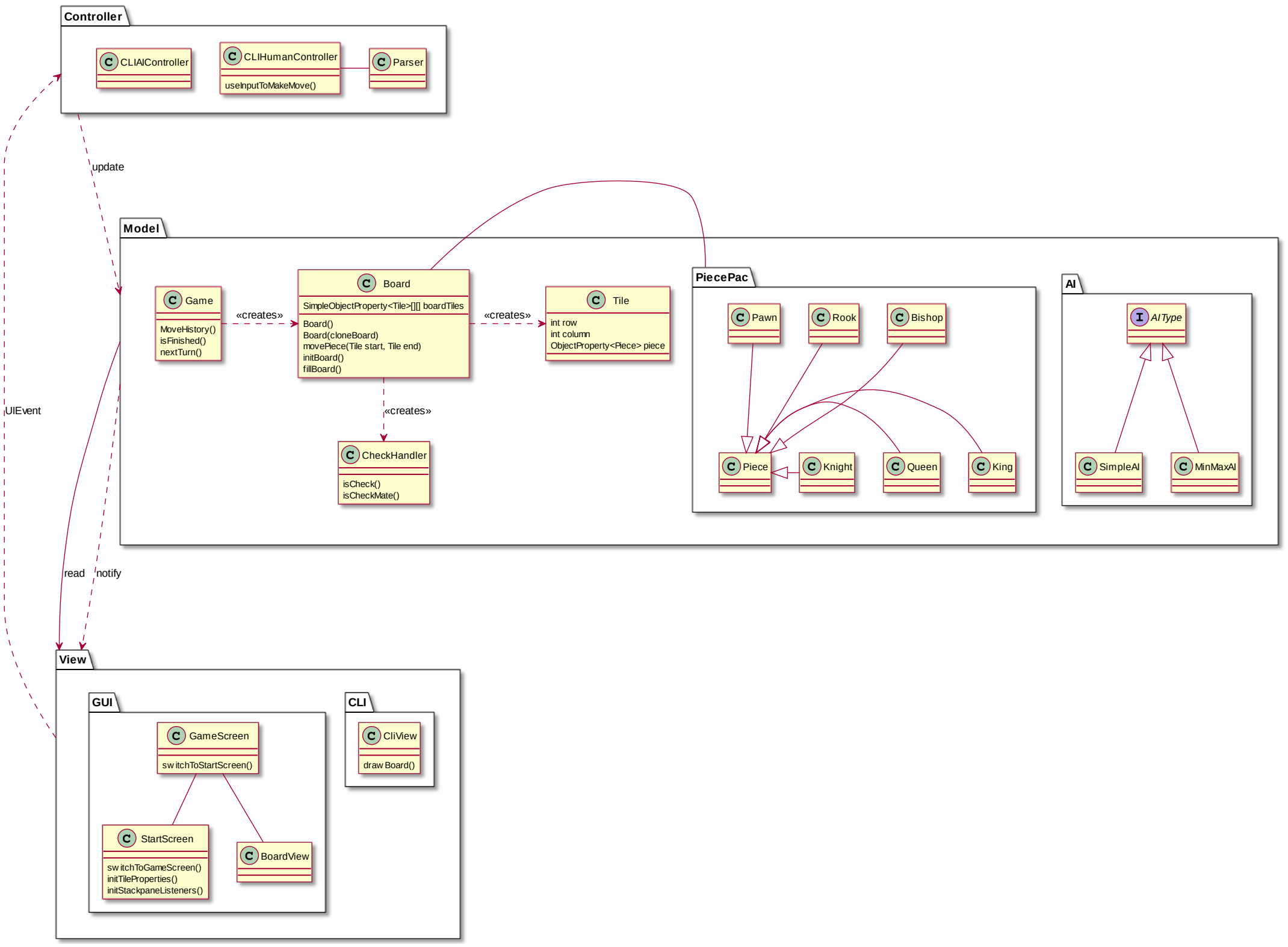
Architekturdokumentation SWENG2021

Arved Boetefür
Etnik Kastrati
Ilja Jamkis
Carl Fedrowitz

Model-View-Controller

Die Struktur des Programms basiert grundlegend auf dem Model-View-Controller Entwurfsmuster, welche aber durch die Verwendung von Properties-Funktionen für Objekte im Modell, der Verwendung von Binding auf die Tiles zwischen Model und GUI und der Strukturierung der GUI in der View durch FXML ein paar erwähnenswerte Besonderheiten hat.



Model

Das Model enthält die Datenstruktur des Spiels, also die typischen Objekte eines Schachspiels und die Funktionen, die den Objekten ihre Spiellogik geben.
Die zentralen Klassen und Komponenten des Models sind: Game, Board, Pieces, CheckHandler und die AI, welche im Folgenden kurz beschrieben werden.

Das Game ist die Datenstruktur, die den Rahmen für den spiellogischen Teil des Programms bildet. Hier wird das Board und die MoveHistory erzeugt. Das Game überprüft, ob das Spiel noch läuft und es wird der nächste Turn initialisiert.

Das Board ist das Herz der Spiellogik. Fast alle wichtigen Funktionen des Spiels gehen über dieses Objekt/Klasse. Beim Konstruktoraufruf wird ein 2D-Array erzeugt, welches mit Tiles gefüllt werden. Und dann werden die Spielfiguren auf die Tiles gesetzt. Jedes Tile hat eine Position, die nur das Tile kennt, das Piece selbst kennt seine Position aber nicht.
Die Funktionen, die das Board hat, haben mit dem Füllen, dem Entfernen und dem Bewegen von Pieces auf dem Schachfeld und dabei der Einhaltung der Regeln zu tun. Zu erwähnen ist noch, dass neben dem normalen Board Konstruktor auch ein Copy-Konstruktor existiert, der für die Abfrage von "Schach" und "Schachmatt" zum Einsatz kommt. Damit können Züge auf ihre Gültigkeit geprüft werden, ohne das hinterher gesetzte Flags zurückgesetzt werden müssen. Darauf wird aber im Kapitel zum Copy-Board Konstruktor noch einmal eingegangen.
Die wichtigste Funktion des Boards ist die movePiece-Methode. Sie bekommt zwei Tiles übergeben und bewegt dann die Figur, wenn es denn möglich ist, vom einen Tile zum anderen.

