# Entwicklung einer allgemeinen Schnittstelle zwischen Ludii und dem GBG Framework


| | |
|---|---|
| vorgelegt von: | Ann Weitz |
| Matrikel-Nr.: | 011 107 450 |
| Adresse: | Am Sandberg 28 |
| | 51643 Gummersbach |
| | annweitz@posteo.net |

eingereicht bei: Prof. Dr. Wolfgang Konen


Gummersbach, 03.02.2022

# Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

| | |
|---|---|
| Gummersbach, 03.02.2022 |  |
| Ort, Datum | Rechtsverbindliche Unterschrift |

# Inhalt

# Abbildungsverzeichnis

# 1 Einleitung

Diese Arbeit wird sich mit der Erstellung einer allgemeinen Schnittstelle zwischen dem General Board Game Playing Framework und Ludii befassen. Sie wird die Konzeption und die anschließende Umsetzung erklären und welche Hürden dabei aufgetreten sind. Ebenso ist es ein Ziel die Funktionsweise von Ludii und der Schnittstelle so zu erklären, dass die Umsetzung weiterer Spiele danach problemlos möglich ist.

## 1.1 Ludii

Ludii ist ein General Game System, dass im Rahmen des Digital Ludeme Projektes entwickelt wurde. Das Projekt hat vom Europäischen Forschungsrat ein Forschungsstipendium über 5 Jahre erhalten und wird seit 2018 an der Universität Maastricht durchgeführt [1]. Ziel des Projektes ist es, die historische Entwicklung traditioneller Spiele im Rahmen der Zeit darzustellen und unser Verständnis für sie zu verbessern.

Ludii im speziellen richtet sich unter anderem an Forscher im Bereich der künstlichen Intelligenz und Spieleentwickler. Es ist ein in Java entwickeltes Programm und basiert auf der Idee von Ludemen. Diese beschreiben Spiele in präzisen, einfach zu verstehenden und auch für Menschen leicht lesbaren Konzepten.

Innerhalb des Systems stehen mittlerweile über 1000 verschiedene Spiele zur Verfügung und diese Zahl wächst stetig weiter. So kamen im letzten Update auf die Version 1.3.1 56 neue Spiele hinzu [2].

## 1.2 Übersicht über die verschiedenen Ludii Klassen

Ludii ist auf einer Vielzahl von verschiedenen Klassen aufgebaut, die im Folgenden erläutert werden. Dies soll zum Verständnis der in der Schnittstelle genutzten Klassen beitragen und den Umgang mit Ludii erleichtern.

### 1.2.1 Game

Ein Game Objekt enthält alle Informationen, die über ein Spiel benötigt werden. Dazu gehören unter anderem der Name des Spiels, die Regeln, wie das Spielbrett aufgebaut ist und Optionen wie Spieleranzahl und Spielbrettgröße. Ebenfalls enthalten sind Beschreibungen für menschliche Spieler, sowie welche AI Klasse als beste für dieses Spiel angesehen wird. Das Game Objekt wird einmalig zur Laufzeit aus der dem Spiel zugehörigen .lud-Datei erzeugt.

### 1.2.2 Trial

Eine Trial ist eine Instanz eines Spiels, die zu einem bestimmten Zeitpunkt von Spielern gespielt wird. Sie enthält eine Liste der bereits gespielten Spielzüge, sowie Informationen zum Abschneiden der Spieler, wenn bereits bekannt.

### 1.2.3 State

Der State enthält Informationen über den momentanen Zustand des Spiels. Beispiel hierfür sind die Anzahl bereits gemachter Züge, welcher Spieler gerade oder als nächstes am Zug ist und auch die Reihenfolge in der gezogen wird.

### 1.2.4 Context

Ein Context-Objekt beschreibt den Kontext für eine Trial. Es enthält Verweise auf diese, sowie auf das übergeordnete Game-Objekt und den momentanen State. Dies macht es zu der wahrscheinlich nützlichsten Variablen, die an Methoden überreicht werden kann.

