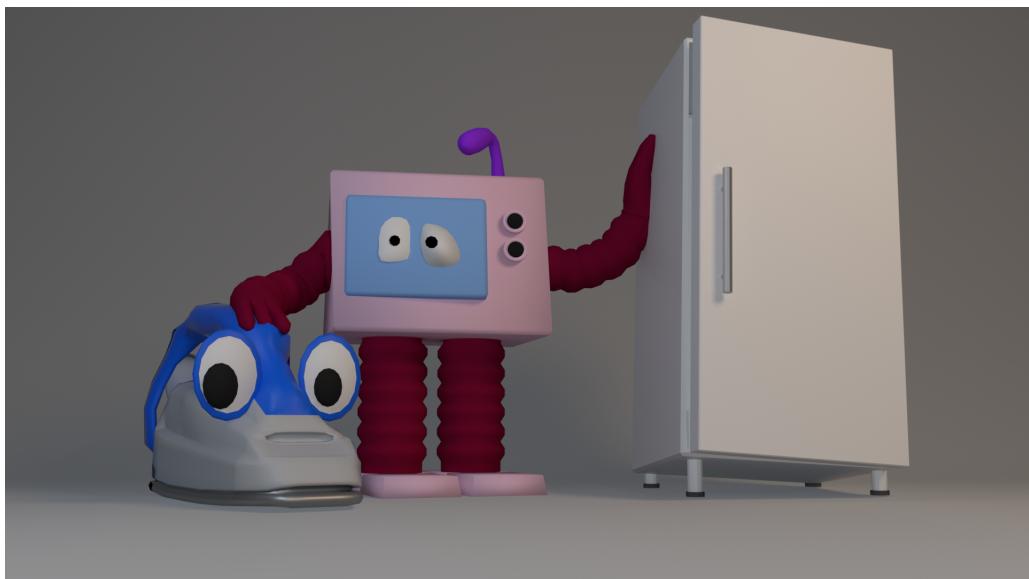


UNIVERSITÄT ULM

Robo Rally

Entwurfsdokument von 0x7DA8



von

Marie Lohbeck
Misam Huynh

Pascal Schiessle
Pascal Weber

Ulrike Kulzer
Nathan Maier

Stand: 26. Juni 2017

Inhaltsverzeichnis

1	Überblick	1
2	Architekturentwurf	2
2.1	Komponentendiagramm	2
2.2	Zugeordnete Anforderungen	4
3	Feinentwurf	5
3.1	Allgemeines	5
3.2	Server	9
3.2.1	Beschreibung der Methoden	9
3.2.2	Klassendiagramme	12
3.2.3	Kommunikationsdiagramme	14
3.3	Util	17
3.3.1	Beschreibung der Methoden	17
3.3.2	Klassendiagramme	26
3.3.3	Kommunikationsdiagramm	34
3.4	Beobachter	36
3.4.1	Beschreibung der Methoden	36
3.4.2	Klassendiagramme	47
3.4.3	Kommunikationsdiagramm	53
3.5	Spieler	55
3.5.1	Beschreibung der Methoden	55
3.5.2	Kommunikationsdiagramm	56
3.5.3	Klassendiagramm	56
3.6	KI	58
3.6.1	Beschreibung der Methoden	58
3.6.2	Kommunikationsdiagramm	58
3.6.3	Klassendiagramm	58
4	Anhang	60
	Glossar	60

1 Überblick

In der Implementierungsanleitung werden alle Entscheidungen und Ereignisse, die die Implementierung betreffen und von unserem Team festgelegt werden, dokumentiert. Ein erster Beschluss, den wir gefasst haben, war die Einigung auf die Verwendung der JMonkey-Engine.

2 Architekturentwurf

2.1 Komponentendiagramm

RoboRally ist ein verteiltes Roboterrennen. Das Spiel besteht aus insgesamt fünf verschiedenen Anwendungen. Auf der einen Seite sind die Anwendungen für *Spieler* (als Mensch), *KI* und *Beobachter* zu nennen. Ihnen gegenüber steht der *Server*, der das Spiel verwaltet und für die Kommunikation zwischen allen anderen Anwendungen zuständig ist. Als fünfte Anwendung kommt der *Leveleditor* hinzu, der allerdings nicht aktiv am Spiel teilnimmt, sondern lediglich Levelentwürfe im richtigen Format zur Verfügung stellt. Aufgrund dieser Aufteilung wird das Komponentendiagramm in der selben Weise gegliedert.

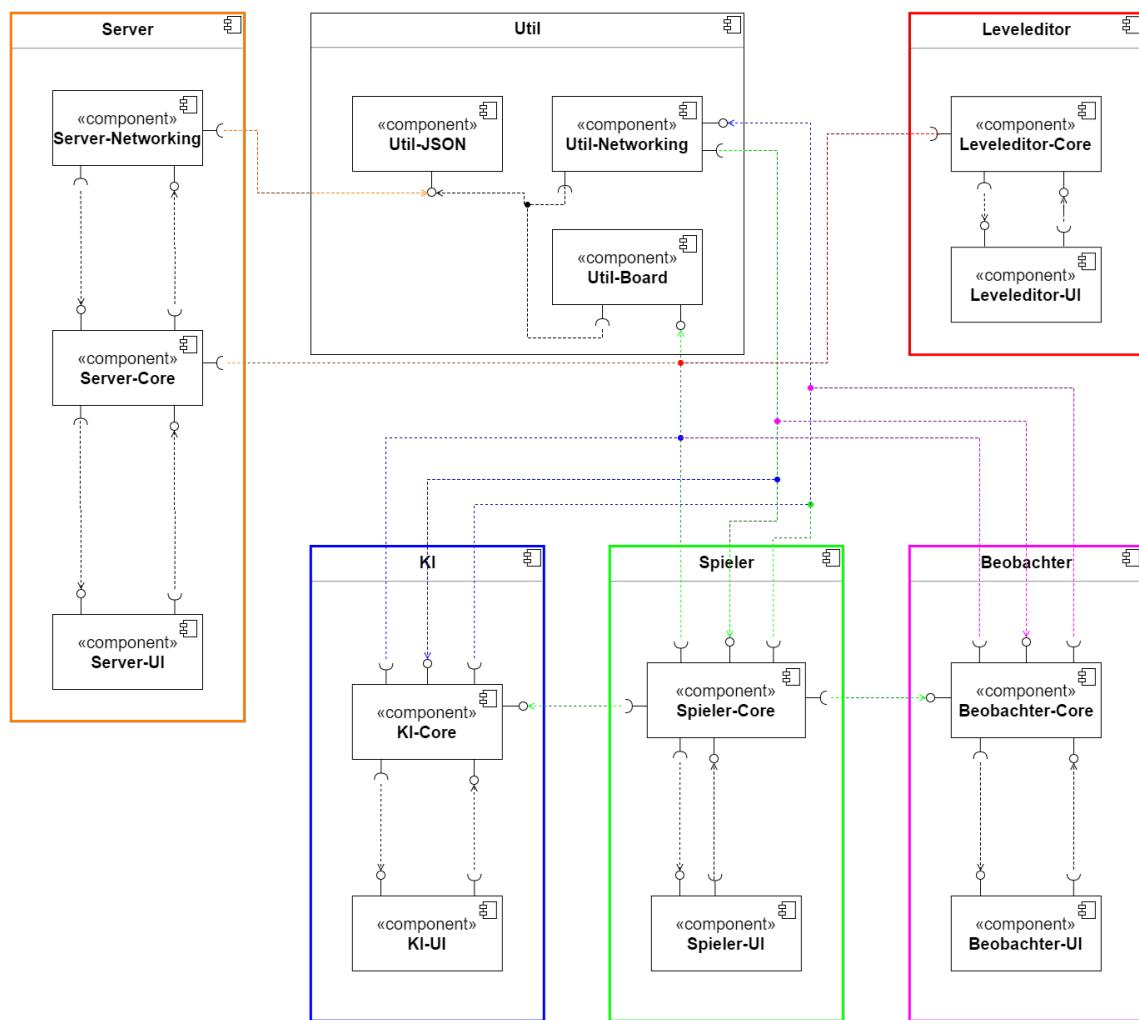


Abbildung 2.1: Komponentendiagramm

Jede Anwendung erhält sinngemäß eine eigene Komponente, was auch der im Lastenheft vorgegebenen groben Einteilung entspricht. Zusätzlich gibt es eine Hilfskomponente *Util*, in welche Funktionalität ausgelagert wird, die von mehreren anderen Komponenten verwendet werden und somit zusammengefasst wird, um Redundanz zu vermeiden.

Zum Diagramm

Die fünf Hauptkomponenten (*Server*, *KI*, *Spieler*, *Beobachter*, *Leveleditor*) sind aufgrund der Übersichtlichkeit in verschiedenen Farben hervorgehoben. Auch bei den Schnittstellenverbindungen wird durch eine entsprechende Farbe angedeutet, welche Komponente sie betrifft.

In diesem Diagramm sind einige Kreuzungen zwischen einzelnen Linien durch Punkte hervorgehoben. Die Punkte markieren, dass die entsprechenden Verbindungen hier zusammengeführt werden. Dies tritt auf, falls eine bestimmte Schnittstelle von mehreren Komponenten genutzt wird. Um zu erkennen, zwischen welchen beiden Komponenten jeweils tatsächlich eine Verbindung besteht, muss auf die Symbole und Pfeile an den Verbindungsenden geachtet werden, denn eine Verbindung kann immer nur von einer bereitgestellten Schnittstelle zu einer entgegennehmenden Komponente führen. Kreuzungen von Linien, die nicht explizit mit Punkt markiert sind, haben keine Verbindung.

Hauptkomponenten

Jede der fünf Hauptkomponenten untergliedert sich jeweils logisch in drei weitere Komponenten, die für die Benutzeroberfläche, die (Haupt-)Logik und die Kommunikation über das Netzwerk mit anderen Komponenten zuständig sind. Sie sind benannt mit *UI*, *Core* und *Networking*. Beim *Leveleditor* entfällt die *Networking*-Komponente, da er nicht mit dem Server kommuniziert. Des Weiteren sind die *Networking*-Komponenten der *KI*, des *Spieler* und des *Beobachters* (also der Clients) so ähnlich, dass sie in einer gemeinsamen Unterkomponente von *Util* zusammengefasst werden.

Der *Core* jeder Hauptkomponente ist jeweils in beide Richtungen mit der zugehörigen *UI*- und *Networking*-Komponente verbunden, stellt ihnen also eine Schnittstelle bereit und nutzt gleichzeitig eine von ihnen. Schließlich müssen sowohl bei der Kommunikation mit dem Benutzer der Anwendung, als auch bei der Kommunikation mit dem Netzwerk, in beide Richtungen Informationen ausgetauscht werden.

Hilfskomponente

In der Komponente *Util* finden sich noch zwei weitere Unterkomponenten *Util-JSON* und *Util-Board*. JSON (JavaScript Object Notation) ist das Format, in welchem bei RoboRally über das Netzwerk kommuniziert wird. Die Komponente *Util-JSON* fasst also Klassen zusammen, die dazu dienen, verschiedene Informationen ins JSON-Format und wieder zurückzuverwandeln. Sie stellt sowohl der *Networking*-Komponente des Servers, als auch der *Networking*-Komponente der Clients eine Schnittstelle bereit und gibt den beiden *Networking*-Komponenten damit die Möglichkeit, das JSON-Format zu verwenden.

Die Komponente *Util-Board* dient dazu, das Spielbrett mit den darauf positionierten Mauern, Checkpoints und Robotern zu verwalten. Der *Leveleditor* kann Spielbretter mit Hilfe dieser Komponente erstellen. Der *Server* benutzt *Board*, um Spielbretter zu erzeugen und zu initialisieren. *Spieler*, *KI* und *Beobachter* benutzen die Komponente um Informationen über das Spielbrett, die sie vom Server zugeschickt bekommen, richtig

zu interpretieren. Deshalb stellt *Util-Board* den *Cores* aller Hauptkomponenten nur eine Schnittstelle zur Verfügung, da *Util-Board* bereits mit der Komponente *Util-JSON* das JSON-Format verarbeiten kann.

Übrige Schnittstellen

Schließlich müssen noch zwei Schnittstellen betrachtet werden: Sowohl der *KI-Core* als auch der *Beobachter-Core* stellen dem *Spieler-Core* eine Schnittstelle bereit. Ersteres ist nötig, weil die *KI* in die Komponente für den Mensch-Spieler integriert wird und einen menschlichen Spieler zeitweise ersetzen kann. Mit der zweiten Schnittstelle macht man sich zu Nutze, dass große Teile der Spieler-Anwendung bereits in der Beobachter-Anwendung implementiert sind und somit in der *Spieler*-Komponente wiederverwendet werden kann.

2.2 Zugeordnete Anforderungen

Server-Core FA-S-03, FA-S-04, FA-S-05, FA-S-10, FA-S-11, FA-S-12, FA-S-13, FA-S-14, FA-S-15, FA-S-16, FA-S-17, FA-S-18, FA-S-19, FA-S-20, FA-S-21, FA-O-10, FA-D-10

Server-Networking FA-O-03, FA-O-18

Server-UI FA-O-09

KI-Core FA-S-16, FA-O-06

KI-UI FA-O-01, FA-O-02, FA-O-04, FA-O-05, FA-D-07

Spieler-Core FA-S-01, FA-S-16

Spieler-UI FA-S-02, FA-S-21, FA-O-01, FA-O-07, FA-D-01, FA-D-02, FA-D-03, FA-D-04, FA-D-05, FA-D-06, FA-D-09

Beobachter-Core FA-D-10

Beobachter-UI FA-O-01, FA-O-02, FA-D-01, FA-D-02, FA-D-03, FA-D-04, FA-D-06, FA-D-09

