Architecture documentation

for the software engineering internship project SoSe 2020

Table of contents

components

separation of responsibilities of the components

The program is divided into two components, the game logic and the user interface from the console interface and the GUI exists. The game logic determines the behavior of the chess game, while the user interface takes over the interaction with the user. The game logic is completely independent of the user interface and does not import classes from the user interface. Thus, the game logic can exist independently as a separate and self-contained package, which makes it easier to infer about the behavior of the game logic.

In general, functionalities are divided into levels of responsibility that are assigned to different methods or classes. Methods of the highest level of abstraction deal with complex concepts that are simply expressed by calling small-scale methods so that the more abstract methods remain precise and concise. These abstract methods ensure that certain preconditions are met by calls to other methods, while less abstract methods carry out concrete calculations and do not check preconditions themselves. This division into rough semantic layers makes it possible to avoid repeated definition and implementation of small-scale tasks. This also means that complex operations and queries can be formulated and implemented with little effort, since details of the calculations are abstracted.

EnvironmentEnvironment

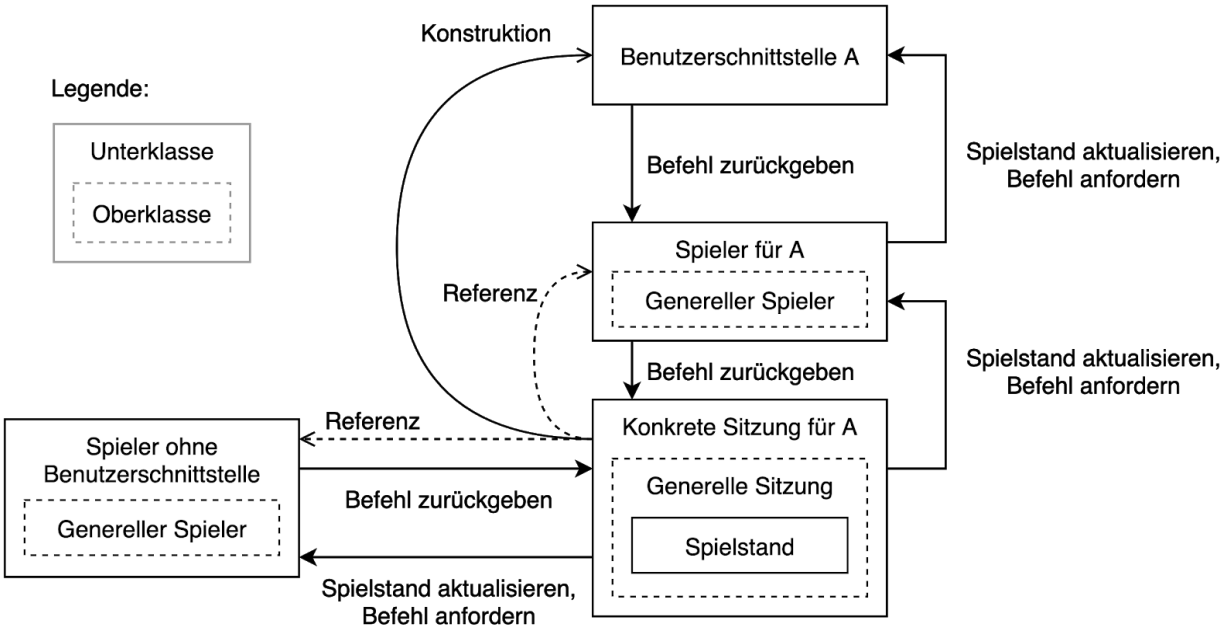
variablesvariables areby theclass **`Environment`** represented and read. The individual Boolean environment variables are declaratively defined by an enumeration type and can therefore be easily expanded. Different parts of the user interface then check which environment variables are set and change their behavior accordingly. This approach allows a compact representation of all user settings thatas a console can be setargument and an uncomplicated transfer to specialized components of the user interface.

Interaction through sessions

The game logic is not concerned with updating the data of the user interface but only with the state transitions in the game itself. The interaction between the game logic and the user interface, be it the GUI or the console interface, isby thepackage **`interaction`** handledin which theprimarily **`session is`** responsible for administration.

A session represents a game session that can consist of several successive but independent games. The two players in a session arebyenter **`players`** modeledwho are informed about the current game status from a session and are called upon toa command. The session thereby controls when changes in the game status are communicated to the players and when a player may make a move or send another command. All changes to the

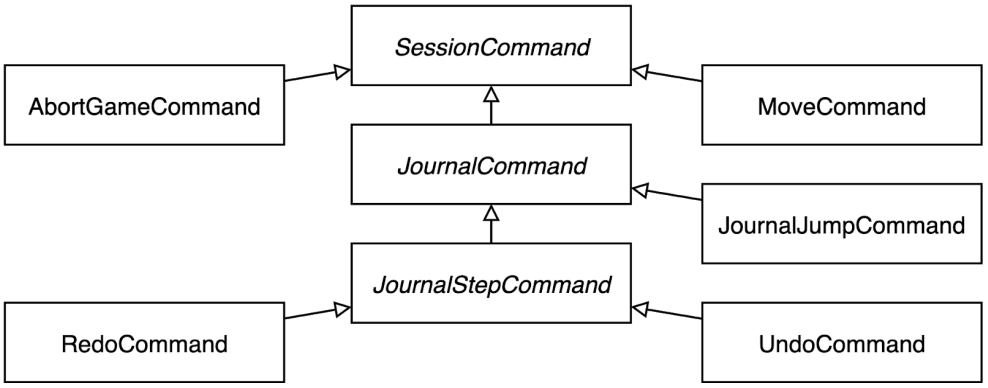
game status and the control of the moves are carried out centrally via the session, so that the logic involved in the course of a game is not duplicated.



