UNIVERSITÄT PADERBORN

*Die Universität der Informationsgesellschaft*

Heinz Nixdorf Institut
Fachgruppe Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

# Interface-Dokument

im Rahmen des Softwaretechnikpraktikums 2017

## Interface-Komitee

**Version** 1.1.1

**Betreuer:** Mario Treiber

Paderborn, den 29. Juni 2017

**Autoren:**

Jonas Böger
Eugen Gerb
Josua Köhler
Björn Luchterhandt
Tobias Pudenz
Thorsten Wichmann
Amin Faez
Ralf Keller
Marcel Kürvers
Julia Peters
Viktor Schellenberg
Bernd Löhr

# Inhaltsverzeichnis

# Changelog

## Version 1.1.1

**Abschnitt 2.3.3** Änderung des Typ beim Attribut *role* von *Role* in *ClientRole*.

**Abschnitt 2.3.4** Im Fließtext *Role* in *ClientRole* geändert.

**Abschnitt 2.3.5** In der Beschreibung des Attributes *clientID* stand dass für die Rolle *SPECTATOR* dieses Attribut auf *NULL* gesetzte werden soll. Da *int* in Java ein primitiver Datentyp ist, ist setzen auf NULL nicht möglich. Daher von setzen auf NULL zu setzen auf $-1$ geändert.

**Abschnitt 2.5** Inkonsistenz zum EMF-Diagramm. Da steht bei *TurnNotification*: *setToken*. Im Text selber aber *moveToken*. Geändert im Text von *moveToken* auf *setToken*

**Abschnitt 2.5.2** Beschreibenden Text zum Attribut *setToken* eingefügt.

**Abschnitt 2.5.6** Die Reason *OUT_OF_BOARD* eingefügt. Sie gibt den Grund an, wenn ein Spieler regulär verloren hat. Die Reason *OUT_OF_BOARD* wieder entfernt. Um die .jar nicht unnötig zu verändern wird für das reguläre Ausscheiden die Reason *DEFAULT* genutzt.

# 1 Einleitung

Das folgende Dokument beschreibt die Interfaces für das Software- und Softwaretechnikpraktikum 2017. Entwickelt wurden die Interfaces von einem Interfacekomitee bestehend aus je einem Mitglied jeder SWTPra- und SoPra-Gruppe. Angeleitet und betreut wurde das Komitee von Herrn Mario Treiber. Alle wichtigen Entscheidungen wurden über Abstimmungen getroffen.

Der Hauptbestandteil des Interface-Dokuments ist ein EMF-Modell, welches die Struktur der Spielkonfiguration und das Netzwerkprotokoll definiert. Dabei handelt es sich um ein Modell von Klassen, deren Zweck darin besteht, den Aufbau der Konfiguration und der Netzwerkbefehle festzulegen.

Die Objekte dieses Modells können unter Verwendung von *emfjson* im JSON-Format serialisiert werden. In dieser Form werden Nachrichten über das Netzwerk übertragen. Auch die Spielkonfiguration wird so in JSON serialisiert und in einer Datei gespeichert.

Abschließend werden Beispiele für die korrekte Nutzung der Spielkonfiguration und des Netzwerkprotokolls gegeben.

# 2 Netzwerk

Die Netzwerk-Kommunikation zwischen Tsuro-Client und -Server findet über ein JSON-basiertes Protokoll statt. Ein Client ist dabei ein Spieler oder ein Zuschauer. Nachrichten werden über eine TCP-Verbindung gesendet und der Standardport für einen Tsuro-Server ist 33098.

Das Diagramm in Abbildung 1 liefert eine Übersicht über das EMF-Modell, bzw. die Klassen des Netzwerkprotokolls. Nachrichten sind dabei ebenso durch Klassen modelliert wie Datenhaltungsobjekte. Dieser Abschnitt erläutert die Nachrichten- und Datenklassen genauer und beschreibt ihren Einsatz.