Die Pieces Klasse ist eine abstrakte Klasse. Unsere Schachfiguren erben von dieser. Wenn im Board eine Figur bewegt werden soll, wird ein Großtteil der Regelbedingungen für einen Zug über die Figuren abgefragt. Eine Besonderheit in dieser Komponente: Um Erweiterbarkeit der Pieces zu ermöglichen, werden die Figuren auf dem Board über konstante Versetzungen des bestimmten Pieces und die Lauflänge bewegt. Z.B kann ein Turm gerade laufen bis zur nächsten Figur bzw Brettgrenze. Die konstante Versetzung, die in der Klasse Constants als Arrays abgelegt sind, sieht für den Turm also wie folgt aus: {-1,0},{0,-1},{1,0},{0,1}, und als Lauflänge BoardLength. Wenn man also weitere Figuren dem Spiel hinzufügen möchte, reicht es sich solch eine Versetzungskonstante und wie weit sie gehen darf, zu überlegen. Spezielle Regeln zu den einzelnen Figuren (z.B. für den Bauern oder den König) sind in den von der abstrakten Piece Klasse erbenden Klassen zu finden.

Der CheckHandler wird für die Spielsituationen "Schach", "Schachmatt" und "Unentschieden" benötigt. Er wird in der movePiece-Methode des Boards erstellt und bekommt ein kopiertes Board übergeben. Das hilft den Code übersichtlicher und besser wartbar zu halten, da gesetzte Flags nicht zurückgesetzt werden müssen, sondern das kopierte Board nach den Abfragen verworfen wird.

Die KI wird beschrieben als AIType, was die übergeordnete Klasse der verschiedenen KI-Typen darstellt. Es gibt zwei verschiedene KI-Typen. Als erstes wäre da die normale KI, welche einen Zug nach einer bestimmten Punktebewertung ausführt. Und es gibt die verbesserte KI, welche mit Hilfe der MinMax-Klasse einen vorrausschauenden Zug mittels Alpha-Beta-Pruning berechnet, und diesen dann ausführt. Beide erben direkt von der AIType Klasse und besitzen alle Methoden und die gleichen Attribute.

View

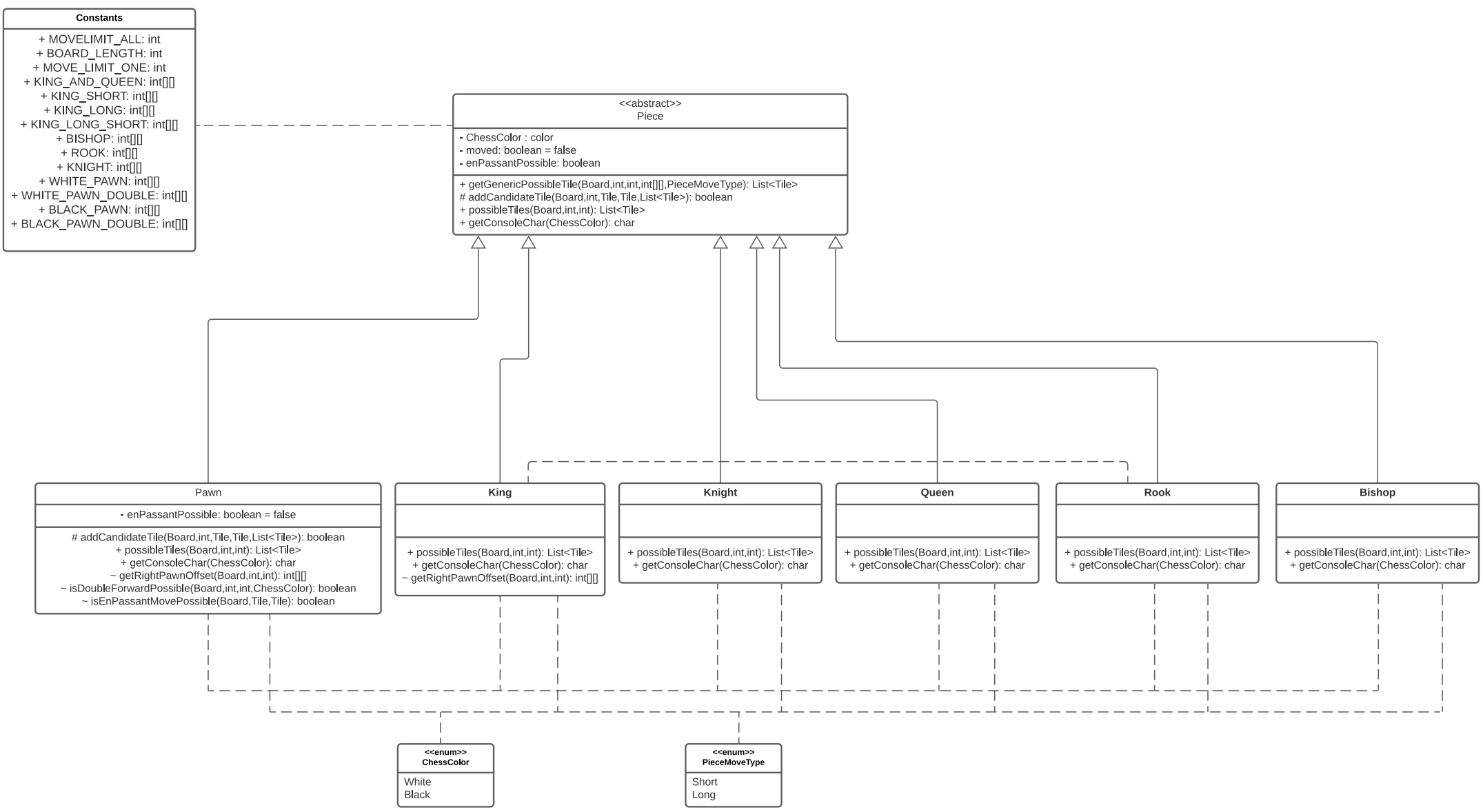
Die View stellt das Model des Programms für den Benutzer visuell dar. Sie liest die Daten des Models und stellt diese entsprechend dar. Wir haben zwei Views: Die CLI und die GUI, sodass das Spiel in zwei Ansichten spielbar ist.
Die CLI -View stellt das Model und damit den Spielzustand in der Konsole dar. Die Interaktion mit dem User läuft daher über Textein- und ausgaben. Die wichtigste Methode in der CLI-View ist die drawBoard-Methode. Nach jeder Zugeingabe wird in dieser Methode der Zustand des Models abgefragt und in der Konsole ausgegeben. In der CLI Main-Methode arbeitet die Methode drawBoard im Wechsel mit der doUserAction-Methode aus dem CLI Controller, welche die Eingaben des Users entgegennimmt und das Model anpasst.
Die GUI bietet dem User eine grafische Spieloberfläche. Die GUI-View besteht hauptsächlich aus den drei Klassen StartScreen, GameScreen und BoardView und zusätzlich der Klasse GUI, in der die start-Methode für JavaFX zu finden ist, quasi die main-Methode für die GUI-View.
Die Informationen aus dem Model werden über zwei Wege an die GUI-View übergeben.