Here is an example of a session that was constructed for user interface A and contains both a player for A and a player who has no user interface. The session serves both players equally and updates the score for both users after a command has been executed. It is important to note that players do not apply their session command to the session and game state themselves, but receive the updated game state from the session. The interaction takes place between the general superclasses, while the instances come from subclasses that contain the specific behavior for their user interface. The user interface A can thus be either the GUI or the console.

and their inheritance structure

Session commandsInstead of only getting moves from the players, the session receives commands from one of the two players. There are a number of different session commands that can be produced by the players. Only the active player whose turn it is may give an order that contains a move. However, both players are allowed to generate commands for moving back and forth in the train history. Which player is allowed to produce which commands is determined by methods on the commands. The following diagram shows the different types of session commands.

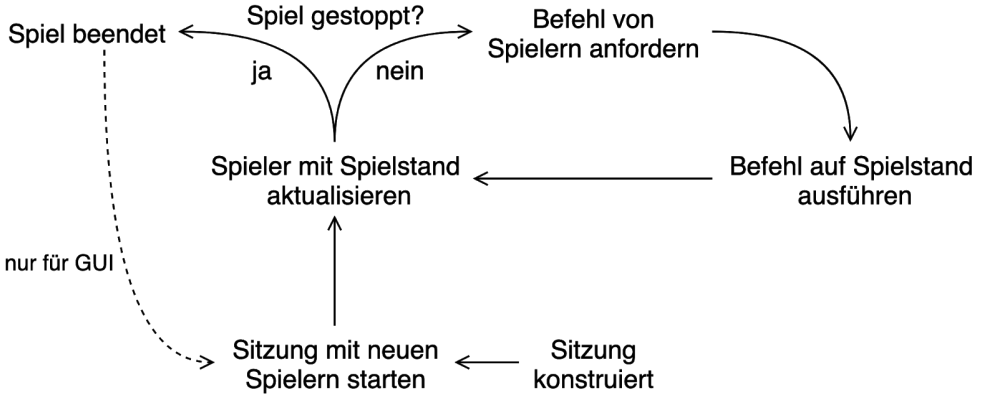


Session Command,an abstract command is displayed for the session. The other commands inherit from this class and implement their respective actions and properties. Commands areby the session by calling the **`applyTo`** activatedmethod. The implementation of this method then executes the respective command. This command structure makes it possible to indirectly transfer method calls to the session across different threads. Since the players sometimes work in separate threads, it must be prevented that they call the session directly. This causes the session to make the changes to itself when it executes a command.

With a **`MoveCommand`** , a move is made on the score. The **`AbortGameCommand`** is used to signal the session that the game should be ended when, for example, the player goes back to the main menu in the GUI. Thesubclasses **`JournalCommand`** implement commands that perform operations on the train history. The train history itself is documented in the corresponding section.

Procedure of a session

The following diagram shows the procedure of a session. A session can play an unlimited number of individual games with each other in turn. At the beginning the session is in a start state and is waiting to be started for the first time.



As soon as the session has started, it creates a new game state and passes it on to the players. The players then update their ads and their internal status. However, players are designed not to access or modify the game save between session updates. The update of a player by the session takes place before the first move of the game, before any change in the game status and after the last move of the game.

Each time a move is used, the active player changes to the previously inactive player. The player with the white pieces is the active player first. Score information always relates to the active player who next makes a move, which is then returned to the session. Since both players are notified of the current game status before each command, it is important to also make statements about the inactive player. The calculation of the state for a player before each application of a command takes place in **`TurnStatus`** and is expressed with one of the values of **`TurnStatus`** . For example, it indicates whether the player is in chess or whether he has won.

Since the inactive player can also issue orders as long as it is not a one turn order, the session asks both players for orders in each cycle of the cycle. (See diagram) If a player has submitted a valid command, the other player is informed that he should no longer produce a command. When the session is stopped, the request to generate a command is also withdrawn for players.

