hasPieceAt(position: Position): boolean
- + getPieceAt(position: Position): Piece
- + getPossibleMoves(position: Position): List<Move>
- + applyMove(move: Move): boolean
- + applyMove(moveString: String): boolean
- + hasEnded(): boolean
- + getResult(): Result
- + toString(): String

Abbildung 3: Game

#### Game:

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

#### Attribute

#### - ruler: RuleProvider

Objekt, welches das Interface RuleProvider implementiert und die Spielregeln zur Verfügung stellt.

#### - board: BoardState

Hier wird der aktuelle Spielstatus bestehend aus

- Stand aller Figuren auf dem Brett
- mögliche Rochaden
- letzter ausgeführter Spielzug
- Anzahl der Züge ohne Bauernzug oder Schlagen von Figuren

gespeichert.

#### - whitePlayer: Player

Der Spieler mit den weißen Figuren.

#### - blackPlayer: Player

Der Spieler mit den schwarzen Figuren.

#### Methoden

#### + Game(user1: User ,user2: User)

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(user1: User ,user2: User, board: BoardState)

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(): BoardState

Gibt den Brettstatus boarddes akutellen Spiels zurück.

#### + getBoard(board: BoardState): void

Setzt den Brettstatus board auf das übergebene Objekt.

#### + getWhitePlayer(): Player

Gibt den weißen Spieler whitePlayer zurück.

#### + getBlackPlayer(): Player

Gibt den schwarzen Spieler blackPlayer zurück.

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

#### + getPieceAt(position: Position): Piece

Gibt die Figur zurück, welche sich an der übergebZüge Position befindet. Ist die Position nicht dibesetz wird null zurückgegeben. ausführ item + getPossiblePositions(position: Position): List<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: Move): void

Führt den übergebenen Zug auf dem Spielbrett board aus.

#### + applyMove(moveString: String): void

Führt den als String übergebenen Zug auf dem Spielbrett **board** aus. Dazu muss der String erst in ein Objekt vom Typ **Move** umgewandelt werden.

#### + hasEnded(): boolean

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

#### + getResult(): Result

Gibt das Ergebnis eines Spiels als Objekt vom Typ **Result** zurück. Ist das Spiel noch nicht beendet, wird **null** zurückgegeben.

#### + toString(): String

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

#### 2.1.2 RuleProvider

# <<interface>> RuleProvider

- getStartState(): BoardState
- + getLegalMoves(position: Position, board: BoardState): List<Move>
- + isLegalMove(move: Move, board: BoardState): boolean
- + hasEnded(board: BoardState): boolean
- + getResult(): Result

# ChessRuleProvider

- + getStartState(): BoardState
- + getLegalMoves(position: Position, board: BoardState): List<Move>
- + isLegalMove(move: Move, board: BoardState): boolean
- + isChecked(white: boolean, board: BoardState): boolean
- + hasEnded(board: BoardState): boolean
- + getResult(): Result
- getPossibleMoves(position: Position, board: BoardState): List<Move>
- isMate(board: BoardState): boolean
- isStaleMate(board: BoardState): boolean
- isDraw(board: BoardState): boolean

Abbildung 4: RuleProvider

## • RuleProvider

Interface, welches alle nötigen Regeln eines Spiels auf einem Schachbrett bereitstellt.

#### Methoden

#### + getStartState(): BoardState

Soll die Anfangskonfiguration eines Brettes für das jeweilige Spiel zurückgeben.

- + getLegalMoves(position: Position, board: BoardState): List<Move>
  Soll auf einem Brett ausgehend von einer Position die Zugmöglichkeiten der sich darauf befindenden Figur berechnen, entsprechend der implementierten Regeln.
- + isLegalMove(move: Move, board: BoardState): boolean

Soll überprüfen, ob ein Zug gemäß der jeweiligen Regeln auf einem bestimmten Brett erlaubt ist.

#### + hasEnded(board: BoardState): boolean

Soll prüfen, ob das Spiel auf einem bestimmten Brett gemäß den jeweiligen Regeln beendet ist.