Zum einen liest sie die Daten des Models, wie auch die CLI-View, zum anderen benachrichtigt das Model die GUI-View aber auch über Änderungen, sodass sich die GUI-View daraufhin aktualisieren kann. Dies funktioniert über den Einsatz von Properties und der Klasse Binding, mit der einige unserer Objekte im Model gewrapt bzw. verbunden sind. So können ChangeListener auf die Objekte angewendet werden, sodass bei einer Veränderung des Objekts die GUI-View benachrichtigt wird über diese Änderung.
Initialisiert werden diese ChangeListener in der switchToGameScreen-Methode der StartScreen Klasse.
Für den Bau und die Strukturierung der sichtbaren GUI Elemente wird FXML verwendet. Die Verwendung von FXML führt zu einer engen Kopplung von View und Controller, sodass wir keine dedizierte Controller Klasse für die GUI-View haben.

Controller

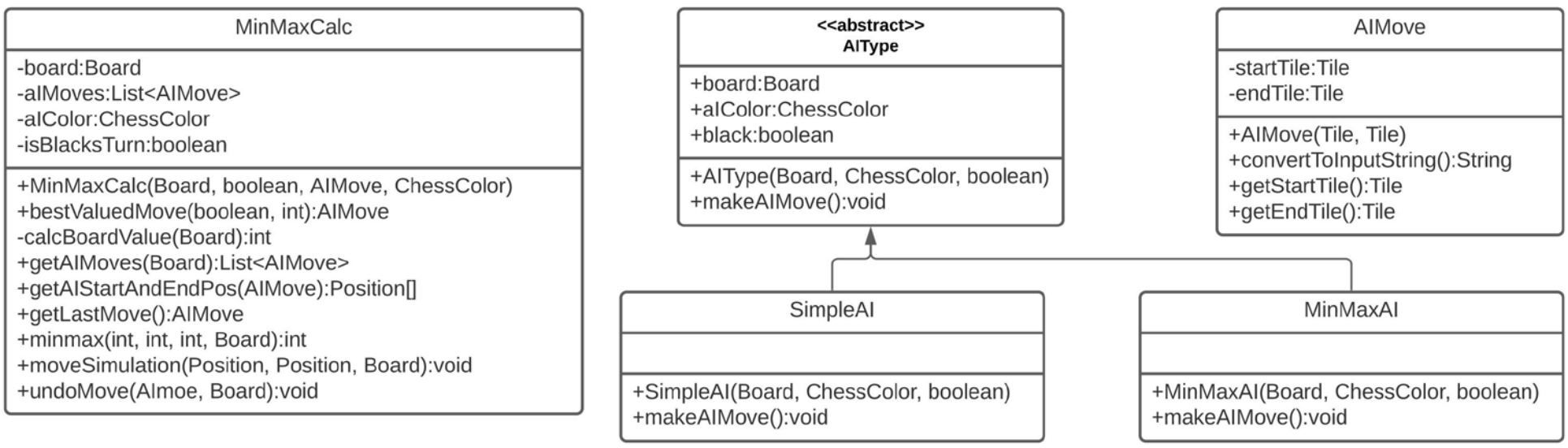
Der Controller verändert anhand der erhaltenen Eingaben das Model. Für die CLI existiert ein dedizierter Controller, der die Texteingaben des Users verarbeitet und daraufhin das Model verändert. Für die GUI gibt es solch einen eigenständigen Controller nicht, da aufgrund von Binding und Properties Controller Funktionalitäten eng zusammen mit View Funktionalitäten implementiert sind.
Der CLI-Controller besteht aus den Klassen CLIHumanController, CLIAIController und Parser
Das Entgegennehmen und Verarbeiten der User Eingaben erledigt die useInputToMakeMove-Methode in der Klasse CLIHumanController. Diese erhält die Eingabe als String, verarbeitet sie mit Hilfe der Klasse Parser zu zwei Tiles und ruft die movePiece-Methode im Board auf. Mehr hierzu im Kapitel zum Controller.

Die Pieces



Für die Implementierung der Figuren haben wir eine abstrakte Klasse „Pieces“ erstellt. Die konkreten Figuren (King, Queen, … usw.) erben von dieser Klasse. Wenn das Spiel die „Tiles“ braucht, wo die Figuren hinziehen dürfen, wird die geerbte Methode „possibleTiles“ aufgerufen. Für jede einzelne Figur ist dort festgelegt, welche Parameter sie braucht, um im Weiteren die benötigten „Tiles“ zurückzugeben. Dabei sind zwei Parameter nötig, um eine Liste der „Tiles“ zu erstellen. Erstens den „PieceMoveType“. Eine enum, die festlegt, ob es eine Figur ist, die über die ganze Länge des Boards geht (PieceMoveType.LONG) oder ob es eine Figur ist, die nur einen Schritt machen darf (PieceMoveType.SHORT). Der zweite Parameter ist eine Menge von „Integer-Arrays“ („Offsets“), die die Information enthalten, in welche Richtungen die Figur auf dem Board versetzt werden darf. Für die meisten Figuren existiert nur eines dieser „Offset“. Für Figuren, wie den „Pawn“, der je nachdem in welcher Situation er sich befindet, andere Züge machen kann, existieren zusätzlich für die Situation angepasste „Offsets“. Falls man neue Figurtypen einfügen wollte, würde das bedeuten, nur zwei Parameter gestalten zu müssen: 1. Wie weit die Figur gehen darf. & 2. Die Richtungen, in die die Figur versetzt wird. Falls mehrere „Offsets“ möglich sind, muss dann noch eine Abfrage eingebaut werden, die den richtigen „Offset“ auswählt, so wie das bei uns für den „Pawn“ und den „King“ geschieht. Die Implementierung einer Iteration über das Board in jeder Figurklasse bleibt erspart, denn man iteriert nur über die „Offsets“ und Länge des „PieceMoveType“ zentral in der Methode „getGenericPossibleTiles“ in der Klasse „Piece“. 4 von 6 Figuren müssen auf nichts als die generische Variante zugreifen, um die möglichen Züge zurückzugeben. So bleibt die Implementierung der Figuren schön schlank und übersichtlich und damit leichter wartbar und für Erweiterungen offen.