Leveleditor-Core FA-O-08

Leveleditor-UI FA-D-08

Util-JSON FA-D-08, FA-D-10

Util-Board FA-S-06, FA-S-07, FA-S-08, FA-S-09

Util-Networking FA-O-03

3 Feinentwurf

3.1 Allgemeines

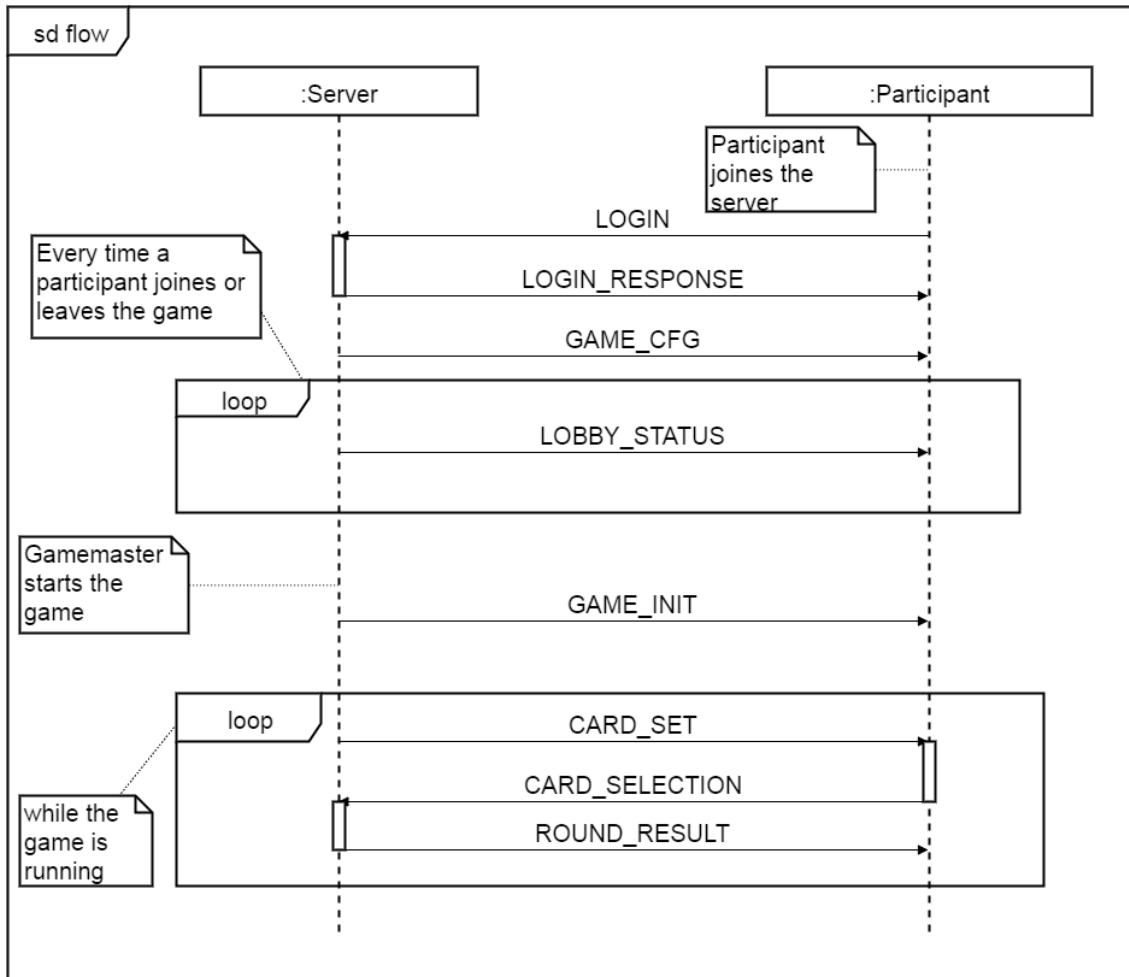
Bemerkung über **Logger**:

In diesem Kapitel werden unter alle Klassen der Anwendung mit deren Methoden und Attributen beschrieben. Ausgespart wird hierbei allerdings in der Regel eine Objektvariable „log“ vom Typ **Logger**, die in sehr vielen Klassen enthalten ist. Wir benutzen Logger, um Konsolenausgaben zu machen, die besser verwaltbar sind als reine System.out.println()-Befehle. Für weitere Informationen, siehe Dokumentation von **Logger**.

Generell gelten für den Server, den Beobachter, den Spieler und die KI, die im Schnittstellenprotokoll vereinbarten Nachrichtenflüsse. Auf jede dieser Komponenten können somit die folgenden Kommunikationsdiagramme aus diesem Protokoll angewendet werden:

Generelle Kommunikation zwischen dem Server und einem Client:

Nach der ersten CardSet-Nachricht wartet der Server die im ausgewählten Spielfeld

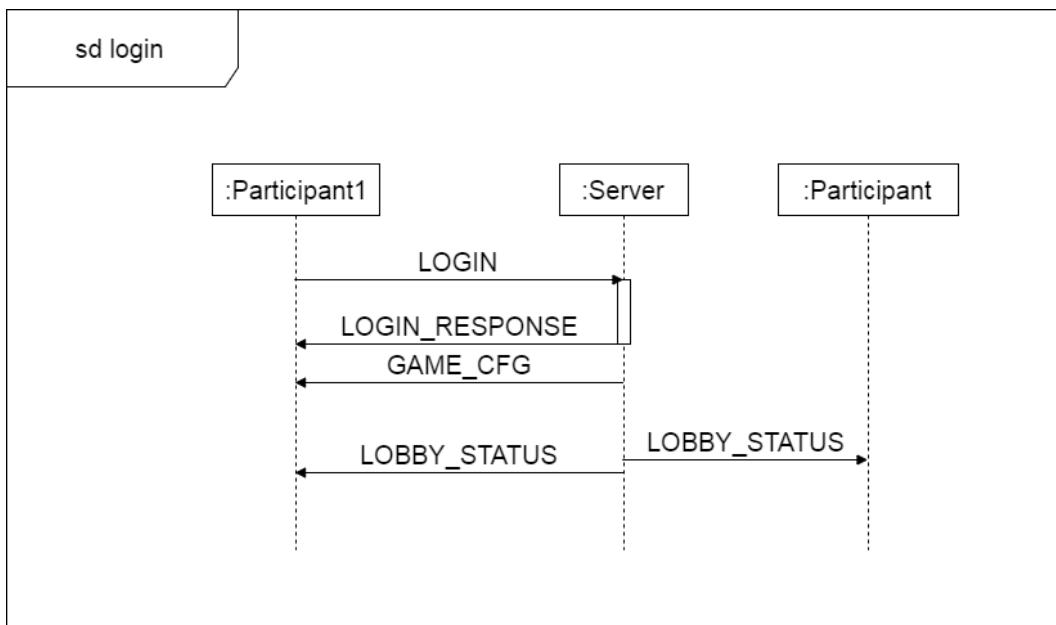


spezifizierte Auswahlzeit ab, in der die Spieler ihre Kartenauswahl treffen und an den Server zurücksenden können.

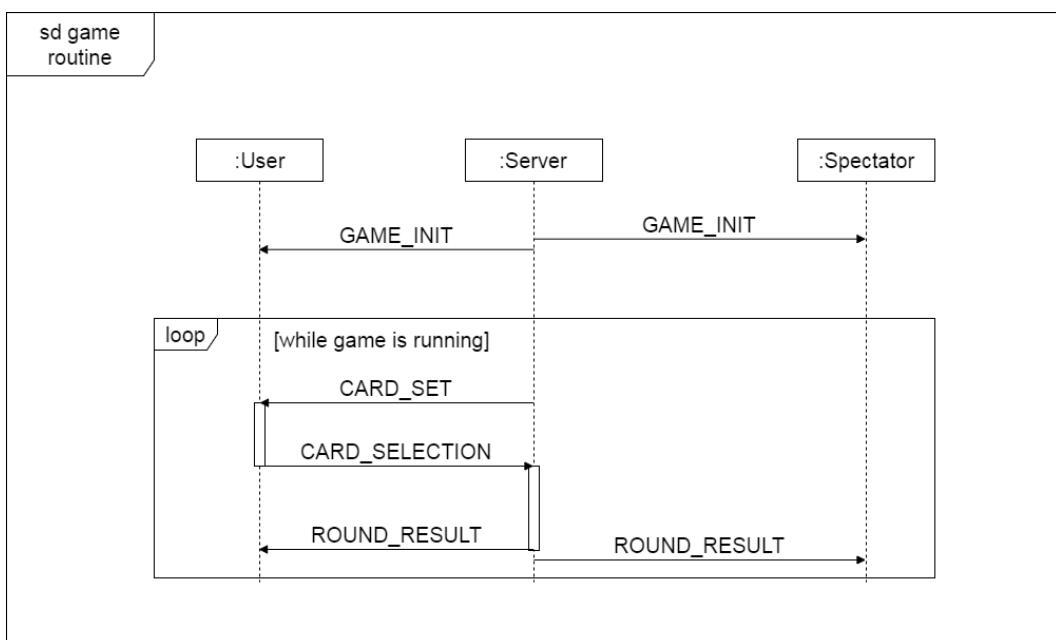
Daraufhin wertet er alle in einer **CARD_SELECTION**-Nachricht ausgewählten Karten aus, sendet eine **ROUND_RESULT**-Nachricht und, solange kein W/N-Event aufgetreten ist, die nächste **CARD_SET**-Nachricht an alle verbliebenen Nutzer.

Anschließend wartet der Server für jeden verbleibenden Nutzer 5 Sekunden sowie die im Spielfeld festgelegte Selektionszeit ab, in der die Nutzer nun ihre nächste Kartenauswahl treffen können, bevor er die nächste Kartenauswahl auswertet.

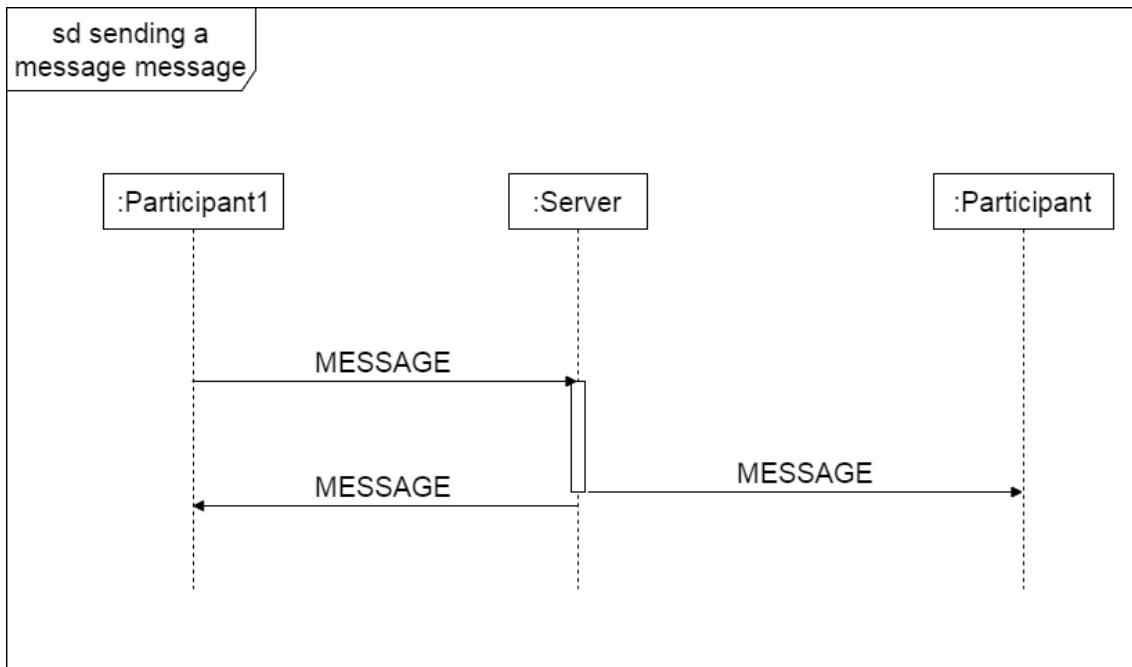
Dem Server vor dem Spielstart beitreten:



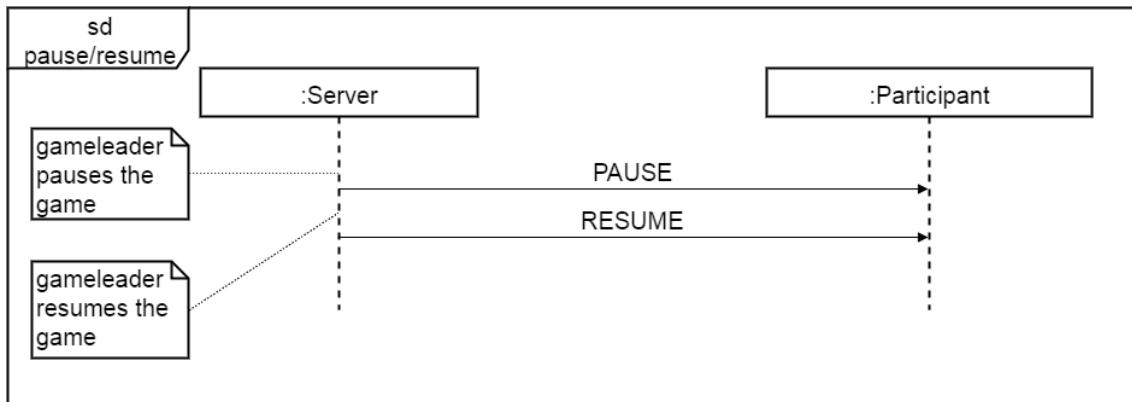
Der Spielablauf nachdem der Spielleiter das Spiel gestartet hat:



Das senden einer *MESSAGE* Nachricht nach einem erfolgreichen Einloggen:



Pausieren und Fortsetzen des Spiels:



3.2 Server

3.2.1 Beschreibung der Methoden

Um den Server zu realisieren, benötigt man natürlich alle Klassen aus der Komponente Server und noch einige Klassen aus der Komponente Util.

Main

Innerhalb der Serverkomponente gibt es die Klasse Main, die eine **main-Methode** enthält und lediglich dazu dient, um das Programm ausführbar zu machen. Die main-Methode erstellt eine Instanz der Klasse Console, die es dem Spielleiter erlaubt, den Server zu bedienen und Befehle zu erteilen. Dann wird die run-Methode der Klasse Console, die Runnable implementiert, aus main heraus gestartet.

Console

Die Klasse Console ist die Schnittstelle zum Spielleiter. Schreibt der Spielleiter etwas in die Kommandozeile, wird es in der **run-Methode** gelesen und der **parseCommand-Methode** übergeben. parseCommand erkennt dann aus dem String falls möglich den gegebenen Befehl und verarbeitet ihn entsprechend weiter.

Der Spielleiter kann folgende Befehle geben:

- Registrierungen zulassen/ablehnen
- Spiel erstellen
- Spiel initialisieren
- Spiel starten
- Spiel pausieren/fortsetzen
- Spiel abbrechen
- Server beenden

Bei Spiel abbrechen ruft die parseCommand-Methode die **stop-Methoden** aller Objekte auf, die zu einem konkreten Spiel gehören. Bei Server beenden werden auch alle übrigen stop-Methoden aufgerufen, einschließlich die von Console selbst.

Soll ein neues Spiel erstellt werden, muss der Spielleiter zusätzlich zum Befehl auch einen Port angeben. Ein neues Game-Objekt wird dann erstellt, wobei der Port im Konstruktor weitergereicht wird.

Das Game-Objekt besitzt dann Methoden, die aus Console aufgerufen werden, um die übrigen Befehle umzusetzen.

Game

Die Game-Klasse enthält die Haupt-Spielroutine und führt alle Teile des Spiels zusammen.

Im Konstruktor des Games wird die Klasse NetworkManager instanziert, um die Kommunikation zu Clients zu ermöglichen.

Solange das Spiel noch nicht gestartet wurde, kann der Spielleiter auch entscheiden, ob Registrierungen momentan grundsätzlich akzeptiert oder abgelehnt werden. Ein solcher Befehl führt zu einem Methodenaufruf von **allowRegistration**. Die Methode ruft wiederum die **registrationAllowed-Methode** des NetworkManagers, der dann den Wert in den atomaren Boolean in den dazugehörigen Acceptor schreiben kann, der ihn wiederum jeder Instanz von ClientCommunication über gibt, die er erzeugt. Bei Spielstart wird die Methode automatisch mit false ausgeführt und kann bis zum Spielende auch nicht mehr neu gesetzt werden.

Bevor ein Game gestartet werden kann, muss es initialisiert werden, also eine Leveldatei zugeordnet bekommen. Dazu wird die Methode **init** auf einen Befehl des Spielleiters hin aufgerufen. Der Spielleiter muss dabei auch eine Dateipfad angeben, unter dem valide Levelkonfigurationen zu finden sind, und init als File von Command übergeben bekommt. Mit diesem File kann init ein GameBoard erzeugen und in der entsprechenden Objektvariable abspeichern.