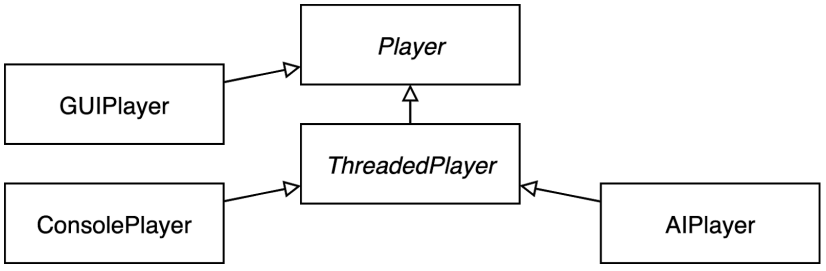
Declarative game modes

A game mode describes which players participate in a game. He is able to start a given session with the required players for this mode. The game mode is given producer functions for players who communicate through a user interface and for players who decide moves with an artificial intelligence. The game mode uses these generators to provide suitable players at the start of the session. The respective session determines what kind of players are actually constructed.

The game modes are defined declaratively, similar to the environment variables, and can therefore be easily expanded with additional options. A game mode is only determined by a description in text form and its behavior when creating a pair of players.

Types of sessions and players

There are subclasses offor both user interfaces **`session`** and **`player`**, which are responsible for interacting with their respective user interfaces. They form the interface between the pure session and a concrete session tied to a user interface. In order for the session to remain pure as a logical concept, these subclasses must contain all the specific knowledge about the user interfaces. An overview of the different players is shown here:



Multithreading in sessions

Both players can give commands to the current session at the same time, even if only one of the players is active. So that both players always remain interactive for the user, players can use **`ThreadedPlayer to`** outsource the generation of commands for the session to a separate thread. The command given to the session is carried out and then both players are asked for a command again. It is possible for players to generate commands in sync without error, as long as both players are not working in sync. This behavior is only used for test purposes.

AI players The AI

game is played by the **`AIPlayer`**, who responds with moves thata search strategy defined in the **`MoveCalculator`** are calculated using. This player also works in a separate thread so that the main thread is not blocked.

By specifying a search strategy, it is determined which algorithm is to be used to search for the best possible move for the current game status. In general, the search algorithms need a position rating, which is a numerical estimate of the strategic value of a score for the AI. This value isbydetermines the **`MoveCalculatortotal`** calculated, whichvalue of all figures of a game from their material value and their position value, in which own figures are evaluated positively and opposing figures negatively.

New search algorithms can be implemented by additional subclasses of **`SearchStrategy`** , so that any complex AI can be easily integrated. The search depth and search strategy of the AI player can be specified in its construction and has a different effect depending on the type of search strategy.

The simple AI is implemented by thestrategy **`ShallowEvaluation`** , which selects from the possible moves the move that generates the score with the best value. The search for this AI is only a pull deep, so that no foresight is possible. Setting a search depth has no effect with this strategy because it only searches one move deep.

In, the complex AI in **`contrastFixedAlphaBeta uses`** a simple alpha-beta search with a variable search depth. As a determined algorithm, this search always selects the same optimal move. The score is only evaluated when the search depth is reached or at the end of a game, hence the name “Fixed”. In order to slightly increase the efficiency of the search, a separate train accumulator is used for the search, which does not accept any further trains if the alpha-beta algorithm cancels the search for the current subtree. Instead of calculating all possible moves of a game and then iterating, this method does not generate any further moves after the search is terminated in a game.

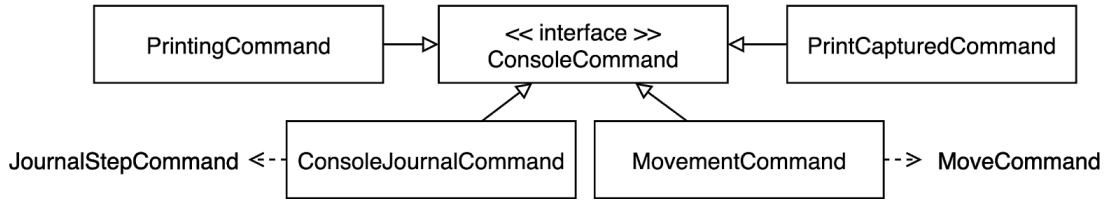
User interface console interfaceinterface

session and player for the console

The specific session for the consolein **`interface isConsoleSession`** realized. This session asks the user about the desired game mode before starting the game and then starts with the corresponding players. For the human player, the **`ConsolePlayer is`** used, which reads commands from the user from the console and outputs the chessboard before each move as text with optional UTF-8 symbols and colors. So that the chessboard is not displayed twice when two players play on the console, only the active player displays the game board in such situations.

This player is waiting for user input in a separate thread so that there is no blocking. Because it is impractical to interrupt waiting for data from the console, this player does not respond when the session prompts them to interrupt their calculation or wait. However, since the input in the console is independent of the output, this player can still return a command in the next cycle without being asked again.

The interaction between the **`CommandInterface`**, which reads and analyzes user input from the console, is similar to that between players and the session using commands, here console commands. The commands implementcommand that **`ConsoleCommand`** and optionally produce a sessionthe player passes on to the session. This symmetry between the two sets of commands allows commands in the console interface to be elegantly converted into session commands. If the user input is incorrect, console commands cannot produce a session, but insteadexecuted directly **`commandissuewhenPrintCommand is`** an error messagea. The following diagram shows the console commands along with the session commands they can produce.



