Architektur Dokumentation

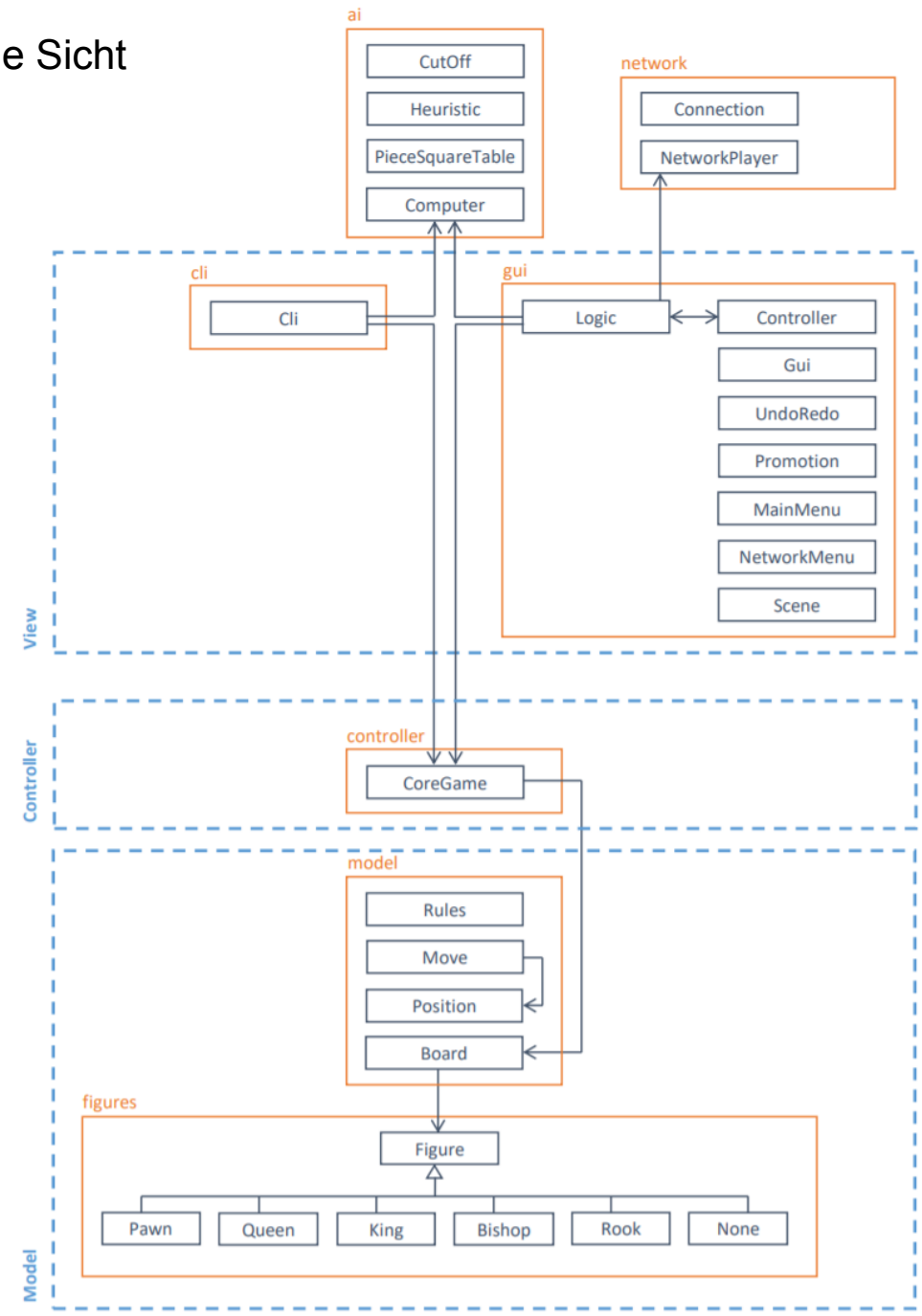
Inhaltsverzeichnis

Nutzersicht

siehe Anforderungsdokumentation[1] (Anwendungsfalldiagramm und Story Cards)

Gesamtprojekt

Statische Sicht



UML-Paketdiagramm[2]

[1] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/documentation/1_Anforderungsdokumentation.pdf
[2] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/Klassendiagramm.png
Dieses Diagramm entspricht nicht den Regeln des UML-Paketdiagramms, sondern wurde so vereinfacht, dass die Klassen und ihre Beziehungen untereinander gut ersichtlich sind.

Wie aus dem Diagramm erkenntlich, orientieren wir uns an dem Architektur-Pattern der Schichtenarchitektur und dem Design-Pattern MVC (Model-View-Controller).

- View: Die Klassen bzw. Packages **CLI** und **GUI** übernehmen dabei die Aufgabe der Darstellung nach außen auf der Konsole bzw. als grafische Schnittstelle.
- Model: Die Klassen **Position**, **Move**, **Board**, **Parser** und **Rules** bilden die Spiellogik. Die Klasse *Board* hat zudem Zugriff auf die einzelnen Figuren.
- Controller: Die Klasse **CoreGame** übernimmt die Kontrolle und definiert das Verhalten des Spiels. Zudem vermittelt diese Klasse zwischen View und Model.

Durch diese Trennung der Komponenten ist es vergleichsweise einfach möglich, Erweiterungen zu realisieren, z.B. eine weitere View-Klasse hinzuzufügen, da die Spiellogik von der Ausgabe getrennt ist. Dadurch ist das Programm relativ unkompliziert erweiterbar und wartbar. Im Laufe des Projektes wurde die Wahl dieses Patterns auch sinnvoll, um Aufgabenbereiche aufzuteilen und so eine möglichst effiziente und agile Entwicklung zu ermöglichen. So haben sich zwei Mitglieder der Gruppe in der zweiten Iteration um die grafische Darstellung nach außen und zwei um die Realisierung eines Computer-Gegners kümmern können, ohne sich gegenseitig in der Entwicklung zu behindern.