Auf einen weiteren Befehl des Spielleiters wird die **start-Methode** aufgerufen. Dies ist allerdings nur möglich, wenn sich die in den Levelkonfigurationen angegebene Spieleranzahl am Server angemeldet hat. Sie bekommt eine Liste von Beobachtern und Spielern für das Spiel übergeben, deren Werte sie in die entsprechenden Objektvariablen schreibt. Jedem Spieler wird dabei sogleich eine PlayerHand zugeordnet, die durch einen Aufruf der statischen Methode **createPlayerHands** in der Klasse PlayerHand erzeugt werden. Dann wird die **run-Methode** der Klasse gestartet, um die Spielroutine zu beginnen.

In der run-Methode werden dann in Schleife die Spielrunden abgehandelt. Die PlayerHands werden über den NetworkManager mit der Methode **sendToPlayer** an die Spieler verschickt. Nach einer Wartezeit werden die zurückgesendeten Programme durch **requestFromPlayer** des NetworkManagers abgerufen und verarbeitet. Die Rundenergebnisse werden berechnet und wiederrum über **sendToAllPlayers** und **sendToAllSpectators** über den NetworkManager verschickt. Falls dabei das Spielende erreicht wird, werden entsprechende Nachrichten an alle Spieler und Beobachter gesendet, das Spiel über das aufrufen aller nötigen **stop-Methoden** beendet und die Schleife wird abgebrochen.

Mehrmals pro Runde muss die atomare Objektvariable pause überprüft werden, weil sie anzeigt, wenn der Spielleiter das Spiel pausieren möchte. Dies geschieht fortwährend in der Wartezeit während der Programmierphase und unmittelbar nachdem das Rundenergebnis an die Clients geschickt wurde. Wird festgestellt, dass pause auf true steht, so wird eine Schleife ausgeführt, die nur überprüft, ob pause wieder auf false gesetzt wurde, in welchem Fall sie wieder verlassen würde und die normale Routine fortgesetzt.

Zwischendurch wird ebenfalls regelmäßig - wieder über Methoden des NetworkManagers - überprüft, ob Chat-Nachrichten vorliegen, und gegebenenfalls an andere Clients weitergeleitet.

Am Ende der Runde wird für alle Spieler eine neue PlayerHand erstellt. Dann beginnt die neue Runde.

Gibt der Spielleiter den Befehl zur Pause, wird die **pause-Methode** aufgerufen. Der Befehl wird unterschiedlich behandelt: Ist die Objektvariable paused = false, wird der

AtomicBoolean pause auf true gesetzt (und damit implizit die Spielroutine angehalten), eine Pause-Nachricht über den NetworkManager an alle Clients gesendet und die Variable paused auf true gesetzt. Ist paused = false passiert genau der umgekehrte Vorgang. Initial stehen paused und pause auf false.

NetworkManager

Der NetworkManager ist für die Koordination der gesamten Kommunikation mit den Clients zuständig. Er wird mit dem Erstellen eines Spiels erzeugt und bekommt einen Port zugewiesen. Im Konstruktor wird Acceptor instanziert und dessen **run-Methode** gestartet. Die Klasse speichert den im Konstruktor zuvor erstellten **Acceptor** und eine Liste von Spieler- und BeobachterIDs inklusive der dazugehörigen Kommunikationsschnittstelle. Zum Steuern der Kommunikation stehen im NetworkManager verschiedene **send-** bzw. **request-Methoden** zur Verfügung.

Sowohl für Beobachter als auch für Spieler steht jeweils eine send-Methode zur individuellen Kommunikation, als auch zum Broadcast bereit. Die Methoden nehmen dann Nachrichten verschiedener Typen entgegen, componieren das Format in JSON mit Hilfe einer Methode aus dem JSON-Package und rufen dann damit die Sendemethoden aus einem oder gegebenenfalls mehreren Player- oder SpectatorCommunication- Objekten auf. Im Falle der individuellen Nachricht benötigt die Methode die eindeutige playerID/spectatorID, um den Empfänger zu ermitteln.

Parallel dazu existieren request-Methoden, die bei einem oder mehreren Kommunikatoren von Clients anfragen können, ob Nachrichten von einem bestimmten Typ verfügbar sind.

Acceptor

Die Klasse Acceptor ist für alle neu eingehenden Verbindungen zuständig. Immer, wenn sich ein Client am Server registrieren möchte und an den **Acceptor** eine Anfrage gestellt hat, wird vom Acceptor eine neue Instanz von **ClientCommunication** erstellt und dessen run-Methode ausgeführt.

ClientCommunication

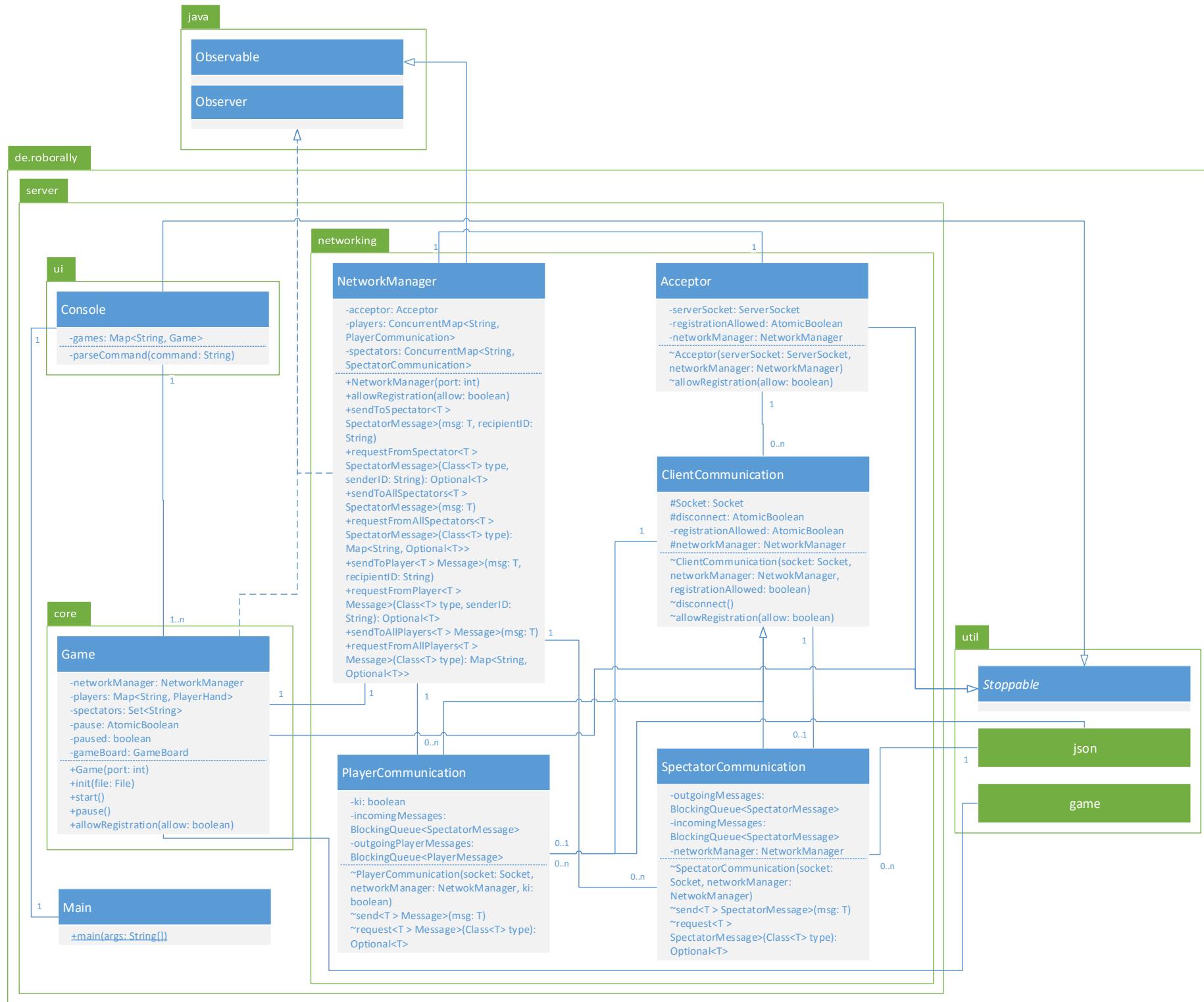
Ein Objekt ClientCommunication wird immer dann erstellt und ausgeführt, wenn ein Client eine Verbindung zum Server aufbaut. Im Konstruktor wird ihr die Information übergeben, ob Registrierungen derzeit erlaubt sind oder nicht. Die **run-Methode** überprüft den Wert: Sind Registrierungen verboten, wird die Methode **disconnect** aufgerufen und die Verbindung zum Client wird wieder getrennt. Ansonsten wird die Information abgefragt, ob es sich bei dem Client um einen Spieler oder einen Server handelt. Je nachdem wird dann entweder ein PlayerCommunication- oder ein SpectatorCommunication-Objekt erzeugt. ClientCommunication vererbt beiden Kommunikatoren.

Player-/SpectatorCommunication

Die Schnittstelle zwischen dem **NetworkManager** und dem individuellen Client wird über eine Instanz von **PlayerCommunication** für Spieler oder von **SpectatorCommunication** für Beobachter realisiert. Beide Klassen erben von **ClientCommunication** und bekommen damit die Möglichkeit den Client via **disconnect** vom Server zu trennen und sich selber damit zu schließen. Beide Klassen sind für einen Client zuständig und führen mit **send-** und **request-Methoden** den Schritt aus, Nachrichten tatsächlich an die Cliebts zu versenden. Dafür lesen sie ständig, ob Nachrichten über den Socket ankommen und schreiben solche dann in eine Queue. Von dort aus können Methoden des NetworkManagers darauf zugreifen, wenn sie die **request-Methode** des Objekts aufrufen. Ebenso kann durch einen Aufruf der **send-Methode** eine zu verschickende Nachricht in eine andere Queue geschrieben werden, von wo aus sie dann abgearbeitet und nacheinander über das Netzwerk verschickt werden. Dabei können Beobachter allerdings nur Spectator-Messages senden und empfangen, während Spieler alle Nachrichten verarbeiten können. Dadurch ist sichergestellt, dass zum Beispiel nicht fälschlicherweise ein Spectator Karten an den Server schicken kann.

3.2.2 Klassendiagramme

Siehe folgende Seiten.



3.2.3 Kommunikationsdiagramme

Bemerkung: Die Diagramme DatenAnfragen und DatenSenden stehen repräsentativ für jeweils mehrere Diagramme. Daten können schließlich auch mit Spielern (Player) statt Beobachtern (Spectator) ausgetauscht werden. Weiterhin besitzt die Klasse NetworkManager neben „sendTo“- und „requestTo“-Methoden auch noch die Äquivalente „sendToAll“ und „requestToAll“. Beim Aufruf dieser Methoden gehen dann entsprechend von NetworkManager mehrere Links aus zu mehreren Instanzen von Spectator beziehungsweise Player, auf denen dann jeweils eine send- oder request-Nachricht verschickt würde. Da dies aber alles nur geringfügige Variationen auf denselben Diagrammen sind, haben wir beschlossen, all diese Fälle durch ein Beispiel darzustellen.