Output of Scores

`Board Printer` summarizes the methods for the preparation of the current article in a class together, care was taken in the expansion. For example, the **`BoardPrinter isboards`** able to output chessup to a size of 26 * 26, which is useful for chess variants with other square playing fields. The user can decide whether he wants the chess pieces to be displayed as letters or as UTF8 symbols and whether a colored checkerboard background should be output in the console (see operating instructions).

The struck figures are displayed in sorted order according to the value of the figures and color. This sorting is implemented by implementing a **`comparator`** so that the logic for comparing figures remains abstract and separate from the user interface.

Selection of game modes

To select the game modes, the options for all game modes are output on the console and one of the abbreviations is expected to start the game with the selected game mode. It is not specified in the console interface which game modes there are or what their names are, since the game modes themselves disclose their description and abbreviation in the form of a list. Incorrect entries in this selection result in an error so that the user can try again.

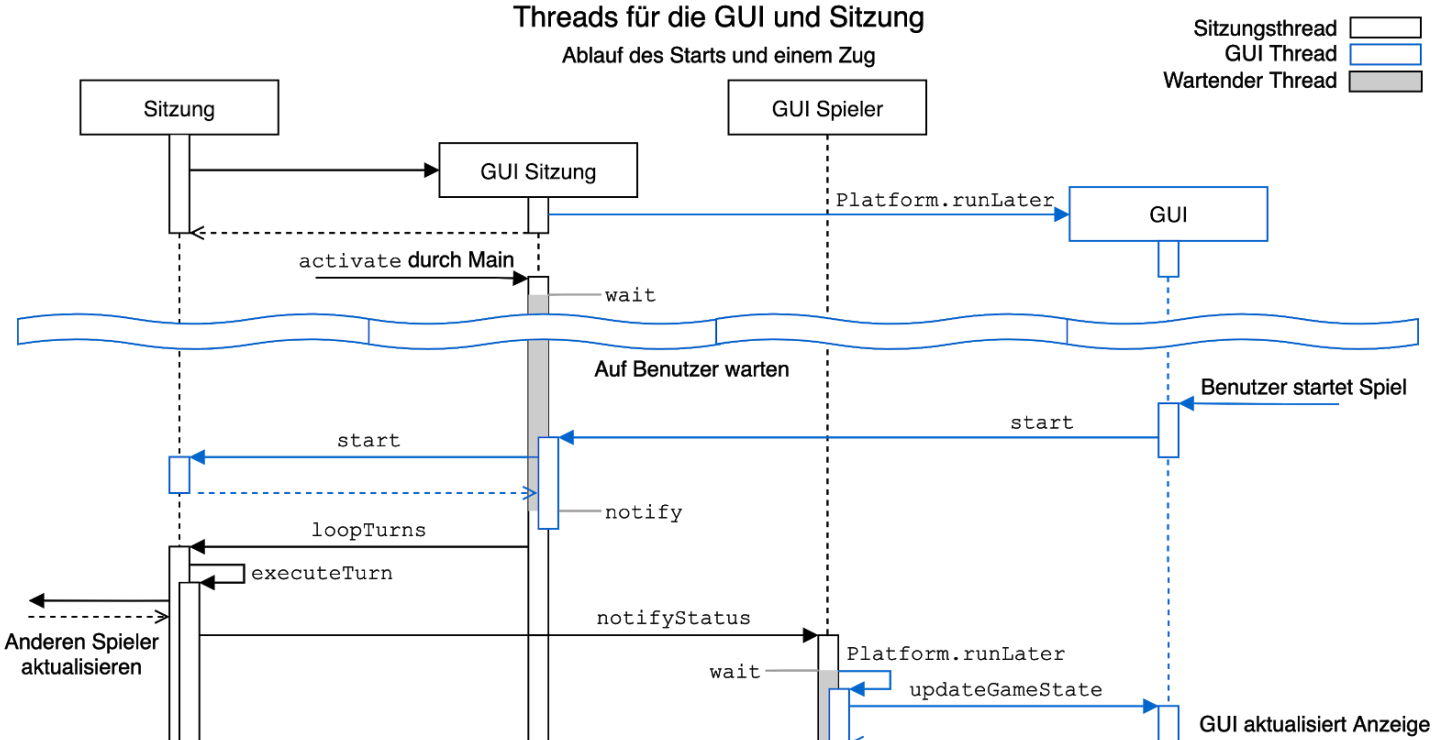
GUIuser interface

sessionand player for the GUI

The following sequence diagram shows the process of starting a session for the GUI and making a move. The end of a game and the restart are not shown here, but they are based on the same mechanisms as can be seen here. The GUI session isby **`GUISession`** realized. Together with theit takes over **`GUIPlayer,flow`** the asynchronous communication between the normal sessionand the GUI. The GUI is started in a separate thread at the start of the session so that the GUI remains interactive, even if synchronous calculations occur in the session.

The **`GUIPlayer`** extends the regular **`player`** and not **`ThreadedPlayer`**because the asynchronous interaction with the GUI has to happen through JavaFX and not with the mechanisms from **`ThreadedPlayer`**that the other asynchronous players use.

