Architektur Dokumentation

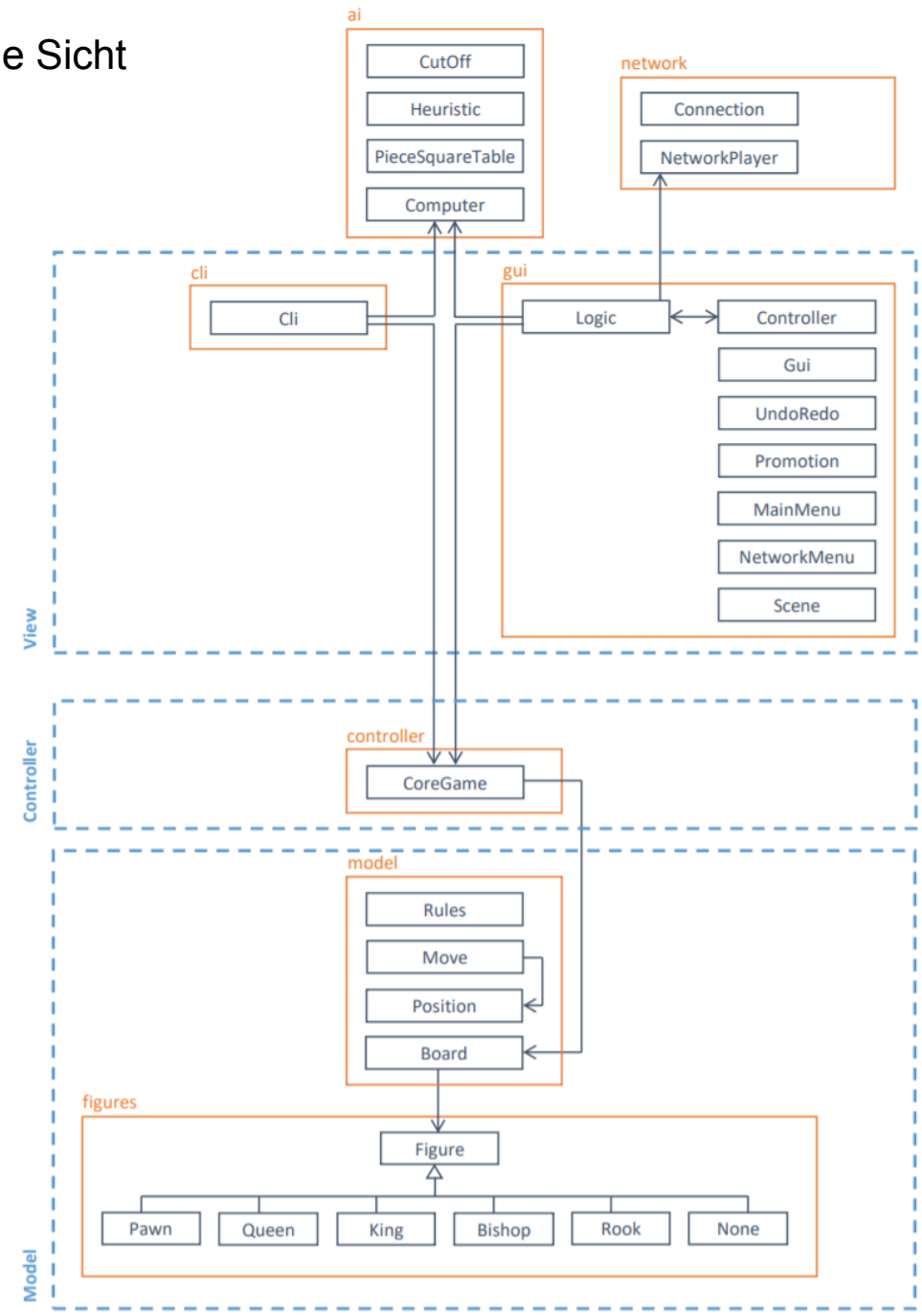
Inhaltsverzeichnis

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

- View: Die Packages CLI und GUI übernehmen dabei die Aufgabe der Darstellung nach außen, also auf der Konsole oder als grafische Schnittstelle. Zudem nehmen sie Eingaben entgegen und passen diese für die Schnittstelle zum Controller an.
- Model: Die Klassen **Board**, **Move** und **Position** dienen vor allem der Speicherung der einzelnen Objekte, wobei Board zusätzlich noch Spielsituationen wie Schach, Schachmatt und Patt erkennt. Das Paket Figures und die Klasse **Rules** beinhalten die Schachregeln.
- Controller: Die Klasse **CoreGame** regelt den Ablauf des Spiels.

Der Controller stellt die Schnittstelle zwischen der View und dem Model dar. Durch diese Struktur benötigt die View kein Wissen über das Model, sondern nur über die Schnittstelle zu dem Controller. So können View und Model problemlos ausgetauscht werden und die einzelnen Komponenten für sich gewartet werden.

Anmerkung zur Projektorganisation: Im Laufe des Projektes erwies sich dieses Pattern auch als sinnvoll, um Aufgabenbereiche aufzuteilen und so eine 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.