+ getResult(board: BoardState): Result

Soll das Ergebnis eines Spiels als **Result** zurückgeben. Ist das Spiel nicht beendet soll **null** zurückgegeben werden.

#### • ChessRuleProvider

Konkreter Regellieferer, welcher die genauen Schachregeln zur Verfügung stellt.

#### Methoden

#### + getStartState(): BoardState

Gibt die Standard Anfangsstellung eines Schachspiels als BoardState zurück.

- + getLegalMoves(position: Position, board: BoardState): List<Move>
  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, board: BoardState): boolean

Prüft, ob ein Zug gemäß den Schachregeln auf dem übergebenen Brett erlaubt ist

#### + isChecked(white: boolean, board: BoardState): boolean

Uberprüft 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, BoardState)** einen Zug erhält, mit welchem der eigene König erreicht werden könnte.

#### + hasEnded(board: BoardState): boolean

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

#### + getResult(): Result

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

- getPossibleMoves(position: Position, board: BoardState): List<Move>
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(board: BoardState): boolean

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(board: BoardState): boolean

Prüft, ob der zu ziehende Spieler Patt gesetzt wurde. Das ist der Fall, wenn der Spieler laut is Checked (boolean, Board State) nicht im Schach steht, und es für keine Figur einen nach get Legal Moves (Position, Board State) erlaubten Zug gibt.

#### - isDraw(board: BoardState): boolean

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, also wenn der **movesWithoutAction**-Wert des **board**-Objekts 50 (oder größer)

#### 2.1.3 BoardState

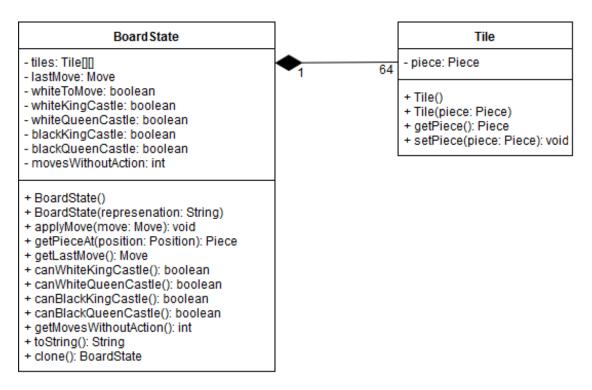


Abbildung 5: BoardState

#### • BoardState

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

#### Attribute

#### - tiles: Tile[][]

Ein Array von Tiles, welches den Aufbau des Spielbretts darstellt

#### - lastMove: Move

Der letzte gespielte Zug.

#### - whiteToMove: boolean

Speichert, ob Weiß am Zug ist.

#### - whiteKingCastle: boolean

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

#### - whiteQueenCastle: boolean

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

#### - blackKingCastle: boolean

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

#### - blackQueenCastle: boolean

Speichert ab, ob Schwarz noch auf der Damenseite rochieren kann.

#### - movesWithoutAction: int

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

#### Methoden

#### + BoardState()

Erzeugt ein leeres Schachbrett und setzt alle Variablen auf ihre Standardwerte.

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

#### + applyMove(move: Move): void

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): Piece

Gibt die Figur an der übergebenen Position zurück. Ist die Position nicht besetzt, wird **null** zurückgegeben.

#### + getLastMove(): Move

Gibt lastMove zurück.

#### + canWhiteKingCastle(): boolean

Gibt zurück, ob Weiß noch auf der Königsseite rochieren kann.

#### + canWhiteQueenCastle(): boolean

Gibt zurück, ob Weiß noch auf der Damenseite rochieren kann.

#### + canBlackKingCastle(): boolean

Gibt zurück, ob Schwarz noch auf der Königsseite rochieren kann.

#### + canBlackQueenCastle(): boolean

Gibt zurück, ob Schwarz noch auf der Damenseite rochieren kann.

#### + getMovesWithoutAction): int

Gibt movesWithoutAction zurück.

#### + toString(): String

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

#### + clone(): BoardState

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.

#### Attribute

#### - piece: Piece

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

#### Methoden

## + Tile()

Konstruktor, welcher ein leeres Feld erzeugt.