Dynamische Sicht

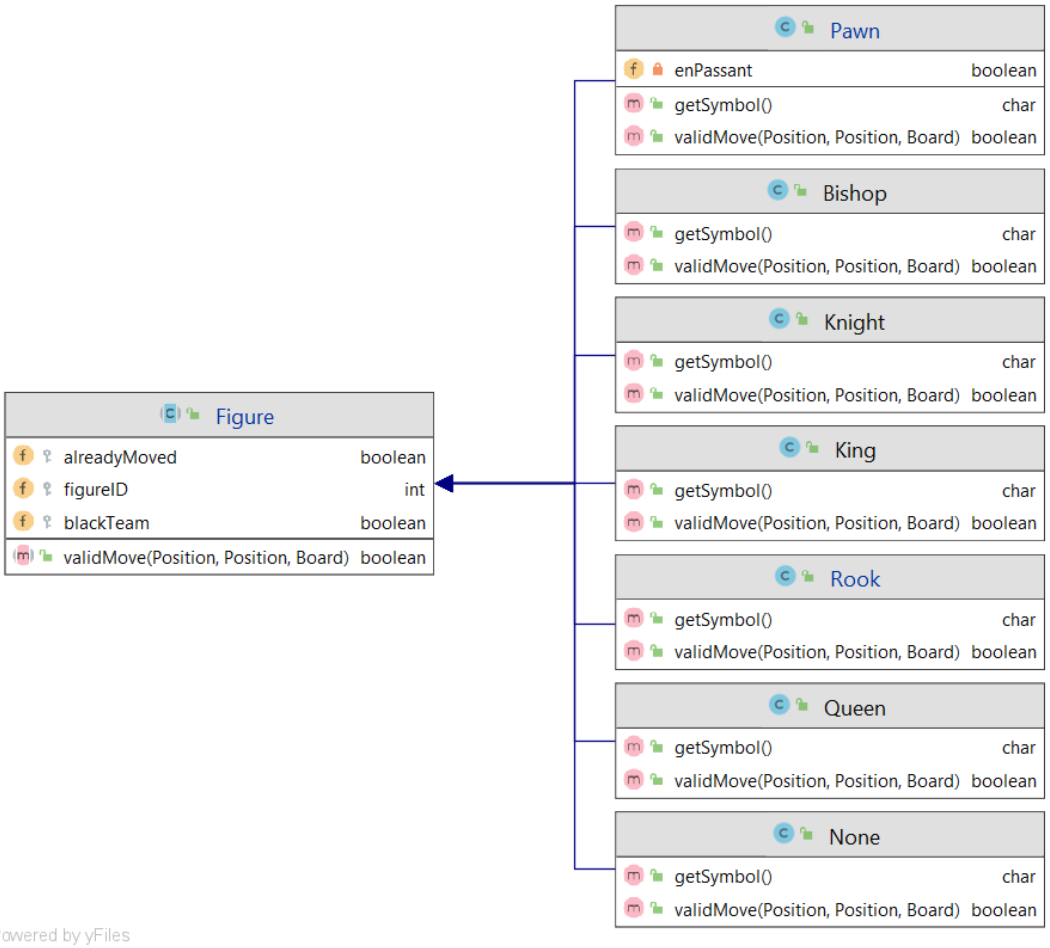
Ein Schachspiel wird entweder von der Cli oder von der Gui aus gestartet. Diese erzeugen dann ein CoreGame und, falls entsprechend ausgewählt, auch einen Computer oder Netzwerkspieler. Kommt es zu einem Schachmatt, Patt oder einem manuellen Spielaustritt, so wird das Spiel beendet.

Im Folgenden wird nochmal im Detail auf die einzelnen Pakete sowohl aus statischer als auch aus dynamischer Sicht eingegangen.

Für ein vollständiges UML-Klassendiagramm siehe
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_chess.png

Statische und dynamische Sicht der Pakete

Paket figures



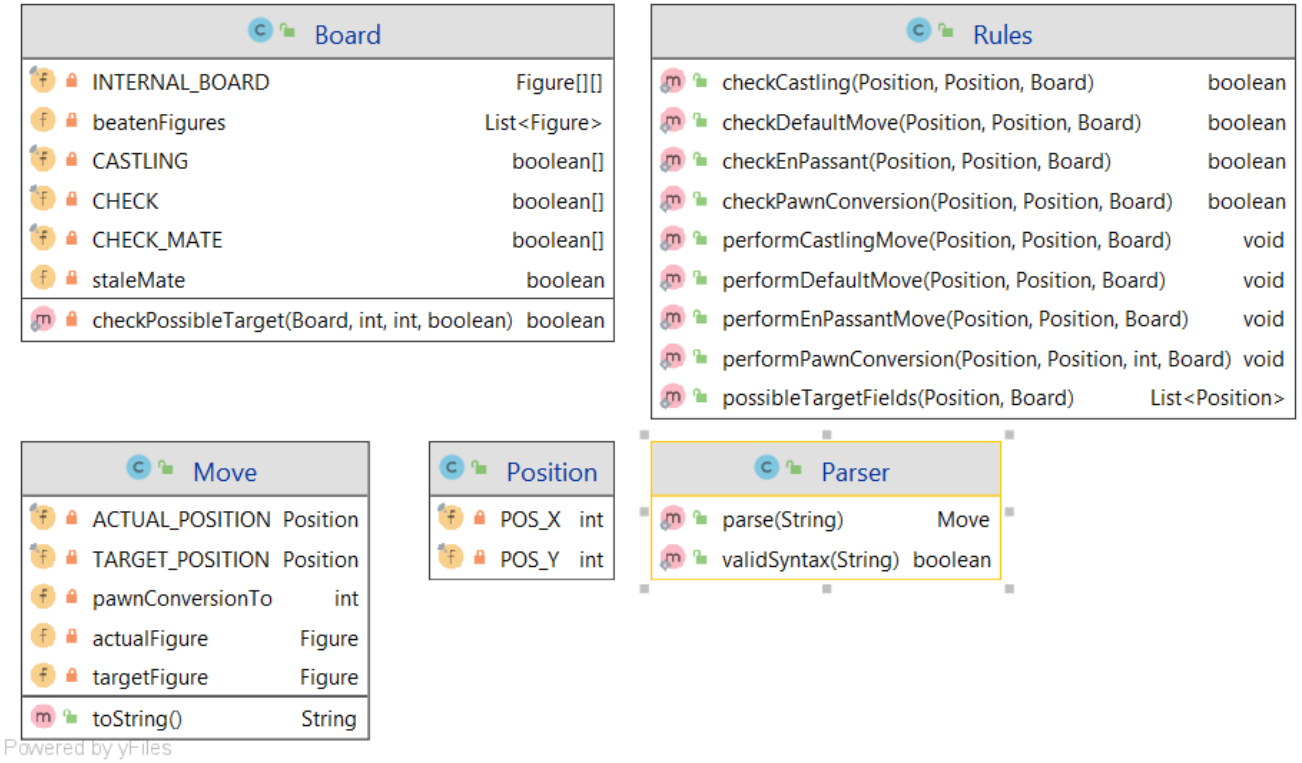
[UML-Klassendiagramm figures][3]

Statische Sicht

In diesem Paket sind alle Figuren vorzufinden. Diese erben alle von der Klasse *Figure*, da jede Figur dieselben Methoden benötigt und nur die Implementierung von *getSymbol()* und *validMove()* variiert. Einzig der Bauer und der Turm besitzen noch zusätzliche Methoden, um die Sonderzüge En-Passant und Rochade ausführen zu können. Durch diese Strukturierung können problemlos noch weitere Figuren erstellt werden oder das Verhalten einer einzelnen verändert werden. Zudem Testen die Figuren lediglich ob der ihnen übergebene move ihrem Bewegungsmuster entspricht und nicht ob sie in der aktuellen Spielsituation diesen auch ausführen können. Dadurch wird die komplexität der Figuren deutlich verringert und es ist möglich unabhängig von den Figuren die allgemeinen Schachregeln abzuändern.

