

Developerhandbuch

Schachspiel



Softwarepraktikum 2019

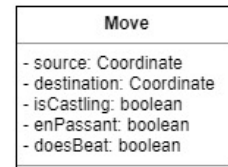
von: Adrian Samoticha, Simon Trapp, Mohamad Halloway

Das Schachspiel wurde nach einer strikten Model-View-Controller Architektur entwickelt. Die einzelnen Schichten werden im folgenden erläutert. Abschließend wird die Funktionsweise des Importes und Exportes von Spieldateien erklärt.

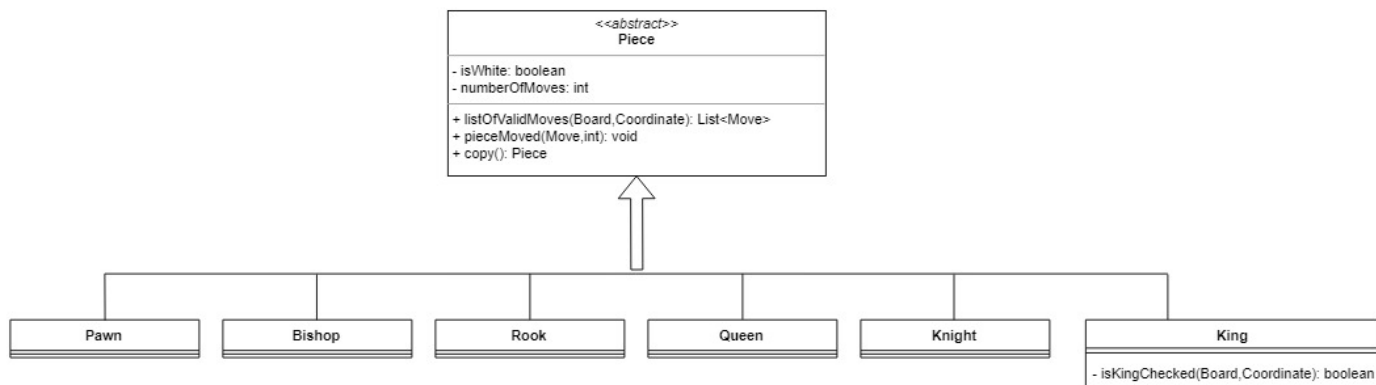
1 Das Model

Das Model enthält das Schachbrett und die Spielregeln.

Ein **Move** stellt ein Zug dar und speichert die Quelle und Ziel-Koordinaten des Zugs, wobei „**Coordinate**“ einfach die $x \in [1, 8]$ und $y \in [a, g]$ Koordinaten enthält



Die Hauptfiguren-Klasse „**Piece**“ ist eine abstrakte Klasse, die alle Schachfiguren-Klassen implementieren, diese Klasse bietet hauptsächlich eine Liste der validen Zügen für die Instanz und lässt sich kopieren um die Simulation der KI zu erleichtern.



Board

Das 8 x 8 Schachbrett wird durch diese Klasse erstellt und wird durch folgende Methoden gesteuert:

- `newGame()` Erneuert das Schachbrett, indem sie es durch ein neues ersetzt auf dem alle Schachfiguren auf ihren richtigen Plätzen stehen.
- `performMove(Move)` Führt einen Zug im Hinblick auf En Passant und IsCastling durch
- `doesMovePreventCheck(Move)` Überprüft durch eine Simulation ob nach diesem Zug der König des spielenden Spielers im Schach steht
- `getKingPosition(boolean)` Gibt die Königskoordinaten der übergebenen Farbe (`true` = weiß, `false` = schwarz) auf dem Schachbrett.

Außerdem steht die Klasse „**GameState**“ zur Verfügung, die das Interface *Serializable* implementiert und alle wichtigen Informationen für das Speichern bzw. Laden des Spiels enthält.

