1. Einleitung zum Netzwerkprotokoll

Dieses Netzwerkprotokoll verletzt zahlreiche REST-Best-Practices. Dies um die Netzwerkkommunikation 1) mit für Sie minimalen (eingewöhnungs) Aufwand, 2) möglichst einfach und fehlervermeident für Ihre Implementierungen und 3) mit einem zu schon gewohnten Methodenaufrufen möglichst ähnlich Verhalten/Stil zu realisieren. Trotzdem aber noch ein paar grundlegende REST-Eigenheiten zu zeigen.
Sollten Sie jemals selbst ein REST-Interface entwerfen, übernehmen Sie bitte diesen Stil nicht, sondern befassen Sie sich zuvor mit Themen wie HTTP-Statuscodes und -Methoden, API-Design, HATEOAS, API-Versionierung, Datenstrukturierung, non-blocking Kommunikation etc. Vielleicht lohnt sich auch direkt ein Blick auf "andere" Ansätze wie GraphQL. Siehe z.B. Kursempfehlung unterhalb. Diese Themen (und eine Netzwerkkommunikation selbst zu entwerfen) gehen aber über die Ziele des Kurses hinaus.
Das im Folgenden beschriebene Netzwerkprotokoll dient dazu, dass Sie im Tournament ein standardisiertes Netzwerkprotokoll vorfinden. Dies ermöglicht Ihnen während des Tournaments gegen andere Studierende in einem freundschaftlichen Wettstreit anzutreten. Hierzu müssen Sie dieses Protokoll in Teilaufgabe 2 und Teilaufgabe 3 integrieren. Die folgende Beschreibung des Protokolls ist immer im Zusammenspiel mit der Spielidee zu lesen, daher das Protokoll bildet die Anforderungen der **Spielidee** ab. Dieses Dokument dient nur zu Beschreibung des Protokolls. Die eigentliche Implementierung des Netzwerkprotokolls erhalten Sie im Verlauf des Semesters von der LV **zur Verfügung gestellt**. Diese können Sie dann einfach in Ihr Projekt einbinden.

Im Folgenden werden die Nachrichten und technischen Hintergründe des Protokolls beschrieben. Die technologische Basis des Nachrichtenaustauschs stellt eine Restschnittstelle dar, daher es wird das HTTP Protokoll verwendet sowie die zugehörigen Operationen GET und POST. Die ausgetauschten Daten bzw. Nachrichten werden im XML Format definiert bzw. erwartet. Damit Sie sich während der Entwicklung keine Gedanken um HTTP Statuscodes machen müssen wird das Protokoll *immer* (auch z.B. bei vom Client versandten falschen Karten) den Code 200 verwenden. Grundlegende Fehler im Zugriff z.B. falsche Endpoints URLs können auch zu anderen Codes wie beispielsweise 404 führen. *Sollten Ihnen HTTP Statuscodes nichts sagen können Sie*

diese einfach komplett ignorieren. Bei Interesse an diesem Thema siehe z.B. RFC 9110.

Weiterführende Materialien: Sollten Sie sich für das Thema Netzwerkkommunikation, REST (inkl. Implementierung Client/Server), XML und XML Schema sowie grundlegende weiterführende Techniken der Netzwerkkommunikation interessieren (diese gehen über unseren Kurs hinaus) dann finden Sie ein passendes **Skriptum hier.** Die darin vermittelten Themen sind auch für andere Kurse wie Distributed Systems Engineering interessant und decken auch die Grundlagen passender Implementierungen mit Spring ab.

2. Modellierung des Netzwerkes im UML Klassendiagramm (Teilaufgabe 1)

Modellieren Sie das Netzwerk *unabhängig* von den technischen Aspekten (REST, XML, HTTP etc.) als “normale” Methoden in einer Klasse. Diese technischen Details werden während der Implementierungsphase von der LV-Leitung bereitgestellt und lassen sich in diesen Entwurf gut integrieren. Jetzt während der Modellierung *abstrahieren* wir uns davon. Sie müssen diese daher hier *nicht* berücksichtigen.

Im Folgenden finden Sie einen Überblick über Client und Server um einen Gesamtüberblick zu erhalten. Welche Informationen davon für die Übung zu berücksichtigen sind entnehmen Sie bitte der jeweiligen Angabe.

Wir empfehlen *jeden relevanten Use Case* der Netzwerkkommunikation mit *mindestens einer Methode* abzudecken. Es kann beim Client (beim Server gilt allgemein immer ein UseCase gleich einer Methode) auch sinnvoll sein, manche Use Cases in mehrere Methoden aufzuteilen. Achten Sie darauf: **1)** keinen relevanten Use Case zu übersehen (Registrieren, Karte austauschen, etc.) und **2)** immer sicherzustellen, dass die Methoden auf die für den Use Case notwendigen Daten zugreifen können (z.B. mit Parametern). Welche Use Cases (ein Use Case = ein Kapitel z.B. 5, 6, 7 usw. im Netzwerkprotokoll) und Daten benötigt werden besprechen wir in den Tutorials, findet sich aber auch in der Dokumentation des Netzwerkprotokolls.

Beispiel anhand des Use Case "Karte austauschen": In der Netzwerkklasse des **Clients** könnte man für diesen Use Case eine `sendMap` Methode vorsehen. *Wichtig* sind immer auch die *Daten* - für diesen Fall ist das mindestens die vom Client erzeugte Kartenhälfte! Um die Kartenhälfte irgendwie (das "wie" sind die hier abstrahierten technischen Details) dem Server zukommen lassen zu können muss `sendMap` auf diese zugreifen können. Eine klassische Lösung dafür wäre ein *Methodenparameter* welcher die zu versendenden Informationen beinhaltet. Daher beispielsweise `public void sendMap(ClientMap map)`. Hier wäre der Parameter `ClientMap map` eine von Ihnen erstellte Klasse welche von Ihren Client-Kartengenerierungsalgorithmen erzeugt wird.
Zugriff auf die Antwort des Servers (den `ResponseEnvelope`) erhalten Sie direkt in der sendenden Methode. Daher Sie können davon ausgehen innerhalb von `sendMap` Zugriff auf ein *Variable* vom Typen `ResponseEnvelope` mit der Antwort des Servers auf die gerade von Ihnen zugesandten Kartenhälfte zu haben. Sie können mit dieser *Variable* alles (speichern, Teile auslesen, als Ganzes weiterleiten, ignorieren usw.) machen was Ihnen zu Variablen bzw. Datenklassen bekannt ist.