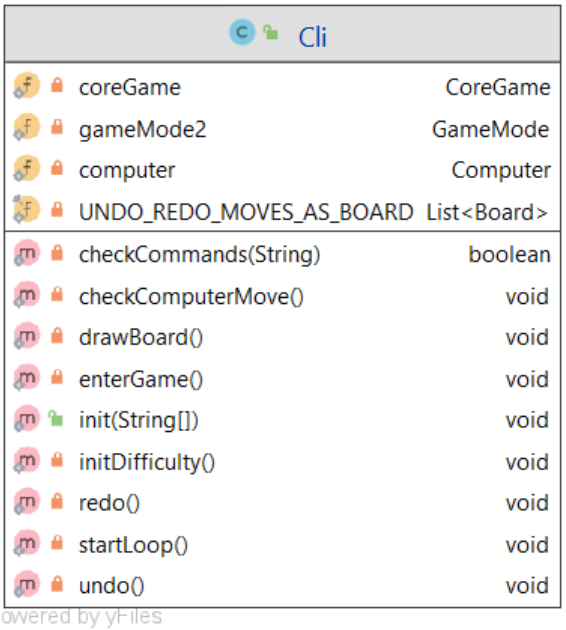
Dynamische Sicht

Ein Schachspiel wird entweder von der Cli oder von der Gui aus gestartet. Diese erzeugen dann ein CoreGame, das sein eigenes Board besitzt, und auch einen Computer- oder Netzwerkspieler, sofern dies ausgewählt wurde. Sobald es zu einer Eingabe durch einen Spieler kommt, nimmt die View diese entgegen und passt sie anschließend für die Schnittstelle zum Controller an. Der Controller ruft dann die entsprechenden Methoden des Boards und der Rules auf. Danach greift die View über den Controller auf das Board zu und aktuallisiert so ihre Ausgabe.

Im Folgenden gehen wir noch einmal im Detail auf die einzelnen Pakete sowohl aus statischer als auch aus dynamischer Sicht ein.

Statische und dynamische Sicht der Pakete

Paket cli



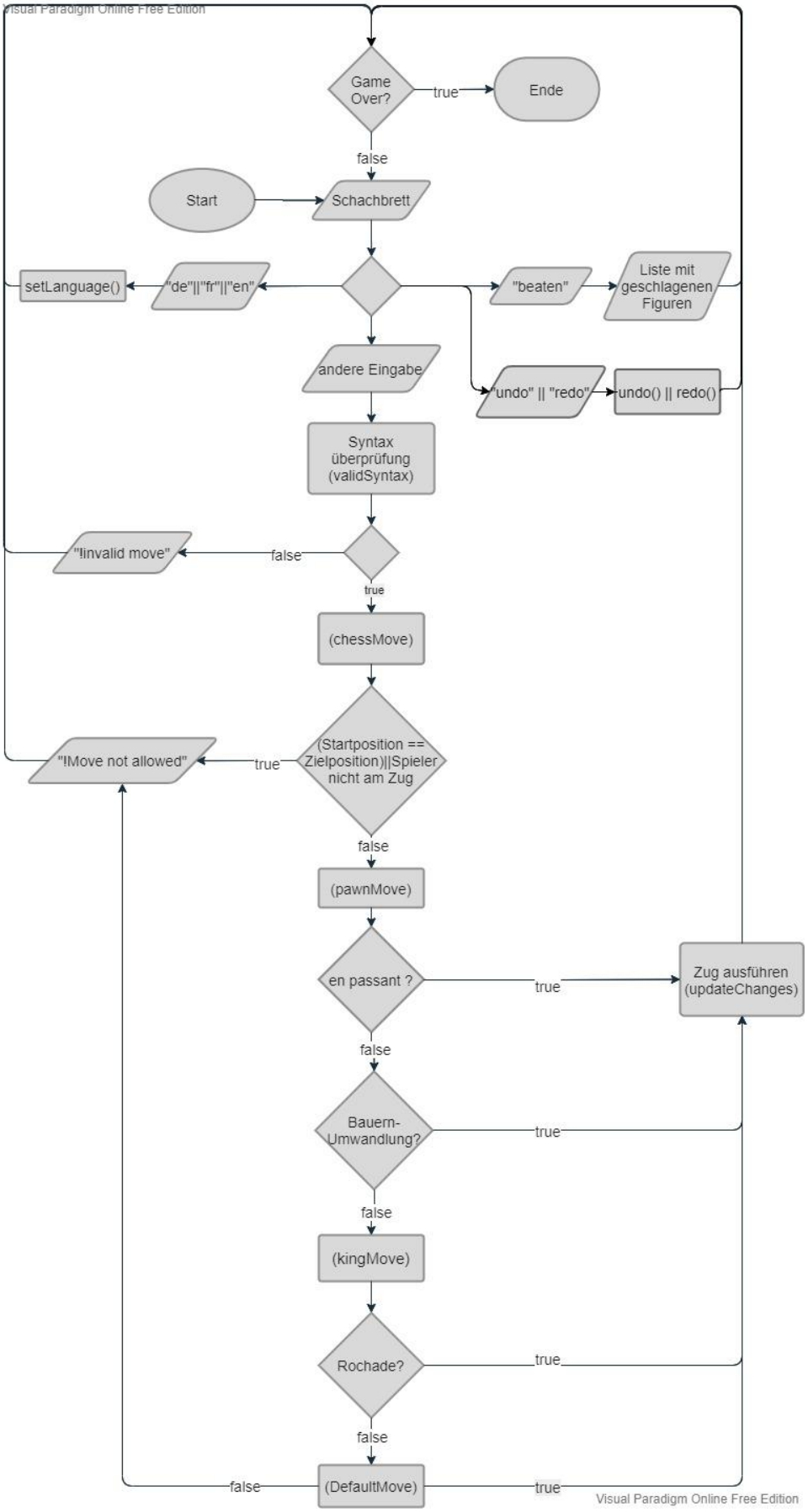
UML-Klassendiagramm cli[3]

Die Klasse **Cli** regelt das Konsolenspiel. Sie nimmt dazu mithilfe eines Scanner-Objekts Eingaben von einem Spieler entgegen und besitzt gleichzeitig eine Computer-Instanz, um auch von diesem Züge entgegenzunehmen. Damit sie eine korrekte Ausgabe tätigen kann, besitzt sie zudem ein CoreGame.