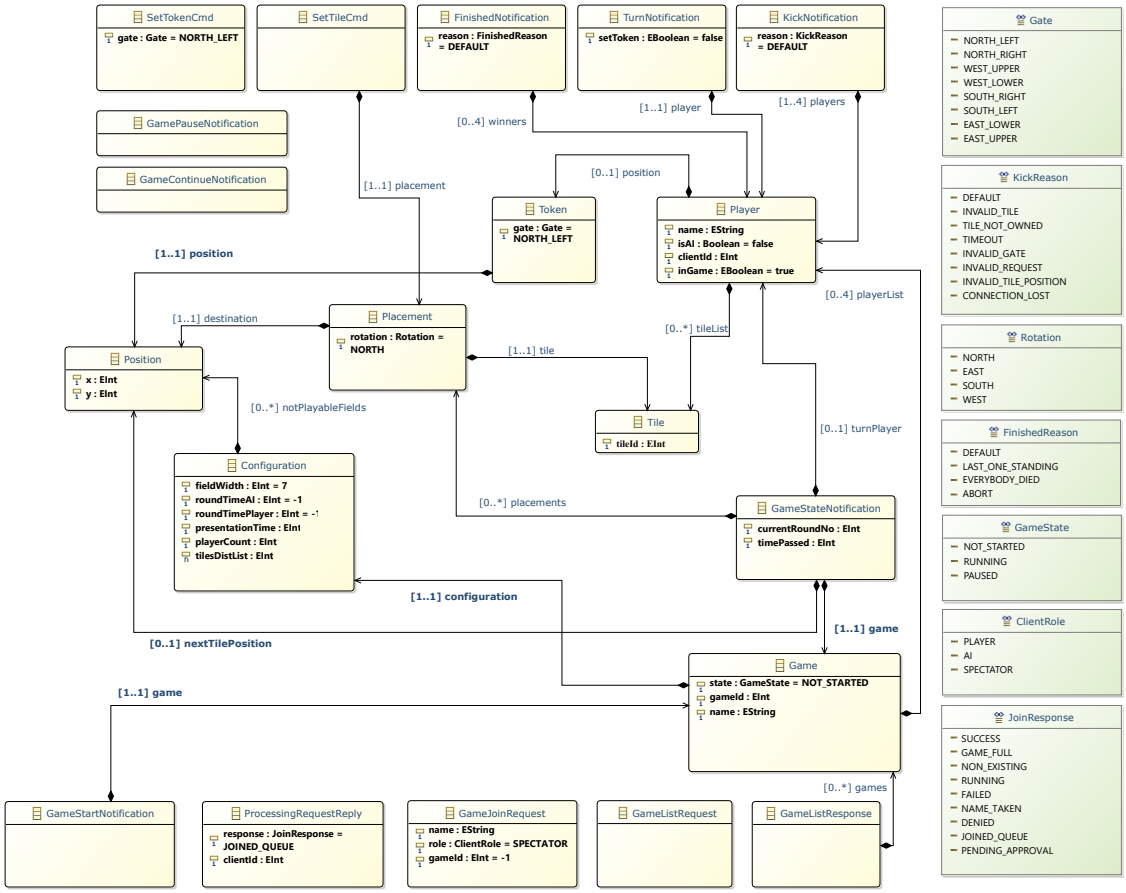


Abbildung 1: EMF-Diagramm der Interfaceklassen und der Konfiguration (in Abschnitt 3 näher erläutert)

Nachrichtenklassen des Typs *Command* werden im EMF-Modell mit *Cmd* abgekürzt. Weiterhin sei gesagt, dass alle Nachrichten der Abschnitte 2.4 bis 2.7 vom Server jeweils an *alle* im Spiel anwesenden Clients gesendet werden.

## 2.1 Trennung der Nachrichten im TCP-Stream

Auf Seiten der Empfänger wird durch ein verbindlich definiertes Trennzeichen die Abgrenzung verschiedener Nachrichten ermöglicht, um diese getrennt zu deserialisieren. Dieses ist das Ascii-Zeichen `10` (Hex: `0A`) bzw. als Escape-Sequenz `\n`. Das Trennzeichen muss als letztes Zeichen *jeder* Nachricht gesendet werden.

## 2.2 Datenklassen

### 2.2.1 Player

Die Klasse *Player* repräsentiert einen Spieler. Spieler haben eine eindeutige *clientId*, durch die sie von der Engine angesprochen werden können. Außerdem haben sie einen Namen und einen Boolean-Flag *isAI*, der festlegt, ob es sich um einen menschlichen Spieler oder eine künstliche Intelligenz (KI) handelt.

**name: EString** - Name des Spielers

**clientId: EInt** - Id des Teilnehmers

**isAI: EBoolean** - Gibt an, ob der Spieler eine KI ist.

**inGame: EBoolean** - Gibt an, ob der Spieler noch im Spiel ist, oder bereits verloren hat.

**tileList: EList<Tile>** - Liste der Wegfelder, die der Spieler auf der Hand hat

**position: Token** - Die Position des Spielers. In der ersten Runde, bevor die Spieler ihre Figur platziert haben, ist dieser Wert auf `null` gesetzt.

### 2.2.2 Tile

Ein Wegfeld wird durch die Klasse *Tile* modelliert. Die Ausprägung von Wegfeldern (wie die Feldenden verbunden sind) wird mittels des Attributes *tileId* festgelegt. Die Abbildung 5 im Anhang definiert die Ids aller möglichen Wegfelder.

**tileId: EInt** - Identifiziert die Ausprägung des Wegfeldes [0..34].

### 2.2.3 Gate

[Enum]

Die Klasse *Gate* ist als Enumeration spezifiziert und gibt einen der Acht Ein- bzw. Austrittspunkte (im Folgenden „Gate“) eines Wegfeldes an. Die Werte und ihre Bedeutungen können der Abbildung 2 entnommen werden.

Weiter ist bei der Angabe von Gates die *Rotation* eines Wegfeldes irrelevant (d.h. *NORTH_LEFT* ist immer das vom Betrachter aus linke obere Gate eines Wegfeldes, egal, welche *Rotation* dieses hat).

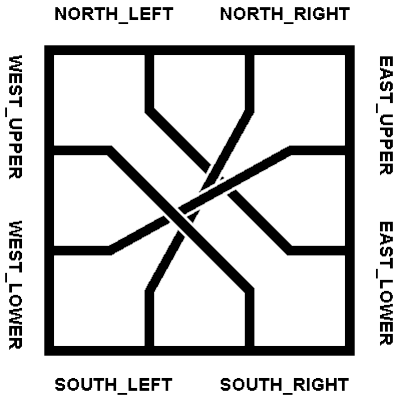


Abbildung 2: Die Benennung der Gates auf den Wegfeldern

#### 2.2.4 Position

Diese Klasse beschreibt eine Position auf dem Spielfeld. Der Ursprung des Koordinatensystems ist dabei die linke obere Spielfeldecke.

