Architekturdokumentation

zum Software Engineering Praktikum Projekt SoSe 2020

Inhaltsverzeichnis

Komponenten **2**
- Trennung der Zuständigkeiten der Komponenten 2
- Umgebungsvariablen 2

Interaktion durch Sitzungen **2**
- Sitzungsbefehle und ihre Vererbungsstruktur 3
- Ablauf einer Sitzung 4
- Deklarative Spielmodi 5
- Typen von Sitzungen und Spielern 5
  - Multithreading in Sitzungen 6
  - Spieler für die KI 6

Benutzerschnittstelle Konsoleninterface **7**
- Sitzung und Spieler für das Konsoleninterface 7
- Ausgabe der Spielstände 8
- Auswahl von Spielmodi 8

Benutzerschnittstelle GUI **8**
- Sitzung und Spieler für die GUI 8
- Komponenten der GUI und ihre Interaktion 10
- Hauptmenü 10
- Einstellungen 11
- Spieldarstellung 11

Spiellogik **12**
- Abbildung des Spielzustandes 12
  - Zustandslose Instanzen für Figuren auf dem Spielbrett 13
  - Zughistorie 14
  - Vererbungsstruktur der Figuren 15
  - Weitere Teile des Zustandes 15
- Berechnung von Zügen 16
  - Komposition von Zügen und ihre Vererbungsstruktur 16
  - Beschränkte Suche nach möglichen Zügen mit Zugakkumulatoren 17
  - Vererbungsstruktur von Zugakkumulatoren 18
  - Erlaubte Züge durch Filtern finden 19
- Ausführen von Zügen 20
  - Züge als Zustandsübergänge 20
  - Spielstatus bestimmen 20

Tests **20**

Komponenten

Trennung der Zuständigkeiten der Komponenten

Das Programm ist in zwei Komponenten gegliedert, die Spiellogik und die Benutzerschnittstelle, die aus dem dem Konsoleninterface und der GUI besteht. Die Spiellogik bestimmt das Verhalten des Schachspiels, während die Benutzerschnittstelle die Interaktion mit den Benutzer übernimmt. Die Spiellogik ist völlig unabhängig von der Benutzerschnittstelle und importiert keine Klassen aus der Benutzerschnittstelle. Somit kann die Spiellogik unabhängig als eigenes und in sich abgeschlossenes Paket existieren, was es einfacher macht, über das Verhalten der Spiellogik zu folgern.

Generell werden Funktionalitäten in Verantwortungsstufen aufgeteilt, die verschiedenen Methoden oder Klassen zugewiesen werden. Methoden der höchsten Abstraktionsstufe beschäftigen sich mit komplexen Konzepten, die durch den Aufruf kleinteiliger Methoden einfach ausgedrückt werden, damit die eher abstrakten Methoden präzise und prägnant bleiben. Diese abstrakten Methoden stellen sicher, dass bestimmte Vorbedingungen von Aufrufen anderer Methoden erfüllt sind, während weniger abstrakte Methoden konkrete Berechnungen ausführen und Vorbedingungen nicht selbst überprüfen. Diese Aufteilung in grobe semantischen Schichten ermöglicht es, eine wiederholte Definition und Implementierung kleinteiliger Aufgaben zu vermeiden. Außerdem führt dies dazu, dass komplexe Operationen und Abfragen mit wenig Aufwand formuliert und implementiert werden können, da Einzelheiten der Berechnungen abstrahiert werden.

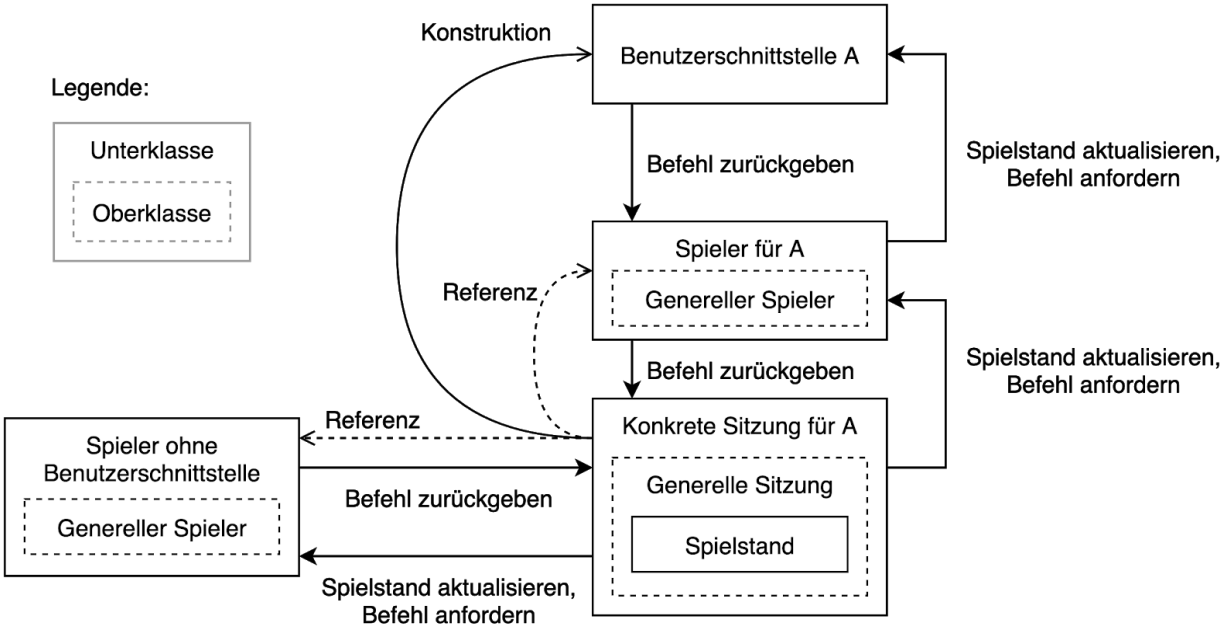
Umgebungsvariablen

Umgebungsvariablen werden durch die Klasse **`Environment`** repräsentiert und gelesen. Die einzelnen booleschen Umgebungsvariablen werden deklarativ durch einen Aufzählungstyp definiert und können somit einfach erweitert werden. Verschiedene Teile der Benutzerschnittstelle überprüfen dann, welche Umgebungsvariablen gesetzt sind und verändern entsprechend ihr Verhalten. Dieser Ansatz erlaubt eine kompakte Darstellung aller Benutzereinstellungen, die als Konsolenargument gesetzt werden können, und eine unkomplizierte Übergabe an spezialisierte Komponenten der Benutzerschnittstelle.

Interaktion durch Sitzungen

Die Spiellogik beschäftigt sich nicht mit der Aktualisierung der Daten der Benutzerschnittstelle sondern nur mit den Zustandsübergängen im Spiel selbst. Die Interaktion zwischen der Spiellogik und der Benutzerschnittstelle, sei es die GUI oder das Konsoleninterface, wird durch das Paket **`interaction`** gehandhabt, in dem maßgeblich die **`Session`** für die Verwaltung zuständig ist.

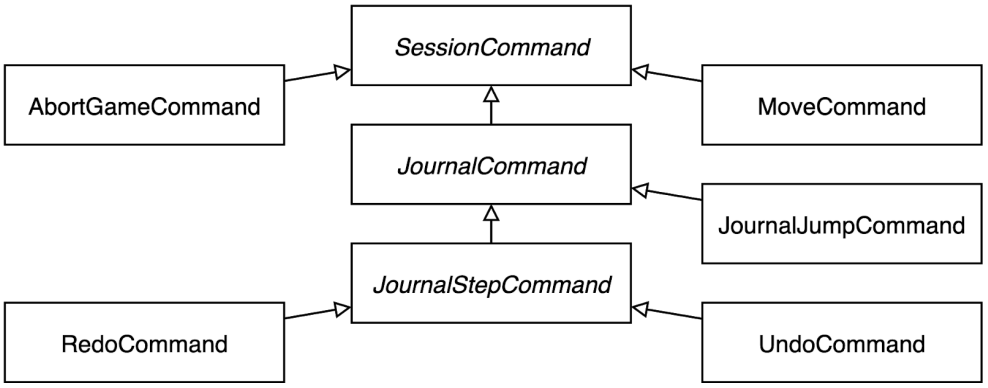
Durch eine Sitzung wird eine Spielsitzung dargestellt, die aus mehreren aufeinanderfolgenden aber unabhängigen Spielen bestehen kann. Die zwei Spieler in einer Sitzung sind durch **`Player`** modelliert, die von einer Sitzung über den aktuellen Spielstand informiert werden und dazu aufgerufen werden, einen Befehl anzugeben. Die Sitzung kontrolliert dadurch, wann Änderungen am Spielstand den Spielern mitgeteilt werden und wann ein Spieler einen Zug machen darf oder einen anderen Befehl senden kann. Alle Änderungen am Spielstand und die Kontrolle der Züge erfolgen zentral über die Sitzung, damit die Logik, die im Ablauf eines Spiels steckt, nicht dupliziert wird.