Sobald das Konsolenspiel gegen einen Freund gestartet wird, wird in einer while-Schleife die Eingabe des Benutzers abgefragt und darauf reagiert. Bei einer Schachzug-Eingabe wird zunächst die Syntax der Zugeingabe geprüft und anschließend, ob der Zug auch semantisch gültig ist. Ist dies der Fall, führt sie den entsprechenden Zug aus. Das Spielfeld wird aktualisiert und das Spiel wartet auf eine neue Eingabe des Spielers.
Im Spiel gegen den Computer ist der Ablauf derselbe bis auf das zuvor noch der Schwierigkeitsgrad ausgewählt werden kann und jeder zweite ZUg von dem Computer ausgeführt wird.
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.
Bei den Eingaben “undo” und “redo” wird auf die MoveHistory aus CoreGame zugegriffen und der entsprechende Spielzustand geladen.

[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_cli.png

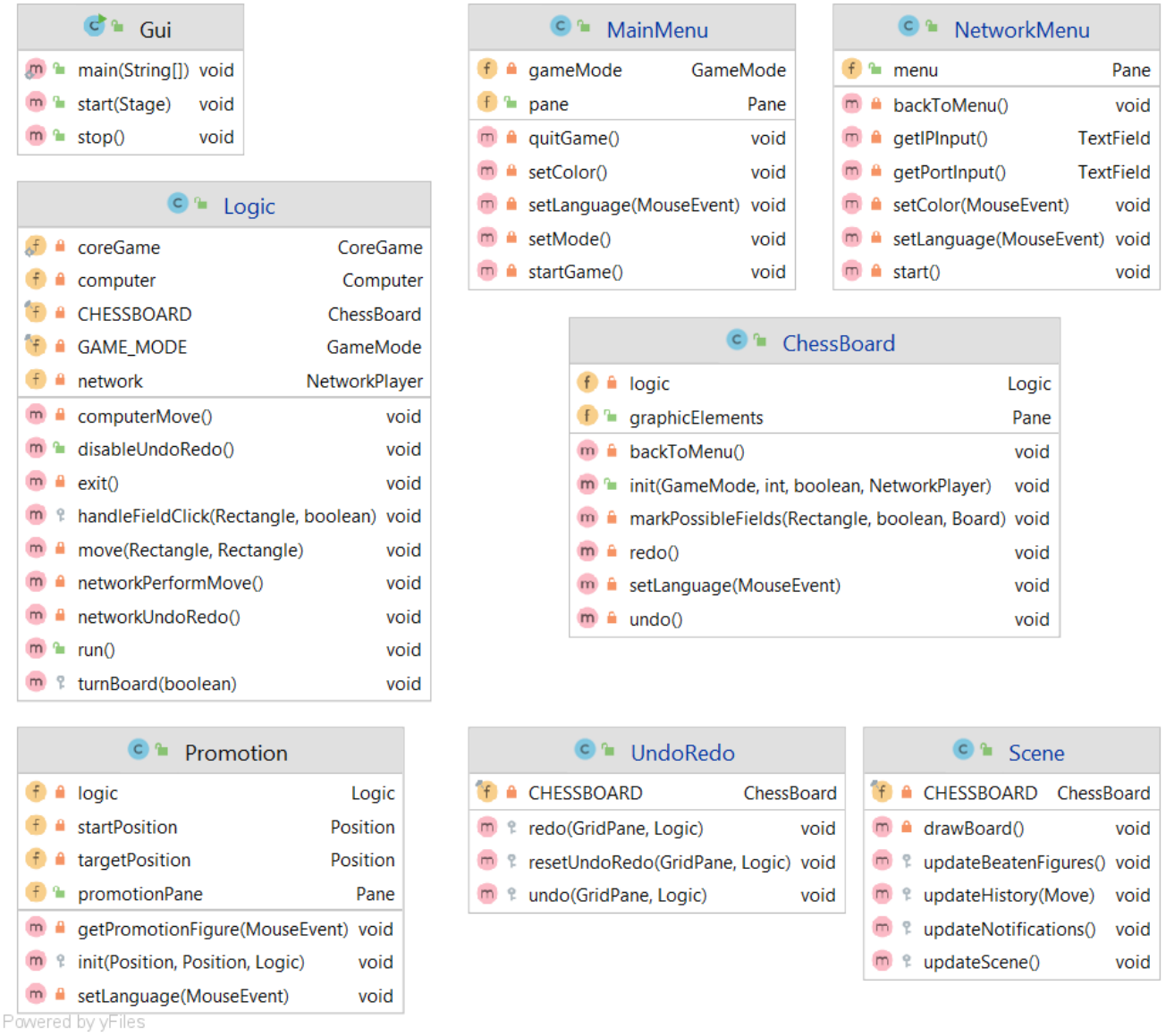
Der Programmablauf für das Spiel auf der Konsole ist im Folgenden skizziert:



[Programmablaufplan cli](#)[4]