**x: EInt** - Spalte des Spielfeldes, beginnend bei 0

**y: EInt** - Reihe des Spielfeldes, beginnend bei 0

#### 2.2.5 Token

Die Klasse *Token* beschreibt die Spielfiguren des Spieles „Tsuro“. Dabei gehört maximal eine Spielfigur zu genau einem Spieler. Die Attribute *position* und *gate* spezifizieren die Lage der Spielfigur auf dem Spielbrett.

**gate: Gate** - Das Gate an dem die Spielfigur steht.

**position: Position** - Die Position des Wegfeldes an dessen *gate* die Spielfigur steht.

#### 2.2.6 Rotation

[Enum]

Die Klasse *Rotation* ist als Enumeration spezifiziert. Sie wird zur Angabe der Drehung eines Wegfeldes verwendet.

**NORTH** - Das Wegfeld wird wie in der Definition im Anhang ausgerichtet.

**EAST** - Das Wegfeld ist aus Initialstellung um 90° im Uhrzeigersinn gedreht.

**SOUTH** - Das Wegfeld ist aus Initialstellung um 180° im Uhrzeigersinn gedreht.

**WEST** - Das Wegfeld ist aus Initialstellung um 270° im Uhrzeigersinn gedreht.

#### 2.2.7 Placement

Die Klasse *Placement* repräsentiert die Platzierung eines Wegfeldes. Sie speichert ein Wegfeld, seine Position und seine Ausrichtung.

**tile: Tile** - Das platzierte Wegfeld

**rotation: Rotation** - Die Ausrichtung des Wegfeldes

**destination: Position** - Die Position auf dem Spielfeld

#### 2.2.8 Game

Die Klasse *Game* repräsentiert ein Spiel.

**playerList: EList<player>** - Liste der *Player*-Objekte aller beteiligten Spieler

**state: GameState** - Status des Spiels

**name: EString** - Name des Spiels

**gameId: EInt** - Id des Spiels

**config: Configuration** - Spielkonfiguration

#### 2.2.9 GameState

[Enum]

Die Klasse *GameState* ist als Enumeration spezifiziert. Sie dient zur Angabe des Spielstatus.

**RUNNING** - Das Spiel läuft.

**NOT_STARTED** - Das Spiel wurde vom Engine-Benutzer noch nicht gestartet.

**PAUSED** - Das Spiel ist pausiert.

### 2.3 Spielbeitritt

Sobald ein Client eine Verbindung mit einem Server aufgebaut hat, kann er dem Pool der wartenden Clients mit einem *GameJoinRequest* beitreten. In diesem Fall muss das Attribut *gameId* im *GameJoinRequest* auf $-1$ gesetzt werden.

Optional kann ein Server auch den Beitritt in ein vom Client gewünschtes Spiel anbieten. Dazu ist es Clients möglich, mit einem *GameListRequest* eine Liste aller vom Server bereitgestellten Spiele anzufordern. Der Server antwortet darauf mit einer *GameListResponse*, die die *Game*-Objekte der angebotenen Spiele beinhaltet. Mit diesen Informationen kann sich der

Benutzer des Clients für ein Spiel entscheiden. Der Client kann dann einen *GameJoinRequest* mit der Id des gewünschten Spiels im Attribut *gameId* senden.

Als Antwort auf einen *GameJoinRequest* sendet der Server eine *ProcessingRequestReply*. Dabei ist abhängig von der *gameId* im *GameJoinRequest* zwischen zwei Fällen zu unterscheiden:

- $gameId = -1$: Die *ProcessingRequestReply* enthält als *response* einen der Werte *FAILED*, *NAME_TAKEN* und *JOINED_QUEUE*.
- $gameId \geq 0$: Bei Bearbeitung des *GameJoinRequests* wird eine erste *ProcessingRequestReply* gesendet, die als *response* bei Fehlschlag einen der Werte *FAILED*, *NAME_TAKEN*, *GAME_FULL*, *NON_EXISTING* und *RUNNING* enthält. Falls die Anfrage auf kein Fehler stößt wird zunächst mit dem Wert *PENDING_APPROVAL* geantwortet. Der Ausrichter kann anschließend über den Spielbeitritt entscheiden. Hat er seine Entscheidung getroffen, so sendet der Server erneut eine *ProcessingRequestReply*, die als *response* entweder *SUCCESS* oder *DENIED* enthält.

Sollte der Server nur den Beitritt in den Pool unterstützen, kann er für alle *GameJoinRequests* mit *gameId* ungleich $-1$ eine *GameJoinReply* mit dem Wert *FAILED* als *response* senden.

#### 2.3.1 GameListRequest

[Client $\rightarrow$ Server]

Durch einen *GameListRequest* fragt ein Client den Server nach einer Liste aller von ihm bereitgestellten Spiele.

#### 2.3.2 GameListResponse

[Server $\rightarrow$ Client]

Durch eine *GameListResponse* sendet der Server eine Liste aller von ihm bereitgestellten Spiele an den anfragenden Client.