[3] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert.
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_figures.png

Paket model



[UML-Klassendiagramm model](#)[4]

Statische Sicht

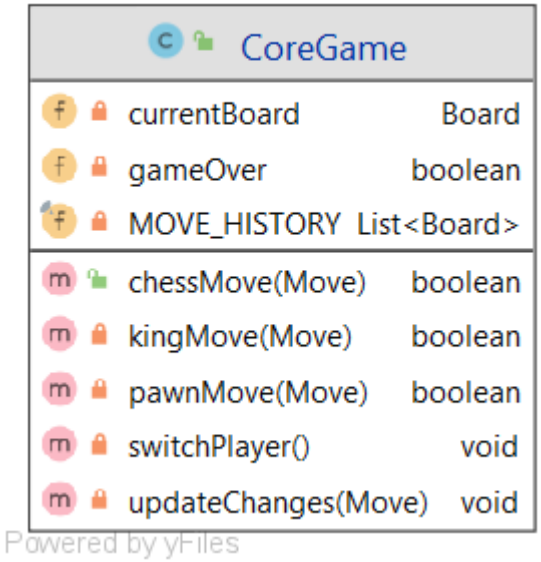
In der Klasse Rules sind die FIDE-Schachregeln implementiert und die Ausführung von diesen. Dazu werden Objekte wie das **Board**, das die **Figuren** enthält, die wiederum eine **Position** haben, benötigt. Zur Ausführung von den Regeln wird auch ein **Move** verwendet, der aus Start- und Zielposition der zu bewegenden Figur besteht.

Durch diese Aufteilung werden saubere Schnittstellen geschaffen, durch die problemlos auf die einzelnen Funktionen zugegriffen werden kann. Zudem kann auch eine Veränderung der Schachregeln leicht erfolgen.

Der **Parser** dient dazu, aus einem String ein Move-Objekt zu erstellen und zu überprüfen, ob die String-Eingabe der Standardform der FIDE-Schachregeln entspricht.

[4] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert.
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_model.png

Paket Controller



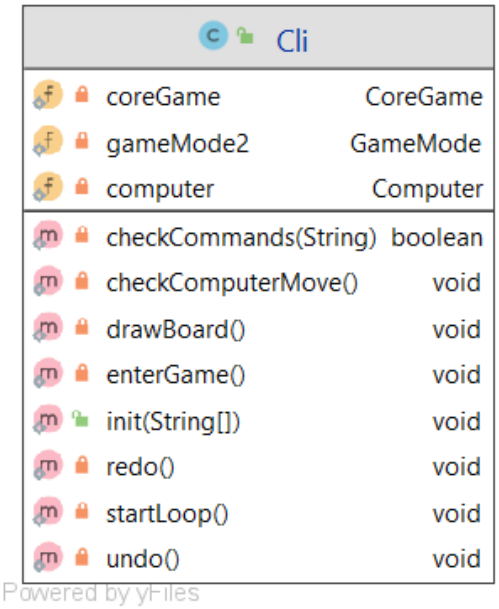
UML-Klassendiagramm controller[5]

Statische Sicht

Die Klasse **CoreGame** regelt den Spielablauf, sobald ein Spieler einen Zug ausführt. Dazu benötigt sie lediglich ein Board, das dann alle weiteren Informationen bereithält. Sie detektiert außerdem, wenn das Spiel durch eine Schachmatt- oder Patt-Situation beendet ist. Sie stellt so die Schnittstelle zwischen der View (CLI und GUI) und dem Model (Paket model und Paket Figuren) dar. Durch diese Struktur benötigt die View kein Wissen über das Model, sondern nur über die Schnittstelle zum Controller. Dadurch können View und Model problemlos ausgetauscht werden.

[5] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert. https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_controller.png

Paket cli



UML-Klassendiagramm cli[6]