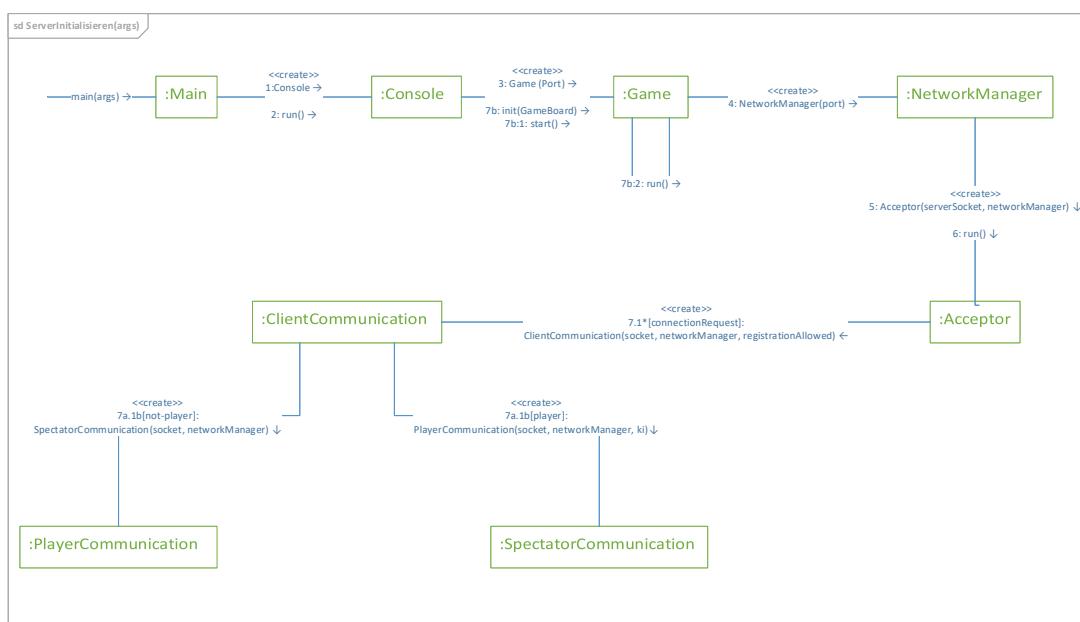


Abbildung 3.1: Server - Kommunikationsdiagramm 01

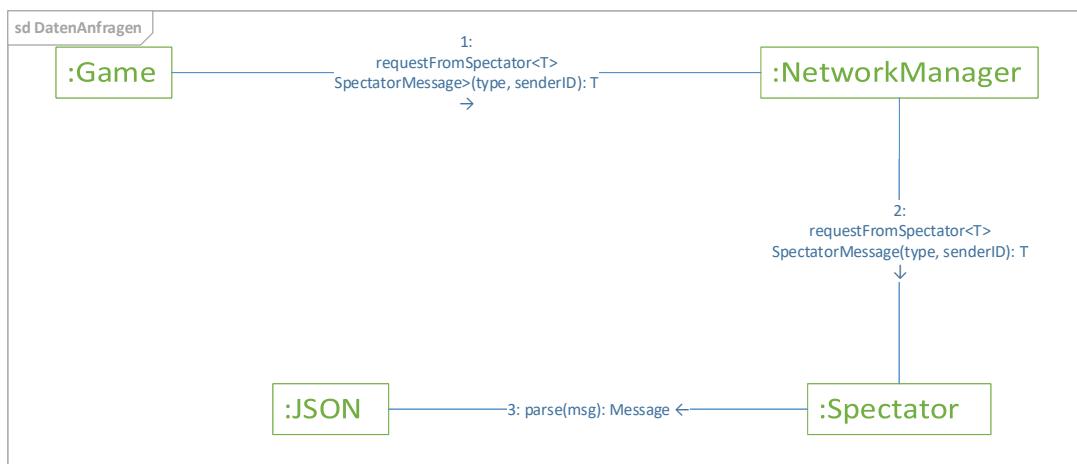


Abbildung 3.2: Server - Kommunikationsdiagramm 02

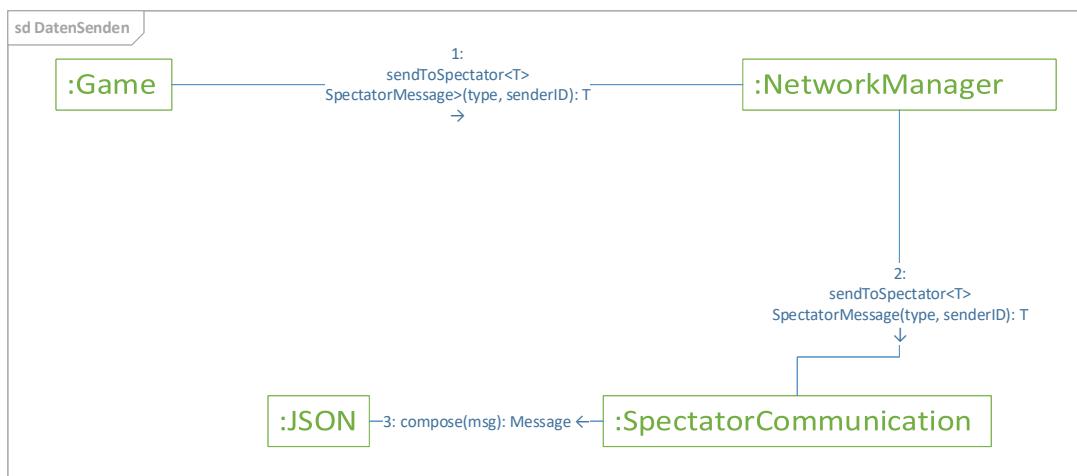


Abbildung 3.3: Server - Kommunikationsdiagramm 03

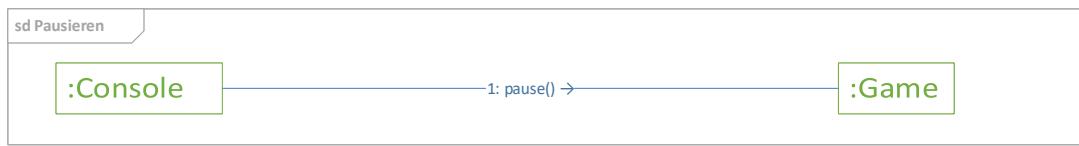


Abbildung 3.4: Server - Kommunikationsdiagramm 04

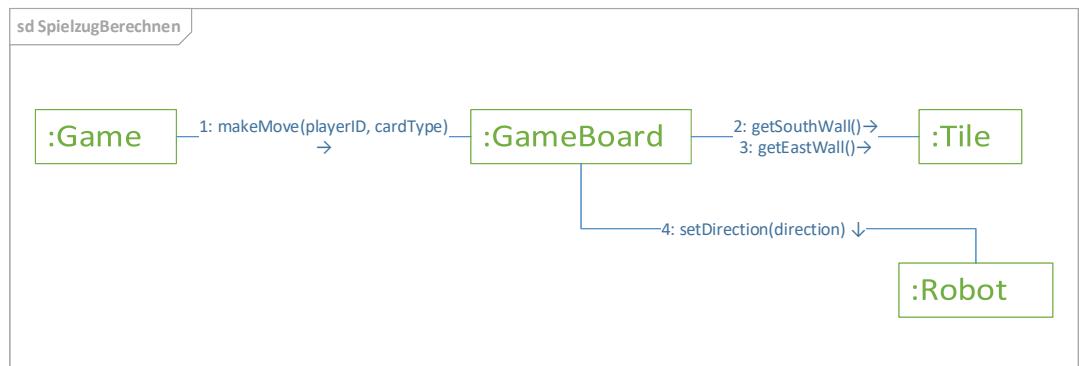


Abbildung 3.5: Server - Kommunikationsdiagramm 05

3.3 Util

3.3.1 Beschreibung der Methoden

Message

Unter dem Interface **Message** werden alle Nachrichten zusammengefasst, die zwischen Server und Clients verschickt werden können. Da alle Nachrichten einen Header besitzen sollen, der Metadaten über die jeweilige Nachricht speichern soll, umfasst das Interface **Message** eine Methode **getHeader**, um diese Informationen auch auslesen zu können. Weiterhin enthält das Interface die Methoden **composeMessage** und **parseMessage**. Diese Methoden dienen dazu, eine Nachricht in ein JSON-Objekt umzuwandeln, welches über das Netzwerk verschickt werden kann (*compose*) und aus einem JSON-Objekt auch wieder das entsprechende Java-Objekt zu erstellen (*parse*). Schließlich beschreibt **Message** die Methode **isValid**, die prüft, ob eine Nachricht korrekt oder fehlerhaft ist. Zudem ist im Interface die statische Methode **instanceOf** implementiert, mit der man von jedem Nachrichtentyp leere Instanzen erstellen kann. Zwar besitzt jeder Nachrichtentyp einen eigenen Konstruktor, doch sie nehmen alle Parameter entgegen. In den Konstruktoren wird das leere Instantiiieren absichtlich verboten, weil Messages in den meisten Fällen nicht leer sein dürfen und niemand versehentlich leere Messages versenden können soll. **instanceOf** ist für Ausnahmefälle gedacht.

Das Interface **Message** besitzt mehrere Subinterfaces, die die Nachrichtentypen weiterhin unterteilen. Interfaces, die direkt von **Message** erben, sind:

- **ServerMessage**: Umfasst alle Nachrichtentypen, die vom Server verschickt werden können
- **ClientMessage**: Umfasst alle Nachrichtentypen, die von Clients verschickt werden können
- **PlayerMessage**: Umfasst alle Nachrichtentypen, die zwischen Spielern und dem Server ausgetauscht werden können (von Beobachtern aber weder empfangen noch gesendet werden können)
- **SpectatorMessage**: Umfasst alle Nachrichtentypen, die **sowohl** zwischen Spielern und dem Server, **als auch** zwischen Beobachter und dem Server ausgetauscht werden können

Beobachter und Spieler können selbstverständlich keine Nachrichten direkt miteinander austauschen.

Es gibt noch vier weitere Interfaces, die die vorherigen Subinterfaces neu zusammenfasst:

- **PlayerServerMessage**: erbt von **PlayerMessage** und **ServerMessage**; umfasst jene Nachrichtentypen, die vom Server an Spieler gesendet werden
- **SpectatorServerMessage**: erbt von **SpectatorMessage** und **ServerMessage**; umfasst jene Nachrichtentypen, die vom Server an Spieler und Beobachter gesendet werden

- **PlayerClientMessage**: erbt von **PlayerMessage** und **ClientMessage**; umfasst jene Nachrichtentypen, die von Spielern an den Server geschickt werden
- **SpectatorClientMessage**: erbt von **SpectatorMessage** und **ClientMessage**; umfasst jene Nachrichtentypen, die von Spielern und Beobachtern an den Server geschickt werden

Alle acht Subinterfaces dienen hier im Übrigen lediglich der weiteren Unterteilung; sie ergänzen die Implementierungsvorschriften von **Message** nicht weiter und sind leer.

Message-Implementierungen gibt es insgesamt zwölf. Alle besitzen einen **Header** als Attribut. Die übrigen Attribute richten sich nach den Informationen, die vom jeweiligen Nachrichtentyp vermittelt werden sollen. Jeder Nachrichtentyp besitzt einen Konstruktor, der diese Attribute setzt und stellt entsprechende Getter neben den vom **Message**-Interface vorgegebenen Methoden zur Verfügung.

Die **LoginMessage** implementiert **SpectatorClientMessage** und wird von einem Client verschickt, der er sich am Server anmelden möchte. Es besitzt das Feld `client`, mit dem der Client, der sich anmelden möchte, selbst beschrieben wird.

Auf eine **LoginMessage** antwortet der Server mit einer **LoginResponseMessage**, die **SpectatorServerMessage** implementiert. Sie enthält einen `status`, der dem Client mitteilt, ob er akzeptiert oder abgelehnt wurde und die User-ID `uid`, unter der der Client vom Server gespeichert wird. Bei Ablehnung wird in `uid` ein default-Wert `VALUE_EMPTY` angegeben, der anzeigt, dass das Feld keinen sinnvollen Inhalt hat.

Nach erfolgreicher Anmeldung versendet der Server an den entsprechenden Client eine **GameCfgMessage**, die **SpectatorServerMessage** implementiert. Sie liefert dem Client alle relevanten Informationen über das aktuelle Spiel. Dazu gehören die Felder `maxSpectators`, also die maximal mögliche Anzahl von Beobachtern im Spiel, `tournament`, dass anzeigt, ob das Spiel im Turniermodus stattfindet, und `gameMap`. In `gameMap` sind dabei alle übrigen Informationen enthalten wie Größe des Spielfelds, Position der Checkpoints und so weiter.

Die **LobbyStatusMessages** implementiert ebenfalls **SpectatorServerMessage** und dient dazu, Clients darüber zu informieren, welche Clients derzeit noch am Server angemeldet sind, während sie sich in der Lobby aufhalten, also vor Beginn des Spieles. Dementsprechend besitzt es neben dem Header das Feld `clients`, dass eine Liste aller angemeldeten Clients ist.

Auch die **GameInitMessage** implementiert **SpectatorServerMessage**. Sie speichert eine Liste aller am Server angemeldeten Clients und eine Liste von `GameInitStates`, die alle Clients am Anfang des Spiels erhalten sollen. Sie wird mit Beginn des Spiels gesendet.

Eine **PlayerHandMessage** wird vom Server verschickt, um Spielkarten an einen Spieler zu verteilen. Sie implementiert also **PlayerServerMessage** und speichert neben dem Header eine `PlayerHand`.

Mit einer **ProgramMessage** sendet ein Spieler am Ende der Programmierphase die von ihm ausgewählten Karten wieder zurück. Sie implementiert **PlayerClientMessage**. Neben dem Program, dass der Spieler zusammengestellt hat, besitzt `ProgramMessage` das Feld `orientation`, das festlegt, welche Blickrichtung der Roboter hat, falls er in dieser Runde neu auf dem Spielfeld eingesetzt werden muss.

Die **RoundResultMessage** verschickt der Server am Ende jeder Runde an alle Clients. Sie implementiert **SpectatorServerMessage** und enthält die **RoundresultEvents** der vergangenen Runde sowie für alle Spieler nochmals eine Liste aus **RoundResultFinalStates**.

PauseMessage und **ResumeMessage** implementieren **SpectatorServerMessage**. Sie werden vom Server verschickt, wenn er das Spiel pausiert beziehungsweise wieder damit fortfährt. Neben dem Header besitzen diese beiden Nachrichtentypen keine weiteren Felder mehr.

Eine **ErrorMessage** wird versendet, wenn der Spielverlauf gestört wurde, zum Beispiel weil das Spiel abgebrochen wurde oder ein Spieler die Regeln verletzt hat. **ErrorMessage** implementiert **SpectatorServerMessage**. Sie enthält einen Fehlertyp `messageErrorType` und einen String `errorContext`, der eine textliche Fehlermeldung speichert.

Die **ChatMessage** implementiert **SpectatorServerMessage** und **SpectatorClientMessage** und kann somit von allen beteiligten Parteien gesendet und empfangen werden. Neben dem Header speichert sie eine Sender-ID, um eine Chat-Nachricht zuordnen zu können und einen String `msg`, der die eigentliche Chatnachricht enthält. Da Chatnachrichten in Robo Rally immer für alle Teilnehmer lesbare Broadcastnachrichten sind, ist jedwede Art von Empfänger-ID unnötig.

MessageFactory

Die Klasse `MessageFactory` stellt für jeden Nachrichtentyp eine statische Methode zur Verfügung, mit der die jeweilige Nachricht bequem instantiiert werden kann. Diese Methoden nehmen zum Beispiel keinen Header entgegen, sondern generieren ihn selbst automatisch. So ist das Erstellen von Nachrichten weniger aufwändig. Da **MessageFactory** selbst nicht instantiiert werden soll, besitzt sie einen privaten Konstruktor, der den öffentlichen Standardkonstruktor versteckt.

MessageUtil

Das Interface **MessageUtil** dient dazu, Hilfsklassen für Messages zusammenzufassen. So wird sie von Objekten implementiert, die als Inhalt von Messages auch über das Netzwerk verschickbar sein müssen. So beschreibt das Interface die Methoden **composeUtil** und **parseUtil**, die die MessageUtil-Implementierungen ins JSON-Format und wieder zurückwandeln. Ebenso ist eine **isValid**-Methode vorgesehen, die von den **isValid**-Methoden der Messages aufgerufen werden kann, die ein Message-Util-Objekt enthalten. Wie das **Message**-Interface besitzt auch MessageUtil eine statische **instanceOf**-Methode, mit der sich ohne die jeweiligen Konstruktoren benutzen zu müssen leere Instanzen von Message-Util-Implementierungen erzeugen lassen.

Card

Um eine Spielkarte zu repräsentieren, wird eine Instanz von `Card` erzeugt, die einen Typ und eine Priorität für die Karte speichert. Der Card-Typ, ist definiert durch einen Wert aus der Enumeration **CardType**. `Card` implementiert das Interface **MessageUtil**.

PlayerCardTuple

Ein PlayerCardTuple ist eine Möglichkeit, eine Karte und einen Spieler miteinander zu assoziieren, also praktischerweise für eine Karte zu speichern, von welchem Spieler sie

stammt. Die Klasse speichert zwei Variablen card und uid, die sinngemäß die Karte und die User-ID des Spielers enthalten, die miteinander assoziiert werden sollen. Die Variablen können über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Da die Klasse das Interface **Comparable** implementiert, besitzt die Klasse zusätzlich noch eine **compareTo**-Methode, die überprüft, ob zwei PlayerCardTuples die gleiche Karte enthalten.

PlayerHand

Eine **PlayerHand** dient dazu, die neun Karten zusammenzufassen, die einem Spieler in jeder Spielrunde zur Verfügung stehen. **PlayerHand** implementiert das Interface **MessageUtil**.

Program

Ein **Program** fasst fünf Karten zusammen, die gemeinsam ein Programm bilden, also eine von einem Spieler zusammengestellt wird. Die Reihenfolge der Karten in einem Program wird ebenfalls gespeichert. **Program** besitzt eine Methode **validateProgram**, die überprüft, ob ein Programm überhaupt nach den Spielregeln zulässig ist. Dafür nimmt es die PlayerHand entgegen, die der jeweilige Spieler zuvor erhalten hatte, um zu überprüfen, ob die zurückgesendeten Karten dem Spieler überhaupt zur Verfügung gestellt wurden. **Program** implementiert das Interface **MessageUtil**.

Checkpoint

Checkpoint ist eine Repräsentation von Checkpoints im Spiel, die mit den Integern fieldX, fieldY und number die Koordinaten auf dem Spielbrett sowie den eigenen Rang speichert, also die Nummer, in welcher Reihenfolge er zwischen den anderen abgelaufen werden muss. Die Variablen können über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Ansonsten implementiert **Checkpoint** das Interface **MessageUtil**.

Client

Client ist eine Repräsentation eines am Server angemeldeten Spielers oder Beobachters. Ein Client besitzt einen Namen, mit dem er im Spiel angemeldet ist, eine **ClientRole**, die besagt, ob jeweiliger Client Spieler oder Beobachter ist, einen **Bool** ai, der ansagt, ob es sich beim Client (falls es ein Spieler ist) um eine künstliche Intelligenz handelt und einen integer uid, die User-ID des Servers für den Spieler. Die Variablen können über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Ansonsten implementiert **Client** das Interface **MessageUtil**.