**games: EList<Game>** - Eine Liste aller bereitgestellten Spiele

#### 2.3.3 GameJoinRequest

[Client $\rightarrow$ Server]

Der *GameJoinRequest* wird vom Client verwendet, um anzufragen, ob er einem bestimmten Spiel beitreten kann. Dabei sendet er seinen Namen und seine Rolle.

**name: EString** - Der Name des Spielers

**role: ClientRole** - Die Rolle des Spielers

**gameId: EInt** - Das vom Client gewünschtes Spiel, $-1$ wenn dem Pool beigetreten werden soll

#### 2.3.4 ClientRole

[Enum]

Die *ClientRole* definiert die Rolle eines Clients im Spiel.

**PLAYER** - Der Client ist ein menschlicher Spieler.

**AI** - Der Client ist eine künstliche Intelligenz.

**SPECTATOR** - Der Client ist ein Beobachter.

#### 2.3.5 ProcessingRequestReply

[Server $\rightarrow$ Client]

Eine *ProcessingRequestReply* wird vom Server an einen Client gesendet, sobald es Neuigkeiten zur Bearbeitung seines *GameJoinRequests* gibt. Sie teilt den Status des Spielbeitritts im Attribut *response* mit.

**response: JoinResponse** - Enthält den Status des Spielbeitritts.

**clientId: EInt** - Falls der Client als Rolle *PLAYER* oder *AI* angegeben hat, so wird ihm in diesem Attribut seine *clientId* bekanntgegeben. Für die Rolle *SPECTATOR* ist *clientId* auf $-1$ zu setzen.

#### 2.3.6 JoinResponse

[Enum]

Die *JoinResponse* informiert über den Status der Bearbeitung eines *GameJoinRequests*.