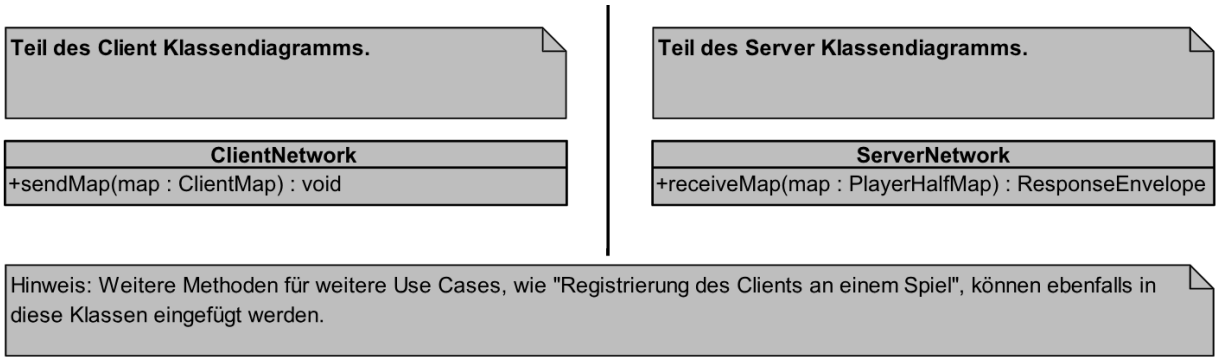
Wenn der Client etwas *versendet*, wird am **Server** das *empfangende Gegenstück* benötigt. Auch hier können Sie wieder mit *einer Klasse* und darin genau *einer Methode pro Use Case* arbeiten. Für den Use Case "Karte austauschen" ergäbe das beispielsweise eine Netzwerkklasse am Server mit z.B. einer Methode `receiveMap`. Auch hier sind wieder die *Daten wichtig*.
So muss der Server *Kartendaten empfangen* (laut Netzwerkprotokoll mit einer `PlayerHalfMap` Typen → empfangende Daten werden zu **Methodenparametern**) und *eine Antwort* an den Client *zurücksenden* (laut Netzwerkprotokoll eine `ResponseEnvelope` Klasse → Antworten an den Client werden zu Rückgabetypen). Dies könnte wie folgt modelliert/implementiert werden: `public ResponseEnvelope receiveMap(PlayerHalfMap map)`. Auch hier können die *technischen Details* wie REST, XML, wie Objekte von/zu XML umgewandelt werden etc. *ignoriert* werden. Die **Bezeichnungen (Namen) der Datenklassen** für den *Netzwerkteil des*

Servers (Parameter und Rückgabetypen der Methoden am Server) finden Sie hier im **Netzwerkprotokoll**, übernehmen Sie diese.

Zur **Verdeutlichung** hier noch einmal als **Klassendiagramm** (siehe oben). Natürlich können Sie von dem gegebenen Beispiel *abweichen* (andere Namen, andere Datenklassen, mehr Methoden, Felder, Konstruktoren etc.). *Beachten* Sie dabei immer die zu *Beginn genannten Dinge*: 1) *alle relevanten Use Cases* abdecken 2) mindestens *eine Methode pro Use Case* 3) die *notwendigen Daten* sind berücksichtigt und 4) beim *Server* sind immer die Bezeichnungen aus dem *Netzwerkprotokoll* für Parameter/Rückgabetypen zu verwenden.

Start der Ausführung: Der **Client** beginnt immer direkt in der `main` Methode und führt danach sequenziell seine Codezeilen ausgehend von dieser aus (wie aus PR1/PR2 bekannt). Der für Ihre Überlegung relevante Beginn der Ausführung beim **Server** ist immer der Empfang der jeweiligen Netzwerknachricht (eine `main` ist vorhanden hat aber darauf wenig Einfluss). Wird beispielsweise eine Karte empfangen springt die Ausführung bzw. beginnt die Auführung in der ersten Zeile der Methode `receiveMap` und arbeitet sich von dort ausgehend sequenziell durch den Code.
Genaue technische Hintergründe erhalten Sie bei Interesse direkt von Spring, sind für die Übung aber nicht notwendig. Durch die Aufrufe dieser Server-Netzwerk-Methoden *erstelle/geändete Zustände* von Objekten die z.B. als *Felder* in der Server-Netzwerkklasse enthalten sind bleiben dabei erhalten. Hierdurch können z.B. Spielinformationen über mehrere verschiedene Server-Netzwerk-Methodenaufrufe hinweg zur Verfügung stehen und Client-Aktionen aufeinander aufbauen.

Etwas fehlt: Sehen Sie sich den Server und dessen `receiveMap` noch einmal an und vergleichen es mit dem Abschnitt über die Kartenhälftenübertragung. Hier wurde ein wichtiges Datum vergessen, können Sie feststellen welches und dieses hinzufügen? Im ersten Tutorial haben wir das ebenfalls erwähnt. Falls Sie unsicher sind was fehlt Fragen Sie in der Übung!