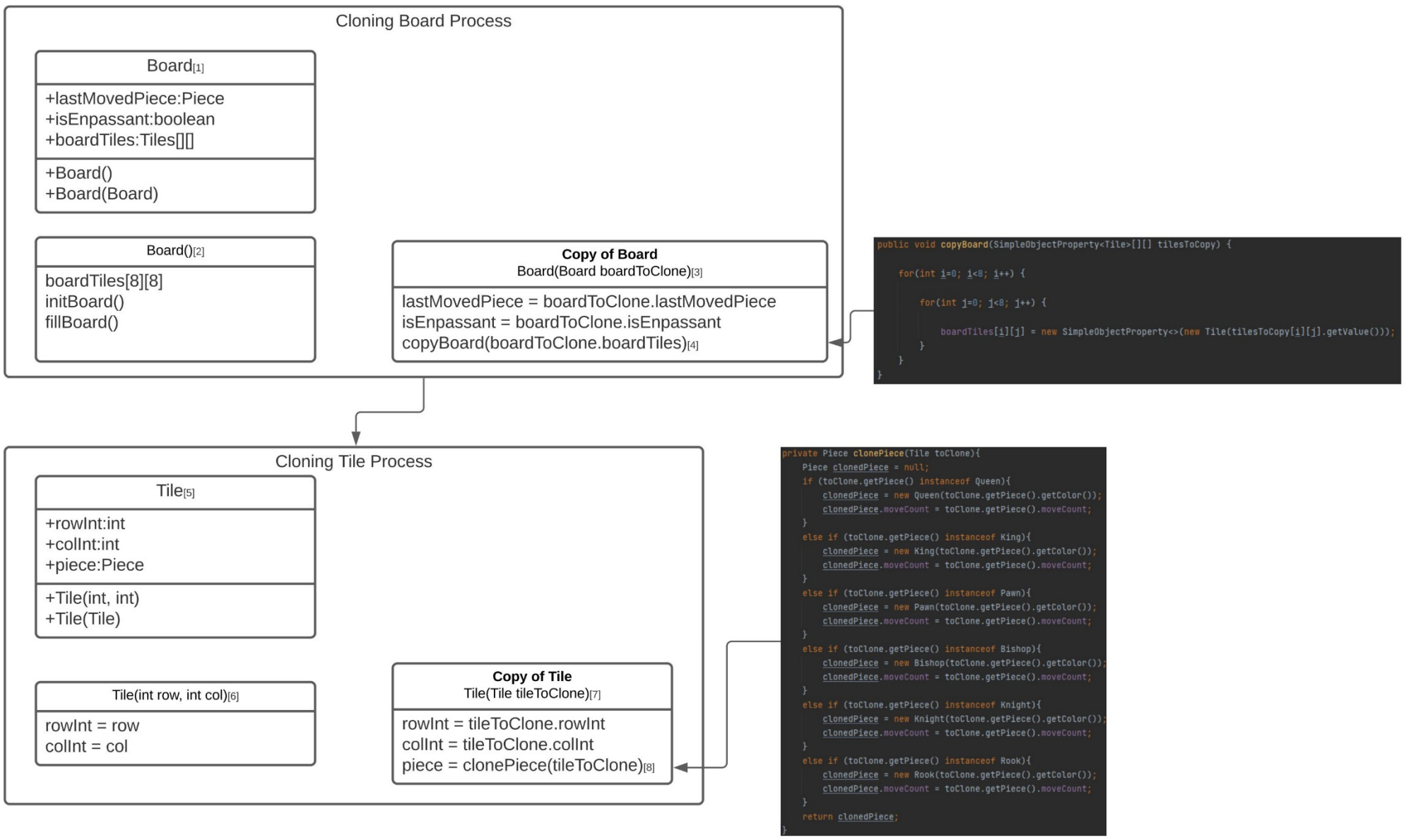
Architektur zur KI



In unserer KI-Architektur nutzen wir eine abstrakte Klasse AIType, welche eine neue Art der KI erschafft. Sie braucht dafür lediglich das aktuelle Board, die Farbe der KI und einen Boolean-Wert, der angibt welche Farbe gerade am Zug ist. Jede KI kann die abstrakte Methode makeAIMove() ausführen. Der erste KI-Typ SimpleAI (einfache KI), soll einfach einen gierigen Zug machen, d.h. bei Möglichkeit eine gegnerische Figur zu schlagen, und zwar immer die Figur mit dem höchsten Punkten. Dazu verwendet sie die MinMaxCalc Klasse. Der andere KI-Typ ist die MinMaxAI (Verbesserte KI). Sie funktioniert genau wie die einfache KI, der einzige unterschied besteht jedoch darin, dass Sie die besten möglichen Reaktionen auf ihren eigenen Zug mit einkalkuliert. Sie geht allerdings bis zu vier Ebenen tiefer und rechnet auf den besten Möglichkeiten des Gegners einen Move aus, und wieder dessen besten Möglichkeiten. Auch sie verwendet für diese Kalkulation die MinMaxCalc Klasse. Kommt die KI zu dem Schluss, dass sie den besten AIMove gefunden hat, führt sie diesen aus. Die schon genannte MinMaxCalc Klasse rechnet immer den besten AIMove aus. Sie verwendet dafür die übergebenen Klassen-Parameter: das Board, den Boolean-Wert über die Farbe welche gerade am

Zug ist, den letzten AIMove und natürlich die Farbe der KI. Das Board ist ein Klon des Aktuellen Boards auf welchen wir Moves simulieren wollen. Der Boolean-Wert wird genutzt um zwischen den Ebenen der Move-Kalkulation zu wechseln. Der letzte AIMove wird immer betrachtet, wenn die nächste ebene erreicht wird, um auf diesen zu reagieren. Bei der Initialisierung wird immer der Wert des Boards berechnet, indem durch das Board iteriert wird und alle Wert-Punkte der Pieces aufsummiert wird. Zudem wird eine Liste erstellt mit allen möglichen AIMoves, welche von der KI ausgeführt werden könnte. Die Kernmethode der Klasse ist die bestValuedMove(boolean, int) Methode. Sie bekommt einen Boolean-Wert übergeben, welcher wieder ansagt, wer gerade in der Kalkulation am Zug ist und einen Integer- Wert, welcher die Tiefe der Kalkulation repräsentiert. In der einfachen KI beträgt dieser 0, und bei der verbesserten KI kann man wählen zwischen 0 bis 4. Die AIMove Klasse definiert nur einen AIMove über ein startTile und endTile. Zudem beinhaltet sie die Methode convertToInputString() mit der ein spezifischer Move als String nach dem Muster !a2-a3 generiert wird.