#### + Tile(piece: Piece)

Konstruktor, welcher ein Feld mit der übergebenen Figur darauf erzeugt.

- + getPiece(): Piece Gibt piece zurück. Wurde kein Piece gesetzt, wird null zurückgegeben.
- + setPiece(piece: Piece): void Setzt piece auf die übergebene Figur.

#### 2.1.4 Player



Abbildung 6: Player

## • Player

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

#### Attribute

- user: User

Der mit diesem Spielerobjekt verknüpfte Benutzer.

#### Methoden

+ Player(user: User)

Konstruktor, welcher user auf den übergebenen User setzt.

#### + getName(): String

Gibt den Namen des verknüpften Benutzers als String zurück.

#### • User

Stellt einen Benutzer der Schach-App dar.

#### Attribute

#### - name: String

Der eindeutige Name eines Benutzers.

#### - playedGames: int

Die Anzahl der gespielten Spiele des Benutzers.

#### - wonGames: int

Die Anzahl der gewonnenen Spiele des Benutzers.

#### - drawedGames: int

Die Anzahl der remisierten Spiele des Benutzers.

#### - lostGames: int

Die Anzahl der verlorenen Spiele des Benutzers.

#### - elo: int

Der Elo-Wert des Benutzers.

#### Methoden

## + User(name: String)

Erstellt einen neuen Benutzer mit dem übergebenen Namen. Setzt **elo** auf 1000, alle anderen Attribute auf 0.

#### + getPlayedGames(): int

Gibt playedGames zurück.

#### + getWonGames(): int

#### Gibt wonGames zurück.

- + addWonGame(): void Erhöht wonGames und playedGames um eins.
- + getDrawedGames(): int Gibt drawedGames zurück.
- + addDrawedGame(): void Erhöht drawedGames und playedGames um eins.
- + getLostGames(): int Gibt lostGames zurück.
- + addLostGame(): void Erhöht lostGames und playedGames um eins.
- + getElo(): int Gibt elo zurück.
- + setElo(Elo: int): void Setzt die Elo des Benutzers auf den übergebenen Wert.
- + getName(): String Gibt name zurück.

## 2.1.5 Result

# Result

- + result: String
- + reason: String
- + Result(result: String)
- + Result(result: String, reason: String)
- + toString(): String
- + getReason(): String

Abbildung 7: Result

#### Result

Stellt ein Ergebnis eines Spiels dar.

#### Attribute

- result: String

Das Ergebnis eines Spiels als String codiert.

- reason: String

Eine optionale Begründung zum Ergebnis als String.

#### Methoden

+ Result(result: String)

Erzeugt ein Objekt mit dem übergebenen String als **result** und einer leeren **reason**.

#### + Result(result: String, reason: String)

Erzeugt ein Objekt mit dem ersten übergebenen String als  $\mathbf{result}$  und dem zweiten übergebenen String als  $\mathbf{reason}$ .

## + toString(): String

Gibt **result** zurück.

+ getReson(): String Gibt reason zurück.

#### 2.1.6 Move

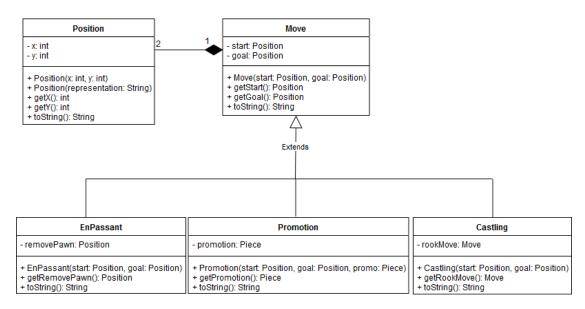


Abbildung 8: Move

#### • Move

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

#### Attribute

#### - start: Position

Position, von welcher aus der Zug ausgeführt wird.

#### - goal: Position

Position, zu welcher der Zug hinführt.

#### Methoden

#### + Move(start: Position, goal: Position)

Erzeugt einen Zug mit der übergebenen Start- und Zielposition.

#### + getStart(): Position

Gibt **start** zurück.

#### + getGoal(): Position

Gibt goal zurück.