Hier wird beispielhaft eine Sitzung dargestellt, die für die Benutzerschnittstelle A konstruiert wurde und sowohl einen Spieler für A enthält als auch einen Spieler, der keine Benutzerschnittstelle hat. Die Sitzung bedient beide Spieler gleichermaßen und aktualisiert den Spielstand für beide Benutzer, nachdem ein Befehl ausgeführt wurde. Es ist wichtig anzumerken, dass die Spieler ihren Befehl für die Sitzung nicht selbst auf die Sitzung und den Spielstand anwenden, sondern sie den aktualisierten Spielstand von der Sitzung erhalten. Die Interaktion findet zwischen den allgemeinen Oberklassen statt, während die Instanzen von Unterklassen stammen, die das spezifische Verhalten für ihre Benutzerschnittstelle enthalten. Die Benutzerschnittstelle A kann also entweder die GUI oder die Konsole sein.

Sitzungsbefehle und ihre Vererbungsstruktur

Statt nur Züge von den Spielern zu bekommen, erhält die Sitzung Befehle von einem der beiden Spieler. Es gibt eine Reihe an verschiedenen Sitzungsbefehlen, die von den Spielern produziert werden können. Dabei darf nur der aktive Spieler, der an der Reihe ist, einen Befehl übergeben, der einen Zug enthält. Beide Spieler dürfen jedoch Befehle für das Vor- und Zurückgehen in der Zughistorie erzeugen. Welcher Spieler welche Befehle produzieren darf, wird durch Methoden an den Befehlen bestimmt. Im folgenden Diagramm werden die unterschiedlichen Arten von Sitzungsbefehlen dargestellt.

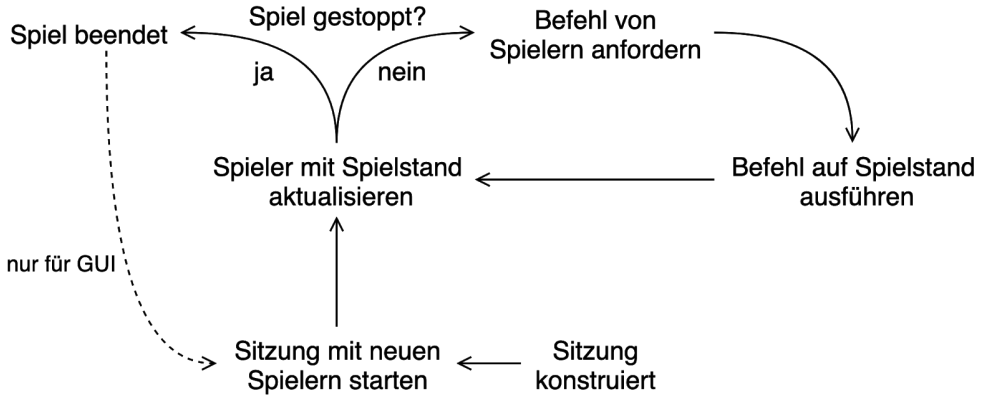


Im **`SessionCommand`** ist ein abstrakter Befehl für die Sitzung abgebildet. Von dieser Klasser erben die anderen Befehle und implementieren ihre jeweiligen Aktionen und Eigenschaften. Befehle werden von der Sitzung durch den Aufruf der **`applyTo`** Methode aktiviert. Die Implementierung dieser Methode führt den jeweiligen Befehl dann aus. Diese Befehlsstruktur ermöglicht es, indirekt Aufrufe von Methoden an der Sitzung über unterschiedliche Threads hinweg zu übertragen. Da die Player manchmal in separaten Threads arbeiten, muss verhindert werden, dass sie die Sitzung direkt aufrufen. Hierdurch führt die Sitzung die Änderungen an sich selbst durch, wenn sie einen Befehl ausführt.

Mit einem **`MoveCommand`** wird ein Zug auf dem Spielstand ausgeführt. Der **`AbortGameCommand`** wird benutzt, um der Sitzung zu signalisieren, dass das Spiel beendet werden soll, wenn zum Beispiel der Spieler in der GUI zurück zum Hauptmenü geht. Die Unterklassen von **`JournalCommand`** implementieren Befehle, die Operationen an der Zughistorie ausführen. Die Zughistorie selbst, wird im entsprechenden Abschnitt dokumentiert.

Ablauf einer Sitzung

Im folgenden Diagramm ist der Ablauf einer Sitzung dargestellt. Eine Sitzung kann eine unbegrenzte Anzahl von einzelnen Spielen mit jeweils anderen Spielern nacheinander durchführen. Am Anfang ist die Sitzung in einem Startzustand und wartet darauf, das erste Mal gestartet zu werden.



Sobald die Sitzung gestartet wurde, erstellt sie einen neuen Spielzustand und gibt diesen an die Spieler weiter. Daraufhin aktualisieren die Spieler ihre Anzeigen und ihren internen Zustand. Die Spieler sind jedoch so konstruiert, dass sie zwischen den Aktualisierungen

durch die Sitzung nicht auf den Spielstand zugreifen oder ihn modifizieren. Die Aktualisierung eines Spielers durch die Sitzung findet vor dem ersten Zug des Spiels, vor jeder Änderung des Spielstands und nach dem letzten Zug des Spiels statt.

Nach jeder Anwendung eines Zuges wechselt der aktive Spieler auf den jeweils vorher inaktiven Spieler. Der Spieler mit den weißen Figuren ist zuerst der aktive Spieler. Informationen über einen Spielstand beziehen sich immer auf den aktiven Spieler, der als nächstes einen Zug macht, welcher dann an die Sitzung zurückgegeben wird. Da beide Spieler vor jedem Befehl über den aktuellen Spielstand benachrichtigt werden, ist es von Bedeutung, auch Aussagen über den inaktiven Spieler zu machen. Die Berechnung des Zustandes für einen Spieler vor jeder Anwendung eines Befehls geschieht in **`TurnStatus`** und wird mit einem der Werte von **`TurnStatus`** zum Ausdruck gebracht. Es wird zum Beispiel angegeben, ob der Spieler im Schach ist oder ob er gewonnen hat.

Da auch der inaktive Spieler Befehle erteilen kann, solange es sich nicht um einen Befehl mit einem Spielzug handelt, fragt die Sitzung beide Spieler nach Befehlen in jedem Durchlauf des Zyklus. (Siehe Diagramm) Wenn ein Spieler einen validen Befehl übergeben hat, wird der andere Spieler benachrichtigt, dass er nun keinen Befehl mehr produzieren soll. Beim stoppen der Sitzung wird die Aufforderung, einen Befehl zu erzeugen, ebenfalls für Spieler zurückgezogen.

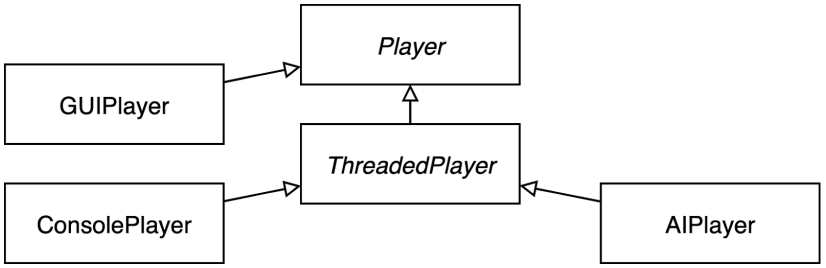
Deklarative Spielmodi

Ein Spielmodus beschreibt, welche Spieler an einem Spiel teilnehmen. Er ist dazu in der Lage, eine gegebene Sitzung mit den erforderlichen Spielern für diesen Modus zu starten. Dabei werden dem Spielmodus Erzeugerfunktionen für Spieler, die durch eine Benutzerschnittstelle kommunizieren, und für Spieler, die mit einer künstlichen Intelligenz Züge entscheiden, übergeben. Der Spielmodus benutzt diese Erzeuger, um passende Spieler zum Start der Sitzung bereitzustellen. Welche Art von Spielern konkret konstruiert werden bestimmt die jeweilige Sitzung.

Die Spielmodi werden, ähnlich zu den Umgebungsvariablen, deklarativ definiert und können daher leicht durch weitere Optionen erweitert werden. Ein Spielmodus wird nur durch eine Beschreibung in Textform und sein Verhalten bei der Erzeugung eines Paars von Spielern bestimmt.

Typen von Sitzungen und Spielern

Es gibt für beide Benutzerschnittstellen Unterklassen von **`Session`** und **`Player`**, die für die Interaktion mit ihren jeweiligen Benutzerschnittstellen verantwortlich sind. Sie bilden die Schnittstelle zwischen der puren Sitzung und einer konkreten an eine Benutzerschnittstelle gebundene Sitzung. Damit die Sitzung als logisches Konzept rein bleiben kann, müssen diese Unterklassen das gesamte spezifische Wissen über die Benutzerschnittstellen enthalten. Zur Übersicht sind hier die verschiedenen Spieler dargestellt:



Multithreading in Sitzungen

Beide Spieler können gleichzeitig Befehle an die laufende Sitzung geben, selbst wenn nur einer der Spieler aktiv ist. Damit beide Spieler für den Benutzer immer interaktiv bleiben, können Spieler durch **`ThreadedPlayer`** die Erzeugung von Befehlen für die Sitzung in einen separaten Thread auslagern. Der zuerst an die Sitzung übergebene Befehl wird ausgeführt und anschließend werden wieder beide Spieler nach einem Befehl gefragt. Es ist ohne Fehler möglich, dass Spieler synchron Befehle erzeugen, solange nicht beide Spieler synchron arbeiten. Dieses Verhalten wird aber nur zu Testzwecken benutzt.