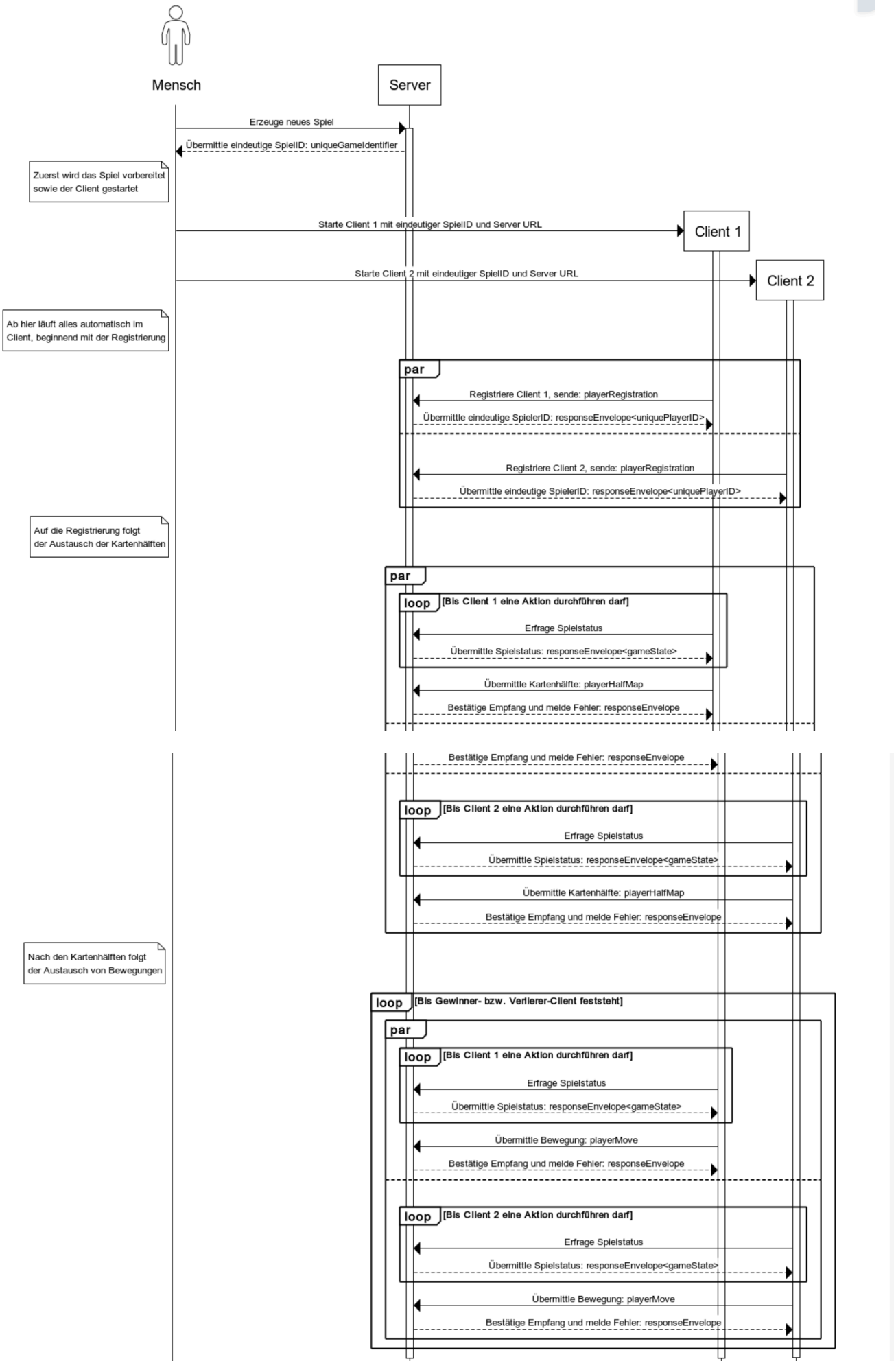


3. Beispielhafter Ablauf eines Nachrichtenaustausches

Zum Einstieg ist unterhalb ein beispielhafter Nachrichtenaustausch, basierend auf dem hier beschriebenen Protokoll, abgebildet. Die genauen Details zu jeder Nachricht finden Sie auf den folgenden Seiten. Hier in der Netzwerkprotokolldokumentation konzentrieren wir uns **nur** auf den ***Nachrichtenaustausch*** zwischen Clients und Server. Ausgaben an den Menschen, internes Verhalten des Clients oder Servers etc. lassen wir hier entsprechend weg.

Ein wichtigste Detail, auf das Sie achten sollten, ist, dass das Spiel rundenbasiert abläuft. Entsprechend müssen Sie wiederholt mit den Statusnachrichten ermitteln ob Ihr Client überhaupt derzeit eine Aktion absetzen darf. Erst sobald dies der Fall ist darf die Aktion durchgeführt werden. Wichtig für: die Übertragung der Kartenhälften **und** der Bewegungen.
Dieses Sequenzdiagramm fokussiert sich auf den *Nachrichtenaustausch* zwischen den Clients und dem Server und verdeutlicht wann/wie diese Systeme miteinander kommunizieren. Für die in **Teilaufgabe 1** verlangten Sequenzdiagramme nicht nur dieses Diagramm "abmalen". Fokussieren Sie sich auf die **Interaktion** *zwischen* Ihren **Klassen** während der gegebenen Szenarien. Ein Beispiel dafür (das zeigt wie solche Sequenzdiagramme aussehen) ist in "Teilaufgabe 1, Aufgabe 3" verlinkt.

Typischer Nachrichtenaustausch in einem vollständigen fehlerfreien Spiel



4. Allgemeine Informationen über den Nachrichtenaustausch

Im Folgenden werden Anfragen des Clients an den Server und erwartete Antworten vom Server zum Client angeführt. Hierbei ist wesentlich, dass die meisten Antworten von einem allgemeinen `responseEnvelope` gekapselt werden. Dieser nimmt die eigentlichen Daten, die übertragen werden sollen auf und bietet jedoch darüber hinaus auch die Möglichkeit mitzuteilen ob eine Anfrage vom Client korrekt oder inkorrekt war.

Darüber hinaus wird im Falle einer inkorrekten Anfrage typischerweise eine Fehlermeldung zusätzlich retourniert. Einen weiteren Spezialfall stellt die Initialisierung eines neuen Spieles dar. Hierzu wird auf einen vorgebenden Endpoint mittels einer HTTP GET Operation zugegriffen und der Server antwortet entsprechend mit einer **SpielID**. Diese schlägt niemals fehl.

5. Erstellung eines neuen Spiels

Bevor zwei Clients gegeneinander antreten können muss ein neues Spiel am Server erstellt werden. Typischerweise wird dies nicht von den KIs bzw. Clients (technisch möglich, nicht empfohlen) sondern von einem Menschen durchgeführt. Hierzu ist es notwendig auf den unten beschriebenen Endpoint zuzugreifen, indem eine HTTP GET Anfrage an diesen gesandt wird. *Ein normaler Webbrowser ist alles was dafür benötigt wird*.

Client-Header: Da die Antwort mit Daten im XML Format definiert wird, sollte die Anfrage einen entsprechenden `"accept"` HTTP Header beinhalten, daher `"accept = application/xml"`. Dies ist nur zu beachten wenn mit dem Client ein neues Spiel erzeugt wird (nicht empfohlen).

5. Erstellung eines neuen Spiels

5.1. Anfrage des Clients oder eines Menschen

Senden Sie eine HTTP GET Operation (alternativ auch mit HTTP POST möglich) an folgenden Endpoint **`http(s)://<domain>:<port>/games`**
Der Server antwortet mit einer XML Nachricht, die eine

eindeutige `SpielID` beinhaltet. Details hierzu sind unterhalb zu finden. Da diese `SpielID` für jedes Spiel eindeutig ist kann diese im Folgenden verwendet werden um Nachrichten einem bestimmten Spiel zuzuordnen und dadurch kann vom Server unterschieden werden für welches Spiel eine eingehende Nachricht gedacht ist. Hierdurch können parallel mehrere Spiele von verschiedenen Spielern durchgeführt werden.

*Um die Ressourcen des Servers zu schonen gilt die Einschränkung, dass maximal 99 Spiele parallel ausgeführt werden dürfen. Sobald die Zahl der gleichzeitig laufenden Spiele die 99 Spiele übersteigt werden ältere (bzw. das älteste) Spiele automatisch entfernt und mit den neueren danach erstellten Spielen ersetzt. Weiters wird jedes Spiel 10 Minuten nach der Spielerzeugung automatisch entfernt, sodass keine weiteren Nachrichten an dieses gesandt werden können. Der Client muss **davor** das Spiel abschließen und sich selbst beenden.*

Debug Modus aktivieren: Ein Debug Modus kann für das zu erstellende Spiel aktiviert werden indem der Parameter `enableDebugMode` mit dem Wert `"true"` an die URL angehängt wird. Entsprechend ändert sich die Endpoint URL auf **`http(s)://<domain>:<port>/games?enableDebugMode=true`**
Sobald dies passiert wird die Business Rule, dass ein Client nicht länger als 5 Sekunden Zeit hat, um eine Aktion zu senden aufgehoben. Alle anderen Business Rules werden weiterhin überprüft.