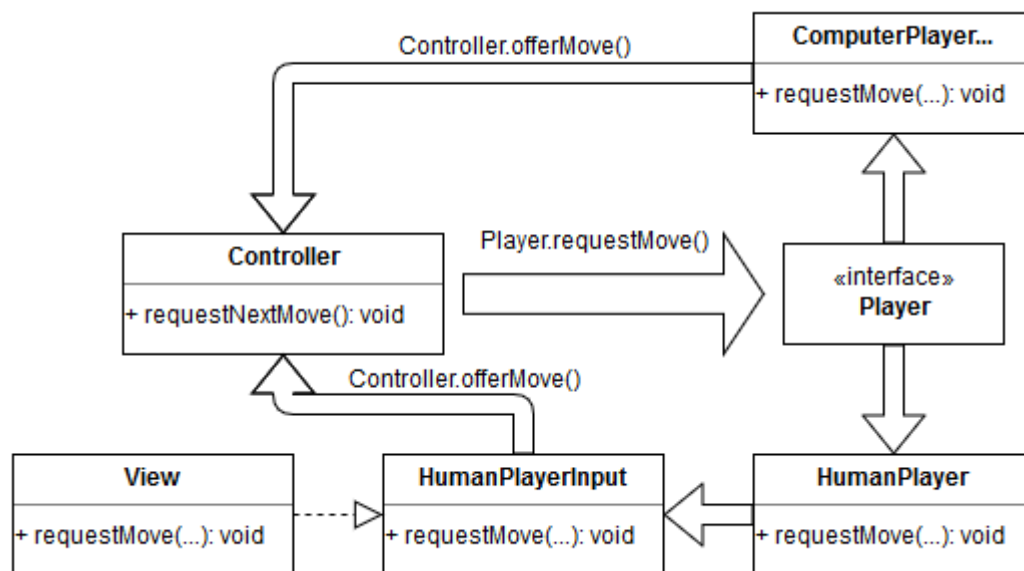
2 Der Controller

Der Controller verwaltet den Spielablauf, die Spieler, die Events und die Zuggeschichte.

a) Der Spielablauf

Bevor ein Spiel gestartet werden kann, muss zuerst ein Controller initialisiert werden. Anschließend kann optional per *Controller.loadGameState()* ein vorhandener Spielstand in den Controller geladen werden. Danach müssen dem Controller mit *Controller.setupNewGame()* die Spielerinstanzen und ein Boolean, der bestimmt, ob ein neues Spiel oder ein geladenes Spiel aufgesetzt werden soll, übergeben werden.

Nachdem per *Controller.start()* das Spiel gestartet wurde, berechnet der Controller zuerst alle validen Spielzüge des aktiven Spielers (aus Performance Gründen, damit diese nicht mehrmals berechnet werden müssen). Dann werden die Könige beider Spieler auf „Schach“ überprüft und bei Bedarf die ControllerListener informiert. Dann wird überprüft, ob sich der momentan aktive Spieler bewegen kann, und falls nicht, wird das Spiel beendet und die Listener informiert. Andernfalls wird vom aktiven Spieler über das Playerinterface *Player.requestMove()* ein Zug verlangt (Der Zug wird **nicht** von dieser Funktion zurückgegeben, da im Falle eines menschlichen Spielers nicht auf diesen gewartet werden würde).



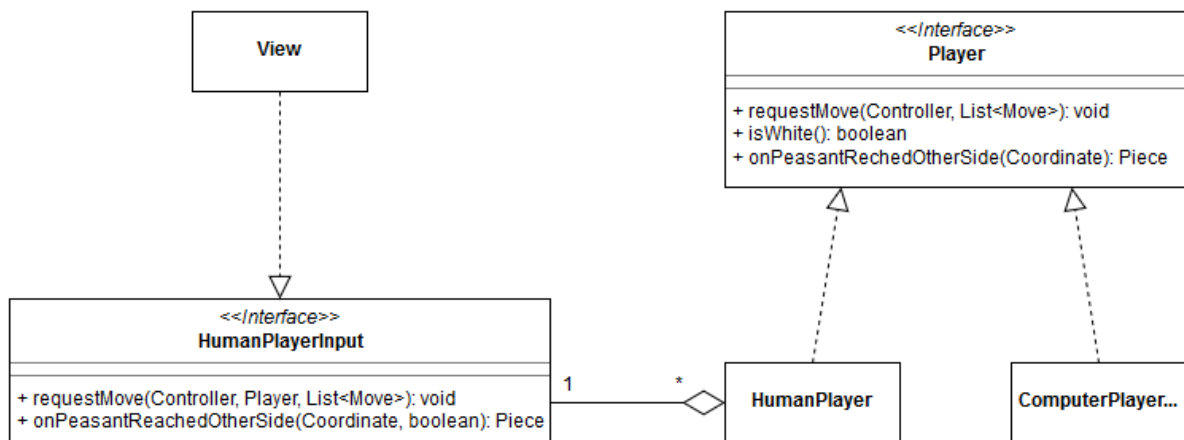
Der *Player* gibt den ausgewählten Zug per *Controller.offerMove()* zurück. Daraufhin wird überprüft, ob sich ein äquivalenter Spielzug in der Liste der validen Spielzüge befindet und falls dies der Fall ist, wird dieser ausgeführt, der Zuggeschichte hinzugefügt und die Listener werden informiert.