Spieler für die KI

Das Spiel gegen die KI erfolgt durch den **`AIPlayer`**, der mit Zügen antwortet, die durch eine festgelegte Suchstrategie im **`MoveCalculator`** berechnet werden. Auch dieser Spieler arbeitet in einem separaten Thread, damit der Hauptthread nicht blockiert wird.

Mit der Angabe einer Suchstrategie wird bestimmt, mit welchem Algorithmus nach einem bestmöglichen Zug für den aktuellen Spielstand gesucht werden soll. Im Allgemeinen brauchen die Suchalgorithmen eine Stellungsbewertung, die eine numerische Abschätzung des strategischen Wertes eines Spielstandes für die KI darstellt. Dieser Wert wird durch **`MoveCalculator`** berechnet, der mit allen Figuren eines Spielstandes aus ihrem Materialwert und ihrem Positionswert einen Gesamtwert bestimmt, in dem eigene Figuren positiv und gegnerische Figuren negativ bewertet werden.

Neue Suchalgorithmen lassen sich durch weitere Unterklassen von **`SearchStrategy`** implementieren, sodass eine beliebig komplexe KI unkompliziert integriert werden kann. Die Suchtiefe und die Suchstrategie des KI-Spielers kann bei seiner Konstruktion angegeben werden und hat je nach Art der Suchstrategie eine unterschiedliche Auswirkung.

Die einfache KI wird durch die Strategie **`ShallowEvaluation`** implementiert, die aus den möglichen Zügen den Zug auswählt, der den Spielstand mit dem besten Wert erzeugt. Die Suche dieser KI ist also nur einen Zug tief, sodass keine Voraussicht möglich ist. Die Einstellung einer Suchtiefe hat bei dieser Strategie keinen Effekt, weil sie immer nur einen Zug tief sucht.

Hingegen verwendet die komplexe KI in **`FixedAlphaBeta`** eine einfache Alpha-Beta-Suche mit einer variablen Suchtiefe. Diese Suche wählt als determinierter Algorithmus immer den gleichen optimalen Zug aus. Der Spielstand wird erst beim Erreichen der Suchtiefe oder eines Spielendes bewertet, daher der Name “Fixed”. Um die Effizienz der Suche leicht zu

steigern, wird ein eigener Zugakkumulator für die Suche verwendet, der keine weiteren Züge akzeptiert, wenn der Alpha-Beta-Algorithmus die Suche für den aktuellen Teilbaum abbricht. Statt alle möglichen Züge eines Spielstandes zu berechnen und dann zu iterieren, werden mit dieser Methode nach dem Abbruch der Suche in einem Spielstand keine weiteren Züge erzeugt.

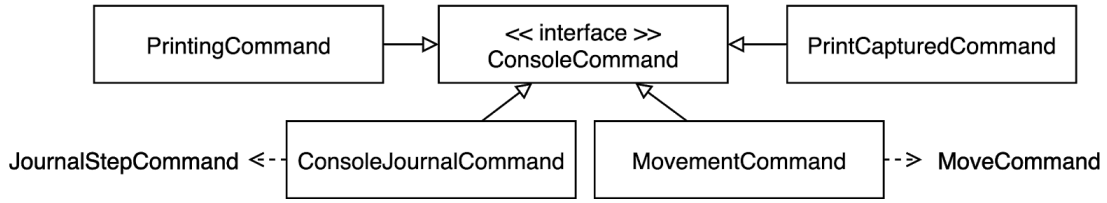
Benutzerschnittstelle Konsoleninterface

Sitzung und Spieler für das Konsoleninterface

Die konkrete Sitzung für das Konsoleninterface wird in **`ConsoleSession`** realisiert. Diese Sitzung fragt den Benutzer vor dem Start des Spiels nach dem gewünschten Spielmodus und startet sich dann mit den entsprechenden Spielern. Für den menschlichen Spieler wird der **`ConsolePlayer`** verwendet, der Befehle vom Benutzer aus der Konsole liest und das Schachbrett vor jedem Zug als Text mit optionalen UTF-8 Symbolen und Farben ausgibt. Damit das Schachbrett nicht doppelt angezeigt wird, wenn zwei Spieler über die Konsole spielen, zeigt nur der aktive Spieler das Spielbrett in solchen Situationen an.

Dieser Spieler wartet auf Benutzereingaben in einem separaten Thread, damit es nicht zu einer Blockierung kommt. Weil es nicht praktikabel ist, das Warten auf Daten von der Konsole zu Unterbrechen, reagiert dieser Spieler nicht darauf, wenn er von der Sitzung dazu aufgefordert wird, seine Berechnung bzw. sein Warten zu unterbrechen. Da in der Konsole die Eingabe jedoch unabhängig von der Ausgabe ist, kann dieser Spieler einen Befehl auch im nächsten Zyklus noch zurückgeben, ohne vorher wieder angefragt zu werden.

Die Interaktion zwischen dem **`CommandInterface`**, was Benutzereingaben von der Konsole liest und analysiert, passiert ähnlich wie zwischen Spielern und der Sitzung mithilfe von Befehlen, hier Konsolenbefehlen. Die Befehle implementieren jeweils **`ConsoleCommand`** und produzieren optional einen Sitzungsbefehl, der vom Spieler an die Sitzung weitergegeben wird. Diese Symmetrie zwischen den beiden Sätzen von Befehlen erlaubt eine elegante Umwandlung von Befehlen im Konsoleninterface in Sitzungsbefehle. Bei falschen Benutzereingaben können Konsolenbefehle auch keinen Sitzungsbefehl produzieren, sondern stattdessen mit der direkten Ausführung eines **`PrintCommand`** eine Fehlermeldung ausgeben. Im folgenden Diagramm sind die Konsolenbefehle zusammen mit dem Sitzungsbefehle, die sie produzieren können, dargestellt.



Ausgabe der Spielstände

`BoardPrinter` fasst die Methoden zur Darstellung des aktuellen Spielstands in einer Klasse zusammen, in der auf die Erweiterbarkeit geachtet wurde. Zum Beispiel ist der **`BoardPrinter`** in der Lage, Schachbretter bis zu einer Größe von 26*26 auszugeben, was für Schachvarianten mit anderen quadratischen Spielfeldern nützlich ist. Der Benutzer kann entscheiden, ob er die Schachfiguren als Buchstaben angezeigt bekommen möchte oder als UTF8-Symbole und ob ein farbiger Schachbretthintergrund in der Konsole ausgegeben werden soll (siehe Bedienungsanleitung).

Die geschlagenen Figuren werden in sortierter Reihenfolge nach Wert der Figuren und Farbe angezeigt. Diese Sortierung wird durch Implementierung eines **`Comparator`** realisiert, damit die Logik zum Vergleichen von Figuren abstrakt und von der Benutzerschnittstelle getrennt bleibt.

Auswahl von Spielmodi

Zur Auswahl der Spielmodi werden die Optionen für alle Spielmodi auf der Konsole ausgegeben und die Eingabe einer der Abkürzungen erwartet, um das Spiel mit dem gewählten Spielmodus zu starten. Hierbei wird nicht im Konsoleninterface festgelegt welche Spielmodi es gibt oder wie sie heißen, da die Spielmodi selbst ihre Beschreibung und Abkürzung in Form einer Listendarstellung preisgeben. Falsche Eingaben bei dieser Auswahl führen zur Ausgabe eines Fehlers, damit der Benutzer es erneut versuchen kann.