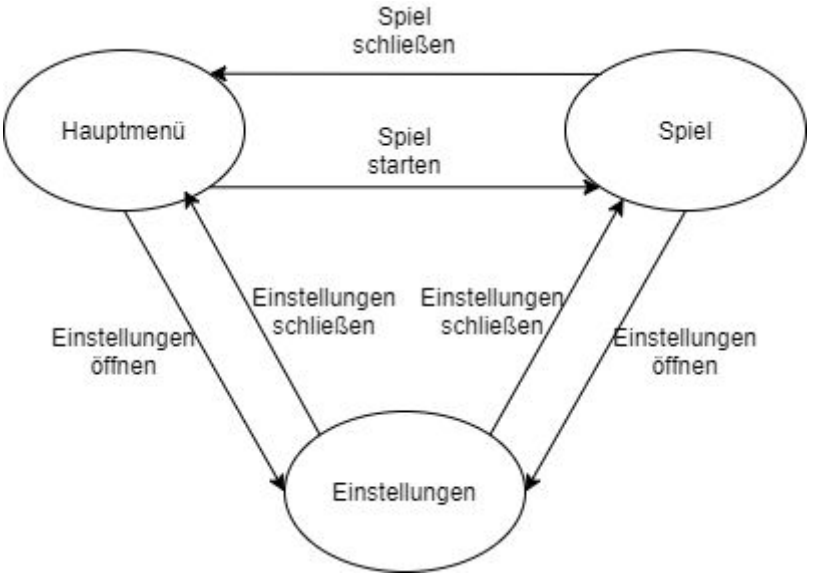
The session generates the GUI after it has been constructed and is only started when the user starts a new game through the GUI. Until it starts, it waits to be woken up by the GUI. When the session requests a command from the GUI player, it calls the GUI in its separate



thread, thus ensuring that the user can enter a command. Then the session waits for an order from one of the players. When a valid move has been made by the user, the GUI player is activated and passes the move on to the session as a command. A similar process takes place when updating the GUI after a train.

The instance of the session in the GUI is not recreated for every game because several games are possible within one execution of the program. Therefore, a session is restarted from the current state after the first start or started after the last game. For a restart during a running game, the waiting state of the players is interrupted so that a new game can be initialized.

Components of the GUI and their interaction



The basic flow of the GUI is relatively simple. When the user starts the program, the main menu opens and it becomes the **`MainMenuView`** set as the root of the scene. From here, as can be seen in the figure, either the settings can be opened or a new game can be started. When the user accesses the settings from the main menu, the user also returns to the main menu by closing the settings. After a game has started, the user has the option of exiting the game and returning to the main menu or opening the settings during the game. The settings are kept independent of the game so that they are retained between games as long as the user does not close the program. The structure of the GUI components is described below.

Main menu

The main menu has a very simple structure. It consists of only one controller and one view. At this point in the program, no model is required, since the main menu does not manage any data and only reacts directly to the user's input.

The **`MainMenuController`** has the task of starting a new game and opening the settings. The controller responds when the user clicks on one of the three images to be seen and then starts the session with the corresponding game mode. The strength of the AI can also

be set using a slider. When you click on one of the pictures for the game modes, the controller calls up the view to display a pop-up for selecting the player color. The controller then starts a new session with the selected mode. The session creates a new game and starts the game. The controller then sets an instance of **`GameView`** as the root of the scene to display the new game.

Settings

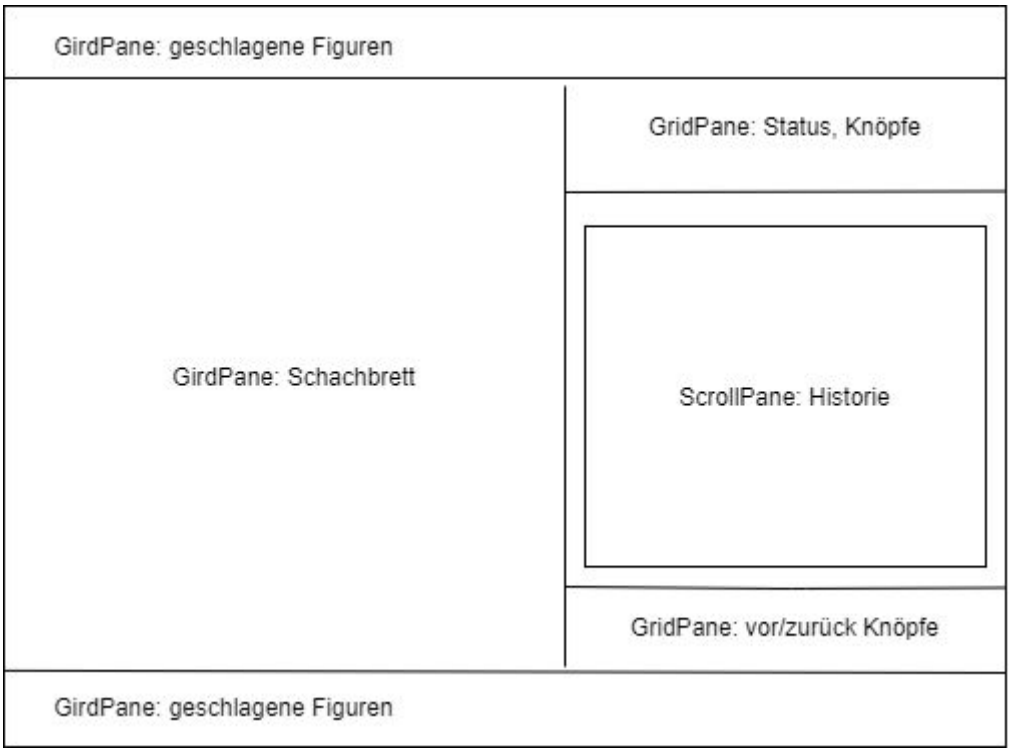
The settings use the MVC model. In the **`SettingsModel`** class, the settings are saved as enums in an EnumMap. This makes it easy to add more options. In the current game, only options that can be true or false are saved, but the **`SettingsModel`** class also allows easy addition of non-Boolean values. Furthermore, the singleton pattern is used because only one instance of the settings is required across the entire game. This makes it very easy to access the settings of different parts of the GUI. Since no switch is available in JavaFX by default, awas used for the settings **`SwitchButton`** created.