**FAILED** - Der Beitritt war aus einem nicht definierten Grund nicht erfolgreich.

**NAME_TAKEN** - Der im *GameJoinRequest* angegebene Name existiert bereits.

**JOINED_QUEUE** - Der Beitritt zum Pool war erfolgreich.

**GAME_FULL** - Das Spiel, dem beigetreten werden sollte, ist voll.

**NON_EXISTING** - Das Spiel, dem beigetreten werden sollte, existiert nicht.

**RUNNING** - Das Spiel, dem beigetreten werden sollte, ist bereits gestartet.

**PENDING_APPROVAL** - Wird direkt nach dem Erhalt eines *GameJoinRequests* mit *gameId* $\geq 0$ gesendet.

**DENIED** - Wird gesendet, wenn der Ausrichter den Beitritt zum im *GameJoinRequests* genannte Spiel abgelehnt hat

**SUCCESS** - Wird gesendet, wenn der Ausrichter den Beitritt zum im *GameJoinRequests* genannte Spiel erlaubt hat.

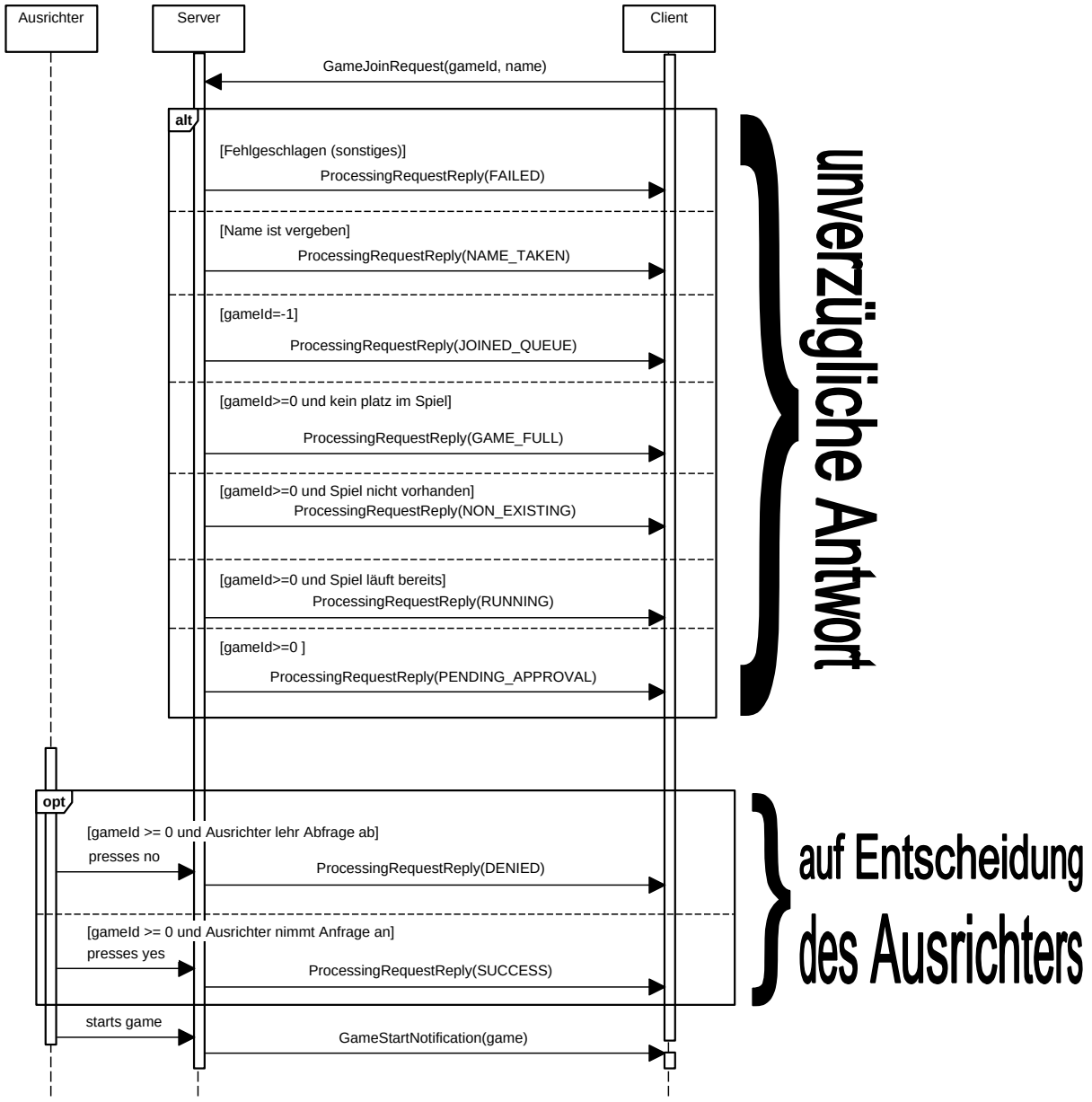


Abbildung 3: Spielbeitritt

## 2.4 Spielstatus

Der aktuelle Spielstatus wird vom Server vor jedem Zug in Form einer *GameStateNotification* an alle Clients gesendet.

### 2.4.1 GameStateNotification

[Server → Client]

Die *GameStateNotification* gibt die aktuelle Spielsituation bekannt.

**game: Game** - Das *Game*-Objekt des aktuellen Spiels

**currentRoundNo: EInt** - Gibt die Nummer der nächsten Spielrunde an (beginnend bei 1 für die erste Spielrunde). Wird um 1 erhöht, sobald alle Spieler genau einen Spielzug durchgeführt haben.

**timePassed: EInt** - Gibt die seit Spielbeginn verstrichene Zeit in Millisekunden an.

**nextTilePosition: Position** - Gibt die Wegfeldposition für den nächsten Spielzug an. *null*, falls dieser noch nicht bestimmbar ist.

**placements: EList<Placement>** - Liste aller Wegfeld-Platzierungen auf dem Spielfeld (Entspricht dem Zustand des Spielfeldes).

## 2.5 Spielbeginn

Sobald der Ausrichter ein Spiel startet, sendet der Server an alle im Spiel anwesenden Clients eine *GameStartNotification*, gefolgt von der ersten *GameStateNotification*. Sobald die im *Configuration*-Objekt angegebene *presentationTime* abgelaufen ist, sendet der Server eine *TurnNotification*. Ist keine *presentationTime* spezifiziert, so werden beide Nachrichten möglichst ohne zeitlichen Abstand gesendet.

Der Client des in der *TurnNotification* genannten Spielers hat nun die im *Configuration*-Objekt definierte Zeit von *roundTimePlayer* bzw. *roundTimeAI*, um an den Server ein *SetTileCommand* zu senden. Ist dies erfolgt und die übermittelten Daten sind gültig, so wiederholt sich der beschriebene Ablauf von *GameStateNotification*, *TurnNotification* und *SetTileCommand* für den nächsten Spieler.

Sobald alle Spieler ihr erstes Wegfeld gelegt haben, sendet der Server erneut eine *GameStateNotification* sowie eine *TurnNotification*, die sich auf den ersten Spieler bezieht. Hier muss als *setToken*-Attribut *true* gesetzt sein, da der Spieler zum Setzen seiner Spielfigur an eine gültige Position am Spielfeldrand aufgefordert werden soll. Der Client des aufgeforderten Spielers muss diesmal innerhalb der gesetzten *roundTime* mit einem *SetTokenCommand* antworten. Der Ablauf zum Setzen der Spielfigur wiederholt sich ebenfalls für jeden Spieler.

Falls ein Client nicht in der vorgegebenen Zeit antwortet, oder seine Nachricht ungültig ist, reagiert der Server mit einem *KickCommand*, das den betroffenen Spieler verlieren lässt.

#### 2.5.1 GameStartNotification

[Server → Client]

Die *GameStartNotification* signalisiert einem Client, dass ein Spiel beginnt, an dem er teilnimmt. Sie kann auch ohne vorherigen *GameJoinRequest* des Clients gesendet werden, wie es z.B. im Turnier erforderlich ist.

**game: Game** - Das *Game-Objekt* des beginnenden Spiels

#### 2.5.2 TurnNotification

[Server → Client]

Die *TurnNotification* fordert einen Spieler zum Zug auf.

**player: Player** - Das *Player*-Objekt des Spielers, der am Zug ist.

**setToken: EBoolean** - Gibt an, ob das Token des Spielers gesetzt werden kann.

#### 2.5.3 SetTileCommand

[Client → Server]

Mit einem *SetTileCommand* gibt ein Client dem Server bekannt, wie sein Spieler ein Wegfeld legen möchte.

**placement: Placement** - Ein *Placement*-Objekt, das die Platzierung des zu legenden Wegfeldes beschreibt

#### 2.5.4 SetTokenCommand

[Client → Server]

Der *SetTokenCommand* wird vom Spieler-Client gesendet, um die eigene Spielfigur zu Beginn des Spieles auf dem gewünschten Ort am Spielfeldrand zu stellen.