### 1.2.5 Model

Das Model existiert als eine Kontrollinstanz für eine Trial. Mithilfe dessen können neue menschliche und computergesteuerte Spielzüge angewendet werden.

### 1.2.6 Action

Actions beeinflussen den Spielzustand eines momentanen Spiels. Eine Action ist dabei eine kleinteilige Einheit, die auch immer nur eine Begebenheit ändert. Dabei kann es sich zum Beispiel darum handeln einen neuen Spielstein auf das Spielbrett zu setzen oder den Zug zu überspringen.

### 1.2.7 Move

Ein Move entspricht einem Spielzug, der von einem Spieler im Spiel gemacht werden kann. Dieser kann, falls erforderlich, aus mehreren Actions bestehen und je nach Spiel auch noch Verweise auf Regeln enthalten, die dann weitere Actions nach sich ziehen.

# 2 Implementierung

## 2.1 Planung

Der zugrunde liegende Plan für die Schnittstelle zwischen Ludii und dem GBG-Framework orientierte sich an der bereits vorhandenen Schnittstelle für Othello, die im Jahr 2020 von Johannes Scheiermann innerhalb eines Praxisprojektes umgesetzt wurde.

In dieser wurde eine Pseudo-Ludii-AI geschaffen, die dann von Ludii verwendet werden kann. Dort wird der Ludii-Context in einen spielspezifischen StateObserver übersetzt, welcher dann um die nächste Aktion gefragt werden kann. Diese wird dann mit der Liste an legalen Spielzügen, die der Ludii Context liefert, abgeglichen, um den passenden zu finden. Dieser wird dann von der Pseudo-Ludii-AI verwendet.

Die Funktionalität für die neue Schnittstelle baut auf dieser Grundlage auf, und liefert einige Klassen und Methoden die implementiert und erweitert werden können um die Schnittstelle einfach und ohne großen Aufwand um neue Spiele zu ergänzen.

Zwischen Ludii und dem GBG-Framework gibt es zwei Unterschiede bei der Spielimplementierung, die beachtet werden müssen:

1. Im Gegensatz zu GBG fängt in Ludii die Spielernummerierung immer bei 1 anstatt bei 0 an.
2. Die Nummerierung der Spielfelder wird sich wahrscheinlich ebenfalls unterscheiden.

Aufgrund dessen muss immer darauf geachtet werden, dass die Werte zwischen den beiden verschiedenen Frameworks nicht simpel übernommen, sondern in das jeweils andere übersetzt werden.

## 2.2 Umsetzung

Die Grundlage für die neue Schnittstelle bildet die Klasse *GBGAsLudiiAI*. Diese erbt von der, von Ludii bereitgestellten, AI Klasse und wird aufgrund dessen als gültiger Agent im Ludii Framework erkannt. Innerhalb der Klasse ist die Methode *initAI* dafür zuständig den jeweiligen GBG Agenten zu laden. Dies passiert entweder über ein Auswahlformular oder über einen festen Pfad, wenn man öfters mit demselben Agenten spielen möchte und etwas Zeit sparen will.

Die Methode *selectAction* sorgt dafür, dass die Anfrage nach einem Spielzug an das richtige Spiel gerichtet wird. Dies passiert über eine switch-case-Anweisung, die nach dem Namen des Spiels geht. Ist der Name des Spiels unbekannt, wird aus der Liste an Spielzügen ein zufälliger ausgewählt und ausgeführt.

Um mehr als einen zufälligen Zug zu machen, ist eine spielspezifische *selectAction*-Methode nötig, welche die anfallende Arbeit erledigt. Dort wird als erstes der Ludii-Context

in einen GBG-StateObserver übersetzt. Dieser wird dann um den nächsten Spielzug gefragt, welcher dann wieder zurück in einen Ludii-Spielzug umgewandelt wird. Je nach Spiel müssen hier auch noch weitere Dinge erledigt werden. Im Spiel Othello beispielsweise, muss dafür gesorgt werden, dass der Agent passt, wenn keine Spielzüge zur Verfügung stehen.

```
private Optional<Move> selectActionYavalath(Game game, Context context, double maxSeconds, int maxIterat
    Optional<Move> returnMove = Optional.empty();

    SystemConversionYavalath index = new SystemConversionYavalath();
    Types.ACTIONS gbgAction = gbgAgent.getNextAction2(new StateObserverYavalathTranslationLayer(context

    FastArrayList<Move> moves = game.moves(context).moves();
    for(Move move : moves){
        if(move.to() == index.getLudiiIndexFromGBG(gbgAction.toInt())) returnMove = Optional.of(move);
    }

    return returnMove;
```