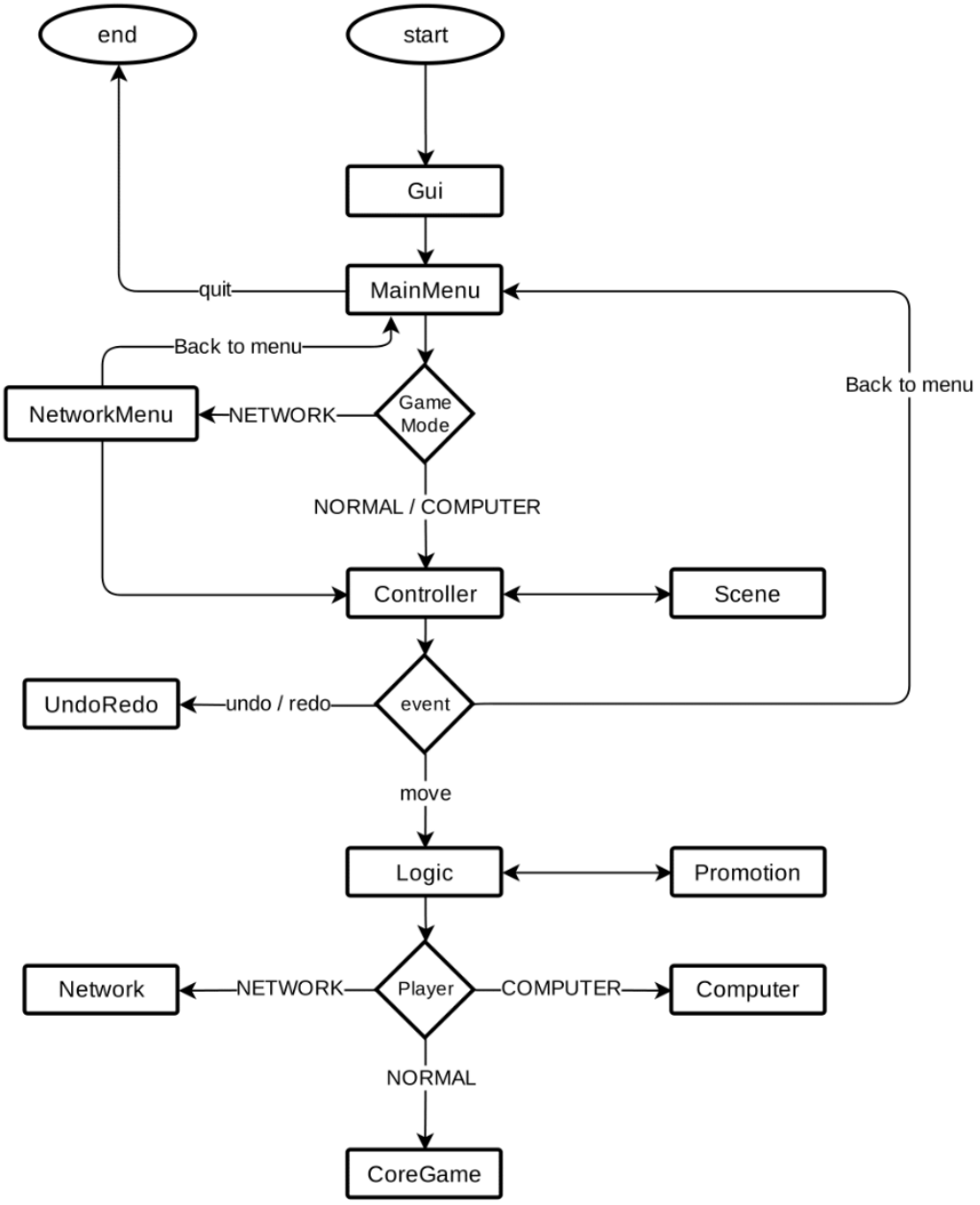
[4] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/Programmablaufplan_CLI.png

Paket gui



[UML-Klassendiagramm gui][5]

[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_gui.png



Ablaufplan Gui[6]

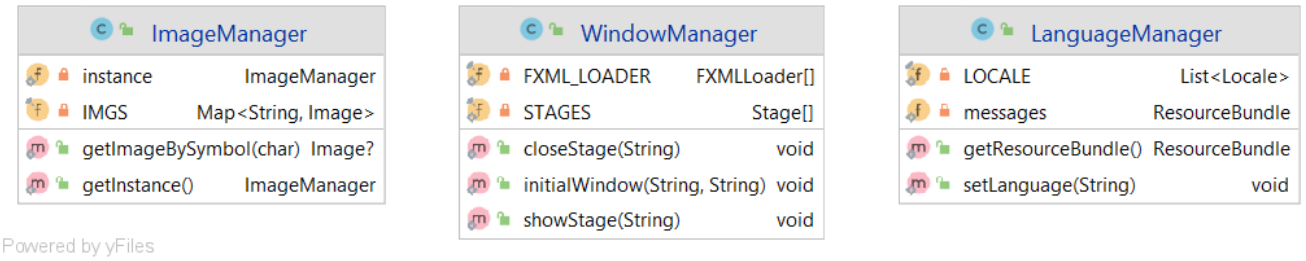
[6] https://projects.isp.uni-luebeck.de/gruppe-10/schach/-/blob/main/images/Ablaufdiagramm_GUI.pdf

Die Gui ist für die grafische Oberfläche verantwortlich.
In der GUI ist die Ein- und Ausgabe entsprechend angepasst, die interne Spiel-Logik bleibt aber dieselbe wie beim Konsolenspiel. Auch hier prüft **CoreGame** die Eingaben und führt die Züge aus. Die Darstellung und die Interaktion mit dem Nutzer erfolgt nun über eine grafische Oberfläche. Hierfür wurde jedes anzuzeigende Fenster in ein FXML-Dokument ausgelagert.
Jedes Fenster bietet die Funktion, die Anzeigesprachen zu wechseln.