**gate: Gate** - Das Gate am Rande des Spielfeldes, auf das die Spielfigur des Spielers platziert werden soll. Die Orientierung wird aus der Sicht des platzierten Wegfeldes angegeben. Falls der Spieler in der Runde zuvor ein Wegfeld in der linken obere Ecke, Position *(0,0)*, gelegt hat, sind einzig die *NORTH* und *WEST* gates valide Angaben zur Positionierung der Spielfigur.

#### 2.5.5 KickNotification

[Server → Client]

Ein *KickCommand* informiert alle Clients, insbesondere den betroffenen Spieler, dass ein bis vier Spieler verloren haben. Des Weiteren gibt er den Grund dazu an. Nach Erhalt eines

KickCommands wird der betroffene Client nicht wieder zum Zug aufgefordert, erhält aber trotzdem alle Nachrichten vom Server wie zuvor. So Kann der weitere Spielverlauf in der Client-Anwendung ebenfalls visualisiert werden.

**players: EList<Player>** - Spieler, die verloren haben

**reason: KickReason** - Der Grund für das Verlieren

#### 2.5.6 KickReason

[Enum]

Die *KickReason* definiert den Grund, aus dem ein Spieler verloren hat.

**DEFAULT** - Wenn das Token eines Spielers das Spielfeld verlassen hat oder mit einem Token eines anderen Spielers zusammengestoßen ist.

**INVALID_TILE** - Der Client hat dem Server eine *tileId* übergeben, die nicht existiert.

**TILE_NOT_OWNED** - Der Client hat dem Server eine *tileId* zu einem Wegfeld übergeben, das der Spieler nicht auf der Hand hat.

**TIMEOUT** - Der Client hat nicht innerhalb der gegebenen Zeitspanne geantwortet.

**INVALID_GATE** - Der Client hat beim Ziehen der Spielfigur ein ungültiges *Gate* angegeben.

**INVALID_REQUEST** - Der Client hat dem Server eine ungültige Nachricht oder eine Nachricht mit falschen Parametern gesendet.

**INVALID_TILE_POSITION** - Der Client hat dem Server eine ungültige Tile-Position übergeben.

**CONNECTION_LOST** - Die Verbindung zum Client ist unterbrochen worden und konnte nicht wiederhergestellt werden.

### 2.6 Spielverlauf

Im weiteren Verlauf des Spieles sendet der Server für jeden Spielzug eine*GameStateNotification* und eine *TurnNotification*, worauf der Client des Spielers, der am Zug ist, innerhalb der dafür vorgesehenen Zeitspanne mit einem *SetTileCommand* antworten muss. Kommt er dieser Pflicht nicht nach, oder ist die übermittelte Nachricht ungültig, so sendet der Server auch hier einen *KickCommand* für den betroffenen Spieler.

Der gesamte Spielverlauf (inklusive Spielbeginn und Spielende) ist in Abbildung 4 als Statechart visualisiert.

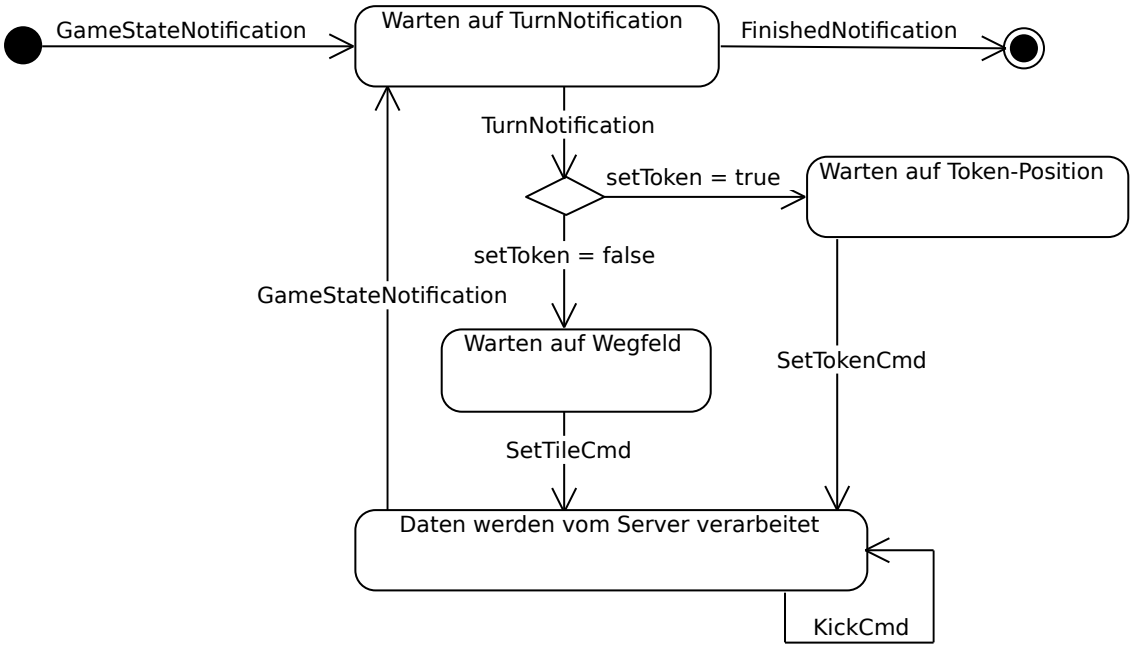


Abbildung 4: Statechart zum Spielablauf

## 2.7 Spielende

Das Spiel wird beendet, wenn nach einem *SetTileCommand* nur noch eine Spielfigur auf dem Feld steht, oder alle noch teilnehmenden Spieler im selben Zug das Spielfeld verlassen. Der Server schickt in diesen Fällen eine *FinishedNotification*.