Abbildung 1: Codeauszug aus der selectActionYavalath-Methode

Um die bestehenden Daten eines Ludii-Context in einen GBG-StateObserver zu übersetzen, braucht jeder spielspezifische StateObserver eine Erweiterung. Dies geschieht mit den StateObserverTranslationLayer Klassen. In diesen muss ein Spielzustand aufgebaut werden, der dem Zustand des Context-Objekts gleicht. Der einfachste Weg ist es, die Liste an bisherigen Spielzügen zu durchlaufen und die Züge in GBG zu wiederholen. Es muss ebenfalls darauf geachtet werden, dass zum Abschluss Variablen wie der nächste Spieler und Punktestände angepasst werden.

Um die Spieler-, sowie Felder Nummerierung zwischen den beiden verschiedenen Systemen übersetzen zu können wurde die Klasse *SystemConversion* geschaffen. Diese nutzt eine von Google Guava bereitgestellte *BiMap*, welche im Gegensatz zu einer normalen Map nicht nur den Schluss von einem Schlüssel auf den Wert erlaubt, sondern es auch ermöglicht einem Wert einen Schlüssel zuzuordnen.

Für jedes Spiel muss eine Klasse geschaffen werden, die von der SystemConversion-Klasse erbt und die *BiMap* füllt, sowie die Grenzen definiert in denen Werte legal sind.

## 2.3 Hürden

Eine der Hürden der Entwicklung bestand darin, die Schnittstelle innerhalb der Ludii Umgebung vollständig lauffähig zu machen. Da dort nicht die Option zur Verfügung steht, das Programm zu debuggen, muss dort mit einem Behelf gearbeitet werden. Dafür stellt die Klasse *Util* im LudiiInterface Ordner des GBG Projekts die Methode *errorDialog* zur Verfügung, die zur Laufzeit Fehlermeldungen ausgibt.

Damit war es möglich einige Probleme zu beheben, die in der programmgesteuerten Verwendung nicht direkt sichtbar wurden. Eines davon betraf die Verwendung der Google Guava Library, welche in den Projekteinstellungen nicht richtig inkludiert war und deshalb nicht beim Erstellen eines Artefaktes berücksichtigt wurde. Dies führte dazu,

dass beim Übersetzen des Ludii Context in das GBG Framework ein Fehler auftrat, der nicht direkt ersichtlich wurde.

Eine weitere Hürde betraf spezifisch das Spiel Othello. Innerhalb von Ludii gab es dort die Besonderheit, dass wenn alle Felder belegt sind, der nächste Spieler noch passen muss, bevor das Spiel beendet wird. Dies führte dazu, dass das Spielende anfangs nicht richtig erkannt werden konnte, da Ludii noch einen Zug erwartete, der GBG Spieler das Spiel aber bereits als beendet ansah und deshalb keinen Zug liefern konnte.

Die Lösung dieses Problems war es, in der Zugauswahl zu prüfen, ob dieser Fall eingetreten ist und dann zu passen.

# 3 Nutzen der Ludii-Schnittstelle

## 3.1 Über die Ludii-GUI

### Erstellen des in Ludii verwendbaren jar-Archivs

Um die GBG Agenten in Ludii nutzen zu können, muss als erstes ein Artefakt des Projektes erzeugt werden. In der Entwicklungsumgebung IntelliJ geschieht dies folgt. Als erstes wird die Projektstruktur geöffnet. Dazu klickt man auf den oben links auf *File*, dann auf *Project Structure*. Dort öffnet man dann den Reiter *Artifacts* und fügt über das + im oberen Bereich des Fensters eine neue *JAR From modules with dependencies* hinzu. Diese beinhaltet dann auch alle für das Programm relevanten Libraries. Zuletzt muss noch der Haken bei *Include in project build* gesetzt werden, wenn das Archiv generiert werden soll und es kann der Pfad festgelegt werden, an dem es gespeichert wird.