Dummy Partner aktivieren: Ein Dummy-KI-Gegner kann für das zu erstellende Spiel aktiviert werden indem der Parameter `enableDummyCompetition` mit dem Wert `"true"` an die URL angehängt wird. Entsprechend ändert sich die Endpoint URL auf **`http(s)://<domain>:<port>/games?enableDummyCompetition=true`**
Sobald dies passiert wird automatisch, sobald sich Ihr Client registriert ein zweiter Client von der LV Leitung zugeschaltet. Dieser bewegt sich zwischen zufällig ausgewählten Feldern auf der Karte und wird in der Regel deshalb eher nur per Zufall ein Spiel gewinnen. Auch die Kartenhälfte welche dieser Client erzeugt ist komplett zufällig.

Beide Optionen (Dummy & Debug) können kombiniert werden, verbinden Sie diese hierzu mit **`&`**.

5. Erstellung eines neuen Spiels

5.2. Antwort des Servers

Diese Antwort wird als Reaktion auf die oberhalb beschriebene Anfrage nach der Erstellung eines neuen Spiels gesendet. Die in XML definierte Nachricht beinhaltet als Hauptbestandteil ein Element namens `uniqueGameID` die eine eindeutige **`SpielID`** beinhaltet (diese ID muss aus exakt 5 Zeichen bestehen). Diese **`SpielID`** muss in den folgenden Anfragen als Teil der Endpoint-Adresse verwendet werden, sodass dem Server bekannt gemacht wird für welches Spiel eine Nachricht gedacht ist. Das Erstellen eines neuen Spiels ist immer möglich und darf niemals fehlschlagen.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="uniqueGameIdentifier">
    <xs:complexType>
      <xs:sequence>

<xs:element name="uniqueGameID" maxOccurs="1" minOccurs="1"
>
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="5" />
              <xs:maxLength value="5" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Beispiel der Nachricht: Die eindeutige `SpielID` eines Spiels wäre im folgenden Beispiel "Ss8Zc".

```
<?xml version="1.0" encoding="UTF-8"?>
<uniqueGameIdentifier>
  <uniqueGameID>Ss8Zc</uniqueGameID>
</uniqueGameIdentifier>
```

6. Registrierung eines Clients

Sobald ein Spiel erstellt wurde und beiden Clients die eindeutige `SpielID` (siehe oben) bekannt ist können sich die Clients für das neue Spiel registrieren. Wichtig dabei ist, dass kein Mechanismus vorgesehen ist um beiden Clients direkt vom Server ausgehend die gleiche `SpielID` mitzuteilen. Daher es ist vorgesehen, dass die `SpielID` einmalig durch einen Menschen erzeugt wird (z.B. indem über einem Webbrowser auf den Endpoint zu Erstellung eines neuen Spiels zugegriffen wird) und dann beispielsweise die Clients mit der `SpielID` als Startparameter gestartet werden. Hierdurch können beide Clients ungefähr zur gleichen Zeit beginnen mit dem Server zu interagieren. Informationen zur Verarbeitung der Startparameter finden Sie auf Moodle in der Angabe zu Teilaufgabe 2.

Es wird erwartet, dass die Registrierung eines jeden Clients zügig nach Spielstart stattfindet. Dies sollte jedenfalls möglich sein, da nur wenige Daten übertragen werden müssen sowie keine, vergleichsweise, aufwändigen Berechnungen/Algorithmen notwendig sind. Außerdem erlaubt dieses Vorgehen fehlerhaft implementierte Clients welche, nicht einmal eine Registrierung schaffen, frühzeitig zu erkennen und deren Spiele abzubrechen. Die beschränkten Ressourcen der zur Verfügung gestellten Systeme können so besser auf teilnehmende Studierende aufgeteilt werden.

6. Registrierung eines Clients

6.1. Anfrage des Clients

Der Client sendet einen HTTP POST Request mit einer XML Nachricht im Body an folgenden Endpunkt: **`http(s)://<domain>:<port>/games/<SpielID>/players`**

Hierbei ist der Teil ‹ `SpielID` › mit der während der Spielerzeugung erhaltenen eindeutigen `SpielID` zu ersetzen. Als Body der Nachricht werden Informationen zum Entwickler des Clients übertragen. Dies wären der Vorname (`studentFirstName`), der Nachname (`studentLastName`) und Ihr u:account username

(`studentUAccount`). Letzteres muss identisch sein zum u:account Benutzernamen welchen Sie beim Login auf Moodle nützen.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="playerRegistration">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="studentFirstName" maxOccurs="1" min
Occurs="1">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:minLength value="1" />
                     <xs:maxLength value="50" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
            <xs:element name="studentLastName" maxOccurs="1" min
Occurs="1">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:minLength value="1" />
                     <xs:maxLength value="50" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
            <xs:element name="studentUAccount" maxOccurs="1" min
Occurs="1">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:minLength value="1" />
                     <xs:maxLength value="50" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

</**xs:schema**>

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<playerRegistration>
  <studentFirstName>Max</studentFirstName>
  <studentLastName>Muster</studentLastName>
  <studentUAccount>musterm87</studentUAccount>
</playerRegistration>
```

Validierung der Daten: Es können grundlegend beliebige Daten eingetragen werden, daher ein Abgleich mit den Daten der Studierenden findet nicht statt (Sie erhalten bei "falschen" Daten, z.B. nicht existenten Vornamen, keine Fehlermeldung). Jedoch ist die Längenbeschränkung zu beachten und die Regelung bezüglich u:account Benutzername. Deren Einhaltung ist für die Bonuspunkte für den Cliententwicklungsfortschritt, KI-Evaluierungen etc. zwingend notwendig:

Client-Fortschritt: Damit die automatische Auswertung Ihres Fortschritts unter *http://swe1.wst.univie.ac.at/* wie gedacht funktioniert müssen Sie, als `studentUAccount`, immer Ihren richtigen u:account Benutzernamen verwenden. Dieser *muss* identisch zu dem u:account Benutzernamen von Moodle bzw. ihrem Moodle Profil sein. Andernfalls können beispielsweise die Bonuspunkte für die Minideadlines und Punkte für andere Bewertungsaspekte der zweiten Teilaufgabe nicht vergeben werden.

6. Registrierung eines Clients

6.2. Antwort des Servers

Der Server antwortet auf die Registrierungsnachricht (siehe oben) mit einer Bestätigung über die Durchführung der Registrierung sowie einer entsprechenden eindeutigen `SpielerID`. Diese `SpielerID` muss bei den folgenden Anfragen mit übertragen werden damit dem Server bekannt ist von welchem Client (bzw. Spieler) diese Anfrage gestellt wurde.

Diese und die folgenden Nachrichten des Servers werden von einem `responseEnvelope` Element umschlossen.
Dieser `responseEnvelope` ist auf zwei Fälle ausgelegt, welche über das `state` Element unterschieden werden können. Beinhaltet das `state` Element den Text **Okay** konnte die Anfrage korrekt verarbeitet werden und die erwarteten Daten (wie in diesem Fall die `SpielerID`) können im `data` Element gefunden werden. Zeigt das `state` Element jedoch den Wert **Error** dann ist ein Fehler aufgetreten z.B., weil eine Spielregel verletzt worden ist. Für diesen Fall ist der Namen einer zugehörigen Fehlermeldung im Element `exceptionName` zu finden und ein Hinweistext über den aufgetretenen Fehler im Element `exceptionMessage` (z.B. mit Details und Tipps dazu warum die zugesandte Registrierungsanfrage nicht korrekt war). Dieses Verhalten wird auch bei allen kommenden Nachrichten angewendet und hilft Ihnen dabei Fehler im Client zu identifizieren.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="state"
type="statevalues" minOccurs="1" maxOccurs="1" />
        <xs:element name="data" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="uniquePlayerIdentifier">
    <xs:sequence>
      <xs:element name="uniquePlayerID"
type="xs:string" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="statevalues">
```

```
      <xs:restriction base="xs:string">
        <xs:enumeration value="Okay" />
        <xs:enumeration value="Error" />
      </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
  <exceptionName/>
  <exceptionMessage/>
  <state>Okay</state>  <data xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:type="uniquePlayerIdentifier">
  <uniquePlayerID>702ecdb6-b363-462c-a63a-5d0a39222c4e</
uniquePlayerID>
 </data>
</responseEnvelope>
```