#### + toString(): String

Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

#### • Position

Stellt eine Position auf einem Schachbrett dar.

#### Attribute

#### - x: int

Stellt die horizontale Koordinate auf dem Schachbrett dar.,

#### - y: int

Stellt die vertikale Koordinate auf dem Schachbrett dar.

#### Methoden

#### + Position(x: int, y: int)

Erzeugt eine Position mit den angegebenen Koordinaten. 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(): int

Gibt  $\boldsymbol{x}$  zurück.

#### + getY(): int

Gibt y zurück.

#### + toString(): String

Gibt eine Repräsentation der Position als String zurück.

#### • EnPassant

Stellt den Bauernzug en passant dar.

#### Attribute

#### - removePawn: Piece

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

#### Methoden

#### + EnPassant(start: Position, goal: Position)

Ruft den Konstruktor von Move auf, berechnet und setzt die PositionremovePawn.

#### + getRemovePawn(): Position

Gibt removePawn zurück.

#### + toString(): String

Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

#### • Promotion

Stellt eine Bauernumwandlung dar.

## Attribute

#### - promotion: Piece

Die Figur, in welche sich der Bauer verwandeln soll.

#### Methoden

### + Promotion(start: Position, goal: Position, promo: Piece)

Ruft den Konstruktor von Move auf und setzt promotion auf die übergebene

Figur.

- + getPromotion(): Piece Gibt promotion zurück.
- + toString(): String

Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

## • Castling

Stellt eine Rochade dar.

#### Attribute

- rookMove: Move

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

#### Methoden

+ Castling(start: Position, goal: Position)

Ruft den Konstruktor von Move auf, berechnet und setzt den ZugrookMove.

- + getRookMove(): Move Gibt rookMove zurück.
- + toString(): String

Gibt eine eindeutige Repräsentation des Zugs als Zeichenkette zurück.

#### **2.1.7 Pieces**

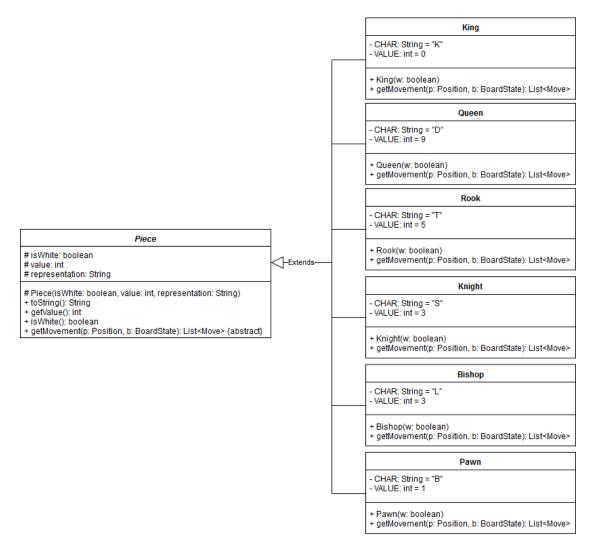


Abbildung 9: Pieces

## • Piece

Stellt das abstrakte Konzept einer Schachfigur dar.

#### Attribute

### - isWhite: boolean

Speichert, ob die Figur weiß ist.

#### - value: int

Speichert den Wert einer Figur ab.

#### - representation: String

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(): String

Gibt representation zurück.

+ getValue(): int

Gibt value zurück.

+ isWhite(): boolean

Gibt isWhite zurück.

## • King

Stellt einen König dar.

#### Attribute

#### - CHAR: String

Die Repräsentation eines Königs, wird auf "K" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState): List<Move>

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: String

Die Repräsentation einer Dame, wird auf "D" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState): List<Move>

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: String

Die Repräsentation eines Turms, wird auf "T" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState)

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: String

Die Repräsentation eines Läufers, wird auf "L" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState)

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: String

Die Repräsentation eines Springers, wird auf "S" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState)

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: String

Die Repräsentation eines Bauerns, wird auf "B" gesetzt.