Game

presentation We chose the MVVM model for the game presentation because we liked the clear separation between the data in the model, the processing of the data in the ViewModel and the display of the data in the view as a clear structure. The general interaction with the user regarding the game is handled by **`GameModel`**, **`GameViewModel`** and **`GameView`** , while specific components are hierarchically divided into their own MVVM structures.

The **`GameModel`** interacts with the **`session`** and saves the data to be displayed in observables. Updates to the data are made when trains are made. Thereacts to changes in this data **`GameViewModel`** corresponding. The data is notat this point on for the **`game`** preparedviewing,as this processing in **`Game viewmodel`** takes place. The **`GameModel`** is called by the **`GameViewModel`** if the user has clicked on a field on the chessboard, and then sends a new move to theif necessary **`session`**.

The **`GameViewModel`** is responsible for data processing and processing the UI events. It binds the data required by the view to the view with the help of JavaFX Properties and uses listenersto changes in the **`GameModel`** to react.



Here is the layout of the **`GameView`** , which is responsible for displaying the specific data. For the chessboard, the GridPane display is obviously particularly useful, since the chessboard is a grid and thus fits well into the structure of the GridPane. In order to be able to quickly display new positions and not have to process moves in the view, all possible figures are loaded once at every point on the chessboard at the start of the program. The visibility of the characters is thenusing bindings and properties in the **`GameViewModel`** handled. The struck figures are created in the same way. Thethus remains **`GameView`** free of game logic.

Game logic Mapping

the game state

The game state is represented by theclass **`GameState`** and **`Board`** and consists of figures, their position and color as well as further information about the movements of the figures in the past. Data is also recorded on the number of total moves and the number of moves since a pawn last moved or when a figure was hit. Which player's turn changes with each move and is referred to here as *color* .

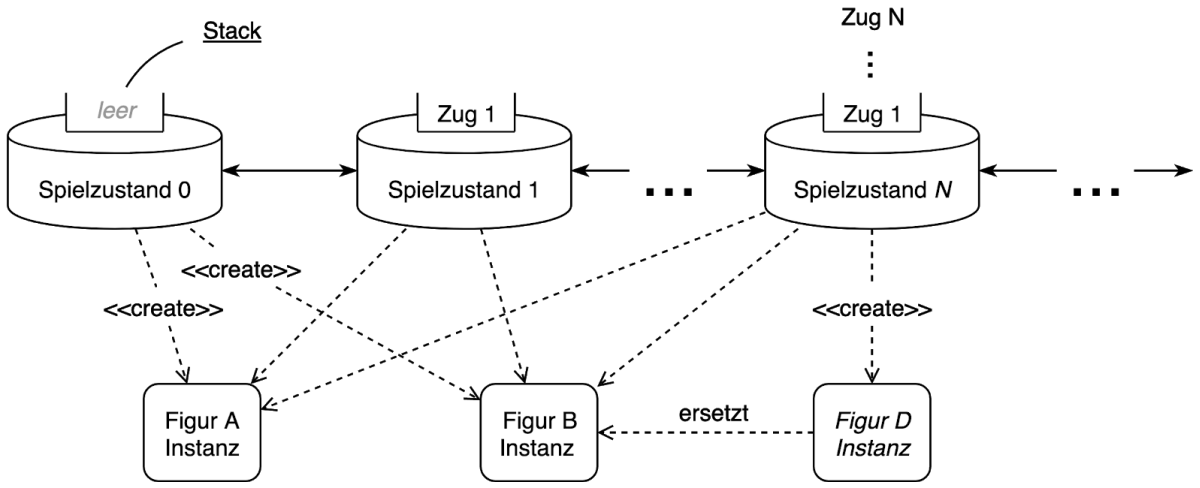
Information about the game and its history is stored in **`GameState`** , while the data about the game board itself and the charactersin **`Board`** are stored. This separation does not mix the responsibilities of the two classes. Although both classes interact a lot, their tasks are different enough to work well as separate classes. The game board can be accessed transparently, because sometimes only information about the game board itself is required and not about the entire game status.