Wie im Ablaufdiagramm sichtbar, dient die Klasse **Gui** allein als Startpunkt für das grafische Spiel und erzeugt das **MainMenu**, das ebenfalls nur weiterleitende Funktionen enthält.
Bei einem Netzwerkspiel wird das **NetworkMenu** aufgerufen, in dem alle wichtigen Einstellungen für das Netzwerk getroffen werden können. Anschließend erfolgt eine Weiterleitung zum Schachbrett.
Bei einem Spiel 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 lediglich eine Weiterleitung zur eigentlichen Methode enthalten.
Die Klasse **Logic** ist für die Schachzüge verantwortlich, kann also aus zwei angeklickten Feldern einen Move erzeugen und regelt auch, welcher Spieler den nächsten Zug machen darf. Dazu besitzt sie eine Schnittstelle zum CoreGame, sowie Netzwerkspieler für das Netzwerkspiel und einen Computer für ein Spiel gegen den Computer.
Die Klasse **Scene** kümmert sich ausschließlich um die Aktualisierung des Displays nach einer Aktion. Sie wird von ChessBoard erzeugt und besitzt auch ein ChessBoard, 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 ein ChessBoard.
Die Klasse **Promotion** beinhaltet ein Auswahlfenster für die Bauernumwandlung. Sie wird von der Logic erzeugt, falls ein Bauer das gegnerische Ende des Spielfeldes erreicht. Sie besitzt außerdem eine Logic, damit sie die Umwandlungsfigur wieder zurückgeben 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[7]

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

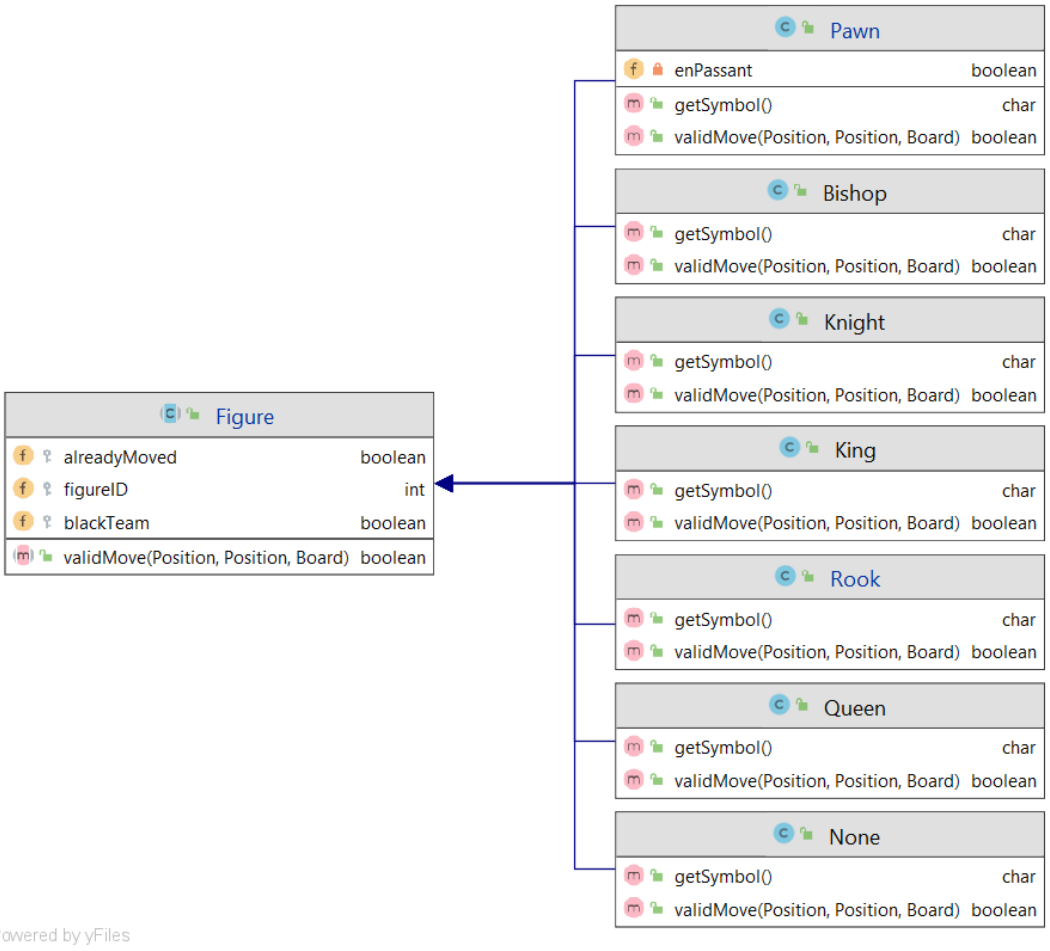
Der **WindowManager** kümmert sich um das Initialisieren, Öffnen und Schließen der verschiedenen Fenster.

Der **LanguageManager** kümmert sich um die Aktualisierung der Sprache in der GUI und in der Konsole.

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.