7. Übertragung einer der beiden Kartenhälften

Im Anschluss an die Registrierung eines Spielers muss der Client eine Kartenhälfte generieren und an den Server übertragen. Hierbei wird jedes Feld individuell definiert, basierend auf dem Typ des Feldterrains sowie einer möglicherweise vorhandenen Burg. Regeln zur Beschaffenheit der Karten sind in der jeweils relevanten Angabe zu finden. Diese werden nicht mit dem XML Schema überprüft bzw. sichergestellt, sondern über die Validierung der Geschäftsregeln am Server.

7. Übertragung einer der beiden Kartenhälften

7.1. Anfrage des Clients

Der Client sendet einen HTTP POST Request mit einer XML Nachricht im Body an folgenden Endpunkt: **`http(s)://<domain>:<port>/games/<SpielID>/halfmaps`**

Hierbei ist der Teil `‹SpielID›` mit der während der Spielerzeugung erhaltenen eindeutigen `SpielID` zu ersetzen. Als Body der Nachricht werden Informationen zur Kartenhälfte verwendet. Diese bestehen aus der eindeutigen `SpielerID`, um den Server wissen zu lassen welcher Client die Daten zugesandt hat und außerdem aus einer Liste von Feldern, die über `playerHalfMapNode` Elemente abgebildet werden. Jedes `playerHalfMapNode` Element wiederum besteht aus einer X (`X` Element) und einer Y Koordinate (`Y` Element), dem Terrain des Feldes und einem booleschen Flag, welches repräsentiert ob auf diesem Feld eine Burg vorhanden ist oder nicht (`fortPresent` Element).

Informationen zur Ordnung/Sortierung der Inhalte: *Wenn Sie Ihre mit den hier besprochenen Nachrichten an den Server übertragene Karte über die Statusnachrichten wieder abfragen wird die Reihenfolge der Felder nicht mehr der Reihenfolge entsprechen, die Sie bei der Übertragung verwendet haben. Nützen Sie daher die X/Y Koordinaten der Felder, um auf die passenden Felder zu identifizieren und auf diese zuzugreifen.*

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="playerHalfMap">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="uniquePlayerID"
type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:element name="playerHalfMapNodes" minOccurs="1" maxOccurs="1">
          <xs:complexType>
```

```
            <xs:sequence>

<xs:element name="playerHalfMapNode" minOccurs="50" maxOccurs="50">
                <xs:complexType>
                  <xs:sequence>

<xs:element name="X" minOccurs="1" maxOccurs="1">
                        <xs:simpleType>
                          <xs:restriction base="xs:integer">
                            <xs:minInclusive value="0" />
                            <xs:maxInclusive value="9" />
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:element>

<xs:element name="Y" minOccurs="1" maxOccurs="1">
                        <xs:simpleType>
                          <xs:restriction base="xs:integer">
                            <xs:minInclusive value="0" />
                            <xs:maxInclusive value="4" />
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:element>
                      <xs:element name="fortPresent"
type="xs:boolean" minOccurs="1" maxOccurs="1" />
                      <xs:element name="terrain"
type="terraintype" minOccurs="1" maxOccurs="1" />
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:simpleType name="terraintype">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Grass" />
```

```
        <xs:enumeration value="Water" />
        <xs:enumeration value="Mountain" />
      </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<playerHalfMap>
  <uniquePlayerID>3560ca14-5bfa-48f3-949b-da0029caf3c8</
uniquePlayerID>
  <playerHalfMapNodes>
    <playerHalfMapNode>
      <X>2</X>
      <Y>1</Y>
      <fortPresent>false</fortPresent>
      <terrain>Water</terrain>
    </playerHalfMapNode>
    <playerHalfMapNode>
      <X>4</X>
      <Y>3</Y>
      <fortPresent>true</fortPresent>
      <terrain>Grass</terrain>
    </playerHalfMapNode>
    <playerHalfMapNode>
      <X>4</X>
      <Y>2</Y>
      <fortPresent>false</fortPresent>
      <terrain>Grass</terrain>
    </playerHalfMapNode>
  </playerHalfMapNodes>
</playerHalfMap>
```

Achtung: Um das Beispiel kurz zu halten wurden nicht alle `playerHalfMapNode` Elemente angeführt. Daher eigentlich wären pro Kartenhälfte mehr `playerHalfMapNode` Elemente notwendig. Dies wird vom zugehörigen Schema auch so überprüft bzw. verlangt.

7. Übertragung einer der beiden Kartenhälften

7.2. Antwort des Servers

Der Server antwortet generisch mit einem `responseEnvelope`. Da aber hier keine Daten anfallen ist das `data` Element in diesem Fall nicht enthalten. Abgesehen davon ist das Verhalten wie in der zuvor beschriebenen Nachricht. Daher im Fehlerfall sind `exceptionName` und `exceptionMessage` mit Details zum Fehler befüllt (z.B. mit Details und Tipps zur zugesandten den Spielregeln widersprechenden Kartenhälfte) sowie das `state` Element mit dem Wert **Error** definiert. Sollte kein Fehler vorliegen ist `state` mit dem Wert **Okay** definiert.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="state"
type="statevalues" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="statevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Okay" />
      <xs:enumeration value="Error" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<responseEnvelope>
  <exceptionName />
  <exceptionMessage />
  <state>Okay</state>
</responseEnvelope>
```