Stateless instances for figures on the game board

So that the positions of the figures do not have to be saved in the instances of the figures, the figures are stored in a hash table in which all figures are stored in one position with the current index of the playing fields as a key. The index counts all fields of the chessboard line by line starting with the first field at the top left. This data structure is efficient since it is almost never necessary to determine its position for a certain figure, but only the figure at a certain position is always interesting. The instances of the figures are called stateless because they do not contain a state that can change after they are instantiated. In addition, these stateless instances of the figures allow the game state to be changed efficiently because the instances of the figures do not have to be modified.

The instances of the figures thus represent the identity of the figures because they only have to be created once across the entire execution of the program. This representation also makes sense in terms of content, because in a real chess game the figures are not changed with every move, but only change their position.

If a figure performs an operation for which the current position of the figure is required, the position of the figure is given to this operation when it is called. The position of a figure is also always available in the context of the call, because all operations that are called from outside the game relate to specific positions and not to a specific figure.



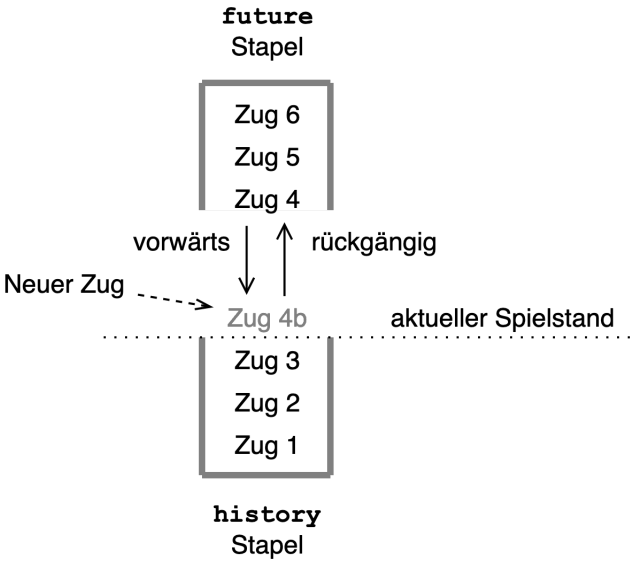
This diagram shows the instances of the figures and the references of the scores to the figures. In individual cases, new instances are created when, for example, a farmer is converted. The train history is shown here in simplified form.

Train historysaved games

So that no complex copies ofneed to be created, it is possible to apply a move to a saved game and then undo it, so that exactly the previous saved game is restored. This method simulates the creation of copies of the game score because operations that would be performed on a temporary copy are instead performed after a move has been made but

before the last game state is restored. The save only allows the last move to be undone, because undoing previous moves out of sequence can bring the game into an undefined state. This is discussed further in the section on trains.

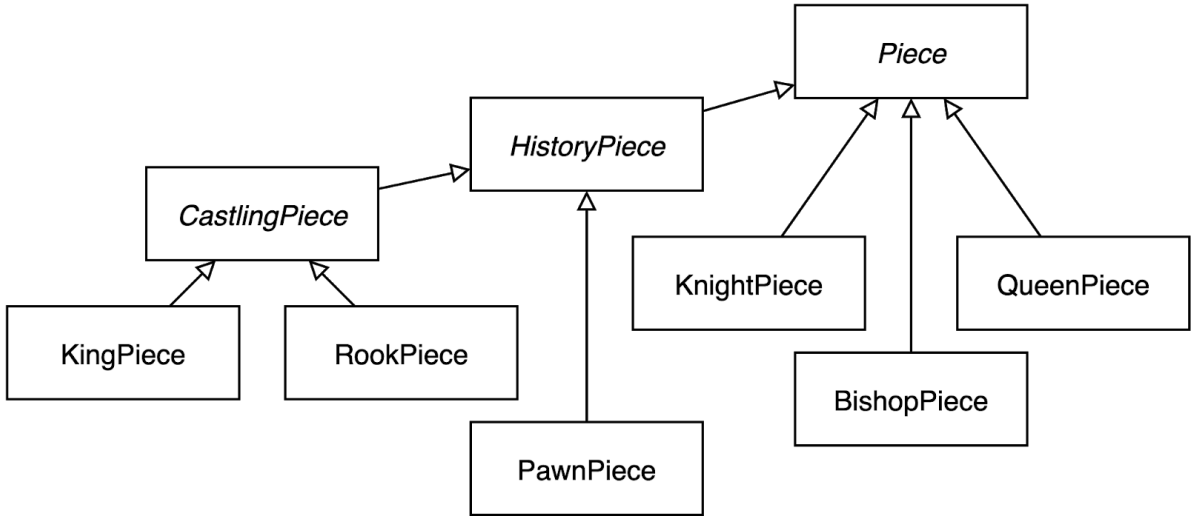
The moves are not saved for calculations on the game, but reversed immediately after the desired calculation on the changed game has been completed. However, if a move is made by a player, the move history in **`MoveJournal saves`** the move. **`MoveJournal`** allows you to undo and redo undone moves by maintaining two stacks of moves. One stack is for the past of the trains used, while a second is for the undone trains of the future.