#### - VALUE: int

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(p: Position, b: BoardState)

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**

#### 2.2.1 Activities

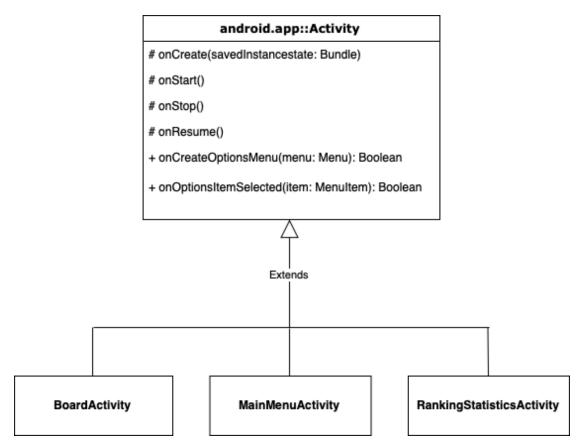


Abbildung 10: Activities

## • android.app.Activity

Ist eine einzelne, fokussierte Aufgabe, die der Benutzer ausführen kann. Da fast alle Aktivitäten mit dem Benutzer interagieren, kümmert sich die Activity-Klasse darum, ein Fenster für Sie zu erstellen.

#### Methoden

#### # onCreate(savedInstanceState: Bundle):

Wird aufgerufen, wenn die Aktivität beginnt. Wenn die Aktivität neu initialisiert wird, enthält das Bundle die Daten, die es zuletzt bei SaveInstanceState

bereitgestellt hat. Ansonsten ist es null.

#### # onStart():

Wird aufgerufen, wenn die Aktivität gestoppt wurde, jetzt aber dem Benutzer wieder angezeigt wird.

#### # onStop():

Wird aufgerufen, wenn man für den Benutzer nicht mehr sichtbar sind.

#### # onResume():

Wird aufgerufen, damit die Aktivität mit dem Benutzer interagieren kann.

#### + onCreateOptionsMenu(menu: Menu): Boolean:

Initialisiert den Inhalt des Standardoptionen-Menüs der Aktivität. Es gibt das Optionsmenü, in dem die Elemente platziert werden sollen. Es gibt true für das angezeigte Menü zurück und bei false wird es nicht angezeigt.

#### + onOptionsItemSelected(item: MenuItem): Boolean:

Wird aufgerufen, wenn ein Element im Optionsmenü ausgewählt wird. Item ist das ausgewählte Menüelement. Es gibt false zurück, um die normale Menüverarbeitung fortzusetzen und true, um sie dann zu verbrauchen.

## BoardActivity

Es werden alle Borad-Aktivitäten angezeigt, wie z.B. das Brett.

# MainMenuActivity

Es werden alle Menü-Aktivitäten angezeigt, wie z.B. die Knöpfe Sofortspiel, Spielersuche und Rangliste/Statistik.

## RankingStatisticsActivity

Es werden alle Ranglisten und Statistiken-Aktivitäten angezeigt, wie z.B. ein Rangliste mit allen Spielern und den jeweiligen Punkten oder eine Statistik über den Spieler angezeigt.

## **BoardActivity**

Hier wird das Spiel angezeigt mit den Buttons "Unentschieden" und "Aufgeben".

#### + drawButton(): void:

Sobald der Benutzer auf diesen Button während des Spieles drückt, wird ein Unentschieden angezeigt. Somit kann er das Spiel abbrechen, ohne das sich etwas an seinem Punktestand ändert. Der Gegner muss diesem zustimmen. Danach gelangt man in die "MainMenuActivtiy".

## MainMenuActivity

Hier wird das Hauptmenü angezeigt mit Buttons "Sofortspiel", "Spielersuche", "Rangliste/Statistik".

#### + quickMatchButton(): void:

Sobald der Benutzer "Sofortspiel" drückt, öffnet sich die "BoardActivity". Dort spielt man gegen einen zufälligen Gegner.

#### + searchPlayerButton(): void:

Sobald der Benutzer "Spielersuche" drückt, öffnet sich die "SearchPlayerActivity". Dort kann man einen anderen Benutzer mit seinem Spielernamen suchen und mit dem Button "Herausfordern" (+ challengeButton(): void) diesen herausfordern.