Es empfiehlt sich, ebenfalls in den Projekt Einstellungen den Ordner *agents* zu exkludieren, da dieser für das Programm nicht relevant ist, aber die Dateigröße um ein Vielfaches vergrößert und somit auch den Build-Prozess in die Länge zieht.

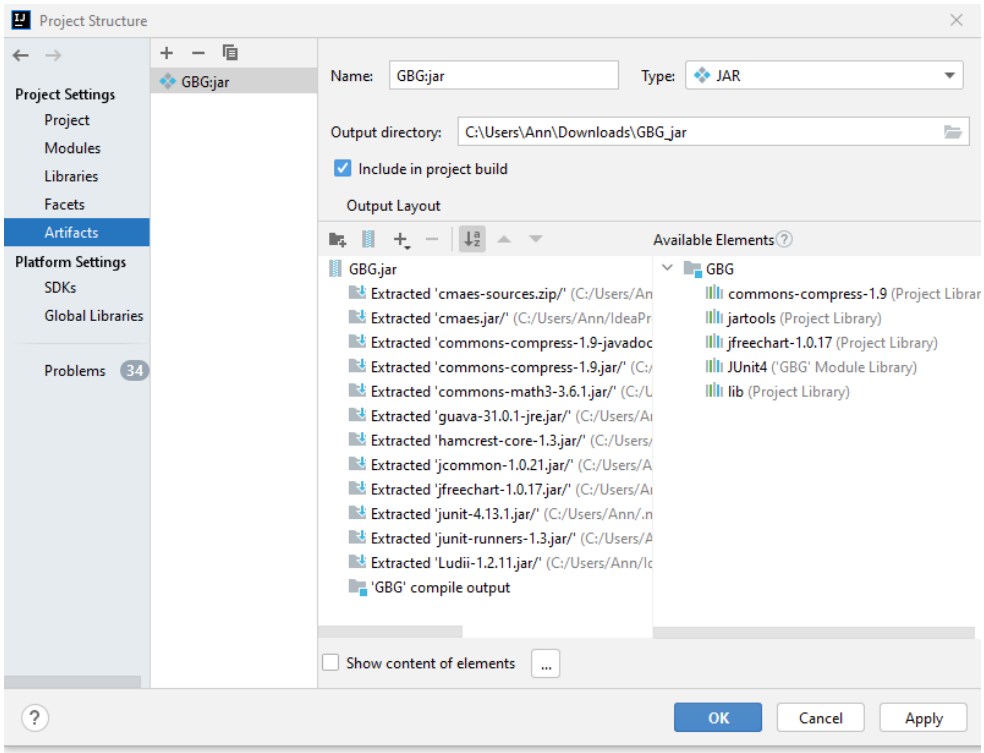


Abbildung 2: Artefakt in den Projekt-Einstellungen einrichten

### Verwenden des Archivs in der grafischen Oberfläche von Ludii

Nachdem die Schnittstelle nun in ein für Ludii verwendbares Format gebracht wurde, muss Ludii geöffnet werden. Es empfiehlt sich die, zum Zeitpunkt der Umsetzung dieses Projekt verfügbare, Version 1.3.0 zu verwenden, die auch im Projektordner *lib* verfügbar ist.

Sobald Ludii geöffnet ist, können wir ein Spiel auswählen. Dies geschieht über *File -> Load Game.* Als Beispiel wird hier Othello genutzt, welches in Ludii unter dem Namen Reversi zu finden ist. Um jetzt den GBG Agenten in Ludii zu laden, klickt man auf einen der Spieler und wählt dann in der Liste der Agenten *From JAR* aus.