GamelinitState

GamelinitStates sind Teil von **GamelinitMessages**. Ein GamelinitState enthält alle relevanten Informationen eines Roboters. So enthält es die User-ID uid des Spielers, dem der Roboter gehört, die Nummer des höchsten Checkpoints, den der Roboter bisher erreicht hat, die Koordinaten an denen sich der Roboter aktuell befindet fieldX und fieldY und die Anzahl an Leben die er besitzt. Da die GamelinitMessage zu Beginn des Spiels gesendet wird, sollten bei einem normalen Spiel alle Angaben außer uid natürlich immer gleich sein. Sie werden hier eingebunden, um dem Schnittstellenprotokoll gerecht zu

werden. Die Variablen können jedenfalls über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Ansonsten implementiert **GameInitStates** das Interface **MessageUtil**.

RoundResultEvent

RoundResultEvents werden in **RoundResultMessages** verschickt. Ein RoundResultEvent beschreibt genau eine Aktion eines Roboters, zum Beispiel wenn er einen Schritt nach vorne zieht, sich dreht, zerstört wird oder auch gewinnt. Zu Beachten ist hierbei, dass zum Beispiel das Ausführen einer Karte, bei der ein Roboter drei Schritte nach vorne zieht, aus drei einzelnen RoundResultEvents besteht. Wenn ein Roboter sich allerdings nur durch den Einfluss eines anderen bewegt, zum Beispiel weil er geschoben wird, ist dies allerdings wiederum kein eigenes Event, sondern wird im beeinflussenden Event berücksichtigt. Ein RoundResultEvent enthält die User-ID uid des Spielers, dem die Aktion zugeschrieben wird, die Nummer des höchsten Checkpoints, den dieser Spieler bisher erreicht hat, die Koordinaten fieldX und fieldY, auf denen der Roboter des Spielers gerade steht, seine Blickrichtung orientation und die Anzahl der Leben, die er besitzt. Weiterhin steht in roundEventType, um welche Art Event es sich hier eigentlich handelt (also zum Beispiel Schritt nach vorne oder Zerstörung). In der Liste affectedUid stehen die User-IDs jener Spieler, die vom aktuellen Event ebenfalls betroffen sind (zum Beispiel durch Schieben). Die Variablen können über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Ansonsten implementiert **RoundResultEvent** das Interface **MessageUtil**.

RoundResultFinalState

RoundResultFinalStates sind ähnlich aufgebaut wie **GameInitStates**, speichern allerdings zusätzlich die Informationen orientation, also die Blickrichtung des Roboters, und onMap, ein **Bool**, der besagt, ob sich der Roboter aktuell auf der Karte befindet. RoundResultFinalStates sind neben den **RoundResultEvents** Teil von **RoundResultMessages**. Sie dienen dazu, den Zustand jedes Roboters am Ende jeder Runde nochmals zusammenzufassen. Alle Variablen können über den Konstruktor gesetzt und über Getter-Methoden gelesen werden. Ansonsten implementiert **RoundResultFinalStates** das Interface **MessageUtil**.

Header

Ein Header ist Bestandteil jeder Message. Er enthält Meta-Informationen über die Message. Er enthält die Sendezeit sentTime, einen String mit der protocolVersion, unter dem die Message läuft, und den eigentlichen Nachrichtentyp type. Er bietet zwei Konstruktoren an: Einer nimmt sentTime und protocolVersion entgegen, der andere setzt diese Werte automatisch. Type muss immer übergeben werden. Ansonsten implementiert **Header** das Interface **MessageUtil**.

GameMap

Es gibt andere Klassen, um das Spielbrett Server- und Clientintern darzustellen (siehe **GameBoard**). Die Klasse GameMap dient hauptsächlich dazu, ein Spielbrett in ein verschickbares Format zu bringen. Mithilfe einer GameMap kann dann ein GameBoard

erstellt werden. Eine GameMap speichert ein creationDate, also das Datum, an dem sie erstellt wurde. Ein String type besagt, dass eine Datei eine Map enthält. Er sollte eigentlich grundsätzlich „MAP“ enthalten. Die Integers minPlayers und maxPlayers begrenzen die Anzahl an Spielern, für welche die Map ausgelegt ist. maxCardSelectionTime bestimmt die Länge der Programmierphasen, also die Zeit, die Spieler in jeder Runde brauchen dürfen, um ihr Programm zu schreiben. width und height geben die Länge und Breite des Spielfelds in Kacheln an. Die Listen checkpoints, walls und assets speichern alle Checkpoints, Mauern und Assets, die sich auf der Map befinden. Alle Variablen können über einen Konstruktor gesetzt und über Getter-Methoden gelesen werden. Zudem besitzt GameMap noch eine statische Methode **parseMapFromFile**, die aus einer gegebenen Datei, falls sie dem benötigten Format entspricht, ebenfalls eine GameMap erzeugen kann. Ansonsten implementiert GameMap das Interface **MessageUtil**.

Asset

Assets sind „Zubehör“ für das Spielbrett. Hier können nach Schnittstellenprotokoll Elemente in einem Level gespeichert werden, die auf die Spielregeln keinerlei Auswirkungen haben, sondern von einzelnen Gruppen beliebig für optionale Grafiken oder ähnliches genutzt werden können. Unsere Gruppe nutzt diese Assets nicht (außer für Testfälle). Da Assets in Messages verschickt werden können, implementiert es **MessageUtil**.

Robot

Eine Instanz der Klasse **Robot** repräsentiert die Spielfigur eines Spielers. Ein **Robot** speichert eine Blickrichtung (orientation), eine Spilbrettkachel, auf der er sich aktuell befindet (tile), eine weitere Kachel, auf der sich der höchstnummerierte Checkpoint befindet, den der Roboter bisher korrekt erreicht hat (lastReachedCheckpointTile), einen boolean, der anzeigt, ob sich der Roboter aktuell überhaupt auf dem Spielfeld befindet (onMap), die Anzahl der Leben, die der Roboter noch hat (lives) und die UserID des Spielers, dem der Roboter gehört (uid).

GameBoard

GameBoard bildet das Spielbrett ab. **GameBoard** ist dabei eine abstrakte Klasse, die an zwei Klassen vererbt: **LogicGameBoard** in der Util-Komponente und **VisualGameBoard** in der Beobachter-Komponente (näheres dazu an den entsprechenden Stellen). Ein **GameBoard** ist im Kern ein zweidimensionales Array aus **Tiles**, das in der Objektvariable board gespeichert ist. Die im Spiel vorhandenen **checkpoints** werden in einer Liste von **Tiles** gespeichert. Um einen Roboter auf dem Spielbrett zu bewegen, stellt die Klasse **GameBoard** die Methoden **setRobotTile** und **letRobotRotate** zur Verfügung. Weiterhin gibt es Methoden zum (Wieder-) Einsetzen von Robotern (**letRobotSpawn**), zum Sterben von Robotern (**letRobotDie**) und zum Erreichen eines Checkpoints (**letRobotReachCheckpoint**).

LogicGameBoard

LogicGameBoard erbt von GameBoard und ist für logische Operationen auf dem Spielbrett zuständig, also wenn es darum geht, Roboter zu bewegen, zu drehen, überprüfen ob sie gegen Wände prallen und so weiter. Die meisten Methoden hierzu sind private, werden

also hier nicht weiter ausgeführt. Die einzige Objektvariable, auf die öffentlich per getter zugegriffen werden kann ist `winner`. Sie bekommt initial einen Leerwert zugewiesen und soll bei Spielende die uid des gewinnenden Spielers enthalten. `LogicGameBoard` besitzt einen Konstruktor, der eine **GameMap** entgegennimmt. Eine öffentliche Methode der Klasse ist **addPlayer**, die eine User-ID entgegennimmt und den user dann zusammen mit einem neuen Roboter intern abspeichert. Weiterhin gibt es die öffentliche Methode **calculateRoundResult**, der die User-ID eines Spielers und ein **CardType** übergeben werden kann. Darin wird dann korrekt ausgeführt, was für die Karte und den Spieler aus der Eingabe passiert. Dazu ruft es teils private, teils public Hilfsmethoden aus der eigenen Klasse auf, die jeweils für ein bestimmtes Ereignis zuständig sind. Als Rückgabe kommt eine Liste aus **RoundResultEvents**, die den eingetretenen Ereignissen entsprechen und als Information über das Netzwerk verschickt werden können. Die übrigen öffentlichen Methoden der Klasse sind **destroyRobot**, **diedRobot**, **spawnRobot** und **disqualifyRobot**. Sie geben alle Listen von **RoundResultEvents** zurück. Sie sind einerseits oben erwähnte Hilfsmethoden für die Methode **calculateRoundResult**, wobei am Namen erkennbar ist, welche Methoden für welche Ereignisse zuständig sind. Public sind die Methoden deshalb, weil diese Ereignisse zusätzlich auch eintreten können, ohne dass direkt eine Karte ausgeführt wird, also **calculateRoundResult** nicht aufgerufen wird, und die Methode dann direkt angesprochen werden kann. Zum Beispiel kann ein Roboter ja auch zerstört werden, wenn es der jeweilige Spieler versäumt hat, seine Karten rechtzeitig zurückzuschicken.

Tile

Eine Instanz der Klasse **Tile** repräsentiert eine Kachel auf dem Spielbrett. Es besitzt eine **Position** auf dem Spielbrett. Außerdem speichert es die Informationen, ob sich rechts und/oder unterhalb der Kachel eine Mauer befindet, ob sich ein Checkpoint auf der Kachel befindet und wenn ja, welche Nummer er hat und gegenfalls einen **Robot**, falls sich einer auf der Kachel aufhält. Um diesen Roboter zu setzen beziehungsweise zu entfernen, stellt die Klasse entsprechende Methoden zur Verfügung.

Position

Eine Position ist eine einfache Hilfsklasse, die eine x- und eine y-Koordinate speichert. Die Einheit der Koordinaten ist die Anzahl an Spielfeldkacheln. Die Koordinaten können mit einem Konstruktor gesetzt und mit Getter-Methoden gelesen werden.

JsonMessageComposer

Um erfolgreich über das Netzwerk zu kommunizieren, müssen Server und Clients Nachrichten über ein definiertes Format austauschen. Die Klasse **JsonMessageComposer** ist dafür zuständig, Nachrichten, die intern beim Arbeiten als Java-Objekte gespeichert sind, in dieses JSON-Format zu übersetzen. Dafür stellt sie folgende Methoden zur Verfügung: **compose**, **composeMessage**

Stoppable

Die abstrakte Klasse **Stoppable** ist eine Erweiterung des Interfaces **Runnable**, welches sie implementiert. Sie fügt also Klassen, die als eigene Threads ausführbar sind, neben

der run-Methode die sie enthalten müssen, auch noch eine **stop**-Methode hinzu, um den entsprechenden Thread wieder bequem beenden zu können. Diese setzt ein AtomicBoolean stop auf true, welches dann in einem Stoppable abgefragt werden kann. Außerdem erbt Stoppable von **Observable**.

NetworkManager

Der NetworkManager im Util-Package dient dazu, die Netzwerkkommunikation auf der Client-Seite zu regeln. NetworkManager erbt von **Stoppable**, ist also als eigener Thread ausführbar. Jeder Client, der über das Netzwerk kommunizieren will, braucht eine eigene Instanz des NetworkManagers. Zu welchem Client ein NetworkManager gehört, wird in der Objektvariable endpoint in Form eines **NetworkingEndpoint** gespeichert. Mit einem **AccesLevel** accesLevel wird gespeichert, welche Rolle der jeweilige Client innehat. Mit einer **InetAdress** serverAdress und einem Integer Port wird der Schnittpunkt zum Server definiert, zu dem der networkManager die Kommunikation aufnehmen soll. Eine **BlockingQueue** outgoingMessages dient dazu, alle Messages, die der Client verschicken will, zwischenzuspeichern. Initial ist sie leer. Der Konstruktor der Klasse nimmt einen **NetworkingEndpoint** entgegen und beschreibt mit dieser Information die Objektvariablen endpoint und accesLevel. Dann besitzt der NetworkManager eine **start**-Methode, die die Objektvariablen serverAdress und Port über Parameter setzt, die stop-Variable (Erbgut von **Stoppable**) false setzt und die run-Methode des Threads aufruft.

Die **run**-Methode des Networkmanagers baut zuerst eine Verbindung zum Server auf, startet einen neuen **Sender**-Thread, der sich darum kümmert, die Nachrichten outgoingMessages zu verschicken, und überwacht dann solange bis er beendet wird den InputStream zum Server, um die Nachrichten des Servers an seinen Client weiterzuleiten.

Um Nachrichten über den NetworkManager verschicken zu können, bietet die Klasse die Methode **send** an, die eine **Message** entgegennimmt und in outgoingMessages schreibt. Ums versenden kümmert sich dann der in **run** erzeugte Sender. Um sich an einem Server als Spieler anzumelden, muss keine **LoginMessage** erst erstellt und über **send** verschickt werden, sondern der NetworkManager bietet hierfür eigens eine Methode **register** an, die dann mit Hilfe der Parameter name (Anmeldename des Clients) und ai (boolean, ob Client KI ist oder nicht) selbst eine LoginMessage erstellt und versendet.

Sender

Der Sender ist eine Hilfsklasse, die auf einem eigenen Thread ausgeführt wird. Sie erbt von Stoppable. Ihre Aufgabe ist es, Nachrichten sequentiell zu verschicken. Sie speichert einen **BufferedOutputStream** und eine **BlockingQueue** aus **Messages** outgoingMessages ab. In die Queue werden von außen diejenigen Nachrichten geschrieben, die der Sender versenden soll, der Stream ist der Ort, auf den die Nachrichten geschrieben werden sollen. Außerdem besitzt sie einen String logMessage, der eine Standardnachricht für den Logger enthält. Wie viele anderen Klassen auch (auch wenn es woanders keine Erwähnung findet; siehe Bemerkung am Anfang des Kapitels Feinentwurf), besitzt diese Klasse auch einen Logger. Weil es sich aber nur um eine Hilfsklasse handelt, wird er an dieser Stelle aber nicht eigens instantiiert, sondern wird gemeinsam mit den anderen Attributen über den Konstruktor von außen gesetzt.

Neben dem Konstruktor besitzt Sender nur noch die **run**-Methode. Dort werden einfach fortwährend alle Messages in der Queue outgoingMessages nacheinander abgerufen und nach Möglichkeit versendet.

NetworkingEndpoint

NetworkingEndpoint ist ein Interface, das von Klassen implementiert werden muss, die den **NetworkManager** benutzen. NetworkingEndpoints sind also Repräsentationen von Clients. Das Interface enthält zwei Getter-Methoden: **getAccessLevel** und **getProtocolVersion**. Ein **AccessLevel** ist hierbei im Grunde eine Unterscheidung zwischen Spieler und Beobachter. Außerdem enthält das Interface die drei Methoden **onServerMessage**, **onConnectionClosed** und **onException**. **onServerMessage** wird vom **NetworkManager** des Clients aufgerufen, wenn er vom Server eine Nachricht empfangen hat. Die Nachricht wird dabei als Parameter übergeben. **onConnectionClosed** wird aufgerufen, wenn der **NetworkManager** ein Verbindungsende feststellt. **onException** wird aufgerufen, wenn beim **NetworkManager** eine Exception auftritt, die er nicht selbst lösen kann. Diese Methode nimmt eine Nachricht und ein **Throwable** entgegen.