[7] 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 figures



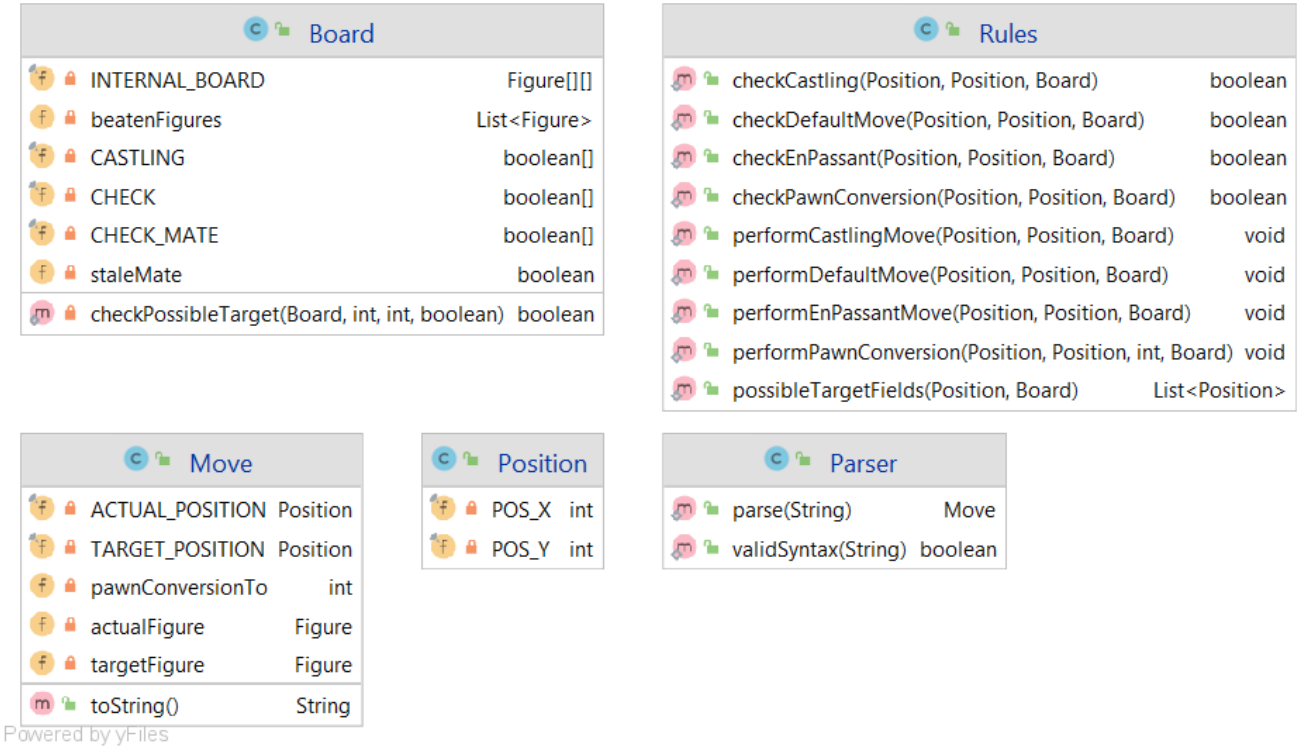
UML-Klassendiagramm figures[8]

In diesem Paket befinden sind alle Figuren. Diese erben von der abstrakten 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 bzw. Rochade ausführen zu können. Durch diese Strukturierung können problemlos weitere Figuren erstellt werden oder das Verhalten einzelner Figuren abgeändert werden.

Dabei ist zu beachten, dass Figuren in der Methode *validMove()* lediglich testen, ob der ihnen übergebene Zug ihrem Bewegungsmuster entspricht, und nicht, ob sie diesen in der aktuellen Spielsituation auch ausführen können. Dadurch wird die Komplexität der Figuren deutlich verringert und es ist möglich, die allgemeinen Schachregeln unabhängig von den Figuren zu ändern.

[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_figures.png

Paket model



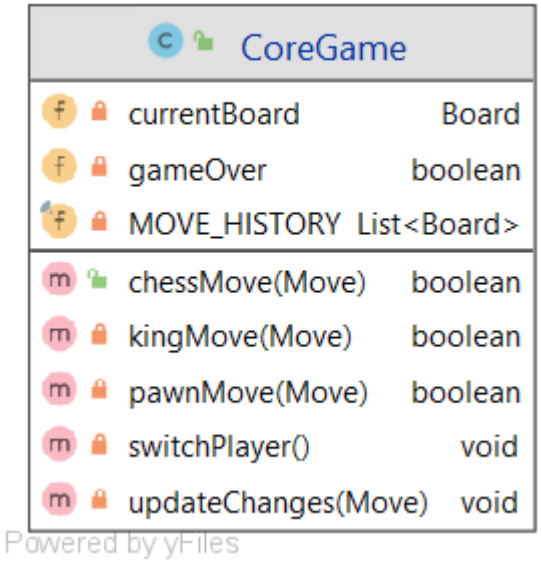
[UML-Klassendiagramm model][9]

In der Klasse **Rules** sind die FIDE-Schachregeln implementiert. Dazu wird Zugriff auf das **Board** benötigt, das die Figuren enthält, die wiederum eine **Position** haben. Zur Ausführung der Regeln wird ein **Move** benötigt, der aus Start- und Zielposition der zu bewegenden Figur besteht.

Der **Parser** wandelt einen String in ein Move-Objekt um und überprüft dabei, ob die String-Eingabe der Standardform der FIDE-Schachregeln entspricht.