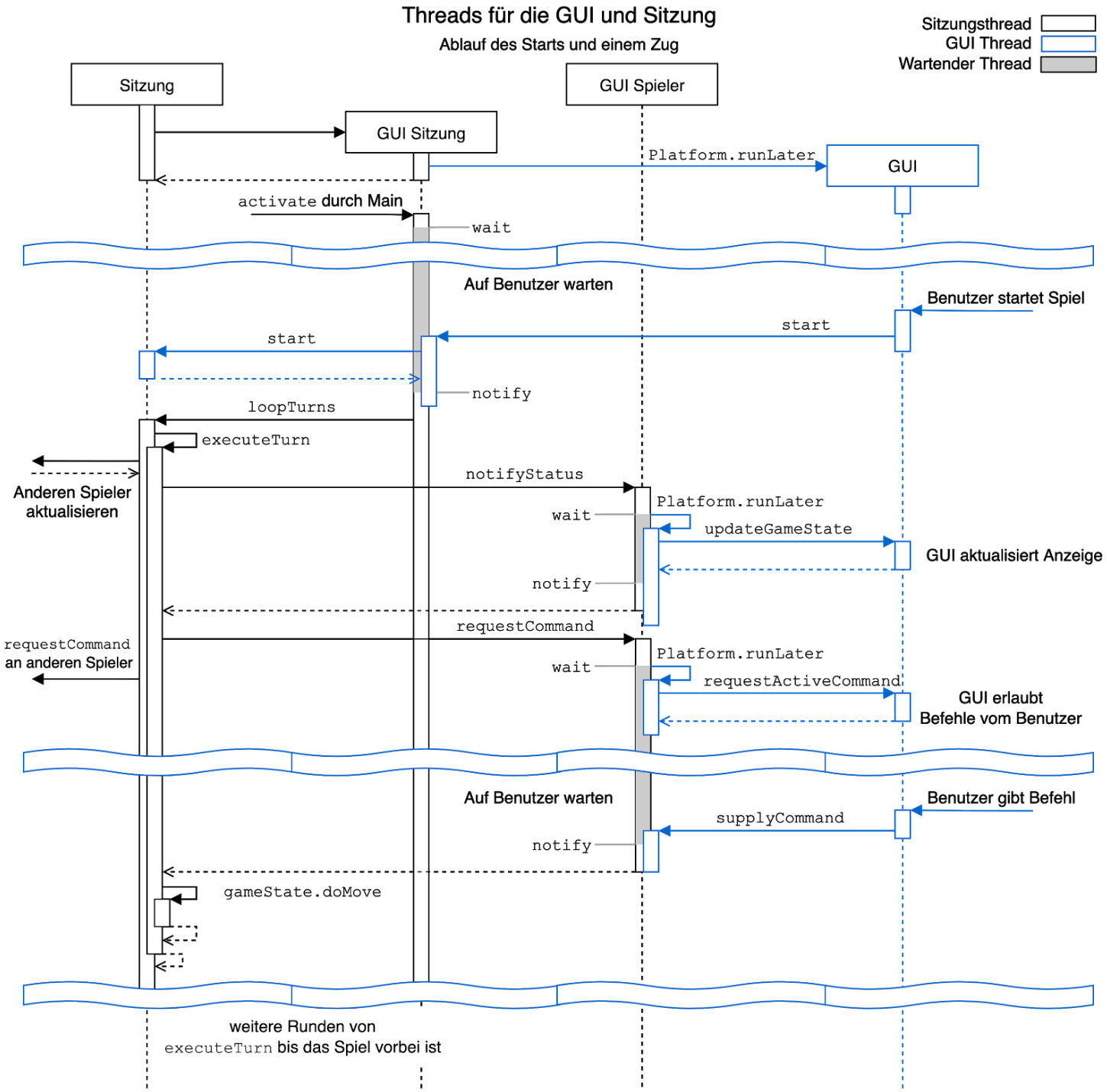
Benutzerschnittstelle GUI

Sitzung und Spieler für die GUI

In folgendem Sequenzdiagramm wird der Ablauf des Starts einer Sitzung für die GUI und das Ausführen eines Zuges dargestellt. Das Ende eines Spiels und der Neustart sind hier nicht ausgeführt, belaufen sich aber auf die gleichen Mechanismen, wie sie hier zu sehen sind. Die Sitzung für die GUI wird durch **`GUISession`** realisiert. Sie übernimmt zusammen mit dem **`GUIPlayer`** die asynchrone Kommunikation zwischen dem normalen Sitzungsablauf und der GUI. Die GUI wird zum Start der Sitzung in einem separaten Thread gestartet, damit die GUI, selbst wenn es zu synchronen Berechnungen in der Sitzung kommt, interaktiv bleibt.

Der **`GUIPlayer`** erweitert den regulären **`Player`** und nicht **`ThreadedPlayer`**, weil die asynchrone Interaktion mit der GUI durch JavaFX passieren muss und nicht mit den Mechanismen aus **`ThreadedPlayer`**, die die anderen asynchronen Spieler benutzen.

Die Sitzung erzeugt die GUI nachdem sie konstruiert wurde und wird erst gestartet, wenn der Benutzer durch die GUI ein neues Spiel anfängt. Bis sie gestartet wird, wartet sie darauf, durch die GUI aufgewacht zu werden. Wenn die Sitzung einen Befehl beim GUI-Spieler

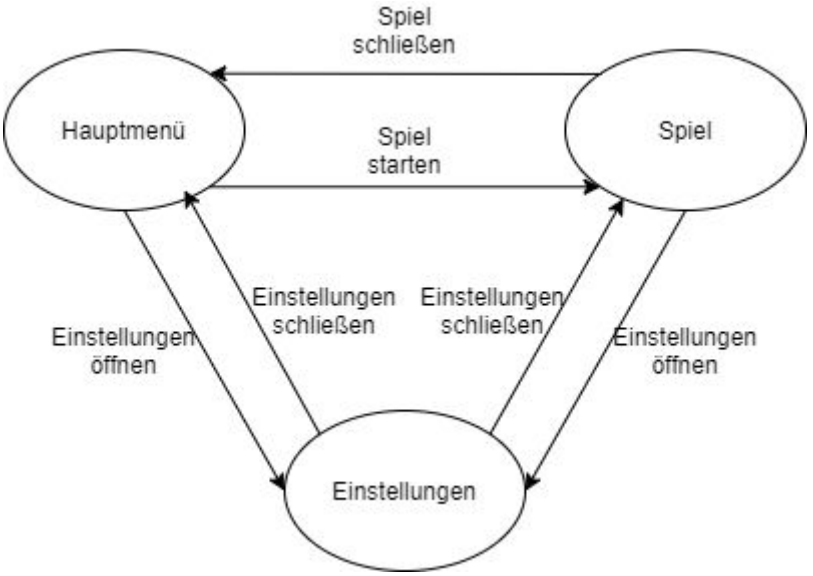


anfordert, ruft er die GUI in ihrem separaten Thread auf und sorgt somit dafür, dass der Benutzer einen Befehl eingeben kann. Dann wartet die Sitzung auf einen Befehl von einem der Spieler. Wenn ein valider Zug durch den Benutzer ausgeführt wurde, wird der GUI-Spieler aktiviert und gibt den Zug als Befehl an die Sitzung weiter. Ein ähnlicher Prozess findet beim Aktualisieren der GUI nach einem Zug statt.

Die Instanz der Sitzung in der GUI wird nicht für jedes Spiel neu erstellt, weil innerhalb einer Ausführung des Programms mehrere Spiele möglich sind. Daher wird eine Sitzung nach dem ersten Start aus dem laufenden Zustand neu gestartet oder nach Beendigung des

letzten Spiels gestartet. Für einen Neustart während eines laufenden Spiels wird der Wartezustand der Spieler unterbrochen, damit ein neues Spiel initialisiert werden kann.

Komponenten der GUI und ihre Interaktion



Der grundlegende Ablauf der GUI ist verhältnismäßig einfach. Wenn der Benutzer das Programm startet, öffnet sich das Hauptmenü und es wird der **`MainMenuView`** als Wurzel der Szene gesetzt. Von hier aus können, wie in der Abbildung zu sehen ist, entweder die Einstellungen geöffnet oder ein neues Spiel gestartet werden. Wenn der Benutzer die Einstellungen aus dem Hauptmenü aufruft, kehrt er durch das Schließen der Einstellungen auch zum Hauptmenü zurück. Nachdem ein Spiel gestartet wurde, hat der Benutzer die Möglichkeit, das Spiel wieder zu verlassen und ins Hauptmenü zurückzukehren oder er kann während des Spiels die Einstellungen öffnen. Die Einstellungen werden unabhängig vom Spiel gehalten, damit sie zwischen Spielen erhalten bleiben solange der Benutzer das Programm nicht schließt. Im Folgenden wird der Aufbau der GUI-Komponenten beschrieben.

Hauptmenü

Das Hauptmenü hat eine sehr simple Struktur. Es besteht nur aus einem Controller und einem View. An dieser Stelle des Programms wird kein Model benötigt, da das Hauptmenü keinerlei Daten verwaltet und nur direkt auf die Eingaben des Benutzers reagiert.

Der **`MainMenuController`** hat die Aufgabe, ein neues Spiel zu starten und die Einstellungen zu öffnen. Der Controller reagiert darauf, wenn der Benutzer auf eines der drei zu sehenden Bilder klickt und startet dann die Sitzung mit dem entsprechenden Spielmodus. Mithilfe eines Sliders kann außerdem die Stärke der AI eingestellt werden. Beim Klick auf eines der Bilder für die Spielmodi wird der View vom Controller dazu aufgerufen, ein Pop-up zur Auswahl der Spielerfarbe anzuzeigen. Daraufhin startet der Controller eine neue Sitzung mit dem gewählten Modus. Die Sitzung erstellt einen neuen Spielstand und startet den

Ablauf des Spiels. Daraufhin setzt der Controller eine Instanz des **`GameView`** als Wurzel der Szene, um das neue Spiel anzuzeigen.