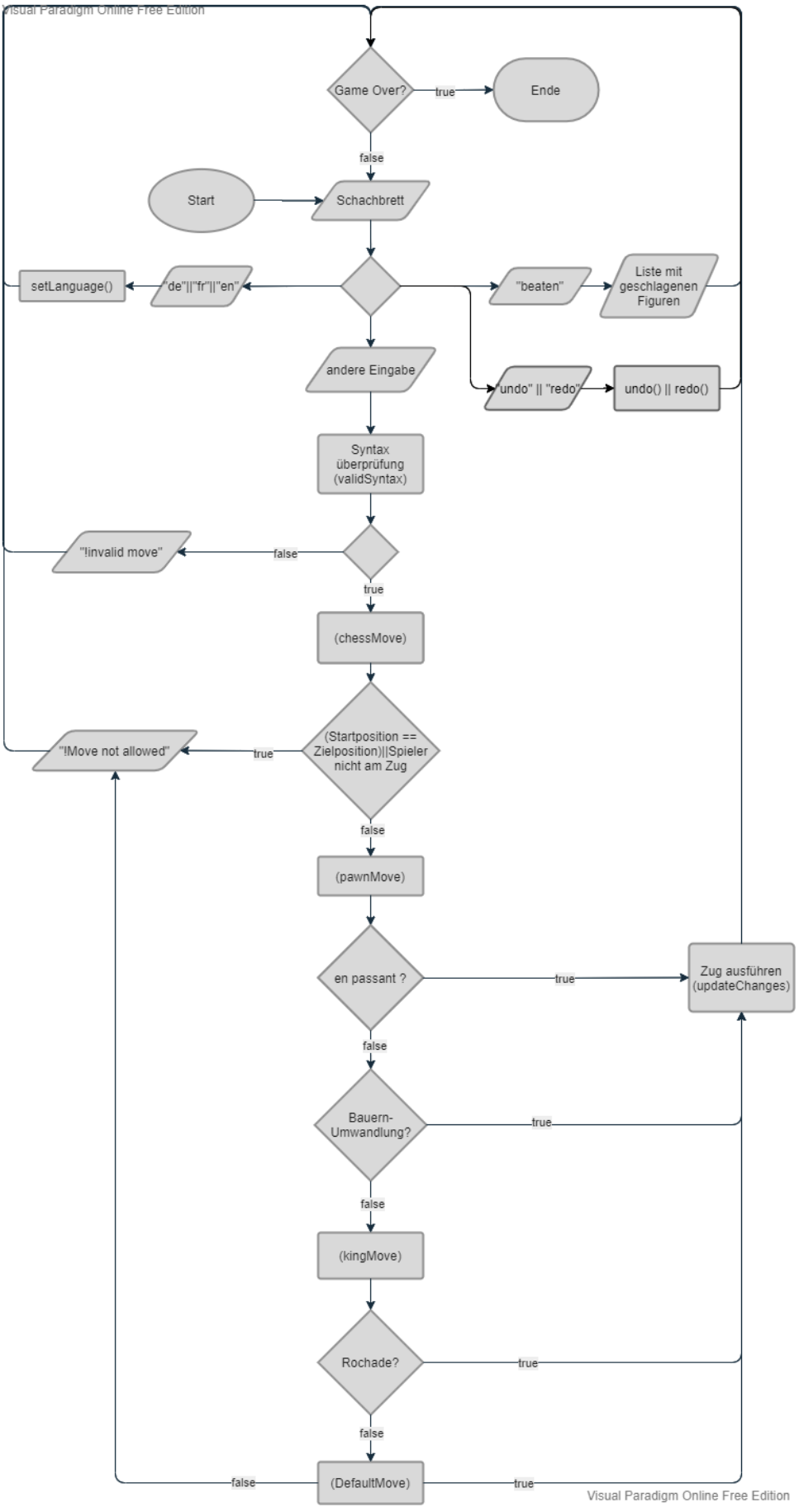
Statische Sicht

Die Klasse **Cli** regelt das Konsolenspiel. Dazu hat sie einen Scanner, um Eingaben von einem Spieler entgegen zu nehmen, und einen Computer, um auch von diesem Züge entgegen zu nehmen. Damit sie eine korrekte Ausgabe tätigen kann, besitzt sie zudem ein CoreGame. Durch die geringe Anzahl an Schnittstellen ist sie einfach modifizierbar.

Dynamische Sicht

Sobald das Spiel gestartet wird, wird in einer while-Schleife die Eingabe abgefragt und darauf reagiert. Bei einer Schachzug-Eingabe prüft das System, ob die Syntax korrekt war, dann ob der Zug semantisch gültig ist und führt anschließend den entsprechenden Zug aus. Das Spielfeld wird aktualisiert und das Spiel wartet auf eine erneute Eingabe des Spielers.
Das Programm reagiert außerdem auf die Eingabe “beaten”, indem die in einer Array-Liste gespeicherten geschlagenen Figuren ausgegeben werden, sowie auf die Spracheingaben “en”, ”de” oder “fr”, indem mithilfe eines Sprachmanagers die Sprache des Spiels auf Englisch, Deutsch oder Französisch gesetzt wird.
Zusätzlich sind als Zusatzfeature die Eingaben “undo” und “redo” gültig, die Züge rückgängig machen und wiederherstellen können. Hierzu werden die ausgeführten Züge in der Zug-Historie abgefragt und die entsprechenden Spielfelder wiederhergestellt.
Der Programmablauf für das Spiel auf der Konsole ist im Folgenden skizziert:

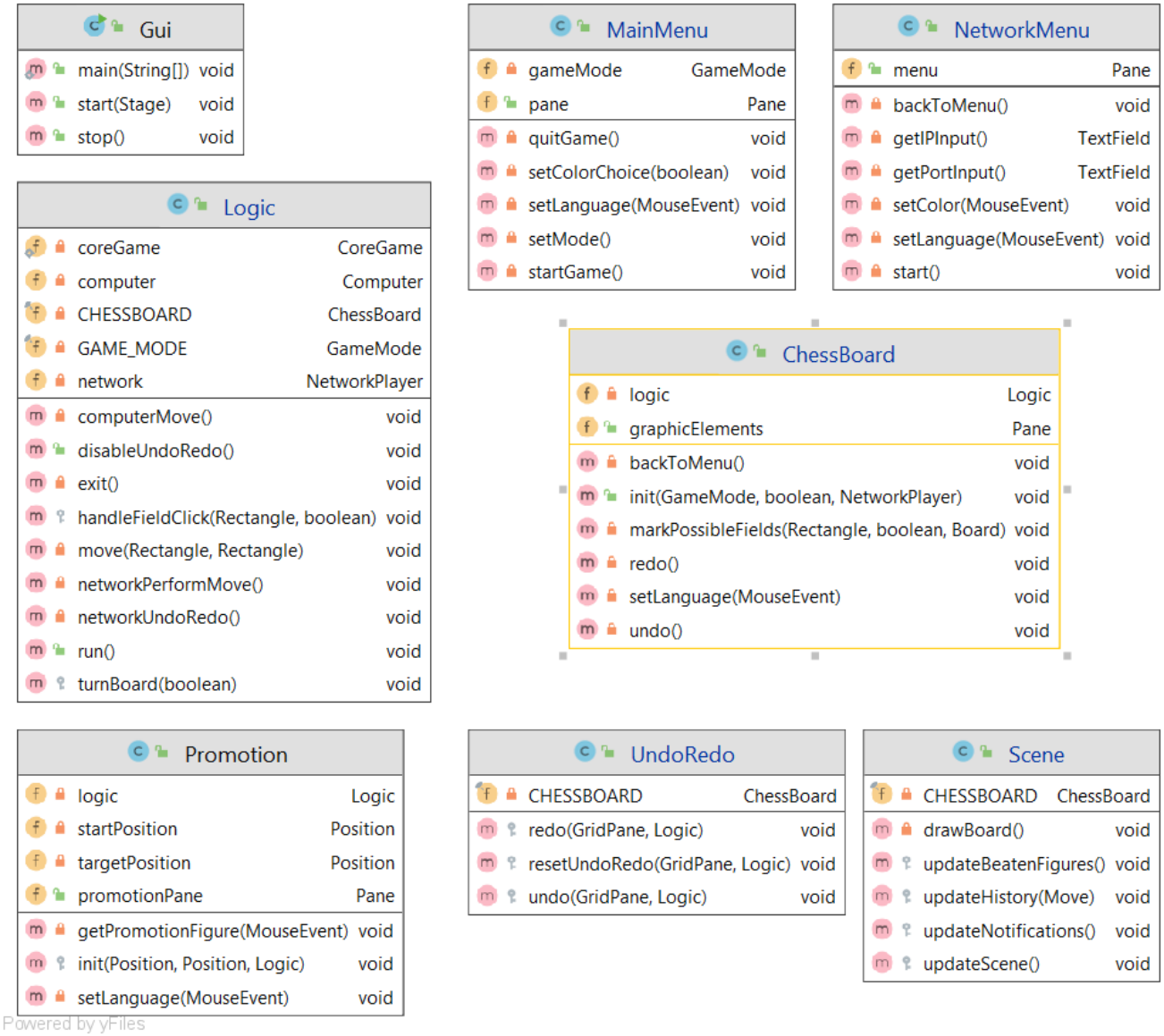
[6] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert.
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_cli.png