Außerdem wird überprüft, ob der Zug einen Bauern ans Ende des Schachbrettes bewegt hat und gegebenenfalls über *Player.onPeasantReachedOtherSide()* eine Ersatzfigur erfragt. Noch dazu wird überprüft, ob der Zug zu einer dritten Wiederholung derselben Situation führt oder ob in 50 aufeinanderfolgenden Zügen weder geschmissen noch ein Bauer bewegt wurde, was beides dazu führt, dass das Spiel mit Resultat unentschieden beendet wird.

Daraufhin wird der aktive Spieler gewechselt und der nächste Zug erfragt. Die Prozedur ab *Controller.start()* wiederholt sich.

b) Das Playerinterface

Das Playerinterface ist die Schnittstelle, über die der Controller mit den Spielern kommunizieren kann.









Besonders ist die Funktionsweise der *HumanPlayer*-Klasse. Da ein menschlicher Spieler im Gegensatz zu Computerspielern eine Eingabemöglichkeit braucht, wurde das *HumanPlayerInput*-Interface geschaffen, welches von der *View* implementiert wird. Dadurch erhält der Programmierer außerdem die Möglichkeit, die Gameloop für Nutzerinput anzuhalten oder Events auszulösen, welche mit der Eingabe in Zusammenhang stehen (z.B. Ton abspielen, „Spieler xy ist an der Reihe“, etc.).

c) Die Spieler „CPU Shallow“, „CPU Deep“ und „CPU Deeper“

Neben dem Spielen gegen einen menschlichen Gegner bietet das Schachspiel zusätzlich eine Reihe an computergesteuerten Spielern an. Folglich wird die Funktionsweise und die Softwarearchitektur dreier dieser Computerspieler beschrieben.

(a) Die Evaluationsfunktion


Den Kern der KI liegt die Evaluationsfunktion, die die Güte einer Situation im Spiel bewertet und es somit ermöglicht, den besten Zug zu ermitteln. Die statische Funktion ist in der *GameStateAnalyzer*-Klasse unter dem Namen *analyzeGameState* zu finden. Sie liefert einen numerischen Wert, der größer wird, umso vorteilhafter die gegebene Situation für den ausgewählten Spieler ist, und kleiner, je mehr der gegnerische Spieler bevorteilt ist. Die Situation wird anhand zweier Merkmale bewertet: Der noch im Spiel vorhandenen, d.h. nicht geschmissenen Spielfiguren, und deren Position auf dem Spielfeld. Jeder Spielfigur wird hierbei ein Wert zugeschrieben, wie aus der folgenden Tabelle ersichtlich wird:

					
10	50	30	30	90	900


Zusätzlich zu dem statischen Wert, wird jeder Figur eine Punktezahl zugeschrieben, die abhängig von ihrer Position ist. Diese wird auf den Basiswert hinzuaddiert und ist von Figur zu Figur unterschiedlich.