### 2.7.1 FinishedNotification

[Server → Client]

Sobald das Spiel beendet wurde, wird an alle Clients die *FinishedNotification* gesendet. Sie gibt die Sieger und den Grund für den Sieg an.

**winners: EList<Player>** - Eine Liste der Gewinner

**reason: FinishedReason** - Der Grund für den Sieg der Gewinner

### 2.7.2 FinishedReason

[Enum]

Die *FinishedReason* definiert den Grund, aus dem ein Spieler gewonnen hat.

**DEFAULT** - Wenn kein anderer Grund passt, oder die Angabe des genauen Grundes nicht implementiert ist

**LAST_ONE_STANDING** - Der Spieler ist der Letzte mit einem Spielstein auf dem Feld.

**EVERYBODY_DIED** - Jemand hat ein Feld gelegt, sodass die zwei verbliebenen Spielfiguren zusammenstoßen würden, oder alle noch aktiven Spieler ihren Spielstein vom Spielfeld bzw. auf ein gesperrtes Feld bewegen müssen.

**ABORT** - Der Spielausrichter hat das Spiel abgebrochen.

## 2.8 Pausieren des Spieles

Der Ausrichter eines Spieles hat die Möglichkeit, dieses jederzeit zu pausieren. In diesem Fall sendet der Server eine *GamePauseNotification*. Ein Client soll dem Benutzer bei einem pausierten Spiel eine entsprechende Meldung anzeigen und keine Spielzüge senden. Auch die Zeitbeschränkungen für die Züge der Spieler werden pausiert.

Sollte ein Client während eines pausierten Spieles einen gültigen Spielzug senden (zum Beispiel, wenn Spielzug und *GamePauseNotification* gleichzeitig gesendet werden), wird dieser vom Server akzeptiert und normal verarbeitet. Die anschließende *GameStateNotification* und die nächste *TurnNotification* werden jedoch erst gesendet, wenn das Spiel fortgesetzt wird.

Sobald der Ausrichter das Spiel fortsetzt, wird eine *GameContinueNotification* gesendet. Die Zeitbeschränkungen der Spieler werden dann wieder aktiv.

### 2.8.1 GamePauseNotification

[Server $\rightarrow$ Client]

Die *GamePauseNotification* wird vom Server gesendet, wenn das Spiel vom Ausrichter pausiert wird.

### 2.8.2 GameContinueNotification

[Server $\rightarrow$ Client]

Die *GameContinueNotification* wird vom Server gesendet, wenn ein pausiertes Spiel vom Ausrichter fortgesetzt wird.

# 3 Configuration

Die Klasse *Configuration* enthält alle spielrelevanten Variablen zum Starten eines Spieles. Diese Informationen werden dem Server vom Spielkonfigurator bereitgestellt.

**fieldWidth: EInt** - Setzt die größe des Spielfeldes fest. Das Spielfeld wird als quadratisch angesehen. Der Wert der Variable ist mindestens 2.

**playerCount: EInt** - Legt die Anzahl der Spieler fest. Es können 2 bis 4 Spieler gegeneinander antreten.

**notPlayableFields: EList<Position>** - Enthält die Koordinaten der nicht bespielbaren Spielfelder.

**tilesDistList: EList<EInt>** - Hat eine feste Größe von 35. Die einzelnen Einträge geben die Anzahl der Wegfelder an, die den Spielern zu Beginn zur Verfügung stehen. Ist z.B. in der Liste an Position 14 eine 3 eingetragen, so sind jedem Spieler zu Beginn 3 Wegfelder mit der ID 14 zuzuordnen.

**roundTimeAI: EInt** - Gibt die Zeit in Millisekunden an, die einer KI zum Durchführen eines Spielzuges zur Verfügung steht. $-1$ bdeutet, dass es keine Zeitbegrenzung gibt.

**roundTimePlayer: EInt** - Gibt die Zeit in Millisekunden an, die einem menschlichen Spieler zum Durchführen eines Spielzuges zur Verfügung steht. $-1$ bdeutet, dass es keine Zeitbegrenzung gibt.

**presentationTime: EInt** - Gibt die Zeit in Millisekunden an, die gewartet wird, bis eine neue Aktion der Teilnehmer gefordert wird.

# 4 Beispiel

## 4.1 De- / Serialisierung von EMF-Objekten

Der folgende Java-Code beschreibt beispielhaft die von uns festgelegte Möglichkeit, Objekte einer Klasse (hier Player) aus dem Tsuro-EMF-Modell mittels emfjson zu serialisieren (EMF-Objekt zu JSON-String), und zu deserialisieren (JSON-String zu EMF-Objekte).