Dann wird man aufgefordert die Datei auszuwählen, die die KI enthält, die geladen werden soll. Dies ist das vorher generierte Artefakt des GBG Projekts. Nachdem dieses ausgewählt wurde, wird man aufgefordert die passende Klasse zu wählen, die die KI enthält, was in unserem Fall *ludiiInterface.general.GBGAsLudiiAI* ist.

Danach muss das Spiel über *Game -> Restart* neugestartet werden. Hier kommt jetzt direkt eine Abfrage des GBG Interfaces, welches Spiel gespielt werden soll. Nachdem das Spiel ausgewählt wurde, kommt ein weiterer Prompt, indem man den Agenten auswählen kann. Sobald dieser bestätigt wurde, kann das Spiel über den Play-Button unten rechts gestartet werden.

## 3.2 Programmgesteuert

Während sich die grafische Oberfläche von Ludii zwar gut anbietet um einzelne Spiele abzuhalten und den Spielverlauf in Echtzeit zu beobachten, ist sie eher weniger gut geeignet um große Mengen von Spielen, für zum Beispiel die Datenerfassung, durchzuführen. Ein weiteres Problem ist die nicht vorhandene Möglichkeit das Programm zu debuggen und so Fehler zu finden und zu beheben.

Um diese Probleme anzugehen, gibt es die Möglichkeit Ludii-Spiele über ein Programm zu erstellen und zu spielen. In der Klasse *LudiiCustomMatch* gibt es eine umfassende Implementierung davon. Führt man diese Klasse aus, kann man durch eine Reihe von Dialogfenstern Spiel, Optionen und Anzahl von Spielen, die gespielt werden sollen, auswählen, sowie den Agenten und die Ergebnisse in einer Datei loggen.

Für spezifische Probleme bietet es sich an eine eigene Klasse zu erstellen, die auf dieses Problem zugeschnitten ist und immer das gleiche Spiel mit denselben Parametern startet.

# 4 Datenerhebung

Mit dem Implementieren der neuen Schnittstelle wurde ebenfalls ein Update auf die Ludii Version 1.3.0 durchgeführt, welche eine Veränderung des Agenten mit sich brachte, der für Ludii in Othello antritt. Während dies in Version 1.0.2 noch ein Biased MCTS Agent war, ist es nun ein MCTS Agent, der eine Move-Average Sampling Technique Playout-Strategie verwendet. Diese Strategie beruht auf dem Prinzip, dass Spielzüge, die in einem bestimmten Spielzustand gut sind, es wahrscheinlich auch in anderen Zuständen sind [3, p.2].

Für die Datenerhebung wurde seitens GBG wieder auf den TD-3-Tupel Agenten „TCL3-fixed6_250k-lam05_P4_H001-diff2-FAm.agt.zip“ zurückgegriffen. Dieser wurde bereits von Herr Scheiermann zum Evaluieren, während der ersten GBG-Ludii-Schnittstelle, verwendet und erzielte dabei eine durchschnittliche Siegesquote von 59% erzielte.

Als Rahmenbedingung für die neue Evaluation werden 1000 Spiele gespielt, in denen der GBG Agent den ersten Zug macht und 1000 in denen dies der Ludii Agent tut.