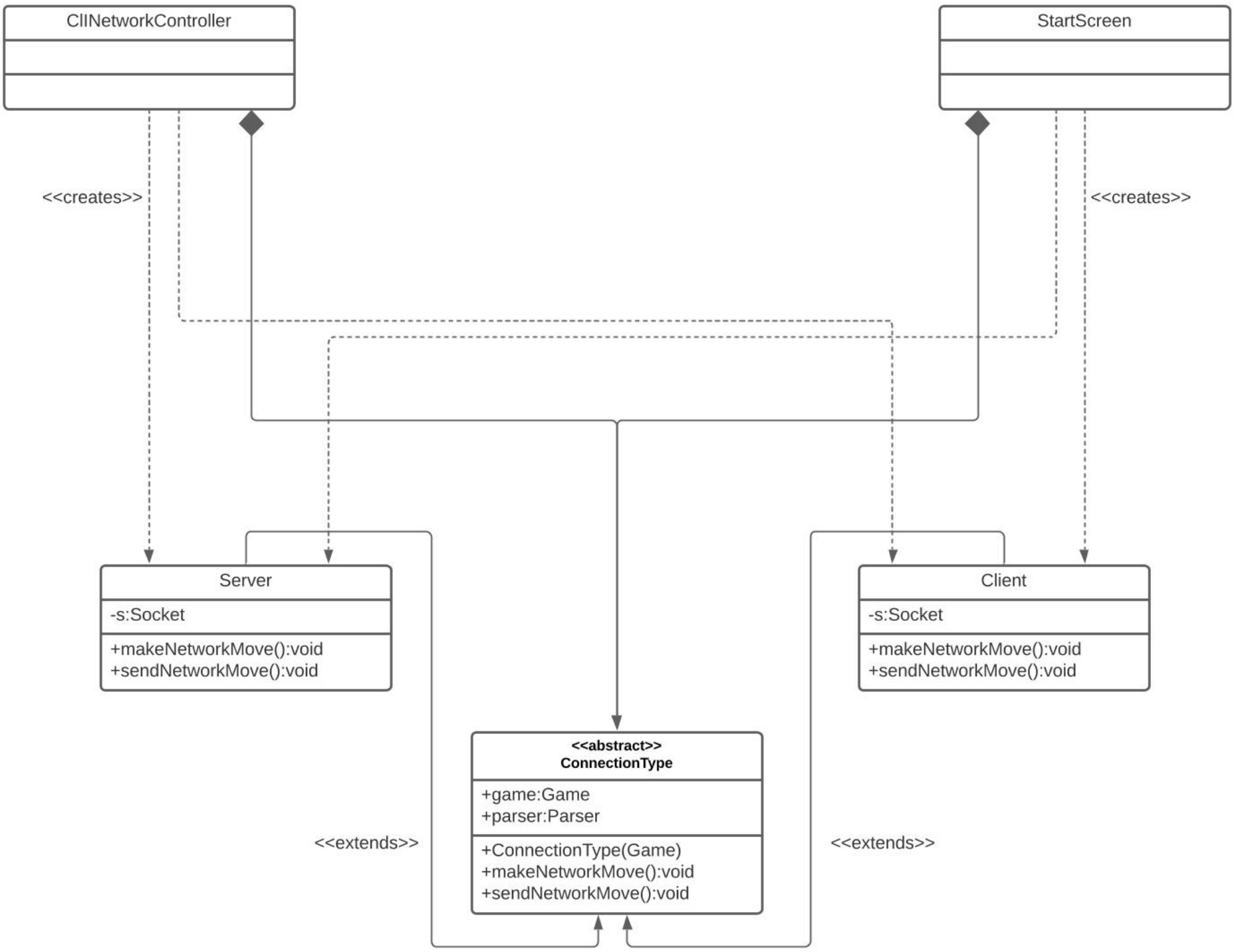
Copy Board Konstruktor



In unserer Architektur verwenden wir häufiger ein kopiertes Board. Immer dann, wenn wir einen oder mehrere Veränderungen auf dem Board simulieren wollen. In der Board Klasse [1] gibt es zwei Konstruktoren, der Normale [2] und der, der das Board kopiert [3]. Im Kopier-Konstruktor werden alle relevanten Flags und die Kopie des originalen 8x8 Arrays der boardTiles kopiert. Die boardTiles werden mittels einer copyBoard-Methode [4] neu erschaffen, indem man auch den Kopier-Konstruktor der Tile Klasse [5] benutzt. Dieser [7] funktioniert auf ähnliche Art wie der aus dem Board. Er kopiert das betrachtete Tile [6] mit all den Werten und mit der Methode clonePiece()[8] ,schließlich auch das Piece auf dem betrachteten Tile. Dieser Aufwand scheint erstmal sehr groß zu sein, jedoch lohnt er sich sehr für unsere Architektur. Abgesehen von den oben genannten Gründen, arbeiten wir auch mit Properties und deren Listener in der GUI. Diese deep copy muss sein, da sonst die Pieces immer auf zwei Tiles referenzieren. Dem ursprünglichen und dem Geklonten. Dies würde zu Fehlern

bei den Tile-Properties führen. Außerdem ist es sehr angenehm, wenn man nach jeder Move-Simulation das geklonte Board verwerfen kann, ohne irgendwelche Veränderung zu beachten.