7. Übertragung einer der beiden Kartenhälften

7.3. Tipps zum Verständnis der Kartenkoordinaten

Allgemein gilt, dass im verwendeten Koordinatensystem links oben der Koordinatenursprung liegt (X=0;Y=0). Rechts vom Ursprung beginnen die X Werte anzusteigen von 0 bis 9 (Kartenhälften wurden untereinander platziert) bzw. 19 (Kartenhälften wurden nebeneinander platziert). Unterhalb des Ursprungs beginnen die Y Werte anzusteigen von 0 bis 4 (Kartenhälften wurden nebeneinander platziert) bzw. 9 (Kartenhälften wurden untereinander platziert). Eine Ausnahme bilden hierbei die beiden Kartenhälften die bei Spielbeginn übertragen werden. Diese beinhalten nur X Werte von 0 bis 9 und Y Werte von 0 bis 4. Wie Kartenhälften kombiniert werden (zu einem Quadrat oder Rechteck oder an welcher Stelle eine Kartenhälfte hinterlegt wird) kann der Client nicht entscheiden oder beeinflussen. Dies Entscheidung fällt der Server zufällig.

Das folgende Beispiel stellt das Koordinatensystem für die Kartenhälfte dar, welche bei Spielbeginn von jedem Client übertragen wird. Die Koordinaten für die komplette zusammengesetzte Karte sind analog definiert. Achtung: Die beiden Kartenhälften werden zufällig zusammengesetzt, daher die erste von einem Client zugesandte Karte landet zufällig auf Position X=0;Y=0 bis X=9;Y=4, X=0;Y=5 bis X=9;Y=9 oder X=10;Y=0 bis X=19;Y=4 (daher Kartenhälften können zufällig untereinander – lange Seiten grenzen aneinander – oder nebeneinander – kurze Seiten grenzen aneinander – platziert werden). Die jeweils vom zweiten Client zugesandte Kartenhälfte füllt den noch freien Platz auf. Als Client sollte man daher erst den Spielstatus mit der vollständigen Karte erfragen bevor Wege berechnet werden, etc.

X=0;Y=0	X=1;Y=0	X=2;Y=0	X=3;Y=0	X=4;Y=0	X=5;Y=0	X=6;Y=0	X=7;Y=0	X=8;Y=0	X=9;Y=0
X=0;Y=1	X=1;Y=1	X=2;Y=1	X=3;Y=1	X=4;Y=1	X=5;Y=1	X=6;Y=1	X=7;Y=1	X=8;Y=1	X=9;Y=1
X=0;Y=2	X=1;Y=2	X=2;Y=2	X=3;Y=2	X=4;Y=2	X=5;Y=2	X=6;Y=2	X=7;Y=2	X=8;Y=2	X=9;Y=2
X=0;Y=3	X=1;Y=3	X=2;Y=3	X=3;Y=3	X=4;Y=3	X=5;Y=3	X=6;Y=3	X=7;Y=3	X=8;Y=3	X=9;Y=3
X=0;Y=4	X=1;Y=4	X=2;Y=4	X=3;Y=4	X=4;Y=4	X=5;Y=4	X=6;Y=4	X=7;Y=4	X=8;Y=4	X=9;Y=4

8. Abfrage des Spielstatus

Die Abfrage eines Spielstatus durch den Client kann erst nach der Registrierung eines Clients beginnen da hierzu die `SpielID` aber auch die `SpielerID` benötigt werden. Bei einer Abfrage über den Spielzustand werden vom Server alle für die Clients relevanten Details zum Spiel zurückgeliefert. Dies sind die vollständige oder teilweise Karte (je nachdem ob erst eine oder beide Clients ihre Kartenhälfte zugesandt haben) sowie Informationen, um die Aktionen der Clients abzustimmen (ob ein Client eine Aktion durchführen soll oder ob ein Client gewonnen bzw. verloren hat).

Da die Clients nur eine beschränkte Zeitspanne zur Verfügung haben um die nächste Spielaktion (Bewegungsoperation oder Übertragung einer Kartenhälfte) durchzuführen muss der Client in regelmäßigen Abständen abfragen, ob er die nächste Aktion berechnen und übertragen muss. Um zu verhindern, dass der Server überlastet wird, muss zwischen zwei vom gleichen Client durchgeführten Abfragen zum Spielstatus **mindestens eine Zeitspanne von 0,4 Sekunden** vergehen.

Ist ein Client zu schnell werden die zugehörigen Anfragen vom Server mit ***Verzögerung*** bearbeitet oder (sofern möglich) Clientausführungen vom Server ***abgebrochen***. Diese Verzögerung übersteigt die angeführte Zeitspanne deutlich. Eine einfache/schnelle, wenn auch nicht schöne und eigentlich zu vermeidende (für uns aber *ausreichende*), Implementierung am Client dafür ist ein `Thread.sleep` zwischen zwei Abfragen des Spielstatus.

Polling: Ist der Fachbegriff für das hier angewandte Verfahren. Dies ist die (auch technisch) für Sie einfachste Möglichkeit beide Clients zu

koordinieren. Komplexere (und damit effizientere) Verfahren lernen Sie bei Interesse in LVs wie DSE kennen.

8. Abfrage des Spielstatus

8.1. Anfrage des Clients

Der Client sendet ein HTTP GET Request an folgenden Endpunkt: **`http(s)://<domain>:<port>/games/<SpielID>/states/<SpielerID>`**

Hierbei ist der Teil `SpielID` mit der während der Spielerzeugung erhaltenen eindeutigen `SpielID` zu ersetzen und `SpielerID` mit der während der Registrierung vom Server mitgeteilten eindeutigen `SpielerID`. Falls beide IDs dem Server bekannt sind antwortet der Server mit dem Spielstatus, andernfalls wird eine entsprechende Fehlermeldung zurückgesandt.

Achtung: Da die Antwort mit Daten im XML Format definiert wird sollte die Anfrage einen entsprechenden `"accept"` HTTP Header beinhalten, daher `"accept = application/xml"`.

8. Abfrage des Spielstatus

8.2. Antwort des Servers

Der Server antwortet mit einem `responseEnvelope`, welches im Falle eines Fehlers (wie bereits bei den vorangegangenen Nachrichten) eine

entsprechende Fehlerbeschreibung beinhaltet und mit dem `state` Element verdeutlicht ob ein Fehler aufgetreten ist oder nicht. Das data Element beinhaltet die eigentlichen Nutzdaten. Daher, im Wesentlichen die Informationen über die Avatare (`players → player`) und die Karte. Für einen Player werden `uniquePlayerID` (eindeutige SpielerID), `firstName` (Vorname), `lastName` (Nachname), `uaccount` (u:account Benutzername der Universität Wien), `state` (Status des Spielers) angehängt. Vorname, Nachname und Matrikelnummer werden basierend auf den Daten die bei der Registrierung des Spielers vom Client übertragen wurden zusammengestellt.
Die `uniquePlayerID` entspricht der bei der Registrierung erstellten und zugesandten eindeutigen `SpielerID`. Letzteres trifft nur für den Datensatz zu welcher den anfragenden Spieler repräsentiert, die `SpielerIDs` der anderen Spieler entsprechen nicht deren realen `SpielerIDs`. Der `state` wiederum zeigt den Status des Clients an, dieser kann entweder sein, dass vom Server für diesen Client ein neuer Befehl erwartet wird (**MustAct**), dass kein Befehl erwartet wird (**MustWait**), dass der Client verloren hat (**Lost**) oder dass der Client gewonnen hat (**Won**).