Einstellungen

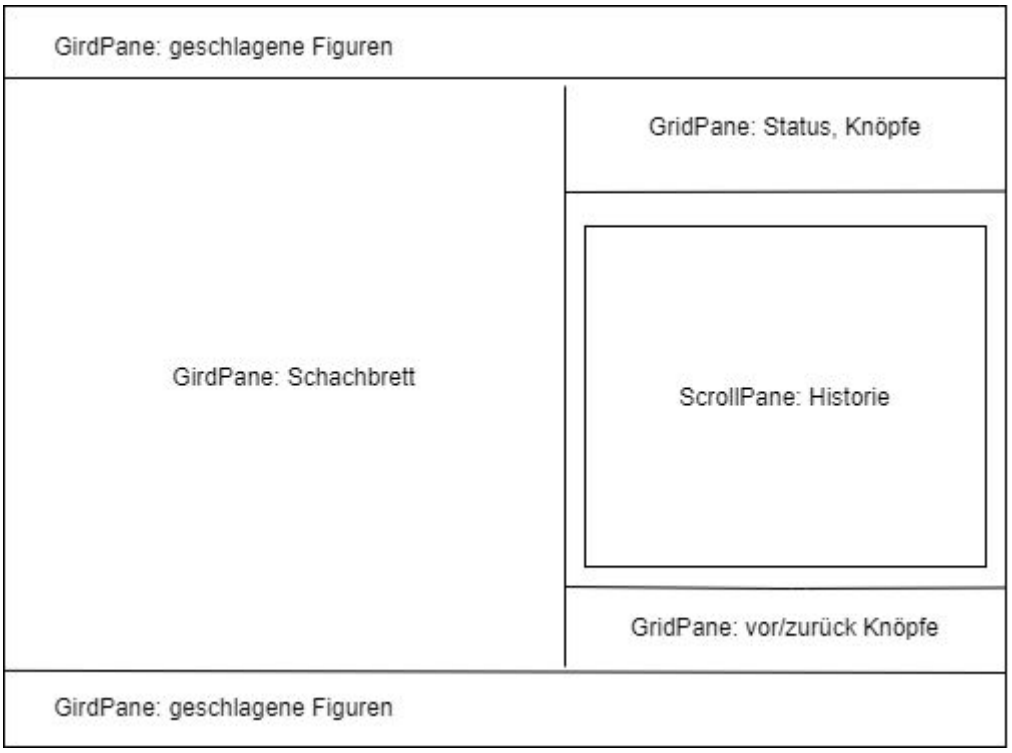
Die Einstellungen verwenden das MVC Modell. In der **`SettingsModel`** Klasse werden die Settings als Enums in einer EnumMap gespeichert. Dies ermöglicht ein einfaches Hinzufügen von weiteren Optionen. Im momentanen Spiel werden nur Optionen gespeichert, die wahr oder falsch sein können, die **`SettingsModel`** Klasse ermöglicht jedoch auch ein leichtes Hinzufügen von nicht boolschen Werten. Des Weiteren wird das Singleton Pattern verwendet, da immer nur eine Instanz der Settings über das komplette Spielgeschehen benötigt wird. Hierdurch ist ein Zugreifen auf die Settings von verschieden Teilen der GUI sehr einfach. Da standardmäßig kein Schalter in JavaFX zur Verfügung steht, wurde für die Einstellungen ein **`SwitchButton`** erstellt.

Spieldarstellung

Bei der Spieldarstellung haben wir uns für das MVVM Modell entschieden, da uns die klare Trennung zwischen den Daten im Model, der Verarbeitung der Daten im ViewModel und der Anzeige der Daten im View als klare Struktur gut gefiel. Die allgemeine Interaktion mit dem Benutzer bezüglich des Spiels wird durch **`GameModel`**, **`GameViewModel`** und **`GameView`** gehandhabt, während spezifische Komponenten hierarchisch in eigene MVVM Strukturen unterteilt sind.

Das **`GameModel`** interagiert mit der **`Session`** und speichert die anzuzeigenden Daten in Observables ab. Aktualisierungen an den Daten werden vorgenommen wenn Züge getätigt werden. Bei Veränderungen dieser Daten reagiert das **`GameViewModel`** entsprechend. Die Daten werden an dieser Stelle noch nicht weiter für den **`GameView`** aufbereitet, da diese Verarbeitung im **`GameViewModel`** stattfindet. Das **`GameModel`** wird vom **`GameViewModel`** aufgerufen, falls der Benutzer ein Feld auf dem Schachbrett angeklickt hat, und sendet dann gegebenenfalls einen neuen Zug an die **`Session`**.

Das **`GameViewModel`** ist zuständig für die Datenaufbereitung und das Verarbeiten der UI Ereignisse. Es bindet die vom View benötigten Daten mit Hilfe von JavaFX Properties an das View und benutzt Listeners um auf Änderungen im **`GameModel`** zu reagieren.



Hier die die Aufteilung des **`GameView`** dargestellt, der für die Anzeige der konkreten Daten zuständig ist. Für das Schachbrett bietet sich offensichtlich besonders die GridPane Darstellung an, da das Schachbrett ein Gitter ist und somit gut in die Struktur der GridPane passt. Um schnell neue Stellungen anzeigen zu können und Züge nicht im View verarbeiten zu müssen, werden zum Start des Programms einmal an jeder Stelle des Schachbretts alle möglichen Figuren geladen. Die Sichtbarkeit der Figuren wird dann mithilfe von Bindings und Properties im **`GameViewModel`** gehandhabt. Genauso werden auch die geschlagenen Figuren erstellt. Somit bleibt der **`GameView`** frei von Spiellogik.

Spiellogik

Abbildung des Spielzustandes

Der Spielzustand wird durch die Klasse **`GameState`** und **`Board`** abgebildet und besteht aus Figuren, deren Position und Farbe sowie weiteren Informationen über die Bewegungen der Figuren in der Vergangenheit. Es werden außerdem Daten über die Anzahl der gesamten Züge und der Anzahl der Züge seit der letzten Bewegung eines Bauern oder dem Schlagen einer Figur festgehalten. Welcher Spieler dran ist wechselt mit jedem Zug und wird hier als *Farbe* bezeichnet.

Informationen über das Spiel und seinen Verlauf werden in **`GameState`** gespeichert, während die Daten über das Spielbrett selbst und die Figuren in **`Board`** gespeichert werden. Durch diese Trennung vermischen sich die Zuständigkeiten der beiden Klassen nicht. Obwohl beide Klassen viel interagieren sind ihre Aufgaben verschieden genug, um auch als separate Klassen sinnvoll zu funktionieren. Es kann transparent auf das Spielbrett

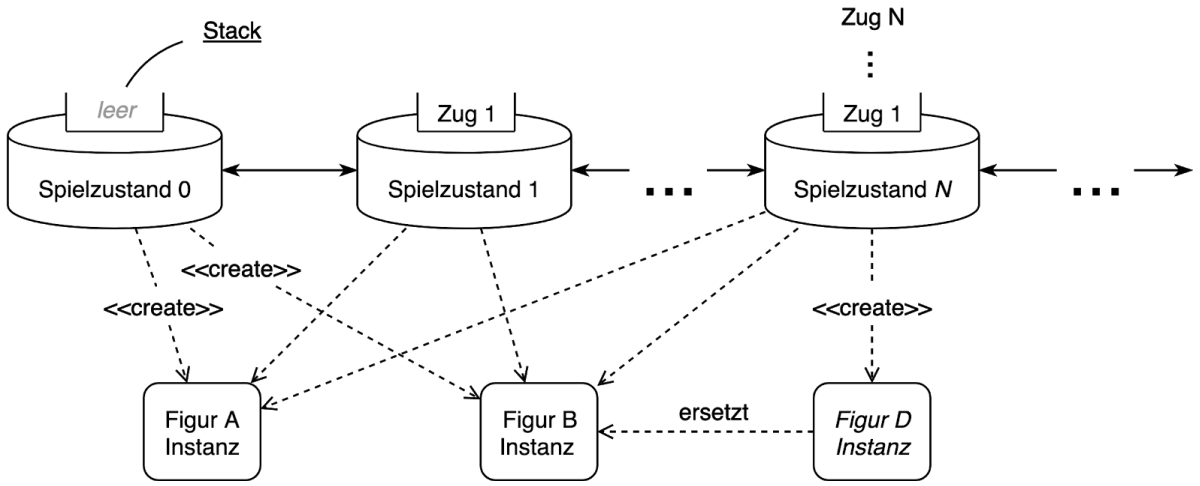
zugegriffen werden, weil manchmal nur Informationen über das Spielbrett selbst benötigt werden und nicht über den gesamten Spielstand.

Zustandslose Instanzen für Figuren auf dem Spielbrett

Damit die Positionen der Figuren nicht in den Instanzen der Figuren gespeichert werden müssen, werden die Figuren in einer Hashtabelle gespeichert, in der mit dem laufenden Index der Spielfelder als Schlüssel alle Figuren an einer Position gespeichert werden. Der Index zählt alle Felder des Schachbretts beginnend mit dem ersten Feld oben links zeilenweise durch. Diese Datenstruktur ist effizient, da es fast nie notwendig ist, für eine bestimmte Figur ihre Position zu bestimmen, sondern immer nur die Figur an einer bestimmten Position interessant ist. Die Instanzen der Figuren werden zustandslos genannt, weil sie keinen Zustand enthalten, der sich nach ihrer Instanziierung ändern kann. Außerdem ermöglichen diese zustandslosen Instanzen der Figuren ein effizientes Ändern des Spielzustandes, weil die Instanzen der Figuren nicht modifiziert werden müssen.