[Programmablaufplan cli][7]

[7] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/Programmablaufplan_CLI.png

Paket gui



[UML-Klassendiagramm gui][8]

Statische Sicht

In der GUI ist die Ein- und Ausgabe entsprechend angepasst, die interne Spiel-Logik bleibt aber dieselbe, da auch hier CoreGame gültige Eingaben prüft und Züge ausführt. Allein die Darstellung erfolgt nun über eine grafische Oberfläche. Hierfür wurde jedes anzuzeigende Fenster in ein fxml-Dokument ausgelagert.

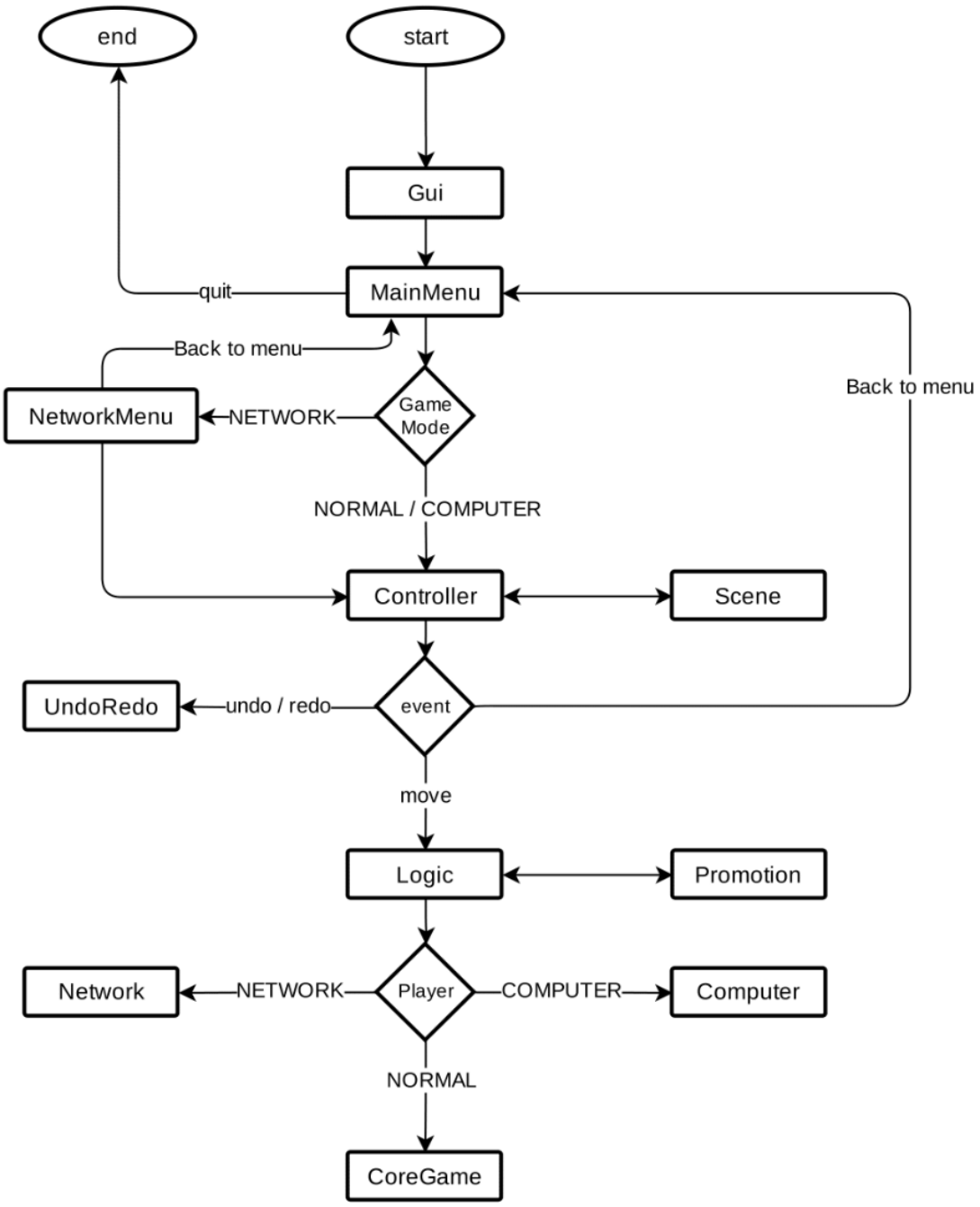
Wir unterscheiden in die Menü-Fenster MainMenu für das allgemeine Menü, in dem ausgewählt werden kann, ob ein Spiel gegen einen Freund, gegen den Computer oder ein Netzwerk-Spiel gestartet werden soll.

Jedes Fenster bietet außerdem die Funktion, die Anzeigesprachen zu wechseln.

[8] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert.
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_gui.png

Startet man vom Hauptmenü nun ein Computer- oder Freundschaftspiel, wird man direkt zum Spielfeld geleitet. Startet man ein Netzwerkspiel, wird das zweite Menüfenster, das NetworkMenu geöffnet, in dem dann IP-Adresse und Port eingegeben werden können, um ein neues Spiel zu starten oder einem bereits bestehenden Spiel beizutreten.
Weiterhin gibt es ein Pop-Up-Fenster, das erscheint, wenn eine Bauernumwandlung auftritt.