Benötigt werden hierzu die Bibliotheken FasterXML-Jackson, sowie emfjson, die von `http://wiki.fasterxml.com/JacksonHome` bzw. `http://emfjson.org` bezogen werden können. Außerdem wird die `JSONHandler`-Klasse verwendet, welche mit dem Interfacedokument mitgeliefert ist.

```
package tsuro_Interface.tests;
import static
   de.upb.swt.swtpra2017.swtinterface.Tsuro_InterfaceFactory.eINSTANCE;
import java.io.IOException;
import de.upb.swt.swtpra2017.swtinterface.*;
import de.upb.swt.swtpra2017.swtinterface.network.JsonHandler;

public class Test {
  public static void main(String[] args){
    TEST_SERIALIZATION();
    TEST_DESERIALIZATION();
  }
  private static void TEST_SERIALIZATION(){
    String output = "";

    // Beispiel Daten
    Player player = eINSTANCE.createPlayer();
    player.setClientId(12);
    player.setIsAI(false);
    player.setName("Charlie Brown");
    Token token = eINSTANCE.createToken();
    token.setGate(Gate.NORTH_RIGHT_LITERAL);
    Position position = eINSTANCE.createPosition();
    position.setX(4);
    position.setY(2);
    token.setPosition(position);
    player.setPosition(token);

    // Beispiel Daten serialisieren
    try {
      output = JsonHandler.serializeJson(player);
    } catch (IOException e) {
```

```
      output = "Error";
    }
    System.out.println(output); //ausgabe JSON-String
  }

  private static void TEST_DESERIALIZATION(){
    String input = //Beispiel JSON-String
        "{" +
            "\"eClass\":\"de.upb.swt.swtpra2017.swtinterface#//Player\","
               +
            "\"name\" : \"Woodstock\"," +
            "\"clientId\" : 15," +
            "\"position\" : {" +
            "\"eClass\" :
               \"de.upb.swt.swtpra2017.swtinterface#//Token\"," +
              "\"gate\" : \"SOUTH_RIGHT\"," +
              "\"position\" : {" +
              "\"eClass\" :
                 \"de.upb.swt.swtpra2017.swtinterface#//Position\","
                 +
                "\"x\" : 3," +
                "\"y\" : 1" +
               "}" +
             "}" +
    "}";
    // Beispiel zu Daten deserialisieren
    Player player = null;
    try {
      player = (Player) JsonHandler.deserializeJson(input);
    } catch (IOException e) {
      e.printStackTrace(System.out);
      System.out.println("Error");
    }
    //Daten verarbeiten
    if(player != null)
      System.out.println(player.getName()); //ausgabe: Woodstock
  }
}
```

# A Tiles

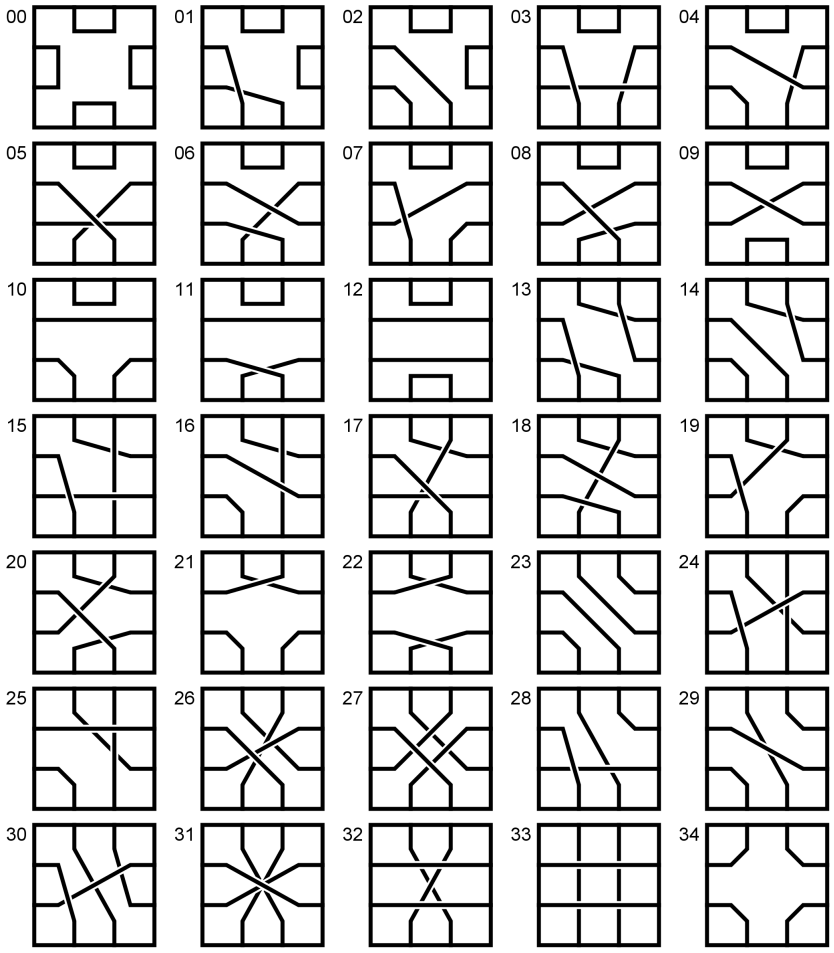


Abbildung 5: Alle Tiles mit eindeutiger ID-Bezeichnung

# Abbildungsverzeichnis