Netzwerkspiel

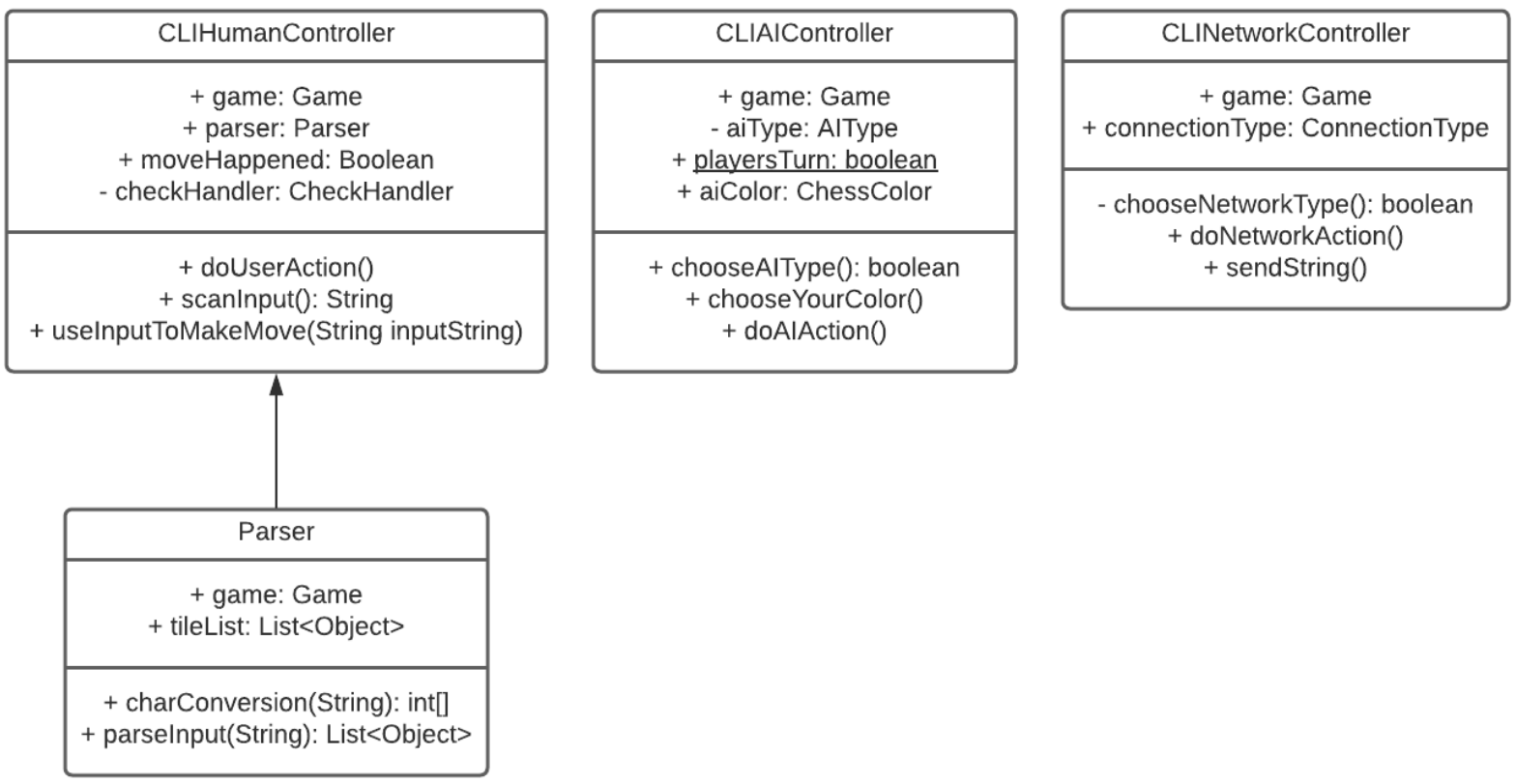


Für unser Netzwerkspiel verwenden wir zwei ConnectionTypes: Den Server und den Client. Beide werden dem übergeordnetem Typ connectiontype zugeordnet (siehe NetworkDocUMLpdf), und erhalten alle Informationen über das derzeitige Spiel (Game). Bei der Initialisierung des Servers erstellt er einen ServerSocket, mit dem Port 4999 (so wie auch unsere Partnergruppe). Dieser wartet dann auf Zugriffe auf den Socket. Der Client wird nach der lokalen IP des Servers gefragt und verbindet sich dann bei korrekter IP über den Port mit dem Server. Die ConnectionType verfügt über zwei wichtige Methoden: makeNetworkMove(), sendNetworkMove(). In der makeNetworkMove() Methode wird als erstes der DataInputStream gelesen(welcher den Zug des NetzwerkPartners enthält) und zu einem String verarbeitet. Dieser wird mit Hilfe der Parser Klasse geparsed und ein Zug ausgeführt. Des Weiteren wird dieser Zug ausgegeben. Die sendNetworkMove() Methode sendet über einen DataOutputStream den zuletzt legal ausgeführten Zug dem Netzwerkpartner als String. Diese Routine wird in der CLI von dem CLINetworkController ausgeführt, und in der GUI vom GameScreen.

SaveLoadManager

Eines der gewählten Zusatzfeature ist das Speichern und Laden eines Spiels.
Als Speicherformat dient .txt Files. Wir haben diese Format gewählt, da es für den Umfang der zu speichernden Informationen eine leicht umsetzbare Variante ist. Außerdem bietet es eine gute Erweiterbarkeit, da das Hinzufügen von mehr oder anderen Informationen in das File sehr schnell und einfach geht.
Die beiden zentralen Methoden sind saveGame und loadGame.
saveGame bekommt beim Methodenaufruf einen absoluten Pfad zu dem zuspeichernden File als Parameter übergeben.
Beim Speichern werden zunächst die Pieces mit ihren Koordinaten und moveCounts in das Textfile geschrieben. Für jedes Piece eine eigene Zeile. Danach wird eine Zeile mit verschiedenen Flags/Infomationen geschrieben, mit denen man später den Zustand des Spiels wiederherstellen kann.
Dann wir eine Zeile mit den Figuren, die bereits geschlagen wurden, geschrieben.
Zuletzt wird noch eine Zeile mit der Liste der Moves aus der Klasse MoveHistory abgespeichert.
Beim Laden werden dann die vier unterschiedlichen "Typen" von Zeilen unterschieden und ausgelesen. Die loadGame-Methode gibt dann eine neu erstellte Instanz vom Game an die aufrufende Methode zurück.
Sowohl in der CLI als auch in der GUI ist das Feature anwendbar. In der CLI muss der User stehts einen Pfad zum Speicherort bzw Ladeort eingeben. Bei der GUI wird ein FileChooser Objekt genutzt, mit dem man bequem über den Fileexplorer Speicher- und Ladeort wählen kann.

Controller



Dies ist das Controller-Package des Spiels. Es beinhaltet die Controller für das PVP-Spiel, das Spiel gegen die KI und für das Netzwerkspiel.

Alle drei Controller greifen über das Game auf das Model zu.

CLIHumanController:
Hier gibt es drei wichtige Methoden;

- *scanInput()*, welches mittels einer Scanner-Methode eine Eingabe liest
- *doUserAction()*, welche diese Eingabe in einen String genannt “input” steckt und anschließend…

- ...*useInputToMakeMove(inputString)* mit “input” aufruft. Hier wird dann zunächst der Parser aufgerufen und anschließend der Zug ausgeführt, bzw. andere Eingaben wie “beaten” etc. abgefangen. Nach weiteren Abfragen, z.B. ob ein Zug legal ist, wird dort auf das Model zugegriffen und ein Zug ausgeführt.