Dynamische Sicht



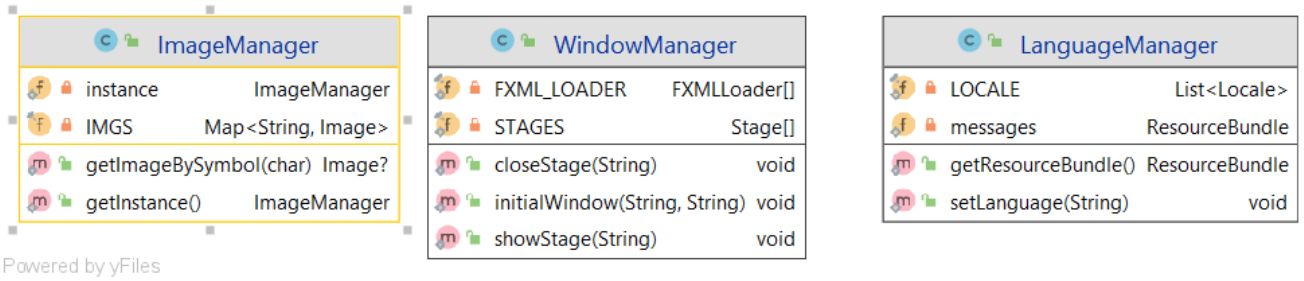
Ablaufplan Gui[9]

[9] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/Ablaufdiagramm_GUI.pdf

Die Gui ist für die grafische Oberfläche verantwortlich.
Wie im Ablaufdiagramm sichtbar ist die Klasse **Gui** einzig und allein als Startpunkt für das grafische Spiel zuständig und erzeugt das **MainMenu**, das ebenfalls nur weiterleitende Funktionen enthält.
Im Fall eines Netzwerk-Spiels, wird das **NetworkMenu** aufgerufen, indem alle wichtigen Einstellungen für das Netzwerk getroffen werden können und erst daraufhin erfolgt eine Weiterleitung zum Schachbrett.
Im Fall eines Spiels gegen den Computer oder gegen einen lokalen Gegner wird direkt zum Schachbrett weitergeleitet.
Das **ChessBoard** enthält die FXML-Datei des Schachbretts mit seinen Einstellungsmöglichkeiten. Um auf alle funktionalen Elemente zuzugreifen besitzt er viele Getter. Die Methoden mit einer einfachen Logik (z.B.backToMenu()) sind direkt im ChessBoard implementiert, wohingegen komplexe Funktionen wie undo, redo oder ein Schachzug eine Methode haben, die nur eine Weiterleitung zur eigentlichen Methode beinhaltet.
Die Klasse **Logic** ist für die Schachzüge verantwortlich, insbesondere dafür welcher Spieler den nächsten Zug machen darf und um aus zwei angeklickten Feldern einen Move zu erzeugen. Dazu besitzt sie eine Schnittstelle zum CoreGame, sowie im Fall eines Netzwerk-Spiels einen Netzwerkspieler oder im Fall eines Spiels gegen den Computer einen Computer.
Die Klasse **Scene** kümmert sich ausschließlich, um die Aktualisierung des Displays nach einer Aktion und wird von ChessBoard erzeugt und besitzt dieses, damit sie auf das zu aktualisierende Schachbrett zugreifen kann.
Die Klasse **UndoRedo** ist für die Logik von undo und redo verantwortlich. Sie wird ebenfalls vom ChessBoard erzeugt und besitzt dieses.
Die Klasse **Promotion** beinhaltet ein Auswahlfenster für die Bauernumwandlung. Sie wird von der Logic erzeugt falls einer Bauer das andere Ende des Spielfeldes erreicht. Zudem besitzt sie auch die Logic, damit sie die Umwandlungsfigur wieder zurück geben kann.

Durch die Auslagerung komplexer Funktionen aus dem ChessBoard, wird die Übersichtlichkeit gewährt und es entstehen klar trennbare Funktionseinheiten die unabhängig voneinander behandelt werden können.

Paket managers



UML-Klassendiagramm managers[10]

Statische Sicht

Der **ImageManager** kümmert sich um alle Bilder, die während des Spiels geladen werden müssen.

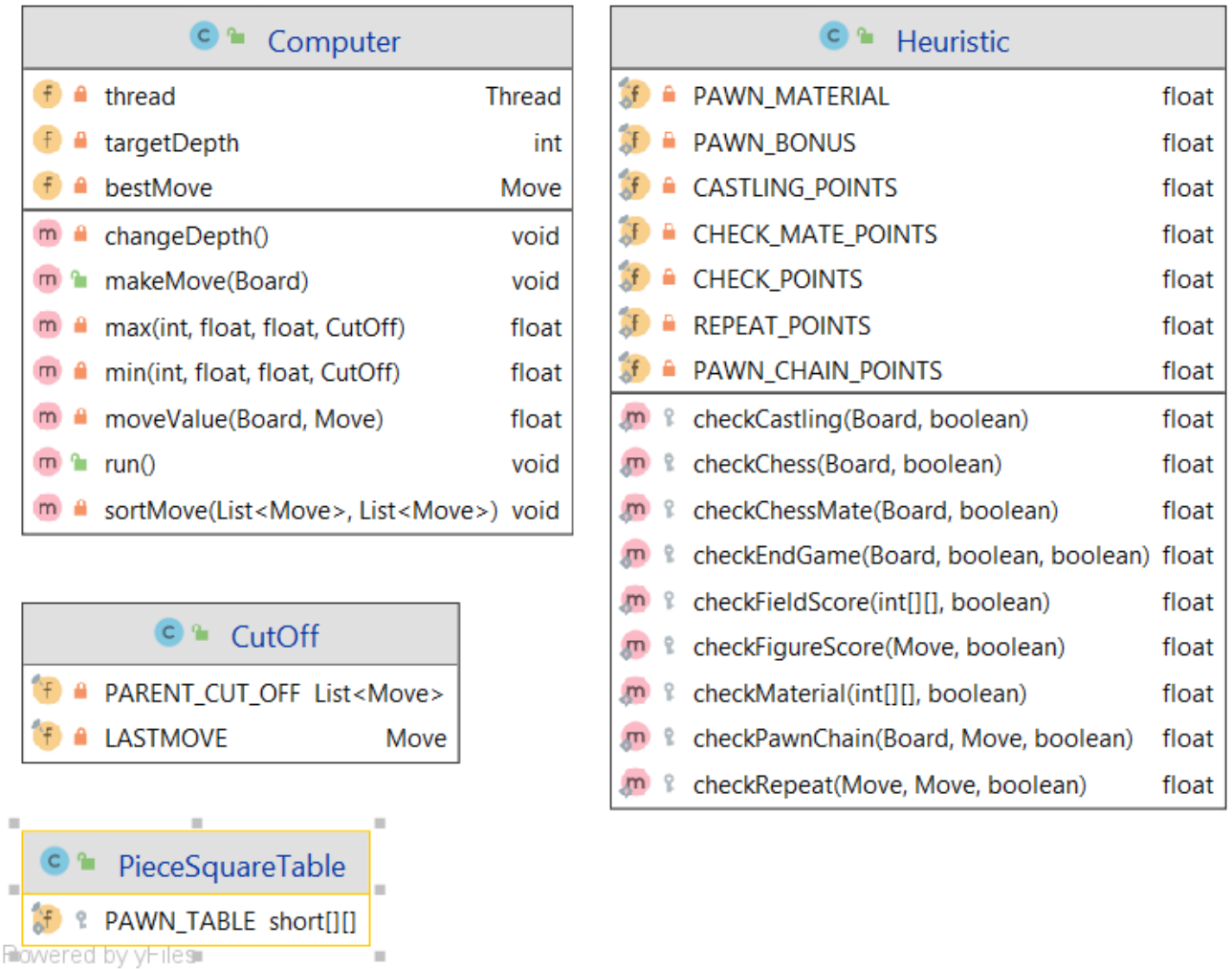
Der **WindowManager** kümmert sich um das Initialisieren, Öffnen und Schließen der verschiedenen Fenster.