#### + rankingStatisticsButton(): void:

Sobald der Benutzer "Rangliste/Statistik" drückt, öffnet sich die "RankingStatisticActivity". Dort kann der Benutzer seine Statistik und den Ranglistenplatz einsehen.

## RankingStatisticsActivity

Nach dem drücken des Buttons "Rangliste/Statistik" in der "MainMenuActivity" öffnet sich die "RankingStatisticActivity". Dort kann der Benutzer zum einen in der Statistik seine Daten einsehen und zum anderen in der Rangliste sich mit anderen Benutzern vergleichen. Dieser kann nach dem Einsehen wieder zur "MainMenuActivity" gelangen.

# SearchPlayerActivity

Nach dem drücken des Button "SearchPlayer" in der "MainMenuActivity" öffnet sich die "SearchPlayerActivity". Dort kann der Benutzer andere Gegner finden und diese mit dem Button "Herausfordern" ( + challengeButton: void) herausfordern.

### 2.2.2 Dialoge

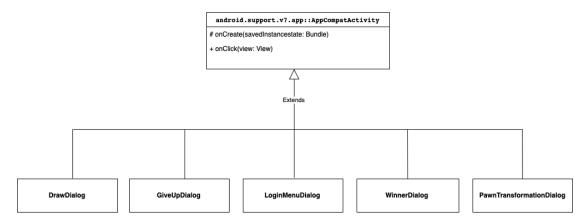


Abbildung 11: Dialoge

# • android.support.v7.app.AppCompatActivity

Die AppComaptActivity wird als Dialog verwendet.

#### Methoden

### # onCreate(savedInstanceState: Bundle):

Erzeugt die Knöpfe. Wenn die Aktivität neu initialisiert wird, enthält das Bundle die Daten, die es zuletzt bei SaveInstanceState bereitgestellt hat. Ansonsten ist es null.

### + onClick(view: View):

Erzeugt die Fuktionalität eines Konpfes wobei ein AlertDialog geöffnet wird.

# DrawDialog

Es wird ein Unentschieden-Dialog angezeigt, wie z.B. das Brett.

# ${\bf Give Up Dialog}$

Es werden alle Menü-Aktivitäten angezeigt, wie z.B. die Knöpfe Sofortspiel, Spielersuche und Rangliste/Statistik.

# LoginMenuDialog

Es werden alle Ranglisten und Statistiken-Aktivitäten angezeigt, wie z.B. ein Rangliste mit allen Spielern und den jeweiligen Punkten oder eine Statistik über den Spieler angezeigt.

# WinnerDialog

Es werden alle Menü-Aktivitäten angezeigt, wie z.B. die Knöpfe Sofortspiel, Spielersuche und Rangliste/Statistik.

# PawnTransformationDialog

Es werden alle Ranglisten und Statistiken-Aktivitäten angezeigt, wie z.B. ein Rangliste mit allen Spielern und den jeweiligen Punkten oder eine Statistik über den Benutzer angezeigt.

#### **Buttons**

# DrawDialog

Hier wird ein Unentschieden-Fenster geöffnet mit den Buttons "Ja" und "Nein".

#### + drawDialog(): void:

Wenn man die Partie mit einem Unentschieden beendet, taucht der "DrawDialog" auf. Man kann bei Bedarf noch einmal gegen den gleichen Gegner spielen, indem man den Button "Ja" betätigt und in die "BoardActivity" wechselt oder in die "MainMenuActivity" gelangt in dem man "Nein" betätigt.

# GiveUpDialog

Hier wird ein Aufgeben-Fenster geöffnet mit den Buttons "Ja" und " Nein".

### + giveUpDialog(): void:

Wenn man auf den Button "Aufgeben" drückt, erscheint der "GiveU-pDialog", ob man wirklich aufgeben möchte mit den Buttons "Ja", wodurch man in die "MainMenuActivity" gelangt und "Nein", wodurch

man in die "BoardActivity" wechselt.

# LoginMenuDialog

Hier wird ein Login-Menü-Fenster geöffnet.