SpectatorEndpoint

SpectatorEndpoint ist ein Interface, das von *NetworkingEndpoint* erbt. Es implementiert die geerbten Methoden **getAccessLevel** und **onServerMessage** als default methods. Ebenso per default wird eine weitere Methode **onSpectatorMessage**, die eine Spectator-ServerMessage entgegen nimmt, also eine Nachricht, die vom Server geschickt und von Beobachtern empfangen werden kann. Da es sich um einen Sonderfall von ServerMessages handelt, wird diese Methode von **onServerMessage** aufgerufen. Dann beschreibt das Interace SpectatorEndpoint noch für jeden SpectatorServerMessage Typen weitere (nicht implementierte) Methoden (z.B. **onGameInitMessage**), die dann wiederum aus **onSpectatorMessage** aufgerufen wird, wenn der entsprechende Nachrichtentyp vorliegt.

PlayerEndpoint

Das Interface PlayerEndpoint ist ebenfalls ein *NetworkingEndpoint*. Da ein Spieler alles können muss, was ein Beobachter auch kann, erbt es aber nicht direkt von **NetworkingEndpoint** sondern von *SpectatorEndpoint*. Die Methoden **getAccessLevel** und **onServerMessage** werden von PlayerEndpoint selbst als default methods implementiert. Weiterhin implementiert es als default die Methode **onPlayerMessage**, die **PlayerServerMessages** entgegennimmt. Wie **onSpectatorMessage** wird diese Methode von **onServerMessage** aufgerufen, wenn dieser Nachrichtentyp vorliegt. Eine weitere (nicht implementierte) des Interfaces, **onPlayerHandMessage**, wird wiederum von **onPlayerMessage** aufgerufen, wenn eine **PlayerHandMessage** vorliegt.

AccessLevel

AccessLevel ist eine Enumeration, die im Grunde schlicht zwischen den Rollen Beobachter und Spieler unterscheidet, indem es ihnen wiederum Werte aus dem Enum **ClientRole** zuweist. Diese können ja zum Beispiel verschiedene Nachrichtentypen verschicken, sollen also auf unterschiedliche Dinge zugreifen können. Da aber ein Spieler alles kann, was ein Beobachter auch kann, bekommen Beobachter das AccessLevel 1, „SPECTATOR“ zugewiesen und Spieler das AccessLevel 2, „USER“, wobei 1 in 2 enthalten ist. Das

Enum speichert das Zugriffslevel in Zahlenform als Objektvariable numericLevel, die über den Konstruktor gesetzt wird. Die Methode **getClientRole** gibt dieses Level dann in Form eines **ClientRole**-Werts aus. Schließlich gibt es noch die Methode **includesLevel**, die schlicht zwei AccesLevel anhand ihrer numericLevel vergleicht und sagt, ob ein Level das andere beinhaltet.

3.3.2 Klassendiagramme

Siehe folgende Seiten.

util

util

networking

networking

util

util

game

messages

messages

util

util

constants

constants

gamemap

gamemap

game

game

player

player

gameboard

gameboard

cards

cards

json

json

constant

constant

constant

<<Enumeration>>

CardType

ERROR
TURN_180
TURN_LEFT
TURN_RIGHT
BACKWARD_1
FORWARD_1
FORWARD_2
FORWARD_3

+CARD_TYPES_COUNT: int

-PRIORITY: int

~CardType(PRIORITY: int)

+getPriority()

Constants

+PROTOCOL_VERSION: String

+SERVER_UNREACHABLE: String

+VALUE_NOT_VALID: int

+VALUE_EMPTY: int

+NUMBER_OF_CARDS_IN_PLAYER_HAND: int

+NUMBER_OF_CARDS_IN_PROGRAM: int

+NUMBER_OF_LIVES: int

+MILLISECONDS_PER_SECONDS: long

+STANDARD_PORT: int

+TILE_NOT_ON_MAP: Tile

-Constants()

<<Enumeration>>

MessageErrorType

ERROR
GAME_ABORT
RULE_VIOLATION
DISQUALIFICATION
UNSUPPORTED_PROTOCOL_VERSION
PROTOCOL_VIOLATION

<<Enumeration>>

EnumName

LOGIN
LOGIN_RESPONSE
GAME_CFG
LOBBY_STATUS
MESSAGE
GAME_INIT
CARD_SET
CARD_SELECTION
ROUND_RESULT
PAUSE
RESUME
ERROR

-PARSESELECT: Map<String, Class<?>

Message>

-COMPOSESELECT: Map<Class<?>

Message>

+getParseSelect()

+getComposeSelect()

<<Enumeration>>

Orientation

ERROR
UP
RIGHT
DOWN
LEFT

+getNextTile (gameBoard: GameBoard, robot: Robot)

+isWall (gameBoard: GameBoard, robot: Robot)

+getDirection (orientation: Orientation)

+rotate (rotate: Rotate)

+turn180()

game

gameboard

GameBoard

```
#WIDTH: int
#HEIGHT: int
#BOARD: Tile[][]
#CHECKPOINTS: List<Tile>
-MINPLAYERS
-MAXPLAYERS
-MAXCARDSELECTTIME
+GameBoard(GAMEMAP: GameMap)
#setRobotTile(ROBOT: Robot, ROTATE: Rotate)
#letRobotRotate(ROBOT: Robot, ROTATE: Rotate)
#letRobotReachCheckpoint(ROBOT: Robot, CHECKPOINT: int)
#letRobotSpawn(ROBOT: Robot, ORIENTATION: Orientation, TILE: Tile)
#letRobotDie(ROBOT: Robot)
+getTile(X: int, Y: int)
+getMinPlayer()
+getMaxPlayer()
+getMaxCardSelectTime()
+getRobot(UID: int)
```

LogicGameBoard

```
-winner: int
-COLLISIONUIDLIST: List<Integer>
-PUSHEDUIDLIST: List<Integer>
-UIDROBOTMAP: Map<Integer, Robot>
-CARDTYPEFUNCTIONMAP:
Map<CardType, Function<Robot,
List<RoundResultEvent>>>
+LogicGameBoard(GAMEMAP: GameMap)
+calculateRoundResultEvent(PLAYERCAR
DTUPLE: PlayerCardTuple)
-turnRoutine(ROBOT: Robot, ROTATE: Rotate)
-turn(ROBOT: Robot, ROTATE: Rotate)
-stepRoutine(ROBOT: Robot, BACKWARD: boolean)
-step(ROBOT: Robot, BACKWARD: boolean)
-checkCollisionen(ROBOT: Robot, ORIENTATION: Orientation)
-checkPushRobot(ROBOT: Robot, ORIENTATION: Orientation)
-checkpointReachedRobot(ROBOT: Robot)
+destroyRobot(ROBOT: Robot)
+diedRobot(ROBOT: Robot)
+spawnRobot(UID: int, ORIENTATION: Orientation)
+disqualifyRobot(UID: int)
-Mitgliedsname
-initCardTypeFunctionMap()
+getWinner()
+addPlayer(UID: int)
```

player

Client

```
-NAME: String
-CLIENTROLE: ClientRole
-AI: Bool
-UID: int
+Client(NAME: String, CLIENTROLE: ClientRole, AI: Bool, UID: int)
+getName()
+getClientRole()
+getUid
+getAi
```

Robot

```
#orientation: Orientation
#tile: Tile
-lastReachedCheckpointTile: Tile
-onMap: boolean
-lives: int
#uid: int
+Robot(ORIENTATION: orientation, TILE: Tile, LASTREACHEDCHECKPOINT TILE: Tile, UID: int)
+getUid()
+decrementLives()
+getLives()
+setLives(LIVES: int)
```

TooManyPlayersException

```
+TooManyPlayersException()
+TooManyPlayersException(msg: String)
```

Erbt von
Exception

cards

Card

```
-CARDTYPE: CardType
-CARDPRIORITY: int
+Card(CARDTYPE: CardType, CARDPRIORITY: int)
+getCardType()
+getCardPriority()
```

PlayerHand

```
-cards: Set<Card>
-program: Programm
~PlayerHand(cards: Set<Card>)
+validateProgram(program: Programm): boolean
+createPlayerHands(numberOfPlayers: int): List<PlayerHand>
```

PlayerCardTuple

```
-UID: int
-CARD: Card
+PlayerCardTuple(uid: int, card: Card)
+getUid()
+getCard()
```

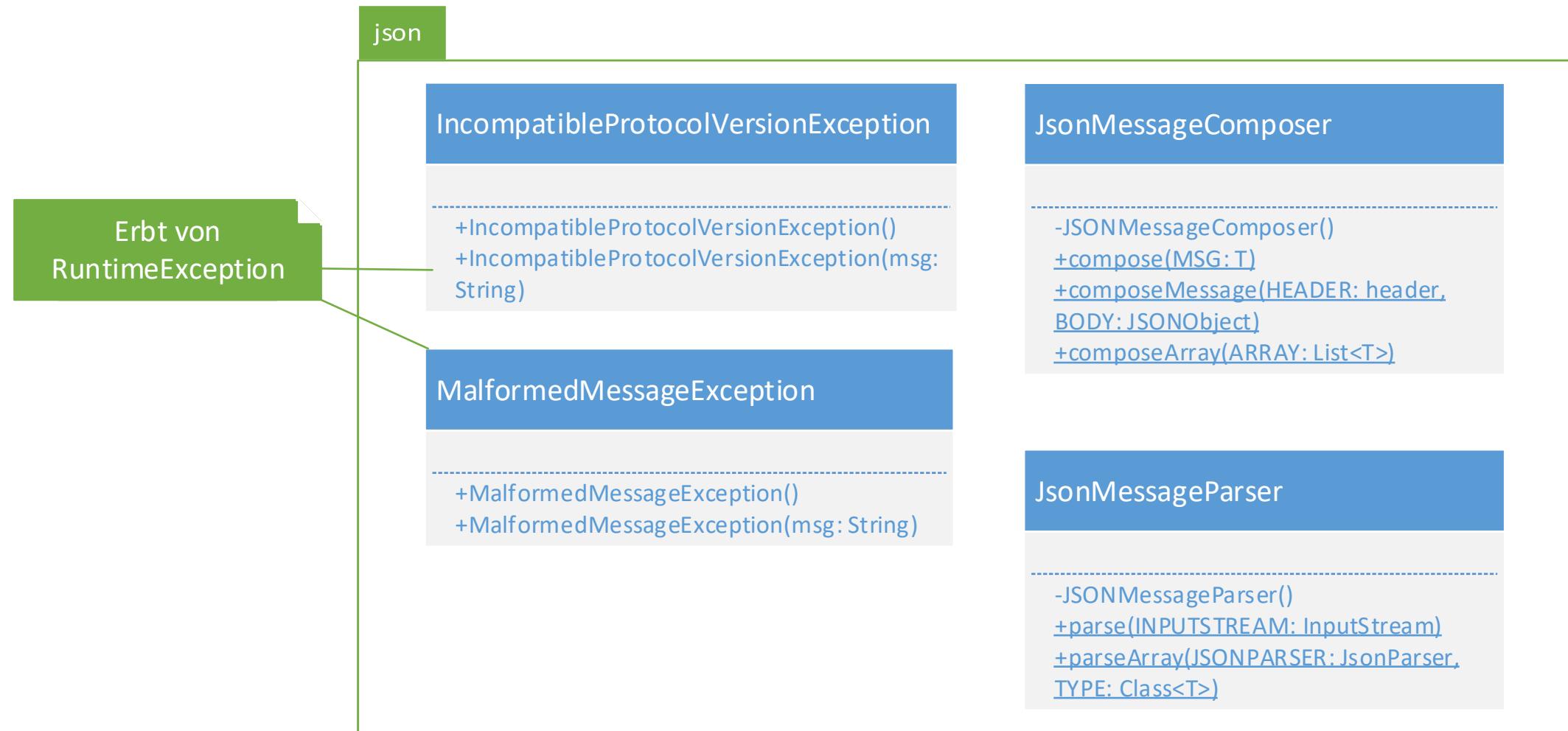
IllegalNumberOfCardsException

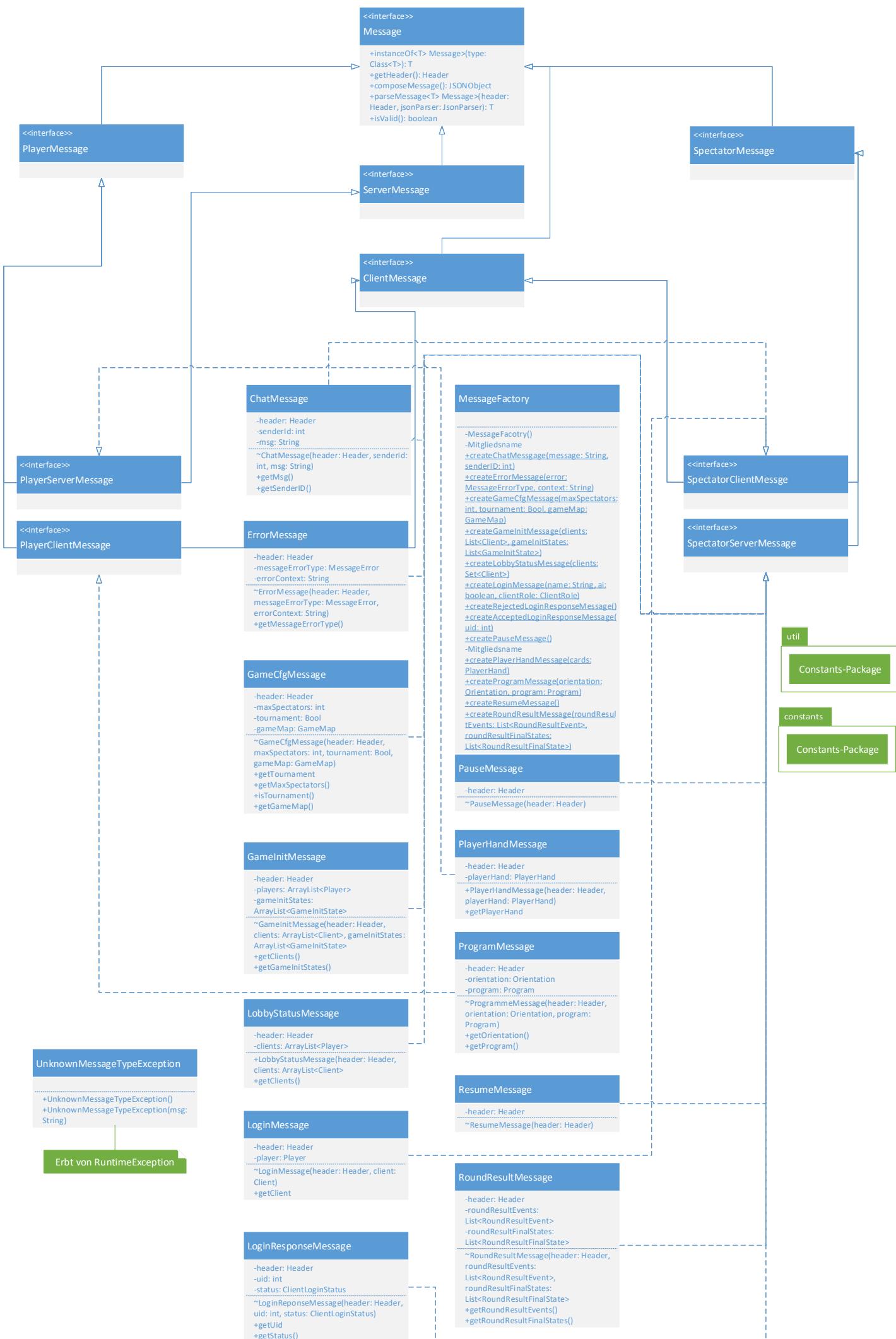
```
+IllegalNumberOfCardsException
+IllegalNumberOfCardsException(msg: String)
```