Der **LanguageManager** kümmert sich um die Aktualisierung der Sprache in Gui und Cli.

Durch die Erstellung der Managerklassen wurde verhindert, dass derselbe Code an vielen Stellen im Programm steht. Dadurch wird eine zentrale Bearbeitung der einzelnen Funktionen ermöglicht.

[10] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert.
https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_managers.png

Paket ai



[UML-Klassendiagramm](#)[11]

Statische Sicht

Die Klasse **Computer** ist in diesem Paket der Hauptakteur, da in ihr die Logik für einen Computerzug steckt. In dieser Klasse wird durch Alpha-Beta-Pruning der bestmögliche Zug für den Computer ermittelt.

Die Güte eines Zuges wird über einen Score ermittelt, der sich aus verschiedenen Aspekten zusammensetzt. Für jeden möglichen Zug wird nun ein solcher Score berechnet und im Suchbaum gespeichert.

Diese Scores werden in der Methode *moveValue()* in der Klasse Computer ermittelt. Hierzu werden die weiteren Klassen Heuristic und PieceSquareTable aufgerufen, die jeweils verschiedene Aspekte eines strategischen Schachspiels und taktische Züge beachten. Die Klasse **PieceSquareTable** beinhaltet Tabellen, die als Bewertungsgrundlage für verschiedene Brettstellungen für die verschiedenen Figuren

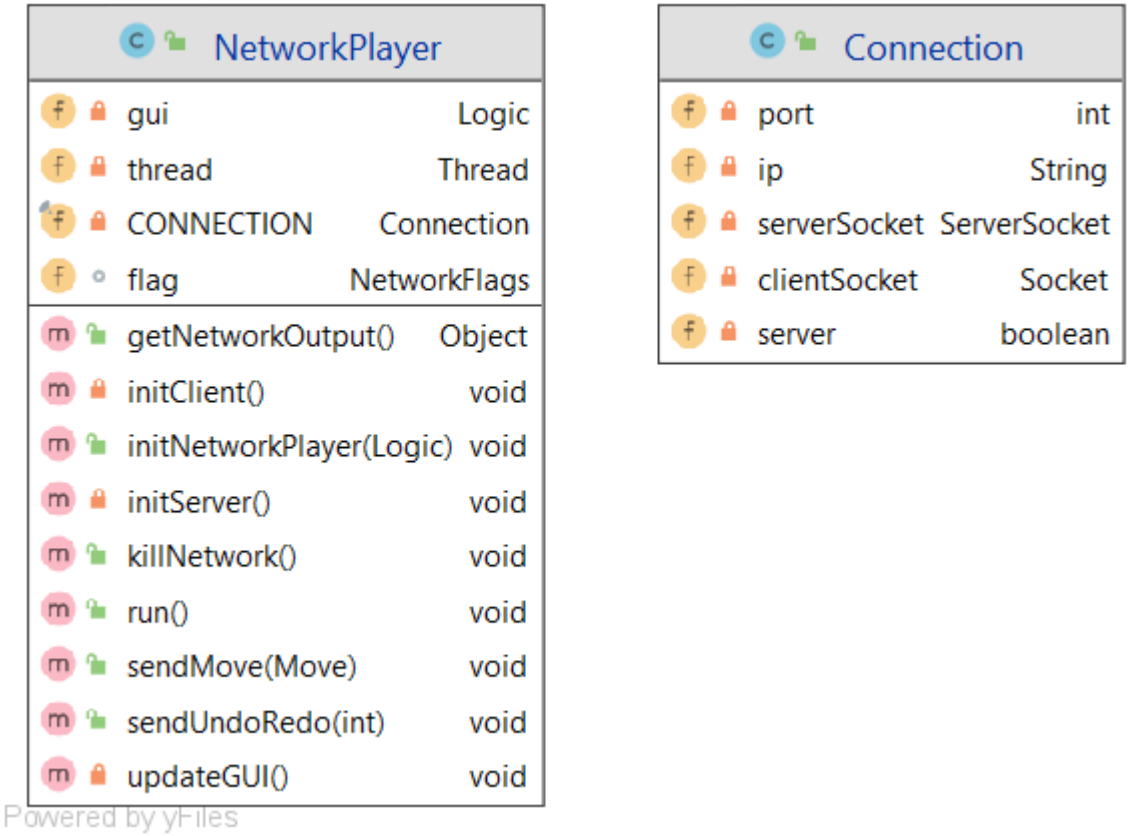
[11] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert. https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_ai.png

genutzt werden. Die Klasse **Heuristic** speichert die Materialwerte der Figuren und ebenso die Bonuspunkte für verschieden taktische Züge wie eine Rochade oder eine Bauernkette.
Die Klasse **CutOff** dient dazu, die KI schneller zu machen, indem die Züge vorsortiert und in der Klasse gespeichert werden, um möglichst viele CutOffs zu generieren. So kann der Suchbaum schneller durchlaufen werden.
Ein Spiel gegen den Computer ist sowohl im Spiel in der Konsole als auch im Spiel mit der GUI möglich. Hierfür wird der vom Computer berechnete Zug übergeben (*getMove()*).