### + loginMenuDialog(): void:

Nach dem Drücken auf den Knopf "Anmeldung" mit Google öffnet sich der "LoginMenuDialog" für den Login, wo man seine Daten angeben kann und nach dem Login in die "ManiMenuActivity" gelangt.

# WinnerDialog

Hier wird ein Gewonnen-Fenster geöffnet mit den Buttons "Menü", "Speichern" und "Noch ein Spiel?"

### + winnerDialog(): void:

Wenn man eine Partie gewonnen hat, erscheint der "WinnerDialog" mit der Meldung, dass man gewonnen hat und mit den zusätzlichen Buttons: "Menü", wodurch man in die "MainMenuActivity" gelangt, "Speichern", wodurch die Spielzüge gespeichert werden und "Noch ein Spiel?", wodurch man in die "BoardActivity" wechselt.

# PawnTransformationDialog

Hier wird ein Bauernumwandlungs-Fenster geöffnet mit den Buttons "Queen", "Rook", "Bishop" und "Knight".

### + pawnTransformation(): void:

Wenn ein Bauer die gegnerische Grundreihe erreicht, erfolgt eine Bauernumwandlung. Dabei werden dann die Möglichkeiten Queen, Rook, Bishop, Knight zum austauschen in dem "PawnTransforamtionDialog" angezeigt. Die umgewandelte Figur wirkt dabei sofort.

#### 2.3 ClientSocket

# ClientSocket

- user: String
- + connectToWS(): void
- + requestBoard(): String
- + sendMove(move: String): void
- + newGame(opponent: String): void
- getUser(): String

Abbildung 12: ClientSocket

### ClientSocket

Baut eine WebSocket-Verbindung zum Server auf. Sendet zudem HTTP-Anfragen an den Server.

#### Methoden

### + connectToWS(): void

Baut eine WebSocket-Verbindung zum Server auf. Die App wird über diese Verbindung benachrichtigt, falls der Gegner einen Zug ausgeführt hat.

## + requestBoard(): String

Sendet eine GET-Anfrage an den Server, die den aktuellen Brettzustand zurückgibt.

# + sendMove(move: String): String

Sendet einen Zug an den Server. Bekommt eine Meldung, je nachdem, ob der gesendete Zug gültig war.

# + newGame(opponent: String): String

Sendet eine Anfrage für die Erstellung eines neuen Spiels. Bekommt eine Meldung, wenn das Spiel erstellt wurde.

# - getUser(): String

Gibt den Nutzernamen des App-Nutzers zurück.

### 2.4 Server

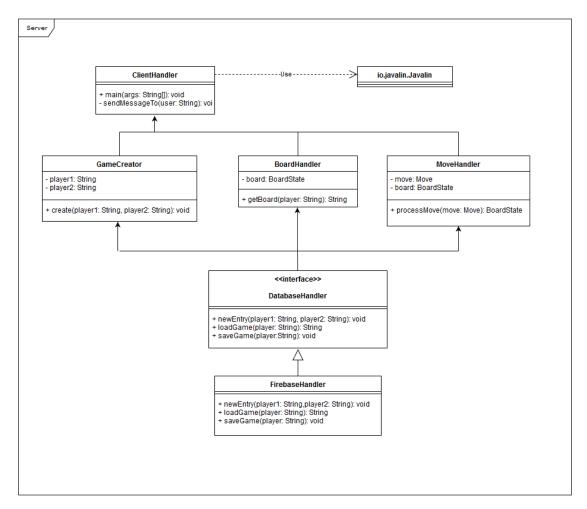


Abbildung 13: Server

# • ClientHandler

Verwaltet eingehende Anfragen an den Server.

### Methoden

# + main(String[] args): void

Die main Methode des Servers. Hier wird der Javalin Server gestartet. Je nach Anfrage wird ein **GameCreator**, ein **BoardHandler** oder ein **MoveHandler** erstellt.

## - sendMessageTo(user: String): void

Sendet eine Benachrichtigung über die WebSocket-Verbindung an den angegebenen Spieler.