Die Instanzen der Figuren stellen also die Identität der Figuren dar, weil sie über die gesamte Ausführung des Programms hinweg nur einmal erstellt werden müssen. Diese Darstellung ergibt auch inhaltlich Sinn, weil bei einem echten Schachspiel ebenfalls die Figuren nicht bei jedem Zug geändert werden, sondern nur ihre Position wechseln.

Wenn eine Figur eine Operation ausführt, für die die aktuelle Position der Figur benötigt wird, dann wird dieser Operation die Position der Figur beim Aufruf mitgegeben. Die Position einer Figur ist außerdem immer im Kontext des Aufrufs verfügbar, weil alle Operationen, die von außerhalb des Spielstands aufgerufen werden, sich auf bestimmte Positionen und nicht auf eine bestimmte Figur beziehen.

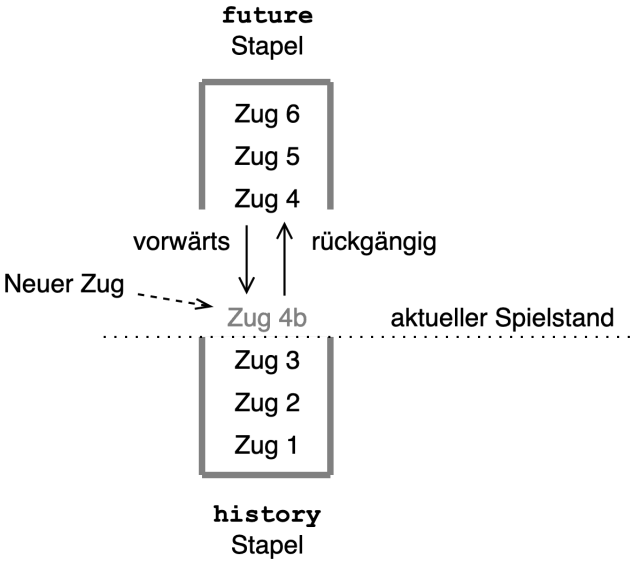


In diesem Diagramm werden die Instanzen der Figuren und die Referenzen der Spielstände auf die Figuren verbildlicht. In Einzelfällen werden neue Instanzen erstellt, wenn zum Beispiel eine Umwandlung eines Bauern stattfindet. Die Zughistorie ist hier vereinfacht dargestellt.

Zughistorie

Damit keine aufwendigen Kopien von Spielständen erzeugt werden müssen, ist es möglich einen Zug auf einen Spielstand anzuwenden und dann wieder rückgängig zu machen, sodass genau der vorherige Spielstand wiederhergestellt wird. Dieses Verfahren simuliert die Erzeugung von Kopien des Spielstandes, weil Operationen, die auf einer temporären Kopie ausgeführt werden würden, stattdessen nach der Anwendung eines Zuges aber vor der Wiederherstellung des letzten Spielstandes ausgeführt werden. Der Spielstand erlaubt nur das Rückgängigmachen des zuletzt angewandten Zuges, weil frühere Züge außerhalb der Reihenfolge rückgängig zu machen, den Spielstand in einen undefinierten Zustand bringen kann. Im Abschnitt zu Zügen wird hierauf weiter eingegangen.

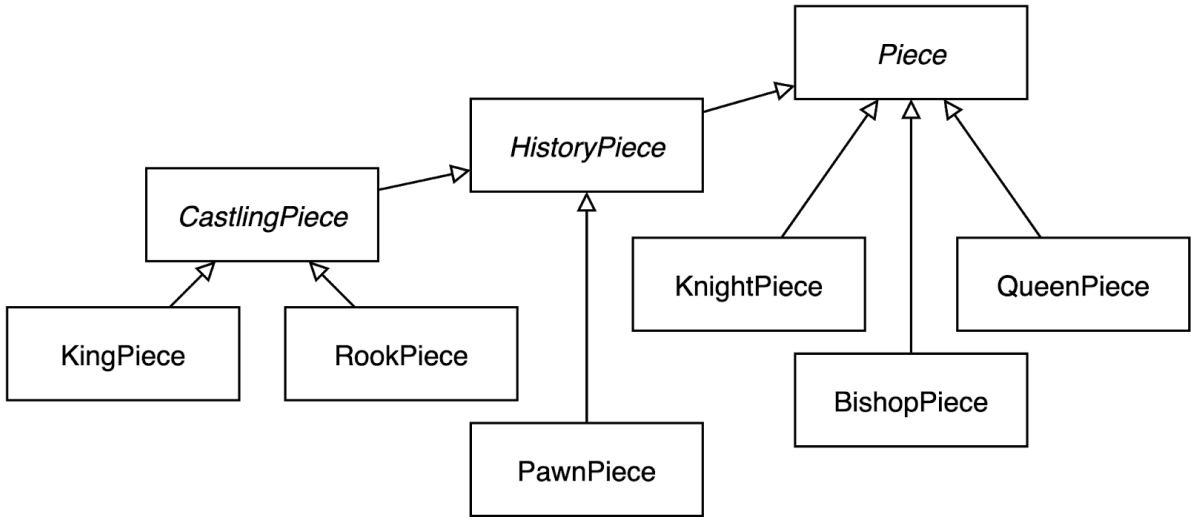
Für Berechnungen auf dem Spielstand werden die Züge nicht gespeichert, sondern direkt nachdem die gewünschte Berechnung auf dem geänderten Spielstand beendet wurde wieder rückgängig gemacht. Wenn ein Zug jedoch von einem Spieler getätigt wird, dann speichert die Zughistorie in `MoveJournal` den Zug. `MoveJournal` erlaubt das Rückgängigmachen und Wiederholen von rückgängig gemachten Zügen, indem zwei Stapel an Zügen unterhalten werden. Ein Stapel ist für die Vergangenheit der angewandten Züge zuständig, während ein Zweiter für die rückgängig gemachten Züge der Zukunft ist.



In diesem Diagramm werden die beiden Stapel und die Bewegung der Züge zwischen ihnen deutlich. Wenn ein Zug rückgängig gemacht wird, dann wird er auf den Stapel für die Zukunft gelegt. Zum Wiederholen eines Zuges wird er wieder von diesem Stapel genommen und auf den Spielstand angewandt, sowie auf den Stapel für die Vergangenheit gelegt. Damit Züge auf dem Stapel für der Zukunft nicht falsch sind, werden sie gelöscht, sobald ein neuer Zug von außen zur Zughistorie hinzugefügt wird.

Vererbungsstruktur der Figuren

Es gibt für jeder der verschiedenen Spielfiguren eine Unterklasse der abstrakten Klasse **`Piece`**, die grundlegende Funktionen enthält, die von mehreren oder allen Unterklassen verwendet werden. Es kann keine Instanz dieser Oberklasse erstellt werden weil es sich immer um eine bestimmte Figur handelt, die eine eindeutige Instanz und einen Typ auf dem Spielfeld hat. Die Unterklassen unterscheiden sich maßgeblich darin, wie sie mögliche Züge generieren. Es gibt außerdem noch die abstrakten Unterklassen **`HistoryPiece`** und **`CastlingPiece`**, die weitere Funktionen bezüglich ihrer Bewegungshistorie zusammen mit dem Spielstand **`GameState`** bereitstellen.



In diesem Diagramm ist die Vererbungshierarchie der verschiedenen Typen von Figuren dargestellt, wobei die Namen abstrakter Klassen schräg gestellt sind. Der Spielzustand hält für Figuren, die von **`HistoryPiece`** erben, den letzten Zug fest an dem sie teilgenommen haben. Der Grund dafür ist, dass diese Figuren bestimmte Züge wie die Rochade nur machen dürfen, wenn sie sich bisher nicht oder im Falle des En Passant, nicht im letzten Zug, bewegt haben. **`CastlingPiece`** stellt eine einfache Abstraktion des Dienstes von **`HistoryPiece`** bereit, der von Figuren, die an der Rochade beteiligt sind, genutzt wird.

Wir haben uns für die Darstellung von verschiedenen Typen von Figuren für Unterklassen entschieden, weil es das Verhalten der Figuren besser unterteilt, als wenn in undifferenzierten Instanzen ein Typfeld zur Unterscheidung der Figuren genutzt wird.