Die folgenden zwei Beispiele zeigen die Positionsevaluationsmatrizen für den weißen Läufer und das Pferd:

	a	b	c	d	e	f	g	h	
8	-2.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-2.0	8
7	-1.0	0.0	0.0	0.0	0.0	0.0	0.0	-1.0	7
6	-1.0	0.0	0.5	1.0	1.0	0.5	0.0	-1.0	6
5	-1.0	0.5	0.5	1.0	1.0	0.5	0.5	-1.0	5
4	-1.0	0.0	1.0	1.0	1.0	1.0	0.0	-1.0	4
3	-1.0	1.0	1.0	1.0	1.0	1.0	1.0	-1.0	3
2	-1.0	0.5	0.0	0.0	0.0	0.0	0.5	-1.0	2
1	-2.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-2.0	1
	a	b	c	d	e	f	g	h	

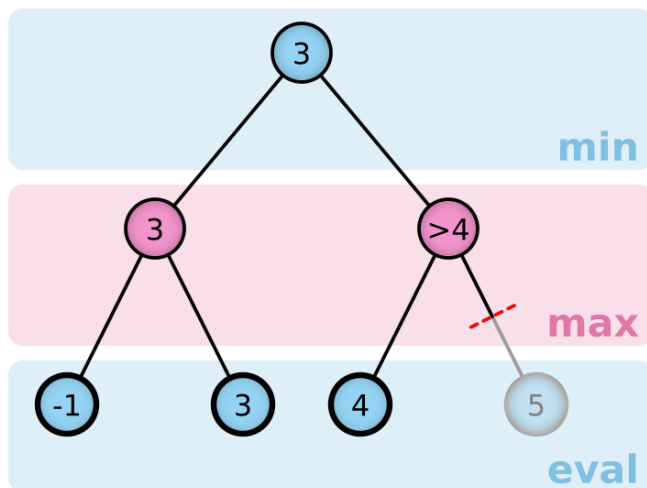


	a	b	c	d	e	f	g	h	
8	-5.0	-4.0	-3.0	-3.0	-3.0	-3.0	-4.0	-5.0	8
7	-4.0	-2.0	0.0	0.0	0.0	0.0	-2.0	-4.0	7
6	-3.0	0.0	1.0	1.5	1.5	1.0	0.0	-3.0	6
5	-3.0	0.5	1.5	2.0	2.0	1.5	0.5	-3.0	5
4	-3.0	0.0	1.5	2.0	2.0	1.5	0.0	-3.0	4
3	-3.0	0.5	1.0	1.5	1.5	1.0	0.5	-3.0	3
2	-4.0	-2.0	0.0	0.5	0.5	0.0	-2.0	-4.0	2
1	-5.0	-4.0	-3.0	-3.0	-3.0	-3.0	-4.0	-5.0	1
	a	b	c	d	e	f	g	h	



(b) Das Entscheidungsfällen

Mithilfe der oben beschriebenen Evaluierungsfunktion kann die KI entscheiden, welcher Zug gespielt werden soll. Die „CPU Shallow“ tut dies auf eine sehr einfache Art und Weise. Jeder mögliche Zug wird simuliert und die darauf folgende Situation mithilfe der Evaluationsfunktion bewertet. Gibt es einen eindeutigen besten Zug, so wird dieser gespielt. Ansonsten wird aus der Menge der besten Züge ein Zug per Zufall gewählt.



Die „CPU Deep“ geht einen Schritt weiter. Mithilfe des sog. „Minimax“-Algorithmus kann sie auch die in der Zukunft zu spielenden Züge berücksichtigen, um somit eine bessere Entscheidung zu fällen, als dies die „CPU Shallow“ tut. Dieser Algorithmus funktioniert, indem er einen Entscheidungsbaum generiert, bei dem sowohl die möglichen Züge der KI, als auch des Gegners in jeweils abwechselnd aufeinander folgenden Knoten repräsentiert werden. Vorgesehen werden ein sog. „maximizing player“, der einen

möglichst hohen Evaluationswert anstrebt, sowie ein „minimizing player“, der niedrige Werte anstrebt. Da die Evaluationsfunktion den zu betrachtenden Spieler (also die KI) entgegennimmt und den für ihn vorteilhaften Situationen höhere Werte zuschreibt, ist der „maximizing player“ immer die KI selbst und der „minimizing player“ ihr Gegner. Um jetzt den besten Zug zu ermitteln, wird jedem Knoten ein Wert zugeschrieben, der jeweils der höchste bzw. niedrigste Wert (je nach Player) aller seiner Kinderknoten ist. Die Blätter erhalten ihren Wert durch die Evaluationsfunktion. Das bedeutet, dass der Minimax-Algorithmus die Bewertung des besten Zuges zurückgibt, unter der Annahme, dass sowohl die KI als auch ihr Gegner immer ihren besten Zug spielen.