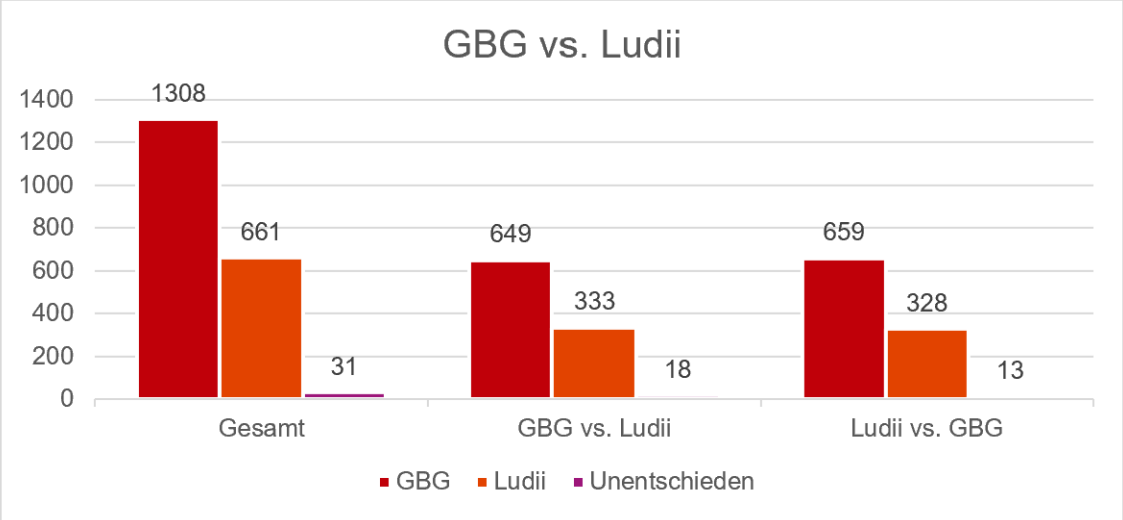


Abbildung 3: Auswertung der Spieledaten zwischen Ludii und dem GBG TD-3-Tupel Agenten

Mit einer durchschnittlichen Siegesrate von 65% ist der Agent jetzt knapp 10% besser als zur letzten Datenerhebung. Ob dies zum Beispiel durch die Verwendung eines anderen Ludii-Agenten zustande kommt ist aber unklar. Bei der Erhebung von Herr Scheiermann wurden damals nur 50 Spiele pro Seite durchgeführt, was das Ergebnis etwas verzerrt haben könnte.

# 5 Umsetzen eines neuen Spiels für die Ludii-Schnittstelle

Um ein neues Spiel für die Ludii-Schnittstelle nutzbar zu machen, müssen zwei neue Klassen geschrieben, sowie zwei vorhandene erweitert werden.

## 5.1 SystemConversion

Die Klasse *SystemConversion* ist dafür zuständig, die, häufig in GBG und Ludii unterschiedlich nummerierten, Spielfelder und Spieler einander zuzuordnen. Dies geschieht über eine bidirektionale Map, die es nicht nur erlaubt den Schlüsseln Werte zuzuordnen, sondern auch umgekehrt von einem Wert wieder auf den Schlüssel zu schließen.

Für neue Spiele muss eine spielspezifische Klasse geschrieben werden, die von dieser Klasse erbt. Es müssen die beiden Methoden *isValidGBGIndex* und *isValidLudiiIndex* implementiert werden, mit denen überprüft werden kann, ob es sich um ein valides Spielfeld handelt, sowie ein Konstruktor geschrieben werden, der die beiden Variablen *indicesMap* und *playerMap* mit den zugehörigen Werten füllt.

In Beiden werden für die Schlüssel die Ludii-Spielfelder, bzw. -Spieler genommen und als Werte die zugehörigen GBG Felder. Bei den Spielern ist es wichtig zu beachten, dass Ludii die Spielernummerierung generell bei 1 beginnt, weshalb diese Unterscheidung notwendig ist. Um die Indizes der Ludii-Spielfelder herauszufinden, kann man in der grafischen Oberfläche von Ludii im Menü *View* den Haken bei *Show Indices* setzen.

Ein Beispiel für diese Implementierung ist die Klasse *SystemConversionOthello*.

## 5.2 State Observer Translation Layer

Die Klasse *StateObserverXYZTranslationLayer* sorgt dafür, dass wir den von Ludii erhaltenen *Context* in ein für unseren GBG Agenten verwendbares Format bringen. Dazu erbt sie von dem bereits vorhandenem spielspezifischem State Observer und implementiert einige neue Methoden, um dies umzusetzen.