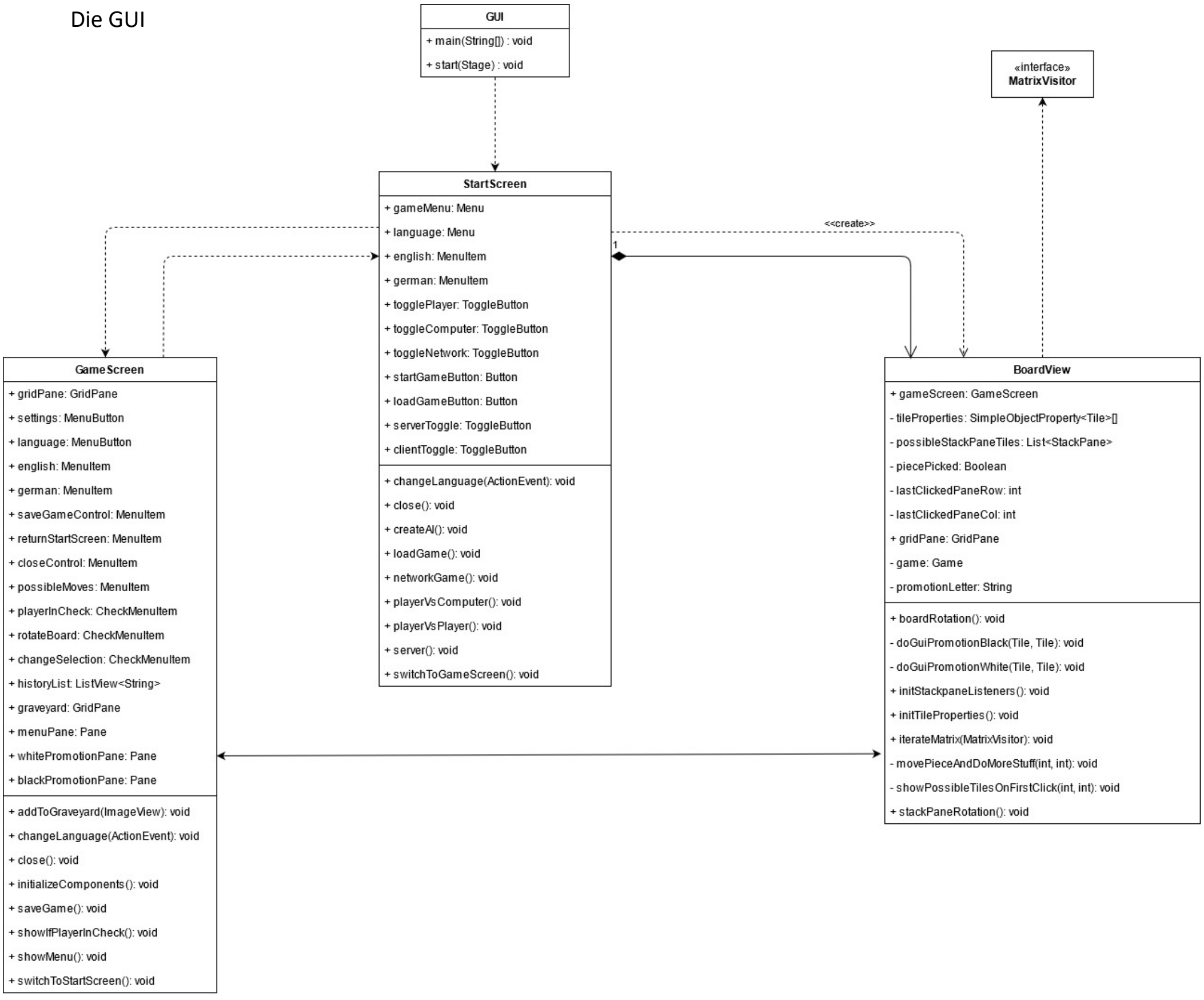
Parser:
Der Parser besitzt die Methode *parseInput(String)*, welche den Input gegen das Eingabepattern matched. Die Eingabe wird in eine Start- und eine Endposition aufgeteilt. Anschließend wird c*harConversion(String)*, mit diesen Positionen als Eingabeparameter aufgerufen. Hier werden die Positionen als Strings zu Positionen als Integer für das Board konvertiert und als positionIndex zurückgegeben.
Start- und Endposition werden zusammen mit einem (potentiellen) promotion letter (für die Pawn-Promotion) als ObjectList an *useInputToMakeMove(String inputString)* übergeben.

CLIAIController:
Der AI-Controller beinhaltet Abfragen für das Spiel gegen die KI sowie eine Methode um KI-Züge auszuführen.

- *chooseAIType()* ist die Abfrage für den Schwierigkeitsgrad der KI mittels Scanner-Methode. Sie gibt einen Boolean zurück, welcher bei der einfachen KI true und bei der MinMax-KI false zurückgibt.
- *chooseYourColor()*, welche mittels Scanner-Methode die chess colors für KI und den Spieler setted
- *doAIAction()*, welche die KI Züge durchführt

CLINetworkController:
Der Network-Controller beinhaltet Abfragen, welche einen Scanner-Input verwenden um festzulegen, ob ein Spieler als Client oder als Server des Spiels agieren möchte und um eine zu verbindende IP-Adresse zu erhalten. Züge werden durch *doNetworkAction()* durchgeführt und mit *sendNetworkString()* an das andere Spiel versendet.

Die GUI



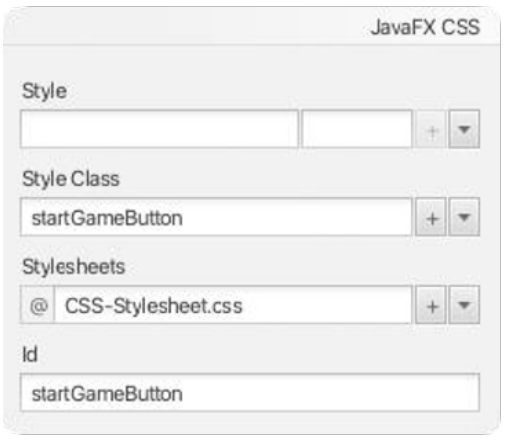
Bereits zu Beginn des Software-Engineering Praktikums haben wir uns Gedanken zu der GUI des Schachspieles gemacht und dementsprechend auch viel Arbeit in diese getan. Dabei sind auch Einflüsse aus vorhandenen Schachspielen im Internet in unsere Entscheidung eingeflossen um ein optimale User Experience zu schaffen.

Für die Umsetzung der GUI haben wir uns entschieden den SceneBuilder zu nutzen, um die FXML-Dateien zu erstellen und dort die Elemente zu bearbeiten. Wir haben uns auch den Vorteil des SceneBuilders zu Nutze gemacht, um ID´s und Methoden-Aufrufe dort direkt zu definieren. Somit sind wir zu der Struktur, wie sie oben im Klassendiagramm gezeigt wird gekommen. Der Startpunkt der GUI, ist die GUI-Klasse. Dort wird über die Main-Klasse die run()-Methode ausgeführt und die Inhalte des StartScreen in die Stage geladen. Ab diesem Punkt interagieren die StartScreen und GameScreen Klasse, wie eine andauernde Schleife.
Die Klasse BoardView enthält die Objekte und Methoden, die zu der Funktionalität des Schachbretts gehören. Zwischen BoardView und GameScreen gehen dabei Abhängigkeiten in beide Richtungen. Die Trennung in zwei Klassen schafft jedoch eine besser Übersichtlichkeit. Näheres zu den Funktionen der BoardView Klasse finden sich im Kapitel “Schnittstelle zwischen Model und View”.
Uns war wichtig, dass wir einen externen Startpunkt haben, also die GUI-Klasse, um die GUI zu initialisieren. Die Klassen StartScreen und GameScreen dienen dabei als Controller der FXML-Dateien um Interaktionen und Handlungen der jeweiligen Screens handzuhaben. Durch das Definieren der