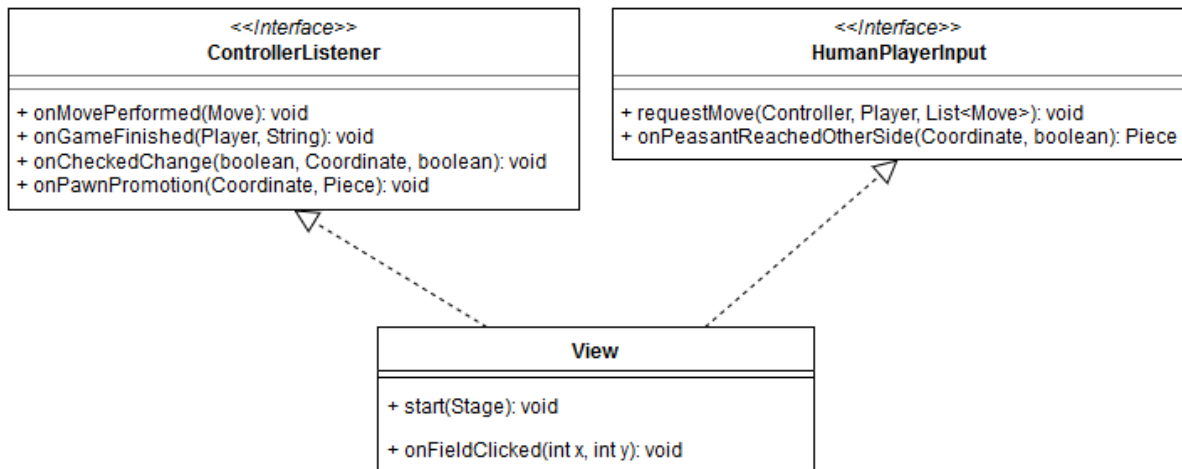
Um die Performanz des Algorithmus weiter zu optimieren, wird zudem die unter dem Namen „Alpha-beta pruning“ bekannte Optimierungsmethode angewendet. Äste des Suchbaums, die zu keiner besseren Situation führen können, als bereits vorher evaluierte Äste, und somit das Resultat des „Minimax“-Algorithmus nicht beeinflussen, werden abgeschnitten, um Rechenzeit zu sparen.

Da das Durchführen des Algorithmus für alle möglichen Züge trotz dieser Optimierung immer noch einige Sekunden dauert, geschieht die Evaluation parallel auf mehreren CPU-Threads. Zu diesem Zweck besitzt die für die „CPU Deep“ zuständige Klasse *ComputerPlayerDeep* eine Subklasse namens *MoveEvaluator*. Am Anfang der Berechnung wird für jeden möglichen Zug ein eben solcher *MoveEvaluator* erstellt, der jeweils einen Zug auf seinem eigenen Thread evaluieren soll. Dadurch kann in einem Bruchteil einer Sekunde eine Entscheidung getroffen werden, wodurch eine flüssige Spielerfahrung ermöglicht wird.

Die „CPU Deeper“ ist der „CPU Deep“ sehr ähnlich. Tatsächlich wird sie von der gleichen Klasse realisiert. Die Klasse hat zwei Eigenschaften, *minDepth* und *maxDepth*, die für die „CPU Deep“ in beiden Fällen 3 betragen. *minDepth* gibt die Tiefe des initialen Suchbaums an. Wird mehr als ein bester Zug gefunden und ist *maxDepth* größer als *minDepth*, so wird die Tiefe graduell erhöht, bis entweder ein eindeutiger bester Zug ermittelt, oder die maximale Tiefe erreicht worden ist. Züge, die schlechter sind als der jeweils beste Zug werden dabei bei jeder Iteration aussortiert, sodass die Anzahl der zu evaluierenden Züge mit zunehmender Tiefe sinkt. Die maximale Tiefe der „CPU Deeper“ beträgt 5.