Program

```
-cards: List<Card>
+Program(cards: List<Card>)
```

Implementiert
IllegalArgumentException





util



constants

<<Enumeration>>

Bool

```

ERROR
TRUE
FALSE
+valueOf(bool: boolean)
+getBoolean()

```

<<Enumeration>>

ClientLoginStatus

```

ERROR
ACCEPTED
REJECTED

```

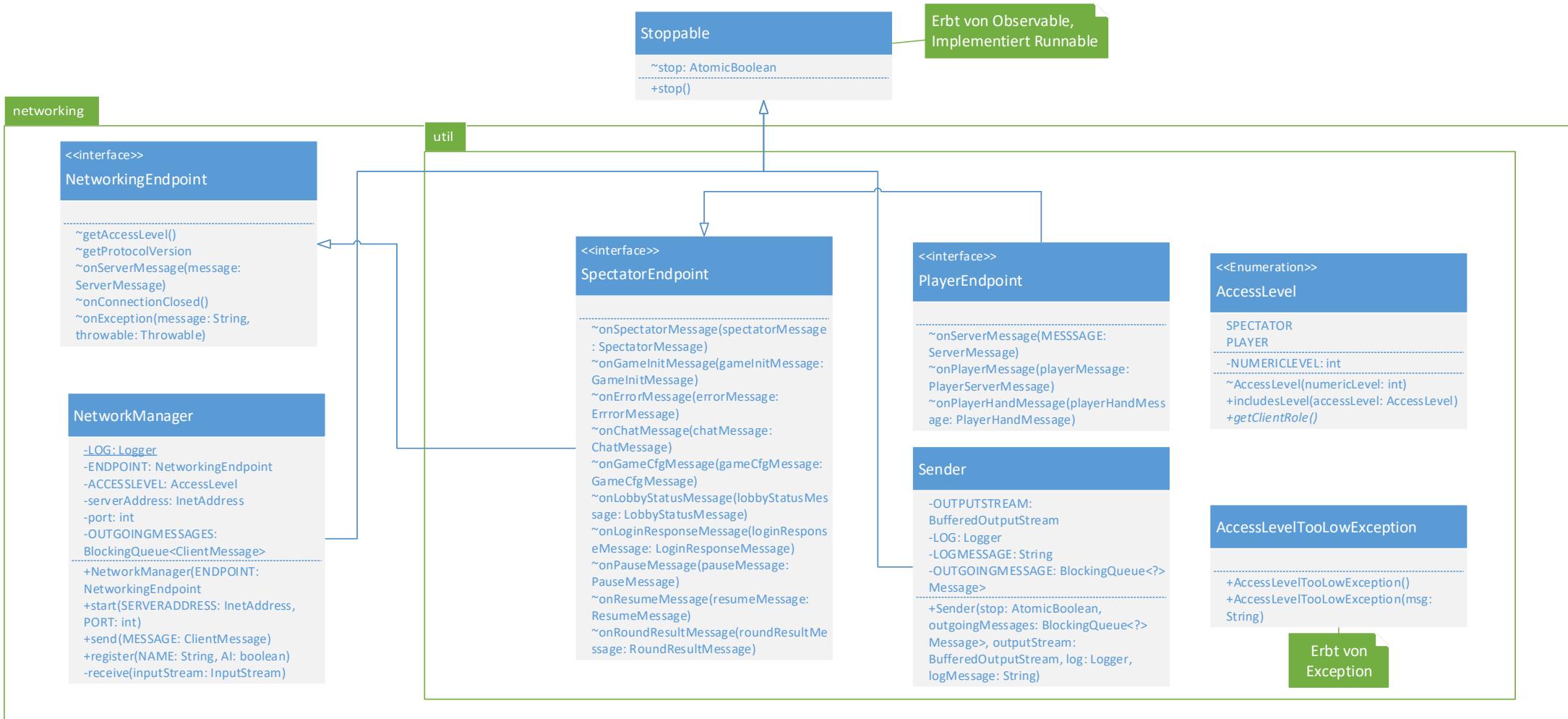
<<Enumeration>>

ClientRole

```

ERROR
USER
SPECTATOR

```



3.3.3 Kommunikationsdiagramm

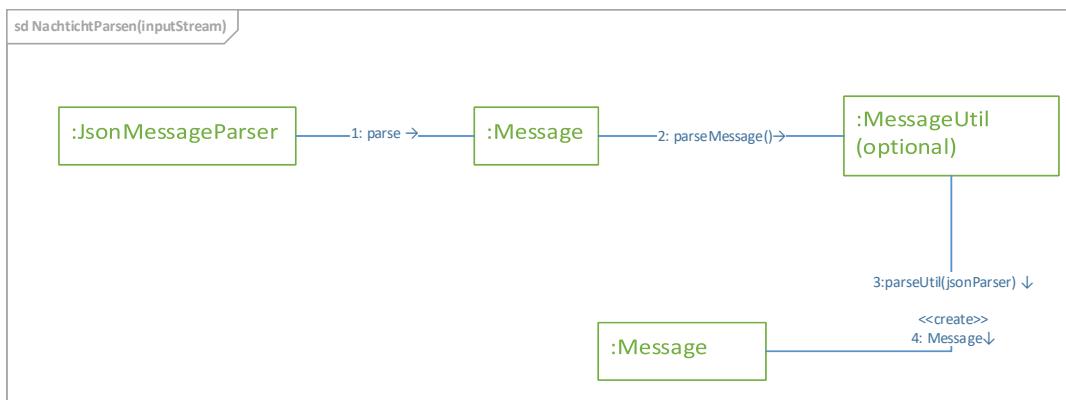


Abbildung 3.6: Util - Kommunikationsdiagramm 01

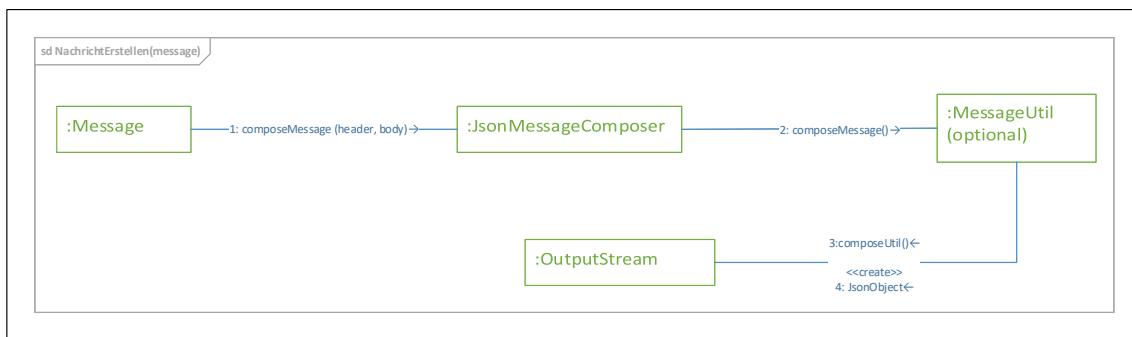


Abbildung 3.7: Util - Kommunikationsdiagramm 02

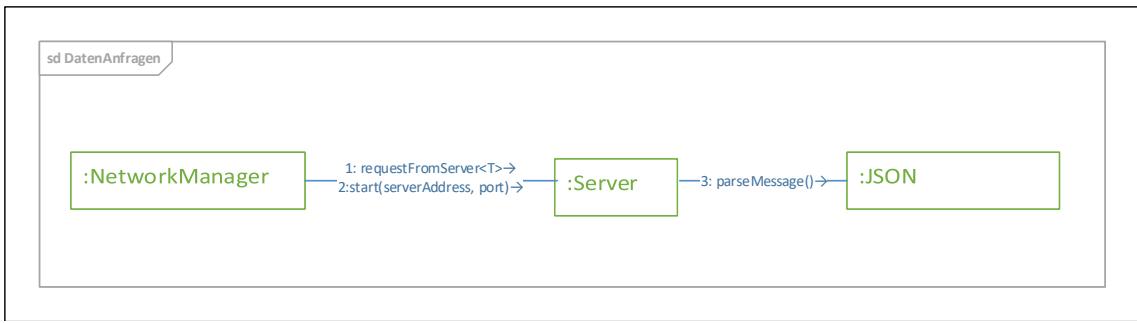


Abbildung 3.8: Util - Kommunikationsdiagramm 03

3.4 Beobachter

3.4.1 Beschreibung der Methoden

Für die Realisierung des Beobachters benötigt man natürlich alle Klassen aus der Komponente Beobachter und noch einige Klassen aus der Komponente Util.

Main

Innerhalb der Beobachterkomponente gibt es die Klasse Main, die eine **main-Methode** enthält und lediglich dazu dient, das Programm ausführbar zu machen. Die main-Methode erstellt und startet eine Instanz der Klasse ExchangePoint, der den Beobachter letztlich verwaltet.

HardReset

Diese Klasse besitzt eine eigene **main**-Methode und ist besonders nützlich, wenn im Beobachter ein Fehler aufgetreten ist. Da die Anwendung zum Beispiel Einstellungen nicht nur temporär, sondern für die Dauer mehrerer Programmaussführungen im Computerspeicher speichert, kann es passieren, dass die Anwendung nach einem Absturz auch nach einem Neustart nicht wieder funktioniert. Das Ausführen der **main**-Methode aus Hardreset leert diesen Speicher wieder und sollte das Problem somit beheben.

ExchangePointImpl

ExchangePoint implementiert **SpectatorEndpoint**, ist also die Schnittstelle des Beobachters zum Netzwerk und benutzt einen **NetworkManager**. Weiterhin besitzt ExchangePoint eine **Engine**, ein **Nifty**, ein **AtomicInteger uid**, ein **VisualGameboard gameBoard**, eine **GameLobby**, einen **IconProvider**, eine **ConcurrentMap uidClient-Map**, eine **GameMap**, ein **AtomicBoolea** connectionClosedExpected, eine **LinkedList visitedScreens**, ein **Object monitor**, ein **RecordedGame** und eine **LobbyStatusMessage delayedlobbyStatus**. ExchangePoint ist auch die Klasse, die von **Main** instanziert und gestartet wird. Er ist gewissermaßen die unterste verwaltende Instanz des Beobachters.

ExchangePoint besitzt eine statische getInstance Methode statt eines öffentlichen Konstruktors, die anhand der statischen Klassenvariable instance sicher stellt, dass nur eine Instanz von ExchangePoint existiert. Die **start**-Methode der Klasse startet die eigene **Engine** in einem neuen Thread. Die **connectToServer**-Methode ist dafür zuständig, den NetworkManager zu starten und mit seiner Hilfe eine Verbindung zum Server herzustellen. Dann implementiert die Klasse, die vom Interface **SpectatorEndpoint** vorgegebenen Methoden, die bestimmen, welche Nachrichten wie behandelt werden. **onEventFinished** dient dazu, nach einer Animation eines Spielzugs die dazugehörige Eventnachricht auf den Spielbildschirm des Beobachters zu schreiben. Die Methoden **getClients** bzw. **getClient** geben alle bzw. einen Client anhand dessen User-ID zurück. Die **reset**-Methode schließt die Verbindung zu einem Server und setzt den Beobachter auf den Ausgangszustand zurück. **quit** beendet die Anwendung, indem es alle Verbindungen und Threads des Beobachters ordnungsgemäß schließt. **gotoScreen** wechselt den Dialog, der dem Beobachter aktuell angezeigt wird. **goBack** tauscht den Dialog, der dem Beobachter aktuell angezeigt wird, wieder auf den davor angezeigten Dialog zurück.

IconProvider

Das Interface IconProvider wird von allen Klassen implementiert, die Spieler-Icons nutzen. Es bietet eine Methode **getIconPath**, die zu der User-ID eines Spielers den Pfad zur passenden Datei liefert, die das Icon enthält, das den Spieler grafisch darstellt.

GameLobby

Die Klasse GameLobby ist dafür zuständig, den Inhalt der Lobby zu steuern, also das, was dem Spieler angezeigt wird, nachdem er sich an einem Server angemeldet hat und bevor das Spiel beginnt. Sie speichert eine **Hashmap** robotTypeMap, die jedem Spieler nach der User-ID einen Robotertyp zuweist (alle Robotertypen sind zu finden im Enum **RobotType**). Die Methoden **addPlayer** und **removePlayer** fügen dieser Map User-Ids unter Zuordnung eines Robotertyps hinzu bzw. entfernen sie. Diese Methoden werden vom **ExchangePoint** aufgerufen, wenn er eine entsprechende **LobbyStatusMessage** erhält. Weiterhin besitzt die Klasse die Methode **createGameBoard**, die vom **ExchangePoint** auf Erhalt einer GameInitMessage aufgerufen wird, also wenn der Aufenthalt in der Lobby beendet und das eigentliche Spiel gestartet wird. Die Methode erstellt ein neues **VisualGameBoard** mit einer übergebenen GameMap und initialisiert es mit der gespeicherten robotTypeMap und einem übergebenen Parameter playerStatusList. Im Übrigen implementiert die Klasse **IconProvider**.

VisualGameBoard

VisualGameBoard erbt von der abstrakten Klasse **GameBoard** aus dem Util-Package. Während **LogicGameBoard** Methoden anbietet, um logische Operationen auf dem Spielfeld vorzunehmen, besitzt VisualGameBoard Methoden, um Ereignisse im Spielgeschehen darszustellen. Die meisten Methoden hierzu sind private, werden hier also nicht genauer ausgeführt. Sie werden von den public Methoden **executeRoundResultEvent** und **init** aufgerufen. **init** sollte hierbei bei Spielbeginn von außen aufgerufen zu werden, um die Roboter aller Spieler im Anfangszustand darstellen zu können. **executeRoundResultEvent** übernimmt die gesamte Visualisierung eines **RoundResultEvents**. Eine weitere öffentliche Methode ist **executeFinalStates**, die die Roboter auf dem Spielfeld entsprechend der **FinalStates** setzt, die am Ende jeder Runde an die Clients verschickt werden. Dies sollte im Normalfall nichts an dem Zustand ändern, der ohnehin zu diesem Zeitpunkt auf dem Spielfeld vorliegt; diese Maßnahme soll lediglich Fehler korrigieren, die im zuvorigen Rundenverlauf aufgetreten sein könnten. Außerdem gibt es die öffentliche Methode **updateRobots**, die die **update**-Methoden aller Roboter auf dem Spielfeld aufruft. Im Übrigen implementiert die Klasse **IconProvider**.

VisualRobot

VisualRobot erbt von der Klasse **Robot** aus dem Util-package und erweitert diese um Funktionalität, um Roboteraktionen grafisch darstellen zu können. So werden einige Robot-Methoden überschrieben und erweitert. Zusätzliche Methoden sind **rotate**, **move**, **stopMovement** und **update**. Die Methoden **rotate**, **move** und **stopMovement** dienen dazu, Drehungen beziehungsweise Bewegungen eines Roboters darzustellen.