Darüber ist die **Spielkarte im Element** `map` angeführt. Dessen Inhalt, einzelne Spielfelder, ist optional und wird erst dann vorhanden sein, wenn zumindest einer der Clients eine Kartenhälfte erfolgreich übertragen hat. Pro Kartenfeld wird angezeigt ob sich Avatare der Clients auf dem Feld befinden (`playerPositionState`), welches Terrain das Feld aufweist (`terrain`), ob sich ein Schatz auf dem Feld befindet (`treasureState`) und ob sich ein Fort auf dem Feld befindet (`fortState`). Weiters werden die Koordinaten angegeben daher X Koordinate (`X`) und Y Koordinate (`Y`).

Das Element `playerPositionState` zeigt an ob kein Avatar auf dem Feld vorhanden ist (**NoPlayerPresent**), der Avatar des Gegnerclients sich auf dem Feld befindet (**EnemyPlayerPosition**), sich der Avatar des anfragenden Clients auf dem Feld befindet (**MyPlayerPosition**), oder Avatare beider Clients (**BothPlayerPosition**). Das Element `terrain` zeigt die Feldart an als **Water**, **Grass** oder **Mountain**. Bezüglich des Schatzes (`treasureState`) kann entweder gemeldet werden, dass sich kein Schatz darauf befindet oder es nicht bekannt ist ob sich auf dem Feld

ein Schatz befindet (**NoOrUnknownTreasureState**). Alternativ wird mitgeteilt, dass der eigene Schatz sich auf dem Feld befindet (**MyTreasureIsPresent**). Sobald ein Schatz "aufgehoben" wurde, daher sich der passende Avatar auf dem Feld befunden hat, auf dem ein Schatz gemeldet wurde wird nicht mehr länger *MyTreasureIsPresent* als wahr angegeben. Bezüglich der Burg weist das Element `fortState` entweder den Wert auf **NoOrUnknownFortState** wenn nicht bekannt ist ob sich auf den Feld eine Burg befindet bzw. dort keine Burg ist oder alternativ wird angezeigt, dass die eigene Burg sich dort befindet (**MyFortPresent**) oder die gegnerische (**EnemyFortPresent**). *Eine einmal aufgedeckte Burg wird in jeder nachfolgenden Abfragen in der Karte vermerkt (gilt auch für nicht aufgehobene aber aufgedeckte Schätze).*

Relevant ist außerdem noch das Element `gameStateId`. Dieses beinhaltet eine zufällig erstellte ID, welche es ermöglicht zu erkennen ob die vom Server angelieferten Statusinformationen sich seit der letzten Abfrage durch Ihren Client geändert haben. Ist dies der Fall hat sich auch die `gameStateId` geändert.
Hintergrund zur `gameStateId`: Es kann der Fall eintreten, dass Ihr Client eine Aktion samt den zugehörigen zwei Status-Abfragen so schnell sendet, dass die Aktion Ihres Client (samt zugehörigen Aktualisierung der Statusinformationen) bis zur zweiten Status-Abfrage noch nicht (fertig) verarbeitet wurde am Server. Ist das der Fall wird die 2. Statusabfrage Daten zurückliefern welche die zugesandte Aktion noch nicht berücksichtigen.

Weiters lässt sich über das `collectedTreasure` Flag im `player` Element erkennen ob der jeweilige Client den Schatz bereits gefunden hat oder nicht. Diese Information wiederum kann verwendet werden, um zu erkennen ob der Client noch nach dem Schatz oder bereits nach der gegnerischen Burg suchen muss.

Informationen zur Ordnung/Sortierung der Inhalte: *Die Details zum Spieler und die Details zur Karte (Felder) werden vom Server ungeordnet retourniert. Verwenden Sie daher die `PlayerID` (für den Spieler) und die `X/Y` Koordinaten (für die Karte bzw. Felder), um die jeweils passenden Informationen für Sie abzufragen bzw. zu ermitteln.*

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/
XMLSchema" elementFormDefault="qualified">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="state"
type="statevalues" minOccurs="1" maxOccurs="1" />
        <xs:element name="data" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
    <xs:complexType name="gameState" >
      <xs:sequence>
        <xs:element ref="players" minOccurs="1" maxOccurs="1" />
        <xs:element ref="map" minOccurs="0" maxOccurs="1" />

<xs:element ref="gameStateId" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  <xs:element name="players">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="2" minOccurs="1" ref="player" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="player">
    <xs:complexType>
      <xs:sequence>

<xs:element ref="uniquePlayerID" minOccurs="1" maxOccurs="1" />
        <xs:element ref="firstName" minOccurs="1" maxOccurs="1" /
>
        <xs:element ref="lastName" minOccurs="1" maxOccurs="1" /
>
        <xs:element ref="uaccount" minOccurs="1" maxOccurs="1" />
```

```
            <xs:element name="state"
type="playerGameStatevalues" minOccurs="1" maxOccurs="1" />

<xs:element ref="collectedTreasure" minOccurs="1" maxOccurs="1" /
>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="uniquePlayerID" type="xs:string" />
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="lastName" type="xs:string" />
    <xs:element name="uaccount" type="xs:string" />
    <xs:element name="collectedTreasure" type="xs:boolean" />
    <xs:element name="map"  minOccurs="1" maxOccurs="1">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="mapNodes" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="mapNodes">
      <xs:complexType>
        <xs:sequence>

<xs:element minOccurs="0" maxOccurs="100" ref="mapNode" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="mapNode">
      <xs:complexType>
        <xs:sequence>

<xs:element minOccurs="1" maxOccurs="1" ref="playerPositionState
" />
            <xs:element minOccurs="1" maxOccurs="1" ref="terrain" />

<xs:element minOccurs="1" maxOccurs="1" ref="treasureState" />
            <xs:element minOccurs="1" maxOccurs="1" ref="fortState" />
            <xs:element minOccurs="1" maxOccurs="1" ref="X" />
            <xs:element minOccurs="1" maxOccurs="1" ref="Y" />
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="playerPositionState" type="playerStatevalues" />
  <xs:element name="terrain" type="terrainStatevalues" />
  <xs:element name="treasureState" type="treasureStatevalues" />
  <xs:element name="fortState" type="fortStatevalues" />
  <xs:element name="X">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0" />
        <xs:maxInclusive value="19" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="Y">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0" />
        <xs:maxInclusive value="9" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="gameStateId" type="xs:string" />
  <xs:simpleType name="statevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Okay" />
      <xs:enumeration value="Error" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="playerStatevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="NoPlayerPresent" />
      <xs:enumeration value="EnemyPlayerPosition" />
      <xs:enumeration value="MyPlayerPosition" />
      <xs:enumeration value="BothPlayerPosition" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="terrainStatevalues">  </xs:simpleType>
    <xs:restriction base="xs:string">
```