Dynamische Sicht

Wird in der CLI oder GUI ein Computerzug angefordert, wird die Methode *makeMove()* in der Klasse **Computer** aufgerufen. Diese startet einen neuen Thread. In diesem Thread wird nun das Alpha-Beta-Pruning ausgeführt. Hierfür wird ein Suchbaum generiert. Dieser besitzt eine maximale Tiefe, die mit der Anzahl der geschlagenen Figuren erhöht wird. Der Suchbaum wird zudem durch CutOffs verkleinert. Ist die erwartete Suchtiefe erreicht, so wird der beste Zug in bestMove gespeichert. BestMove ist hierbei der Zug mit dem höchst möglichen Score.
Um den Score zu berechnen werden die Klassen **Heuristic** und **PieceSquareTable** von der Methode *moveValue()* aufgerufen. Sie berechnen den Wert des Zuges und geben diesen als den jeweils aktuellen bestMove zurück. Ist der Suchbaum fertig durchlaufen, so wird der Thread beendet und der berechnete Zug kann abgerufen werden.

Paket network



UML-Klassendiagramm network[12]

Statische Sicht

Beim Netzwerkspiel haben wir uns für eine Client-Server-Architektur entschieden. Ein Spieler startet das Spiel und bildet somit den Server, ein zweiter Spieler tritt dem Netzwerkspiel als Client bei. Sobald die Verbindung hergestellt ist und das Spiel initialisiert ist sind Server und Client als gleichwertig zu betrachten und beide durchlaufen dieselbe *run()*-Methode.

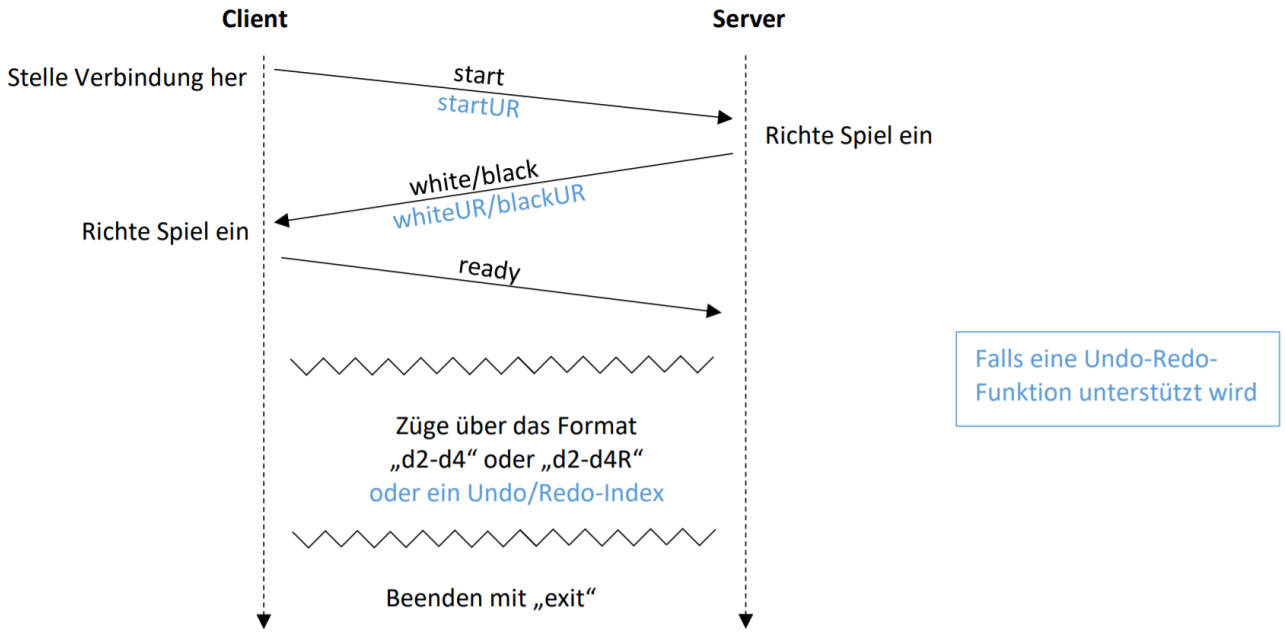
Da dieses Feature nur in Verbindung mit der grafischen Oberfläche funktionieren muss, ist die Klasse auch die Einzige, die auf das Package network zugreift. Hierzu wird die Klasse **Logic** im Package gui genutzt, die in der run-Methode den Spielmodus abfragt. Ist dieser im Netzwerk-Modus, so wird das Netzwerkspiel in der GUI aufgebaut. Die Klasse **NetworkPlayer** in dem Package network kümmert sich hierbei um die Logik hinter einem Netzwerk-Spiel und die Klasse **Connection** um die eigentliche Verbindung.

[12] Dieses UML-Diagramm entspricht nicht der Norm und wurde aus Übersichtsgründen reduziert. https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/uml%20class%20diagrams/uml_network.png

Dynamische Sicht

Hierbei kommunizieren sie über einen vorher festgelegten freien Port, z.B. Port 5555, wobei der Spieler, der den Server stellt, diesen Port freigeben muss, damit der Client-Spieler darauf zugreifen kann. Der Server-Spieler muss dem Client-Spieler entsprechend (extern) seine IP-Adresse mitteilen. Zudem legt der Server-Spieler zu Anfang fest, mit welcher Farbe gespielt werden soll. Die Farbe des Client-Spielers wird dem Client als “black” oder “white” übermittelt. Haben Beide nun das Spiel eingerichtet, beginnt der als weiß Spielende mit dem ersten Zug.
Es wird davon ausgegangen, dass alle gesendeten Züge sowohl syntaktisch als auch semantisch korrekt sind. Dies wird im jeweiligen Controller d vor dem Verschicken der Züge überprüft.

Die Kommunikation läuft nun wie folgt ab:



Hierbei dient das Versenden von “start” dazu, dass der Client sich mit dem Server verbindet. Zudem wird durch das Anhängen von “UR” dem Server mitgeteilt das der Client die Undo-Redo-Funktion besitzt.
Das Versenden von “white/black” von dem Server an den Client dient, dazu dass dem Client seine Spielerfarbe zugewiesen wird. Hier bedeutet das Anhängen von “UR”, dass der Server die Undo-Redo-Funktion unterstützt.
Anschließen versendet der Client “ready” an den Server, um mitzuteilen, dass er sein Spiel initialisiert hat und nun bereit ist, Züge auszutauschen.