ID´s und die onAction-Funktion die der SceneBuilder uns angeboten hat, haben wir dann die Funktionen der Inhalte und ihre Verbindungen zur Logik in diesen Klassen durchgeführt.

```java
@FXML
public Button startGameButton

@FXML
public void switchToGameScreen(ActionEvent event) throws IOException{
   FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("/gameScreen.fxml"));
   root = fxmlLoader.load();
   scene = new Scene(root);

   stage.setWidth(1150);
   stage.setHeight(800);

   stage.setScene(scene);
   stage.show();

   gameScreen = fxmlLoader.getController();
   if (game.isAIGame() && game.getAIColor() == null){
      gameScreen.setAiType(new SimpleAI(game.getBoard(), ChessColor.Black, true));
   }
   else {
      gameScreen.setAiType(aiType);
   }
   gameScreen.setGame(game);
   gameScreen.initTileProperties();
   gameScreen.setImages();
   gameScreen.initStackpaneListeners();
   gameScreen.setStage(stage);
}
```

Wie bereits erwähnt haben wir uns sehr früh Gedanken zur GUI gemacht und waren gewillt diese so zu gestalten, dass wir dem Nutzer eine moderne, angenehme und spielerische Interaktion mit unserem Spiel anbieten können. Um dies umzusetzen haben wir viel Arbeit in das Design investiert. Die Umsetzung haben wir mittels CSS-Stylesheets vorgenommen, welche wir auch im SceneBuilder definiert haben. Dabei haben wir auch aufwendig, viele Use-Cases miteinbezogen, wie beispielsweise das hovern und klicken der Buttons in Zusammenhang mit der grafischen Änderung dieser.




```css
#startGameButton{
    -fx-background-color: #7fa650;
    -fx-text-fill: #fff;
}
#startGameButton:hover{
    -fx-background-color: #95bb49;
}
#startGameButton:pressed{
    -fx-background-color: #82A542;
}
```

Schnittstelle zwischen GUI (View & Controller) und Model

Wenn man ein Spiel mittels der GUI spielen will, werden, bevor der GameScreen dargestellt wird, verschiedene Listener auf Objekten des Models und der GUI initialisiert. Diese können auf Objekten aus dem Model angewendet werden, weil diese Objekte als Properties gewrapt wurden.

Initialisierung der StackPanes

Die 64 StackPanes des GridPanes vom Schachfeld werden für Mausklick Events initialisiert. Sobald eines der StackPanes geklickt wurde, läuft eine Folge an Anweisungen ab:

Zuerst wird überprüft, um was für eine Art Klick es sich handelt. Wenn noch kein StackPane geklickt wurde, heißt das, dass noch keine Figur (z.B. für eine Figurbewegung auf dem Schachbrett)

ausgewählt wurde. Deshalb wird hierfür überprüft, ob das geklickte StackPane eine Figur enthält und es die richtige Farbe hat. Wenn die Bedingungen erfüllt sind, werden die StackPanes, die für die bestimmte Figur mögliche Felder für einen Zug sind, farbig markiert durch Aufruf der „showPossibleTiles“ Methode.

Anderseits, wenn bereits ein StackPane geklickt wurde, wird für den zweiten Klick abgefragt, welches StackPane geklickt wurde. Ist es ein StackPane, das auch eines war, dass in der „showPossibleTiles“ Methode farbig markiert wurde, dann wird im Model die Figur bewegt.

Initialisierung der ChangeListener für das 2D TileArray

Wenn eine Figur bewegt wurde, also in dem Model auf einem Tile die Figur verändert wird, werden ChangeListener aktiviert. Auch diese werden beim Starten eines GUI-Spiels für alle 64 Tiles initialisiert. Die GUI View wird also darüber informiert, ob eine Veränderung stattgefunden hat und wo diese stattgefunden hat. Wird der Listener getriggert, werden Lösch- und Zeichenmethoden für die Images auf den entsprechenden StackPanes ausgeführt. Vom ChangeListener erhalten wir sogar die Information, welche Figur vor der Änderung und welche nach der Änderung auf dem Tile stehen, sodass wir diese Information nicht mehr aus dem Model rauslesen müssen, sondern direkt mit der Benachrichtigung erhalten

Initialisierung des ChangeListeners für die beatenPieces in der GUI

Da die Liste mit den „beaten Pieces“ im Model als ListProperty deklariert ist, wird durch eine Veränderung der Liste der ChangeListener in der View getriggert und es wird ein Image auf den Figurenfriedhof gezeichnet

Initialisierung des ChangeListeneres für die Liste der MoveHistory

Wie schon die Liste der geschlagenen Figuren, wird auch die Liste mit den gemachten Zügen in der GUI automatisch aktualisiert, wenn sie im Model verändert wird, da auch diese Liste als ListProperty deklariert ist.

Stellungnahme zu PMD.SurpressWarnings

<u>TooManyFields</u>
Klasse Constants, Zeile 8: Die Constants Klasse ist eine reine Datenklasse und besteht nur aus Feldern. Dies erspart uns sehr viel Implementierungsarbeit und Zeilen an Code, vor allem in der Bewegungslogik der Pieces.
Klasse GameScreen, Zeile 36: Die GameScreen Klasse enthält eine solch große Anzahl an Feldern, da wir sehr viele FXML Felder nutzen müssen um die verschiedenen Knöpfe anzusprechen.
StartScreen Zeile 37: Die StartScreen Klasse enthält eine solch große Anzahl an Feldern, da wir sehr viele FXML Felder nutzen müssen um die verschiedenen Knöpfe anzusprechen.
<u>ExcessiveParameterList</u>
Klasse Pawn, Zeile 46, addCandidateTile(): In dieser Methode werden alle 5 Übergabeparameter gebraucht um das korrekte CandidateTile zu berechnen. Damit werden dann die möglichen Positionen des Bauern berechnet. Da es nur ein Übergabeparameter zu viel ist und es ein zu großer Eingriff in die Logik wäre diese Methode aufzusplitten, haben wir uns dazu entschieden, sie so zu lassen.
Klasse Piece, Zeile 104, addCandidateTile(): Da dies eine abstrakte Methode ist muss sie auch dieselben 5 Übergabeparameter haben, wie die Methode in Pawn.
<u>CyclomaticComplexity</u>
Klasse Piece, Zeile 50, getGenericPossibleTiles(): Diese Methode wird für jedes Piece genutzt. Dadurch müssen wir nicht extra solch eine Methode in den erbenden Pieces implementieren. Außerdem ist die Komplexität dieser Methode nur 1 zu hoch. Deshalb haben wir uns dazu entschieden diese Methode so zu lassen.