Weitere Teile des Zustandes

Bei der Erstellung von Instanzen von Figuren wird mit einem laufenden Zähler den Figuren eine eindeutige Identifikationsnummer zugeteilt, die dazu dient, sie kollisionsfrei in der Bewegungshistorie zu identifizieren. Es wird außerdem eine Liste der bereits geschlagenen Figuren geführt. Es werden auch die Positionen der beiden Könige separat festgehalten, damit überprüft werden kann, ob ein bestimmter Zug den König in Schach versetzt oder das Spiel durch ein Schachmatt beendet ist. Die Bewegungshistorie wird für jedes

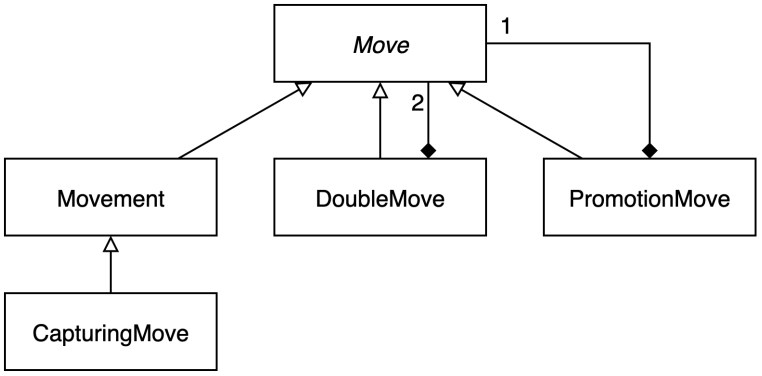
`HistoryPiece` als Stack gespeichert, damit auf vorherige Positionen zurückgegriffen werden kann.

Berechnung von Zügen

Komposition von Zügen und ihre Vererbungsstruktur

Züge beschreiben mögliche Veränderungen des Spielstandes und werden durch die abstrakte Klasse **`Move`** und deren Unterklassen abgebildet. So wie bei den Klassen für die Figuren ist hier die generelle Oberklasse abstrakt, weil alle Züge eine spezifische Änderung darstellen und daher zwar ein geteiltes Grundverhalten haben, aber differenziert implementiert werden. Die Figuren konstruieren mögliche Züge in ihren verschiedenen Formen für die verschiedenen Arten von Zügen.

Einfache Bewegungen werden mit der Klasse **`Movement`** ausgedrückt, die einen Zug durch eine Start- und Endposition auf dem Spielbrett ausdrückt. Züge, die Figuren schlagen, werden mit **`CapturingMove`** dargestellt. Die Klassen **`PromotionMove`** und **`DoubleMove`** enthalten selbst wieder eine bzw. zwei Züge, weil sie keine Spezialisierung eines vorhandenen Zuges sind, sondern eine Zusammensetzung von anderen Zügen. Diese Komposition von Zügen erlaubt es sehr einfach neue Arten von Zügen zu konstruieren und zusammenzusetzen, weil für neuartige Konstruktionen keine neuen Strukturen erstellt werden müssen sondern die existierende Klassen zusammen benutzt werden können.



In diesem Diagramm wird dargestellt, wie Züge voneinander erben und einander teilweise auch rekursiv enthalten können. Hierbei entsteht eine Komposition von ein oder zwei Zügen in **`PromotionMove`** und **`DoubleMove`**. In der Oberklasse Move werden mehrere Methoden definiert, die Informationen über einen bestimmten Zug liefern und dazu verwendet werden, einen Zug mit einer vom Benutzer eingegebenen Bewegung zu vergleichen oder zu testen, ob ein Zug ein Feld auf dem Spielbrett angreift. Bei den Unterklassen, die andere Züge enthalten, greifen diese Methoden auch auf die Informationen der Unterklassen zu.

Alle Züge können Informationen darüber liefern, welche Bewegung sie prinzipiell repräsentieren, damit für den **`FulfillmentFilter`** einfach festgestellt werden kann, ob ein Zug durch eine vom Benutzer angegebene Bewegung beschrieben wird. Bei einfachen Zügen ist diese symbolische Bewegung gleich der eigentlichen Bewegung. Dahingegen wird

die symbolische Bewegung bei **`DoubleMove`** und **`PromotionMove`** entweder automatisch aus den Teilzügen berechnet oder in bestimmten Fällen durch den Konstruktor gegeben.

Beschränkte Suche nach möglichen Zügen mit Zugakkumulatoren

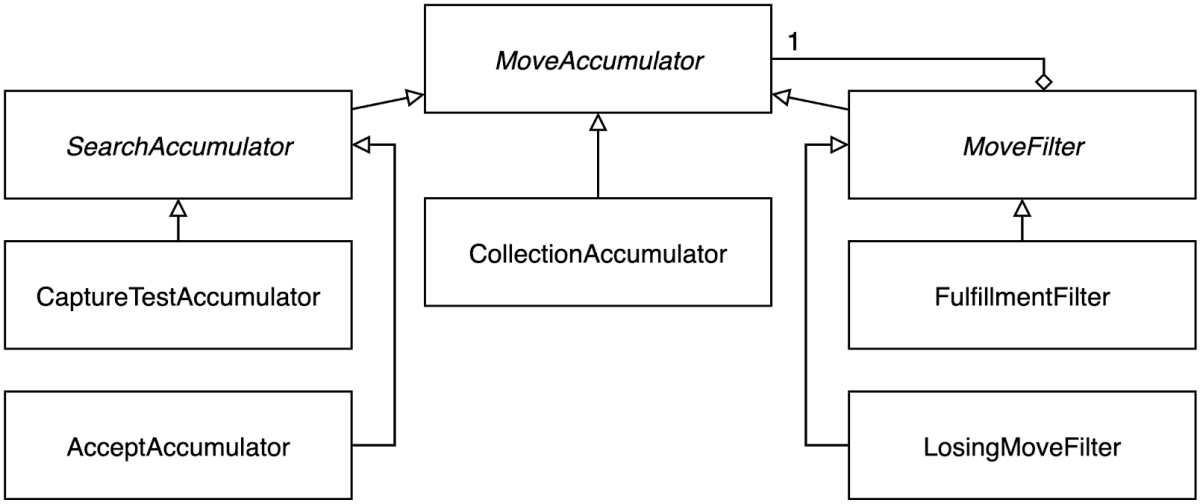
Um alle Züge zu finden, die in einer bestimmten Stellung möglich sind, müssen für alle Figuren alle ihre möglichen Züge gefunden werden. Dafür iteriert die Spiellogik über die Figuren und lässt alle Figuren ihre möglichen Züge generieren. Wie jede Figur Züge generiert hängt vom Typ der Figur, der Position der Figur und allen anderen Figuren auf dem Spielbrett ab. Bei manchen Figuren hängen die möglichen Züge auch vom letzten Zug der Figur ab.

Da die Instanzen der Figuren ihre Position und Bewegungshistorie nicht selbst speichern, wird ihre Position und eine Referenz zum Spielzustand zur Berechnung der Züge übergeben. Die möglichen Züge werden nicht vom Aufruf der Methode **`Piece.accumulateMoves`** zurückgegeben, weil dann oft Listen vereinigt und damit auch kopiert werden müssten. Stattdessen wird eine Instanz eines sogenannten Zugakkumulators, der von der Klasse **`MoveAccumulator`** erbt, übergeben. Dieser Zugakkumulator enthält eine Referenz zum Spielstand und die Position der Figur für die Züge berechnet werden. Die konkrete Implementierung der Zugberechnung in jeder Figur berechnet die Züge, die für den bestimmten Typ der Figur möglich sind und fügt sie dann dem Zugakkumulator hinzu.

Während der Berechnung der Züge greift die Unterklasse einer spezifischen Figur auf Methoden ihre Oberklassen zu, weil viele Operationen in dieser Berechnung sehr ähnlich sind. Zum Beispiel wird das Berechnen von Zügen in einer geraden Linie ohne Unterbrechung mit **`accumulateRayMoves`** durchgeführt. In Piece gibt es zur Berechnung von Zügen einige Methoden, die wie in der Einführung beschrieben abnehmende Grade der Abstraktion haben und sich gegenseitig aufrufen. Wenn eine bestimmte Figur jedoch für die Berechnung eines Zuges kein Standardverhalten braucht, kann sie auch kleinteilige und weniger abstrakte Methoden aufrufen. Diese Struktur minimiert die Duplikation der Implementation von Regeln und Verhaltensweisen.

Im einfachsten Fall sammelt der Zugakkumulator alle berechneten Züge in einer Liste, damit sie später ausgewertet oder angezeigt werden können. Jedoch werden manchmal gar nicht alle möglichen Züge benötigt, da oft nur geprüft werden muss, ob ein bestimmtes Feld angegriffen wird oder ob der König im Schach steht. Für diese unterschiedlichen Verhalten bei der Iteration über alle möglichen Züge noch während ihrer Berechnung gibt es einige verschiedene Zugakkumulatoren.

Vererbungsstruktur von Zugakkumulatoren



In diesem Diagramm ist die Vererbungsstruktur der Zugakkumulatoren dargestellt. Die abstrakte Oberklasse **`MoveAccumulator`** definiert Methoden und Funktionen, die generell für die Berechnung von Zügen benötigt werden, wie eine Referenz auf den Spielzustand und die Position der aktuellen Figur, für die Züge berechnet werden. Generell werden berechnete mögliche Züge mit **`addMove`** zu einem Zugakkumulator hinzugefügt. Dann entscheidet der Zugakkumulator, je nachdem was seine Aufgabe ist, was mit dem berechneten Zug getan wird.