[9] 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



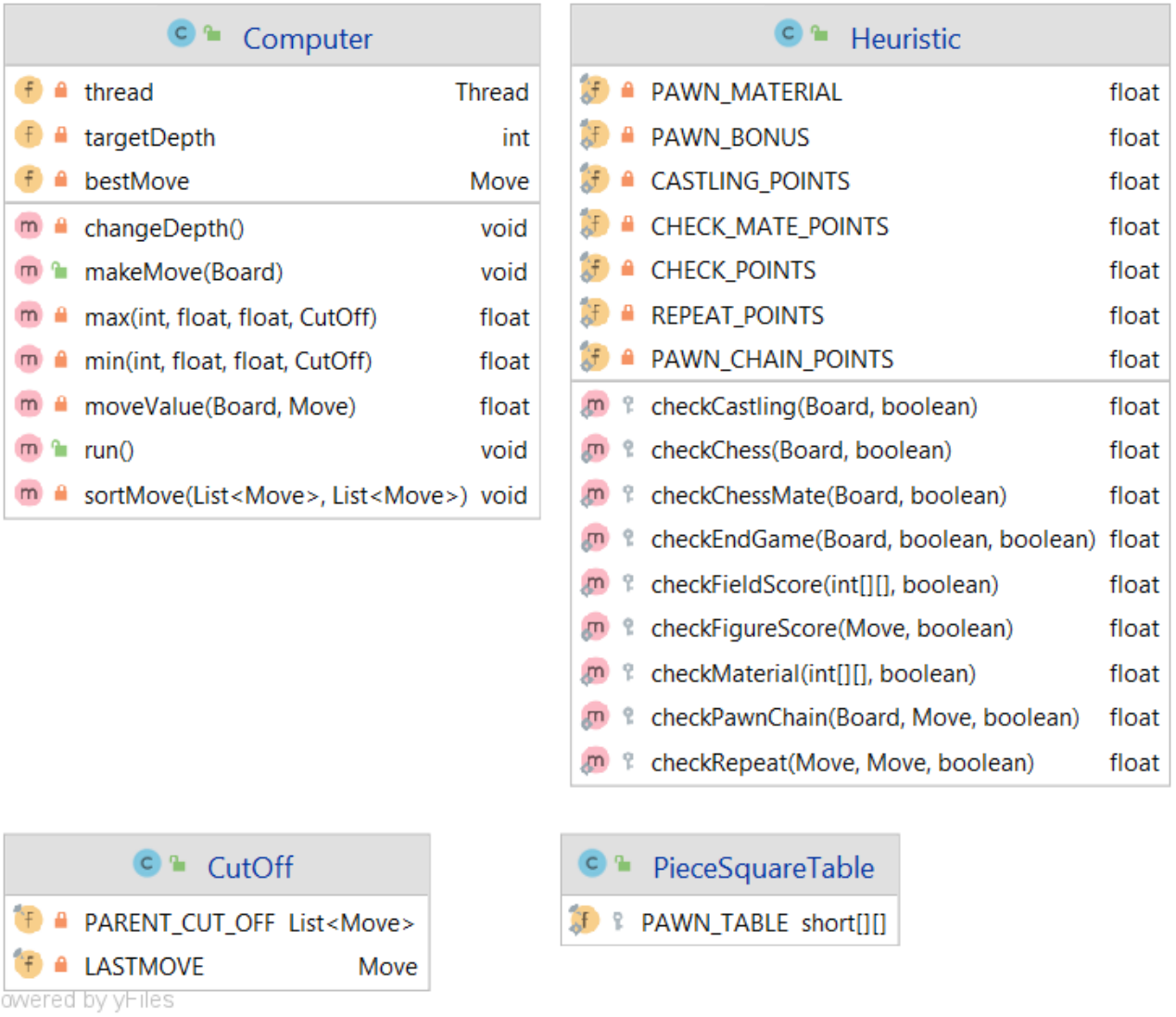
UML-Klassendiagramm controller[10]

Die Klasse **CoreGame** regelt den Spielablauf, sobald ein Spieler einen Zug ausführt. Dazu benötigt sie lediglich ein Board, das alle weiteren Informationen enthält. Sie detektiert außerdem, wenn das Spiel durch eine Schachmatt- oder Patt-Situation beendet wird.

Für die Zusatzfunktion Undo/Redo speichert CoreGame die gültigen Spielzustände in einer Array-Liste (MoveHistory). Soll in der Cli oder in der Gui ein Undo bzw. Redo ausgeführt werden, wird der entsprechende Eintrag aus dieser Liste geladen.

[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_controller.png

Paket ai



[UML-Klassendiagramm](#)[11]

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

Die Klasse **PieceSquareTable** beinhaltet Tabellen, die als Bewertungsgrundlage für verschiedene Brettstellungen für die verschiedenen Figuren genutzt werden.

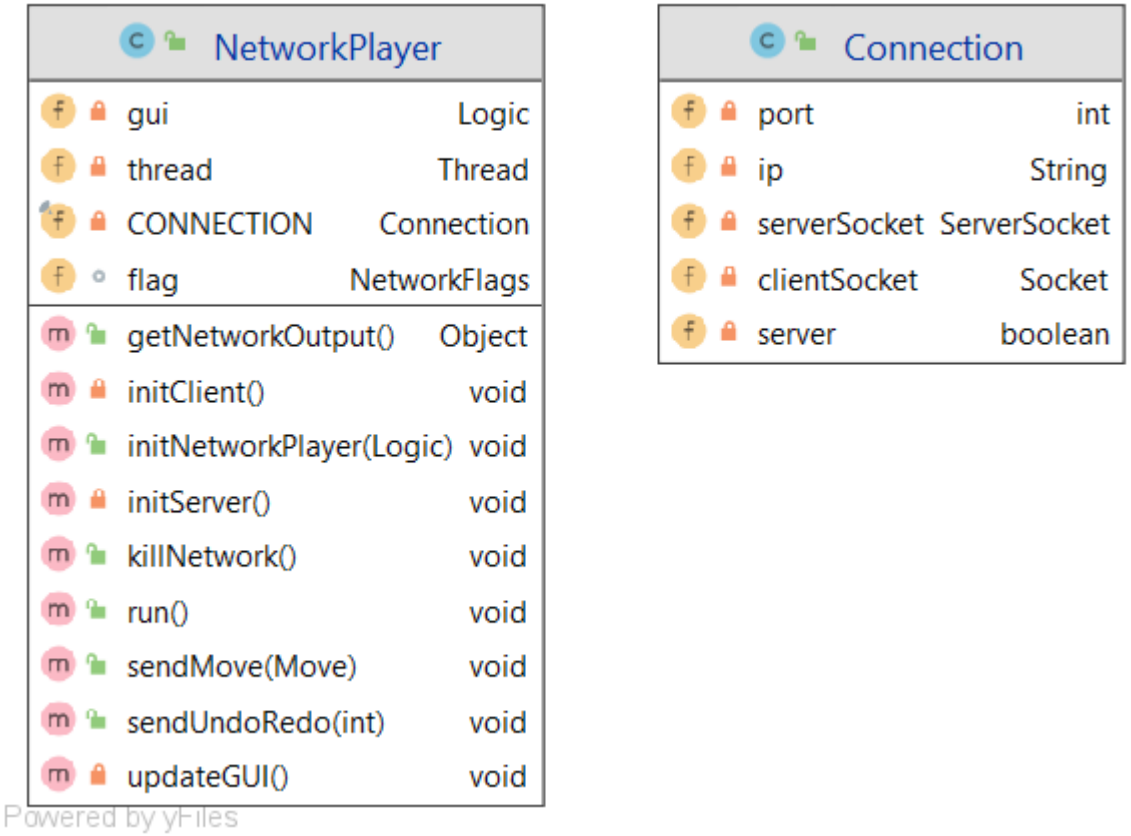
Die Klasse **Heuristic** speichert die Materialwerte der Figuren und ebenso die Bonuspunkte für verschieden taktische Züge wie eine Rochade oder eine Bauernkette.