```
        <xs:enumeration value="Water" />
        <xs:enumeration value="Grass" />
        <xs:enumeration value="Mountain" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="playerGameStatevalues">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Won" />
        <xs:enumeration value="Lost" />
        <xs:enumeration value="MustAct" />
        <xs:enumeration value="MustWait" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="treasureStatevalues">
      <xs:restriction base="xs:string">
        <xs:enumeration value="NoOrUnknownTreasureState" />
        <xs:enumeration value="MyTreasureIsPresent" />
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="fortStatevalues">
      <xs:restriction base="xs:string">
        <xs:enumeration value="NoOrUnknownFortState" />
        <xs:enumeration value="MyFortPresent" />
        <xs:enumeration value="EnemyFortPresent" />
      </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
  <exceptionName />
  <exceptionMessage />
  <state>Okay</state>
  <data xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="gameState">
    <players>
      <player>
        <uniquePlayerID>6c4362b5-5636-41b8-aaae-6daba4aea91b</
uniquePlayerID>
```

```
        <firstName>Max</firstName>
        <lastName>Mustermann</lastName>
        <uaccount>musterm55</uaccount>
        <state>MustAct</state>
        <collectedTreasure>false</collectedTreasure>
      </player>
      <player>
        <uniquePlayerID>2e31d1c8-e91d-4f55-88ad-ffcd1dae48f0</
uniquePlayerID>
        <firstName>Maxin</firstName>
        <lastName>Musterfrau</lastName>
        <uaccount>mustefm74</uaccount>
        <state>MustWait</state>
        <collectedTreasure>false</collectedTreasure>
      </player>
    </players>
    <map>
      <mapNodes>
        <mapNode>
          <playerPositionState>NoPlayerPresent</
playerPositionState>
          <terrain>Water</terrain>
          <treasureState>NoOrUnknownTreasureState</
treasureState>
          <fortState>NoOrUnknownFortState</fortState>
          <X>2</X>
          <Y>1</Y>
        </mapNode>
        <mapNode>
          <playerPositionState>NoPlayerPresent</
playerPositionState>
          <terrain>Grass</terrain>
          <treasureState>NoOrUnknownTreasureState</
treasureState>
          <fortState>NoOrUnknownFortState</fortState>
          <X>6</X>
          <Y>4</Y>
        </mapNode>
      </mapNodes>
    </map>
```

```
    <gameStateId>430bdf29-5e61-4950-8408-ac40e9fb1176</
gameStateId>
  </data>
</responseEnvelope>
```

Beispiel gekürzt: *Um das Beispiel kurz zu halten wurden nicht alle `mapNode` Elemente angeführt. Daher eigentlich wären pro Karte zumindest die Hälfte bis alle möglichen mapNode Elemente notwendig. Dies wird vom zugehörigen Schema auch so überprüft bzw. verlangt. Wenn noch keine Kartenhälften erfolgreich übertragen ist `map` zwar vorhanden, aber leer. Beinhaltet also keine `mapNode` Elemente.*

9. Übertragung einer Bewegung

Sobald ein Client an der Reihe ist (daher im Spielstatus angezeigt wird, dass die nächste Aktion gesendet werden muss) kann der Client eine Spielbewegung übertragen. Dies ist allerdings erst möglich sobald die Kartenhälften beider Clients bereits übertragen worden sind. Die Übertragung der Bewegungen erfolgt immer schrittweise, daher benötigt z.B. die Bewegung zwischen zwei Grasfeldern 2 Bewegungen in Richtung (ein “Bewegungsnachricht“ eines Clients überträgt immer nur eine einmalige Bewegung in eine Bewegungsrichtung) des entsprechenden Feldes um das alte Grasfeld zu verlassen und um das neue zu betreten. *Weitere Details und Beispiele hierzu sind in der Spielidee zu finden.*

9. Übertragung einer Bewegung

9.1. Anfrage des Clients

Der Client sendet einen HTTP POST Request an folgenden Endpunkt: **`http(s)://<domain>:<port>/games/<SpielID>/moves`**

Die Anfrage beinhaltet im Body XML Daten und wird auch mit XML Daten beantwortet. Daher sollte die Anfrage einen entsprechenden `"accept"` HTTP Header beinhalten, daher `"accept = application/xml"`. Die Daten, welche im Body übertragen werden bestehen aus der **SpielerID,** welche bei der Registrierung des Spielers dem Client mitgeteilt worden ist (`uniquePlayerID`) sowie dem Bewegungsbefehl (`move`). Der Bewegungsbefehl beinhaltet die Richtung in welche sich der Avatar, welcher durch den Client gesteuert wird, bewegen soll. Daher der Avatar kann sich nach oben (**Up**), links (**Left**), rechts (**Right**), sowie unten (**Down**) bewegen.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/
XMLSchema" elementFormDefault="qualified">
  <xs:element name="playerMove">
    <xs:complexType>
      <xs:sequence>

<xs:element ref="uniquePlayerID" minOccurs="1" maxOccurs="1" />
          <xs:element ref="move" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="uniquePlayerID" type="xs:string" />
  <xs:element name="move" type="moveValues" />
  <xs:simpleType name="moveValues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Up" />
      <xs:enumeration value="Down" />
      <xs:enumeration value="Left" />
      <xs:enumeration value="Right" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<playerMove>
```

```
  <uniquePlayerID>1c44419e-304a-4caa-8901-b7d8c62898b5</
uniquePlayerID>
  <move>Right</move>
</playerMove>
```

9. Übertragung einer Bewegung

9.2. Antwort des Servers

Der Server antwortet generisch mit einem `responseEnvelope`. Da aber hier keine Daten anfallen ist das `data` Element in diesem Fall nicht enthalten. Abgesehen davon ist das Verhalten wie in der zuvor beschriebenen Nachricht. Daher im Problemfall sind `exceptionName` und `exceptionMessage` mit Details zum Problem befüllt (z.B. mit Details und Tipps zur zugesandten den Spielregeln widersprechenden Bewegung) sowie das `state` Element mit dem Text **Error** gefüllt. Sollte kein Fehler vorliegen ist `state` mit dem Wert **Okay** befüllt.

XML Schema der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage"
type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="state"
type="statevalues" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="statevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Okay" />
      <xs:enumeration value="Error" />
```

```
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Beispiel der Nachricht

```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
   <exceptionName />
   <exceptionMessage />
   <state>Okay</state>
</responseEnvelope>
```