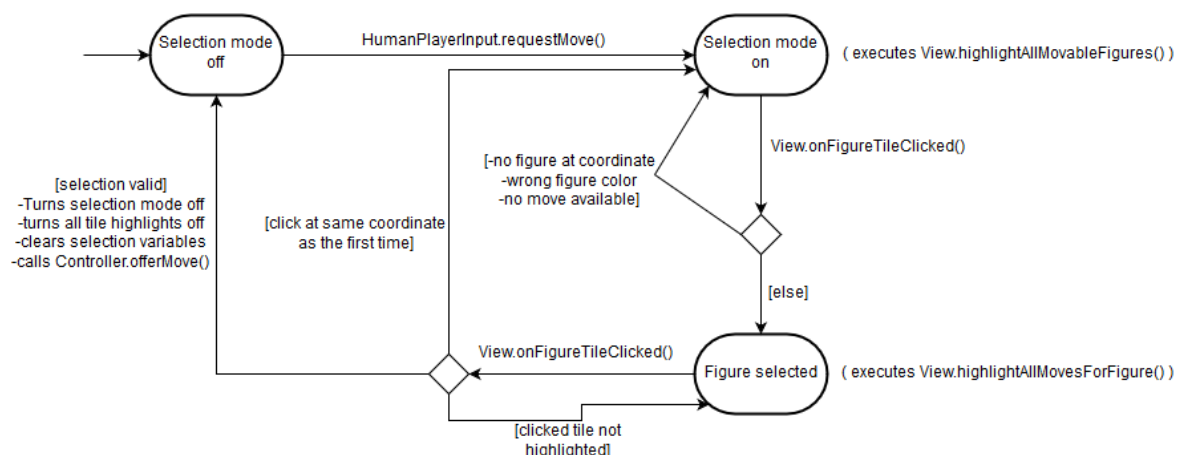
3 Die View

Die View ist die Schnittstelle zwischen menschlichem Benutzer und dem Programm.



Die Klasse „View“ repräsentiert die Benutzeroberfläche des Programms. Sie zeigt das Schachbrett und alle weiteren Bedienelemente (Menü, AlertBoxes, Move history,...). Außerdem ist sie der Einstiegspunkt in das Programm. Sie initialisiert und startet den Controller, dessen Events sie über das Interface „ControllerListener“ entgegennimmt.