Es werden außerdem Methoden definiert mit denen die Unterklassen dem Spielzustand und den Figuren signalisieren können, ob die Berechnung von weiteren Züge stattfinden soll und ob die Berechnung von Zügen, die keine Figuren schlagen, stattfinden soll. Diese Methoden existieren, um die Berechnung der möglichen Züge einzuschränken und zu beschleunigen, wenn der Akkumulator keine weiteren Züge oder keine weiteren nicht schlagenden Züge braucht.

- Die abstrakte Klasse **`SearchAccumulator`** dient dazu, Züge nur so lange zu berechnen, wie eine Bedingung, die von implementierenden Unterklassen definiert wird, noch nicht erfüllt ist. Wenn die Bedingung erfüllt wurde, wird die Berechnung weiterer Züge angehalten und keine weiteren Züge akzeptiert. Der erfüllende Zug wird nach der abgeschlossenen Berechnung aus dem Zugakkumulator zur Weiterverarbeitung zurückgegeben.
- Der **`AcceptAccumulator`** ist die einfachste Implementierung des **`SearchAccumulator`** und stellt nur fest, ob überhaupt ein Zug möglich ist. Er wird verwendet um zu bestimmen ob ein Spieler patt ist. Außerdem kann er als innerster Zugakkumulator in einer Filterkette stehen, um einen einzigen akzeptierten Zug aufzunehmen. Die Bedingung dieses Zugakkumulators ist immer erfüllt, damit ein Zug, der vom vorangestellten Filter akzeptiert wurde, gespeichert wird.

- Der **`CaptureTestAccumulator`** überprüft mit Hilfe der Methode **`Move.isAttacking`**, ob es einen Zug gibt der eine bestimmte Position angreift. Dieser Akkumulator wird benutzt um zu testen ob ein bestimmtes Feld, insbesondere das Feld des Königs, angegriffen wird.
- Der abstrakte **`MoveFilter`** ist ein Zugakkumulator, der Züge nur dem enthaltenen inneren Zugakkumulator hinzufügt, wenn sie eine Bedingung, die durch Unterklassen definiert wird, erfüllen. Diese Klasse wird für die Implementierung anderer Filter verwendet, damit die Filterlogik nicht dupliziert werden muss, sondern mit dieser Klasse abstrakt verfügbar gemacht wird.
- Der **`LosingMoveFilter`** erbt direkt von **`MoveAccumulator`** und beinhaltet einen weiteren Zugakkumulator, dem er nur Züge hinzugefügt, die den eigenen König nicht ins Schach bringen. Somit werden von den berechneten Zügen nur legale Züge erlaubt, wenn zum Beispiel Züge, die der Spieler eingegeben hat, validiert werden.
- Mit dem **`FulfillmentFilter`** wird geprüft, ob eine Bewegung durch einen legalen Zug dargestellt wird. Hierbei wird die Methode **`Move.fulfillsMovement`** benutzt, die feststellt, ob ein komplexer Zug durch eine gegebene einfache Bewegung dargestellt wird. Es kann nicht durch einen direkten Vergleich überprüft werden, ob ein Zug durch eine Bewegung dargestellt wird, weil Züge teilweise komplexe Kompositionen sind.
- Von **`MoveAccumulator`** erbt direkt der **`CollectionAccumulator`**, der Züge einfach nur in eine Collection tut, die bei seiner Konstruktion übergeben wurde. Dieser Zugakkumulator wird für die Auflistung aller möglichen Züge benutzt. Er fordert immer die Berechnung weiterer Züge und auch nicht-schlagender Züge, da alle Züge gesammelt werden sollen.

Erlaubte Züge durch Filtern finden

Um alle erlaubten Züge für einen bestimmten Spielstand zu finden, wird ein **`LosingMoveFilter`** mit einem **`CollectionAccumulator`** konstruiert. Dann berechnen alle Figuren Züge, die sie dem Zugakkumulator hinzufügen. Der filternde Zugakkumulator überprüft für jeden hinzugefügten Zug, ob die Ausführung dieses Zuges den König in Schach versetzt und fügt den Zug nur zu seinem inneren Zugakkumulator hinzu, wenn das nicht der Fall ist. Hierbei kommt das Anhalten der Zugberechnung noch nicht ins Spiel, weil beide Zugakkumulatoren an allen Zügen interessiert sind.

Bei der Validierung eines Zuges vom Spieler wird ein **`FulfillmentAccumulator`** mit der vom Spieler angegebenen Bewegung konstruiert und dann auch über alle Figuren iteriert. Diesem Zugfilter wird als innerer Zugakkumulator ein **`LosingMoveFilter`** übergeben, damit nur erlaubte erfüllende Züge akzeptiert werden. Am Ende dieser Filterkette steht ein einfacher immer akzeptierender Zugakkumulator um den gefundenen validierten Zug nach der Suche zurückgeben zu können.

Es wird jeweils überprüft, ob ein berechneter Zug der Bewegung vom Spieler entspricht. Sobald ein Zug gefunden wurde, der dieser Bewegung entspricht, wird der erfüllende Zug gespeichert und die Berechnung weiterer Züge gestoppt, da für eine Bewegung immer nur höchstens ein Zug existiert. Somit müssen nur ungefähr die Hälfte der möglichen Züge

berechnet werden bevor ein erfüllender Zug gefunden wird. Da der **`FulfillmentAccumulator`** vor dem **`LosingMoveFilter`** steht, müssen nur erfüllende Züge auf Legalität überprüft werden.

Ausführen von Zügen

Züge als Zustandsübergänge

Da bei der Überprüfung der Legalität eines Zuges der Zug testweise ausgeführt wird, um zu testen, ob der König in diesem neuen Zustand angegriffen wird, muss der Spielzustand temporär modifiziert werden, da keine Kopie erstellt wird. Hierzu wird der Zug ausgeführt, die gewünschte Bedingung überprüft und dann der Zug wieder rückgängig gemacht.

Züge beschreiben nur die Änderung des Spielzustandes und ihr Verhalten ist nur auf dem Spielzustand definiert, von dem sie konstruiert wurden. Ein Zug kann zwar auf einen Spielzustand angewandt werden für den er nicht konstruiert wurde, aber im Allgemeinen kommt es dann zu einem Fehler, wenn die Figur an der Startposition der Bewegung des Zuges nicht existiert. Diese Limitierung ist akzeptabel, weil es nicht vorkommen sollte, dass Züge von unterschiedlichen Spielständen gemischt oder vertauscht werden.

Mit **`GameState.applyMove(Move)`** wird ein Zug auf den Spielzustand angewandt. Diese Methode stellt sicher, dass die anderen Teile des Zustandes korrekt verändert werden und der Zug im richtigen Moment angewandt wird. Das Gegenstück dazu bildet die Methode **`GameState.reverseMove()`**, die den zuletzt angewandten Zug rückgängig macht und darauf achtet, dass alle Veränderungen, die für die Anwendung eines Zuges vorgenommen wurde, in umgekehrter Reihenfolge rückgängig gemacht werden.

Spielstatus bestimmen

Der Spielstatus zeigt an, ob sich ein Spieler im Schach oder Schachmatt befindet, oder ob das Spiel remis ist. Der Spielzustand wird bestimmt, indem geprüft wird, ob der König angegriffen werden kann und ob ein Zug möglich ist, der den König aus dem Schach bringt. Das Spiel ist auch remis, wenn festgestellt wird, dass zu wenig Material im Spiel übrig ist, um ein Schachmatt erzeugen zu können oder die 75-Züge-Regel eintritt. Die 50-Züge-Regel besagt nur, dass ein Spieler ein Remis einfordern kann, aber nicht muss. Diese Regeln wird nicht durch die Spiellogik umgesetzt, weil die Einigung auf ein Remis bzw. das Einfordern eines Remis nicht im Umfang der Anforderungen enthalten ist. Dahingegen ist das Spiel mit der 75-Züge-Regel ohne Zutun der Spieler remis und wird daher auch durch die Spiellogik umgesetzt.

Tests

Um die zahlreichen Tests, die Züge ausführen und überprüfen müssen, einfacher und eleganter zu machen, gibt es durch **`GameTestUtils`** eine Reihen an Hilfsfunktionen. Sie erlauben es, Züge in der Schreibweise des Konsoleninterfaces zu spezifizieren, damit Züge nicht in das interne Koordinatensystem für Positionen umgerechnet werden müssen.

Außerdem wird hiermit vermieden, Strukturen zur Sicherstellung der Legalität oder Illegalität von Zügen in Tests zu duplizieren.

Für das Testen des Konsoleninterfaces gibt es Hilfsmethoden, die einen **`InputStream`** mit einer bestimmten Zeichenkette erzeugen oder sicherstellen, dass einem **`PrintStream`** eine bestimmter String übergeben wurde. Da den Konstruktoren der Klassen im Konsoleninterface statt **`System.in`** und **`System.out`** auch ein eigener **`PrintStream`** bzw. **`InputStream`** übergeben werden kann, ist es möglich, deren I/O-Verhalten zu testen.