Die bisherige Vorgehensweise besteht darin, den Ludii *Context* über einen Konstruktor weiterzugeben und mit Hilfe dessen, und der darin vorhandenen Liste an allen bisher ausgeführten Spielzügen, einen neuen State Observer aufzubauen.

```
public StateObserverYavalathTranslationLayer(Context ludiiContext, int playerID){
    super();
    this.ludiiContext = ludiiContext;
    this.playerID = playerID;
    updateGameState();
}

private void updateGameState(){
    SystemConversionYavalath conversion = new SystemConversionYavalath();
    //Get a list of all moves made in the context
    List<Move> ludiiContextMoves = ludiiContext.trial().generateCompleteMovesList();
    for(Move x : ludiiContextMoves){
        // x.to() returns the position of the stone we place, translate that into a action value
        int actionInt = conversion.getGBGIndexFromLudii(x.to());
        int j = actionInt% ConfigYavalath.getMaxRowLength();
        int i = (actionInt-j)/ConfigYavalath.getMaxRowLength();
        // x.mover() returns who made the move, use that to adjust the state observer board accordingly
        board[i][j].setPlayer(conversion.getGBGPlayerFromLudii(x.mover()));
    }
    currentPlayer = conversion.getGBGPlayerFromLudii(playerID);
    setAvailableActions();

}
```

Abbildung 4: Java Code von StateObserverYavalathTranslationLayer

Ein Beispiel für die Implementierung sind die Klassen *StateObserverOthelloTranslationLayer* oder *StateObserverYavalathTranslationLayer*. Letztere enthält dabei nur eine einfache grundlegende Funktionalität zum Setzen von Spielsteinen, während erstere auch das in Othello vorhandene passen berücksichtigen muss, was stattfindet, wenn ein Spieler gerade keinen Zug machen kann.

## 5.3 GBGAsLudiiAI

Die Klasse *GBGAsLudiiAI* ist für die Kommunikation zwischen dem GBG und dem Ludii-Framework zuständig. Sie erbt von der, von Ludii bereitgestellten, *AI* Klasse. Dort müssen zwei Dinge angepasst werden.

Als erstes muss eine spielspezifische *selectAction* Methode geschrieben werden, die einen gegebenen Context nimmt und den passenden Move zurückgibt. Dazu wird ein neuer StateObserverTranslationLayer des Spiels erstellt und die nächste GBG-Action angefragt. Abschließend wird die Liste der nächsten Spielzüge des Ludii-Context durchsucht und der der Action entsprechende Move ausgewählt.

Als letztes muss dann in der generellen *selectAction* Methode innerhalb der switch-Anweisung noch ein Fall für das neue Spiel geschrieben werden, der die spielspezifische Methode aufruft. Dies erfolgt unter dem Namen des Spiels wie er in Ludii bekannt ist, da die switch-Anweisung auf dem Spielenamen aus dem game-Objekt basiert.

Beispiele für die spielspezifischen *selectAction*-Methoden sind wieder Othello und Yavalath. Während in Yavalath nur die Action abgefragt und übersetzt wird, wird in Othello auch noch geprüft ob überhaupt ein Move gemacht werden kann, da es Situationen gibt, wo der Spieler passen muss. Falls dies der Fall ist, wird ein *PassMove* erstellt und zurückgegeben.

## 5.4 LudiiCustomMatch

Die Klasse *LudiiCustomMatch* muss nur ergänzt werden, wenn auch über diese Spiele abgehalten werden sollen. Wird ein Spiel implementiert, dass nur innerhalb von Ludii gestartet werden soll, muss hier auch nichts gemacht werden.

Möchte man aber über diese Klasse spielen, muss nur das Spiel in der *games* Variable registriert und die switch Anweisung um einen Fall für das Spiel erweitert werden. Dort muss eine neue *GBGAsLudiiAI* erstellt werden, die die ID des Spiels innerhalb des Konstruktors erhält. Dann muss über den GameLoader des Ludii Frameworks das Spiel geladen werden.

## 5.5 Unit-Tests

Um zu testen, ob das StateObserverTranslationLayer des Spiels richtig funktioniert, werden zwei verschiedene Fälle überprüft. In Fall 1 wird von einem Spiel ausgegangen, dessen Spielzüge und Ausgang bereits bekannt ist. Die Züge werden ausgeführt und dann ein neuer StateObserver erstellt. Dann wird geprüft, ob der Ausgang derselbe ist und ob die Scores mit dem Übereinstimmen was erwartet wird.