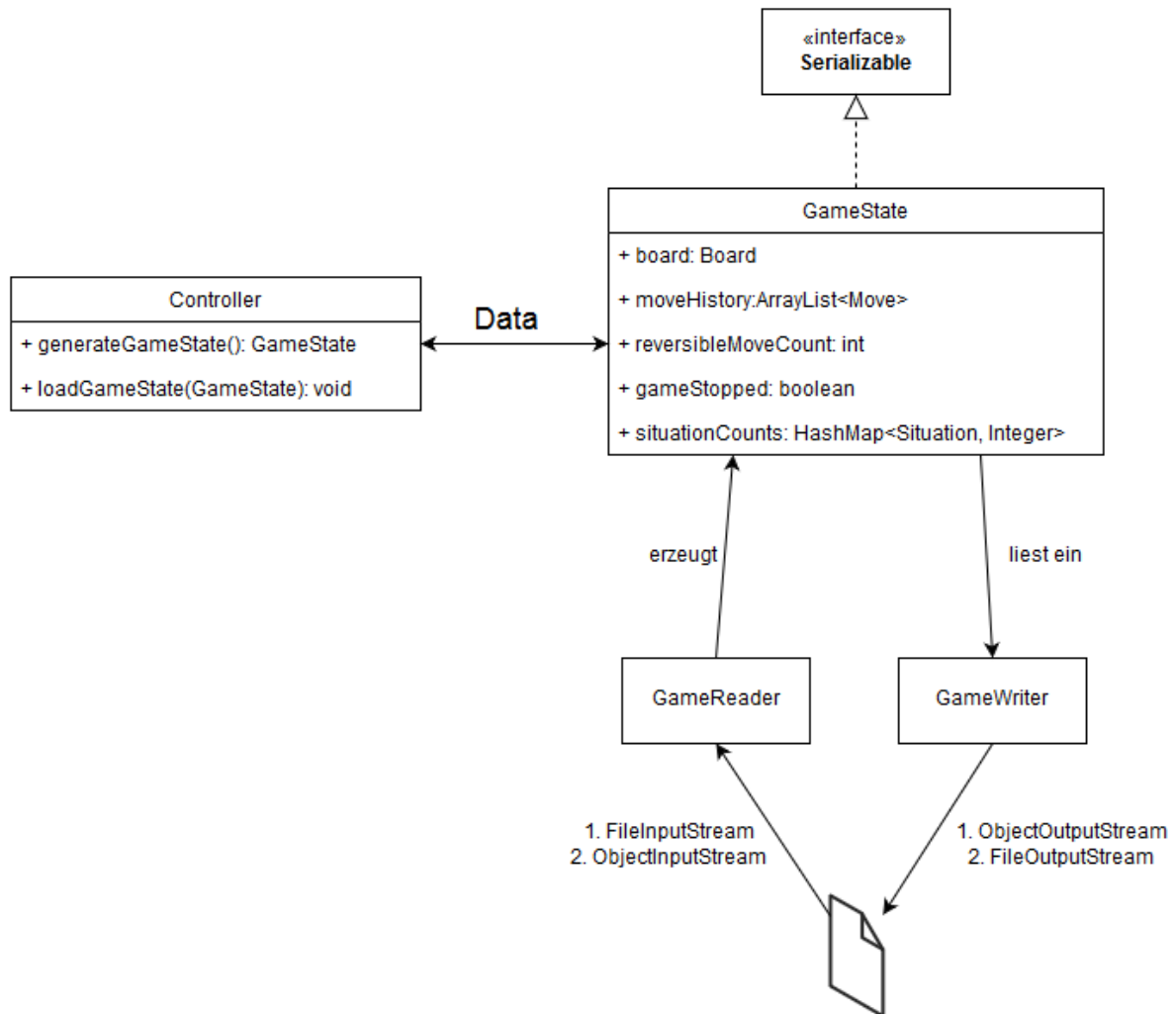
Weiterhin dient der View als Eingabemöglichkeit für einen menschlichen Spieler. Diese Funktionalität wird durch das Interface „HumanPlayerInput“ realisiert. Über dieses kann ein „HumanPlayer“ im Spielverlauf (durch den Controller aufgefordert), nach Erreichen des anderen Spielfeldrandes mit einem Bauer eine neue Spielfigur, oder einen „Move“ abfragen. Der „View“ versteht diese Move-Abfrage als Signal, in den Auswahlmodus zu wechseln, in dem das Schachbrett zur Auswahl von Feldern durch den Benutzer freigegeben ist. Klickt der Nutzer nun auf ein Feld des Schachbretts, so wird von der „GridTile“ oder der „FigureTile“ die Funktion `View.onFieldClicked(x,y)` aufgerufen, welche quasi einem Schritt in einem Zustandsautomaten entspricht, der folgendes Diagramm umsetzt:



Durch erfolgreiche Auswahl wird der Move an den Controller per `Controller.offerMove()` übergeben. Die „game loop“ ist somit rekursiv.

4 Der Dateiimport/-export

Das folgende Modell skizziert grob, wie das Speichern und Laden eines Spielstandes funktioniert:



Um den Spielverlauf und das derzeitige Spielfeld zu speichern, generiert der **GameWriter** über *Controller.generateGameState()* einen **GameState**, welchem der Controller alle spielrelevanten Daten übergibt. Dieser implementiert das Interface **Serializable**, das von Java selbst zur Verfügung gestellt wird. Dieses Interface ermöglicht es, den **GameState** per Stream im Dateisystem zu speichern und zu laden. Zum Laden kann der **GameReader** genutzt werden. Dieser lädt die Spieldatei per Stream und generiert daraus einen `ObjectInputStream`, aus welchem wieder ein **GameState** erzeugt werden kann. Dieser wird über *Controller.loadGameState()* wieder an den Controller übergeben, welcher die Daten aus dem **GameState** übernimmt.