## • 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  ${\bf Database Handler}.$ 

# • 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 Aktivitätsdiagramm

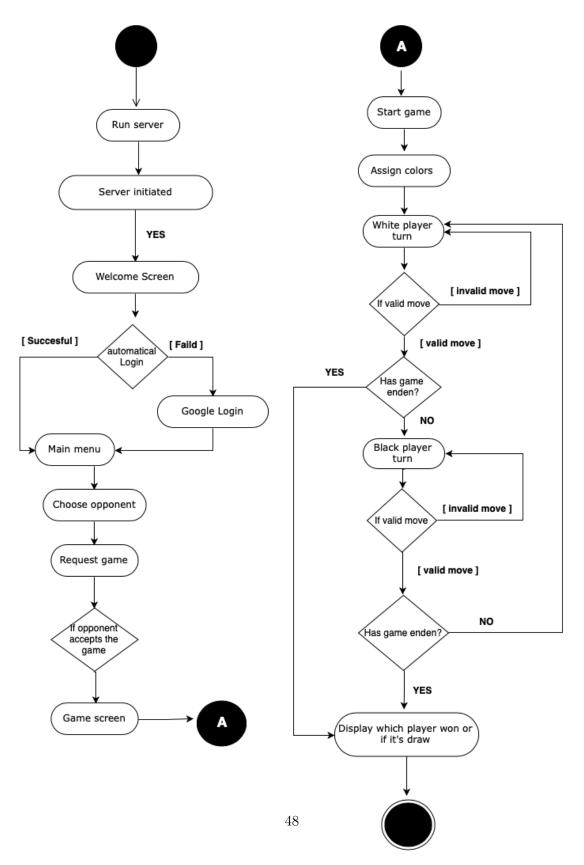


Abbildung 14: Aktivitätsdiagramm

# 4 Sequenzdiagramm

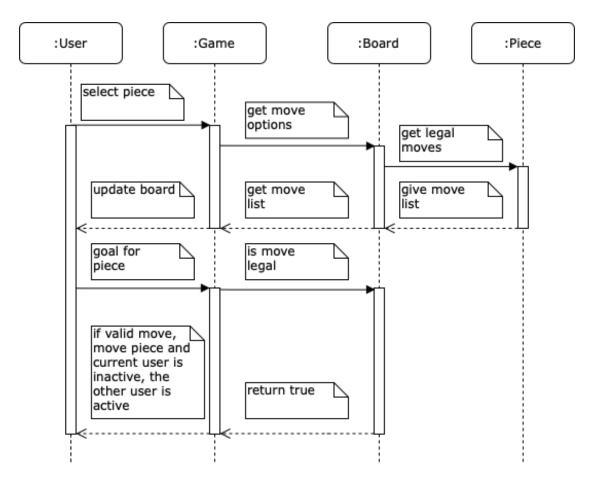


Abbildung 15: Sequenzdiagramm 1

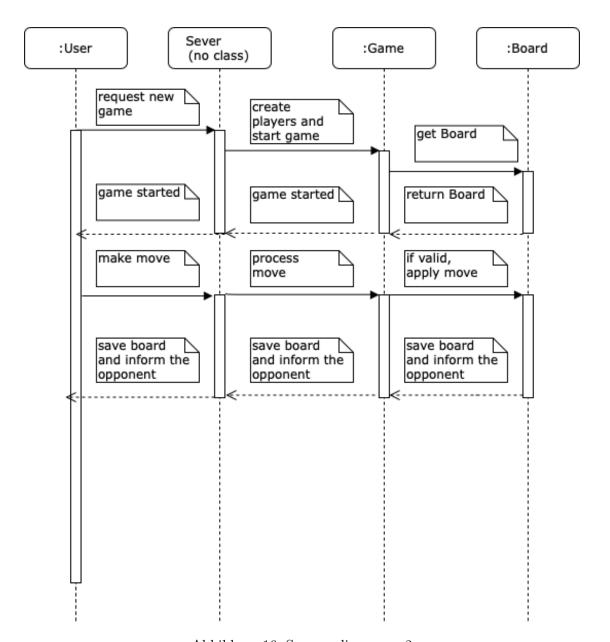


Abbildung 16: Sequenzdiagramm 2