Für Fall 2 wird ein Spiel zwischen 2 Ludii Agenten abgehalten und wenn dies fertig ist ein StateObserver erstellt. Diesmal wird dann geprüft, ob die Ergebnisse des StateObserver mit denen des Ludii Context übereinstimmen.

Ein Beispiel für diese zwei Tests ist die Klasse *StateObsOthelloTranslationLayerTest*.

## 5.6 Nützliche Ludii Methoden

Folgende von Ludii zur Verfügung gestellte Methoden können während der Umsetzung eines neuen Spiels für die Schnittstelle Verwendung finden:

- **Game Methoden**
  - Game.moves(Context).moves(): Gibt eine FastArrayList vom Typ Move zurück, die alle legalen Spielzüge enthält, die ausgehend vom Context gemacht werden können.
  - Game.apply(Context, Move): Führt den Spielzug Move im Context aus. Nützlich für Tests.
- **Context Methoden**
  - Context.trial().generateCompleteMovelist(): Gibt eine Liste an Moves zurück, die alle bisher im Spiel gemachten Spielzüge enthält.
  - Context.trial().over(): Prüft, ob das Spiel vorbei ist oder noch läuft.

- Context.trial().status.winner(): Gibt die ID des Gewinners des Spiels zurück.
- Context.score(PlayerID): Gibt den Score des Spielers zurück

## 5.7 Aktualisieren der Schnittstelle auf eine neue Ludii Version

Nach dem Erstellen der ersten Version der Ludii-Schnittstelle gab es einige größere Updates, die die Projektstruktur zu großen Teilen änderten.

Zum Aktualisieren der Ludii Schnittstelle auf neuere Ludii Versionen, sollte es reichen die vorhandene Ludii jar Datei durch die Neuere zu ersetzen und diese dann in der Projektstruktur als Library zu hinterlegen. Sofern es bei Ludii keine größeren Änderungen gab, funktioniert die Schnittstelle dann wie gewohnt weiter.

# 6 Nützliche Links zum Umgang mit Ludii

Es gibt einige Seiten, die zum Verständnis von und im Umgang mit Ludii sehr nützlich sind. Diese möchte ich hier kurz auflisten und vorstellen, um den Einstieg zu erleichtern.

- https://github.com/Ludeme/Ludii : Das GitHub-Repository vom Ludii Projekt. Enthält den kompletten, und umfangreich kommentierten, Source-Code und ist sehr nützlich, um schnell verschiedene Dinge nachzuschauen.
- https://ludiitutorials.readthedocs.io/en/latest/index.html : Ludii Tutorials enthält grundlegende Anleitungen und Erklärungen, die den Einstieg erleichtern. Besonders nützlich sind die Seiten *Running Trials, Ludii Programming Cheat Sheet* und *Basic API for AI Development.*
- https://ludii.games/downloads/LudiiGameLogicGuide.pdf : Der Ludii Game Logic Guide ist ein umfangreicher technischer Bericht zum Ludii Projekt und enthält detailreiche Erklärungen zum kompletten System und der Logik dahinter.
- https://ludii.games/download.php : Auf der Download Seite der Ludii Homepage findet sich stets die aktuelle Version des Projektes. Ebenfalls stehen dort alle älteren Versionen zur Verfügung, sowie Links zu weiteren Dokumentationen.

# Literaturverzeichnis

[1] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, & C. Browne, Ludii - The Ludemic General Game System, In Proceedings of the 24th European Conference on Artificial Intelligence, pp. 411-418, 2020, doi 10.3233/FAIA200120

[2] Ludii Release Notes (1.3.1), Abgerufen am 24.01.2022 von https://ludii.games/downloads/Patch1.3.1.txt

[3] Tak, M. J. W., Winands, M. H. M., & Björnsson, Y, Decaying Simulation Strategies in IEEE Transactions on Computational Intelligence and AI in Games, Volume 6, Issue 4, pp. 395-406, 2014, doi 10.1109/TCIAIG.2014.2310782

# Anhang

Der gesamte entwickelte Quellcode ist in dem GitHub-Repository https://github.com/WolfgangKonen/GBG einsehbar.