Die Klasse **CutOff** beschleunigt die KI, indem die Züge vorsortiert und in der Klasse gespeichert werden, um möglichst viele CutOffs zu generieren. So kann der Suchbaum schneller durchlaufen werden.

[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

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 davon abhängig ist, welcher Schwierigkeitsgrad zu Beginn gewählt wurde. Im einfachen Modus beträgt die Suchtiefe zwei, im Mittleren vier und im schwierigsten Modus fünf. Im schwierigen Modus wird die Suchtiefe zusätzlich mit der Anzahl der geschlagenen Figuren erhöht. Der Suchbaum wird zudem durch CutOffs verkleinert. Ist die erwartete Suchtiefe erreicht, so wird der beste Zug in der Variable bestMove gespeichert. BestMove ist hierbei der Zug mit dem höchstmöglichen Score.
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.
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, wird der Thread beendet und der berechnete Zug kann abgerufen werden.

Paket network



UML-Klassendiagramm network[12]

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.

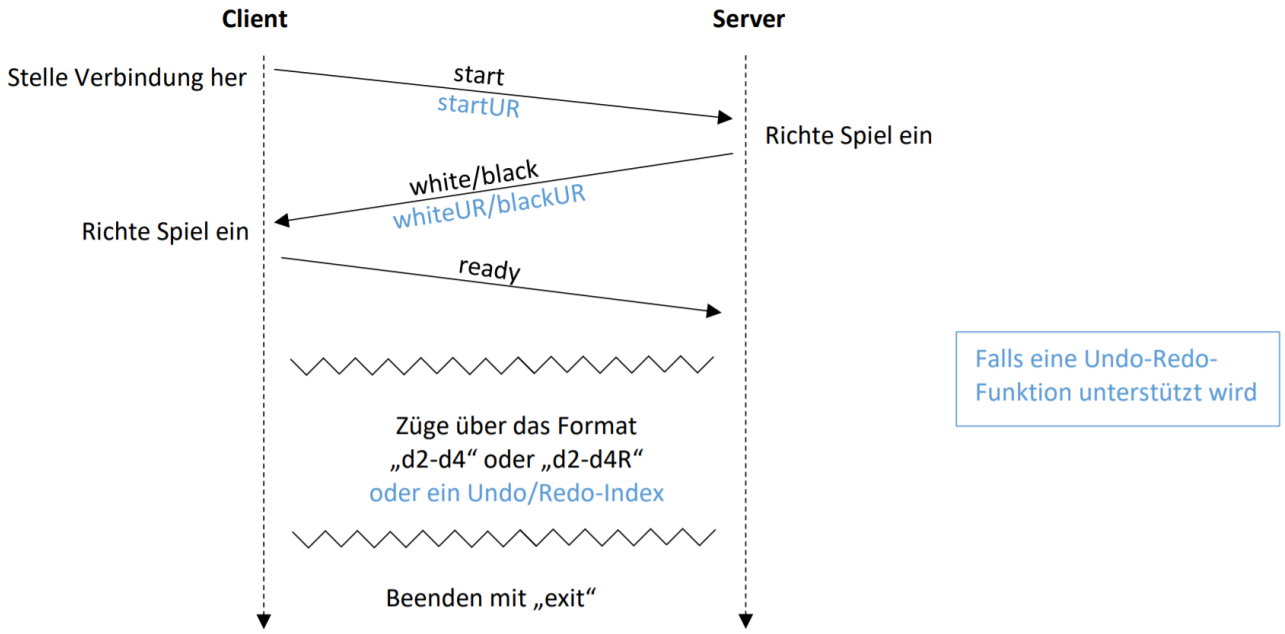
Da dieses Feature nur von der grafischen Oberfläche unterstützt wird, ist die Klasse auch die Einzige, die auf das Package network zugreift. Hierzu wird die Klasse **Logic** im Package gui genutzt, die in der *run()*-Methode den Spielmodus abfragt. Ist dieser im Netzwerk-Modus, wird das Netzwerkspiel 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.

Client und Server kommunizieren über einen vorher festgelegten freien Port, z.B. Port 5555. Der Spieler, der den Server stellt, muss diesen Port freigeben, damit der Client-Spieler darauf zugreifen kann. Der Server-Spieler muss dem Client-Spieler entsprechend (extern) diesen Port und seine IP-Adresse mitteilen.

[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

Zudem legt der Server-Spieler zu Anfang fest, mit welcher Farbe er spielt. Die Farbe des Client-Spielers wird dem Client als “black” oder “white” übermittelt. Haben beide nun das Spiel eingerichtet, beginnt Spieler Weiß mit dem ersten Zug.
Es wird davon ausgegangen, dass alle gesendeten Züge sowohl syntaktisch als auch semantisch korrekt sind. Dies wird im jeweiligen Controller 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, dass der Client die Undo-Redo-Funktion besitzt.
Das Versenden von “white/black” von dem Server an den Client vermittelt die Spielfarbe des Clients. Das Anhängen von “UR” bedeutet, dass der Server die Undo-Redo-Funktion unterstützt.
Anschließend sendet der Client “ready”, um mitzuteilen, dass er sein Spiel initialisiert hat und nun bereit ist, Züge auszutauschen.