This diagram shows the two stacks and the movement of the trains between them. When a move is undone, it is put on the stack for the future. To repeat a move, it is removed from this stack and applied to the game as well as placed on the stack for the past. So that trains on the stack are not wrong for the future, they are deleted as soon as a new train is added to the train history from the outside.

Inheritance structure of the characters

There is a subclass of the abstract classfor each of the different characters **`Piece`**, which contains basic functions that are used by several or all subclasses. No instance of this superclass can be created because it is always a specific figure that has a unique instance and type on the field. The subclasses differ significantly in how they generate possible moves. There are also the abstract subclasses **`HistoryPiece`** and **`CastlingPiece`**, which provide further functions with regard to their movement history together with thegame **`GameState`** .



This diagram shows the inheritance hierarchy of the different types of figures, with the names of abstract classes slanted. The game state holds for characters by **`HistoryPiece`** inherit the last train you took part in. The reason for this is that these figures may only make certain moves such as castling if they have not yet moved or in the case of the en passant, not on the last move. **`CastlingPiece`** provides a simple abstraction of theservice used by **`HistoryPiece`** characters involved in castling.

We decided to display different types of figures for subclasses because it divides the behavior of the figures better than if a type field is used in undifferentiated instances to distinguish the figures.

Other parts of the state

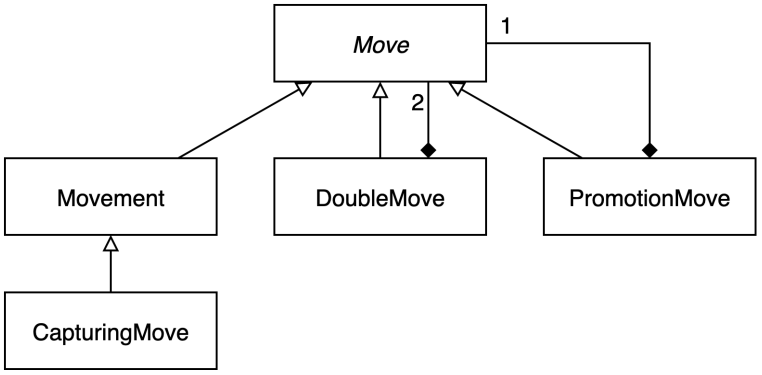
When creating instances of figures, the figures are assigned a unique identification number with a running counter, which is used to identify them in the movement history without collisions. A list of the figures already beaten is also kept. The positions of the two kings are also recorded separately so that it can be checked whether a particular move puts the king in check or the game is ended by a checkmate. The movement history isfor each **`HistoryPiece`** saved as a stackso that previous positions can be accessed.

Calculating moves

Composition of moves and their inheritance structure

Trains describe possible changes in the game status and are represented by the abstract class **`Move`** and its subclasses. 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.

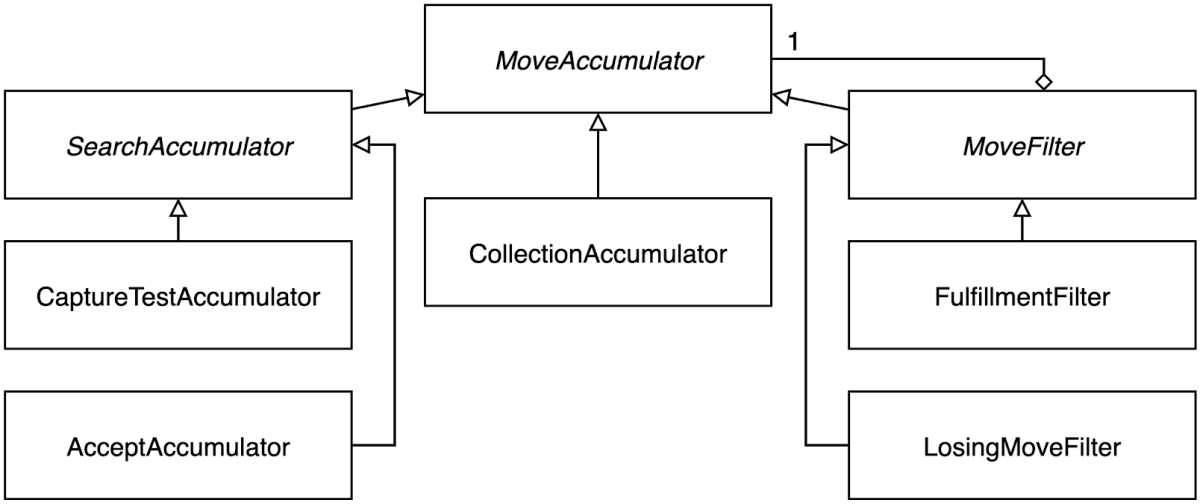
Da die Instanzen der Figuren ihre Position und Bewegungshistorie nicht selbst speichern, wird ihre Position und eine Referenz zum Spielzustand zur Berechnung der Züge übergeben. Die möglichen Züge werden nicht vom Aufruf der Methode

`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.