ChatModel

Diese Klasse ist für das Verwalten von Chats zuständig. Nifty GUI besitzt bereits eine fertiges Interface **Chat**, dessen Instanz **ChatControl** wir verwenden können. Da ChatControl aber ein Controller für genau ein Nifty-**Element** ist und jedes Element nur auf einem **Screen** existiert, muss man für jeden Dialog, in dem man ein Chatfenster zur Verfügung stellen will, einen eigenen **ChatControl** haben. Wir möchten das Chatten sowohl auf dem Spielbildschirm als auch in der Lobby ermöglichen. Also gibt es mehrere **ChatControl**-Instanzen. Um diese bei Ereignissen wie eintreffenden Chat-Nachrichten nicht alle einzeln ansprechen zu müssen, existiert die Klasse **ChatModel**. Die Methoden **receiveMessage**, **addClient** und **removeClient** rufen einfach passende Methoden des **Chat**-Interfaces für alle Chat-Instanzen auf. Weitere Methoden von ChatModel sind **addChat**, die einen Chat zu ChatMap hinzufügt und dabei dem Chat alle Clients hinzufügt und alle Nachrichten schickt, die die anderen Chats schon bekommen haben, **toggleVisible**, die alle Chats auf einmal abhängig von der isVisible-Variable sichtbar oder nicht sichtbar schaltet, und **reset**, die alle Chats auf den Ausgangszustand zurücksetzt. Außerdem gibt es die Methode **handleUserInput**, die von Nifty aufgerufen wird, wenn ein Client eine Chatnachricht abschickt.

ChatModel besitzt eine statische getInstance Methode statt eines öffentlichen Konstruktors, die anhand der statischen Klassenvariable instance sicher stellt, dass nur eine Instanz von ChatModel existiert. ChatModel speichert in einem boolean isVisible, ob Chats gerade sichtbar geschaltet ist oder nicht. In der Objektvariable chatMap - eine Hashmap, die Nifty-**Chat**-Objekten **ScreenElementPairs** zuordnet - ist dann schließlich gespeichert, welche Instanz einer **Chat**-Implementierung zu welchem Dialog und zu welchem Element daraus gehört.

ConsoleModel

Im Beobachter kann man von jedem Dialog aus eine Konsole öffnen, mit deren Hilfe man die Netzwerkkommunikation (zu debug-Zwecken) besser nachvollziehen kann. Man kann von hier aus das Log-Level einstellen, also welche Art von **Logger-Ausgaben** hier ausgegeben werden sollen. Sonstige Befehle kann man von hier aus nicht geben, es wird lediglich angezeigt, welche Nachrichten zwischen Server und Client ausgetauscht werden. Ähnlich wie **ChatModel** fasst die Klasse **ConsoleModel** die Konsolen auf allen **Screens** zusammen, damit deren Verhalten nicht Verhalten nicht einzeln erzeugt werden muss. Auch ConsoleModel besitzt eine **getInstance** statt eines öffentlichen Konstruktors, damit nicht mehrere Instanzen existieren können. Alle **Consolen** des Objekts sind in der Variable consoleMap gespeichert und einem **ScreenElementCommandTuple** zugeordnet. Ebenfalls besitzt die Klasse eine **addConsole**-Methode, die consoleMap eine neue Konsole hinzufügt und sie dabei auf den Stand der bisher in der Liste stehenden Konsolen bringt, und eine **toggleVisible**-Methode, die die Sichtbarkeit aller Konsolen an- bzw. abschaltet. Zudem gibt es noch eine **writeToConsole**-Methode, die von Nifty benutzt werden kann, um einen Text in alle Konsolen anzuzeigen. Uch ConsoleModel besitzt eine statische getInstance Methode statt eines öffentlichen Konstruktors, die anhand der statischen Klassenvariable instance sicher stellt, dass nur eine Instanz von ChatModel existiert.

EventListModel

EventListModel steuert die Anzeige auf dem Spielbilschirm, die alle Events im Spiel (also z.B. SpielerX hat Checkpoint Y erreicht, Spieler Z wurde zerstört und so weiter) schriftlich anzeigt. Um dieses Eventfeld zu realisieren, wird eine nifty-**ListBox** benutzt. Alle ListBoxes, die von EventListModel gesteuert werden sollen, sind zusammen mit dem zugehörigen nifty-**Element** in der listBoxElementMap gespeichert. Da aber eigentlich nur auf dem Spielbildschirm diese Eventnachrichten eingeblendet werden und sonst kein anderer **Screen** solch ein Eventfeld besitzt, sollte diese Map in der Regel nur einen Eintrag besitzen.

Auch EventListBox besitzt eine statische getInstance Methode statt eines öffentlichen Konstruktors, die anhand der statischen Klassenvariable instance sicher stellt, dass nur eine Instanz von ChatModel existiert. Weiterin gibt es die Methoden **addEvent**, mit deren Hilfe Nifty **EventListBoxItems** in dieses Eventfeld schreiben kann, **addAllEvents**, die dasselbe für mehrere Items macht, und auch hier die Methoden **toggleVisible**, **addListBox** und **reset**, welche die Sichtbarkeit der Anzeige einstellen, neue ListBoxes korrekt hinzufügen und die ListBoxes wieder in den Ausgangszustand zurücksetzen.

PlayerStatusModel

Ebenso wie die Event-Liste gibt es im Spiel eine Anzeige, welche alle Spieler mit einigen zusätzlichen Informationen auflistet (siehe **PlayerStatus**). Unter diesen Informationen ist das einzige, was sich während dem Spiel ändern kann die erreichten Checkpoints und die Leben der Spieler. Um diese Informationen aktuell zu halten, gibt es die Methoden **destroy** und **checkpointReached**. Wenn ein entsprechendes Ereignis eintritt, werden sie aufgerufen und aktualisieren die Angaben in der Anzeige.

EventListBoxItem

Auf dem Spielbildschirm gibt es eine Anzeige, die Ereignisse im Spielgeschehen in schriftlicher Form festhält. Solche Ereignisse sind z.B. SpielerX hat Checkpoint Y erreicht, Spieler Z wurde zerstört und so weiter. Jedes Ereignis wird durch eine Instanz der Klasse EventListBoxItem repräsentiert. Diese speichert nur einen eventString ab, in dem eben jene Beschreibung des jeweiligen Events steht. Der eventString wird im Konstruktor erzeugt, dem ein **RoundResultEvent** übergeben wird, aus welchem dann die Informationen für eventString gezogen werden. Zusätzlich sorgt der Konstruktor noch dafür, dass eine besondere Anzeige aufgerufen wird, sollte das eingegebene RoundResultEvent ein win-Event sein. Den eventString erhält man von außen über die **toString**-Methode der Klasse.

PlayerStatus

Dies ist eine Hilfsklasse, die ein paar wesentliche Informationen über einen Spieler zusammenfasst. Diese Informationen wären die User-ID uid, ein String icon, ein boolean ai, ob es sich beim Spieler um eine KI handelt, die Anzahl der verbleibenden Leben lives und die Nummer des höchsten Checkpoints checkpoint, den der Spieler bisher in richtiger Reihenfolge erreicht hat. uid, name, icon und ai können über den Konstruktor gesetzt werden (checkpoint und lives sind initial immer gleich gesetzt) und für alle Variablen existieren Getter-Methoden. Zusätzlich zum Konstruktor besitzt die Klasse die Methode **createMultiple** die mit Hilfe eines Sets aus **Clients** gleich mehrere PlayerStatus erzeugt.

Außerdem gibt es die Methoden **destroy** und **checkpointedReached**, die beim Aufruf schlicht die Variablen `lives` und `checkpoint` verringern bzw. erhöhen.

ScreenElementPair

Dies ist eine Hilfsklasse, die zwei Nifty-Objekte **Screen** und **Element**, die miteinander assoziiert werden sollen, zusammenfasst. Für weitere Erleuterungen zu diesen Klassen siehe deren JavaDoc.

ScreenElementCommandsTuple

Diese Hilfsklasse ist eine Erweiterung von **ScreenElementPair** und speichert einen **Screen** und ein **Element** noch mit Nifty-**ConsoleCommands** zusammen. Für weitere Erleuterungen zu diesen Klassen siehe deren JavaDoc.

CountdownUpdate

Diese Klasse kümmert sich um den Inhalt der Countdown-Anzeige auf dem Spielbildschirm. Auch `EventListBox` besitzt eine statische `getInstance` Methode statt eines öffentlichen Konstruktors, die anhand der statischen Klassenvariable `instance` sicher stellt, dass nur eine Instanz von `CountdownUpdate` existiert. Weiterhin besitzt `CountdownUpdate` eine **update**-Methode, die vom **IntervalUpdater** aufgerufen wird, und damit die Countdown-Anzeige auf den neuesten Stand bringt. Dann besitzt `CountdownUpdate` noch die öffentlichen Methoden **start**, **pause** und **unpause**, deren Funktion sich am Namen erschließt. Schließlich besitzt die Klasse noch eine Methode **clear**, die den Inhalt aus dem Countdown-Label entfernt.

LobbyChangeUtil

Hilfsklasse, die eine statische Methode anbietet, um das Bearbeiten einer **LobbyStatusMessage** zu vereinfachen. Da die Klasse nicht instanziert werden soll, wird der Standardkonstruktor durch einen leeren, privaten Konstruktor versteckt. Die Methode, die `LobbyChangeUtil` zur Verfügung stellt heißt **executeLobbyChanges** und nimmt eine **LobbyStatusMessage** und zwei Listen aus **Consumern** `removeOperations` und `addOperations` entgegen. Die Aufgabe der Methode ist es dann, diejenigen Spieler der Lobby hinzuzufügen oder daraus zu entfernen, dass es der `LobbyStatusMessage` entspricht.

CustomDisplayMode

`CostumDisplayMode` ist eine Klasse, die im Grunde nur dafür zuständig ist, ein **DisplayMode** zu enthalten und dafür die zusätzliche Methoden **toString**, **equals** und **hashCode** anzubieten, da z.B. eine **toString**-Methode in **DisplayMode** nicht vorhanden ist. Für zusätzliche Informationen siehe Java Dokumentation zu **DisplayMode**.

NumberSettings

Dies ist eine Wrapper-Klasse für `Integer`. Sie bietet eine **toString**-Methode für die entsprechende Zahl und ist damit geeignet für Werte in Einstellungen oder in den **Preferences**. Die **toString**-Methode stellt dann die Zahl zusammen mit einem variablen Suffix und Präfix dar. Suffix und Präfix können wahlweise im Konstruktor übergeben werden. Dafür gibt es insgesamt vier verschiedene Konuktoren. Weiterhin gibt es für den gespeicherten `Integer` noch einen Getter.

NiftyConsoleAppender

NiftyConsoleAppender implementiert erbt von der abstrakten Klasse **AppenderSkeleton**. Unser Beobachter besitzt eine integrierte Konsole in der GUI. Um diese Konsole nutzen zu können, das heißt mithilfe des Loggers Text hineinschreiben, wird die Klasse NiftyConsoleAppender benötigt. Sie bietet hierfür die Methode **append** an, die ein **LoggingEvent** entgegennimmt und der **writeToConsole**-Methode des **ConsoleModels** übergibt.

IoUtil

Diese Klasse wird in unserer Anwendung bisher nicht benutzt und ist auch nicht implementiert, zur Vollständigkeit soll sie hier aber aufgeführt werden. Sie stellt eine (bisher leere) Methode **saveRecordedGameToFile** zur Verfügung, die dafür gedacht ist, ein **RecordedGame** in eine Datei auf dem Computer abzuspeichern, sodass aufgezeichnete Spiele nicht verloren gehen, wenn man die Anwendung schließt.

RecordedGame

Im Lastenheft ist als optionales Feature für das Spiel angegeben, eine Record-Funktion zu implementieren, mit der man Spiele aufzeichnen und hinterher nochmals ansehen kann. Um ein Spiel aufzunehmen steht die Klasse RecordedGame zur Verfügung. Sie speichert eine Liste aus **RoundResultMessages** ab und besitzt eine einfache Methode **addRoundResultMessage**, die eine neue, übergebene **RoundResultMessage** in diese Liste einfügt. Diese Methode wird in jeder Runde vom **ExchangePoint** aufgerufen. Allerdings ist im Beobachter bisher noch keine Möglichkeit implementiert, diesen Record dann hinterher auch wiederzugeben.

Engine

Diese Klasse bildet eine Schnittstelle zur JMonkey-Spieleengine. Sie implementiert das Interface **SimpleApplication**, welches Klassen definiert, die Basen für jME3-Anwendungen darstellen. Damit ist Engine für die Visualisierung von **RoundResultEvents** sowie für alle sonstigen visuellen und audioellen Ausgabe der Anwendung zuständig. Neben den Methoden, die dem Interface entstammen und von JMonkey aufgerufen werden sollen, gibt es folgende Methoden: **addEventToQueue**, nimmt ein **RoundResultEvent** entgegen und fügt es der Objektvariable events, eine Liste von **RoundResultEvents**, die noch zur Visualisierung ausstehen, hinzu. **toggleStats** schaltet die Sichtbarkeit von mehreren Statistiken hin und her. **toggleFps** schaltet die Sichtbarkeit der Fps-Anzeige hin und her. **reshapeNiftyJmeDisplay** informiert den **NiftyJmeDisplay**, dass die Auflösung des Spiels geändert wurde. **addModelList** fügt ein neues **Spatial** in die Objektvariable listOfAllModels (eine Liste von **Spatials**) ein. Und **reset** schließlich löst alle **Spatials** aus listOfAllModels von ihren Eltern-Knoten und leert die Liste schließlich, sodass ein neues Spiel gestartet werden kann.

Paths

Damit Elemente auf dem Spielfeld angezeigt werden können, müssen sie auf Ressourcen-Dateien zugreifen, in denen gespeichert ist, wie sie aussehen sollen. Die Klasse Paths ist eine Zusammenfassung mehrerer Pfade, die zu solchen Ressourcen führen. Elemente,

die eine Instanz von Paths besitzen, sind zum Beispiel die Roboter oder die Nummern, durch die die Checkpoints symbolisiert werden. Unterschiedliche Elemente benötigen unterschiedlich viele Ressourcen und damit unterschiedlich viele Pfade. Deswegen gibt es mehrere Konstruktoren, die nur Teilmengen der in Paths verfügbaren Pfade setzt. Alle Pfade können über Getter gelesen werden.

Environment

Die Environment ist verantwortlich für die grafische Darstellung der Elemente, die nicht spielrelevant sind. Das sind in unserem Fall die Küchenzeile, auf der das Spielbrett aufgestellt ist und der Sternenhimmel im Hintergrund. Der Konstruktor nimmt eine x- und eine y-Koordinate entgegen, die die Dimensionen des Spielbretts verkörpern, denn die Größe des Spielbretts ist natürlich relevant für die Gestaltung der Küchenzeile.

MeshProcessing

MeshProcessing stellt die Methode **getMyGeometry** zur Verfügung, die mit einem **Spatial** aufgerufen werden kann, um es in ein **Geometry** umzuwandeln, das dann zusätzlich geometrische Informationen enthält.

ModelLoader

Die Klasse ModelLoader stellt zwei statische Methoden **loadModelByName** und **loadGameBoardModel** zur Verfügung. Ein model ist hierbei eine Implementierung vom JMonkey-Interface **Spatial**, also der mathematische Unterbau eines Objekts, dass letztlich im Spiel animiert werden kann. **loadModelByName** erzeugt dabei Roboter, Wände oder auch die Küchenelemente unterhalb des Spielbretts, wenn man ihr einen entsprechenden Wert aus der Enumeration **RobotType** übergibt. **loadGameBoardModel** kann das model für das Spielbrett selbst erstellen; ihre Parameter sind Höhe und Breite des Spielbretts.

MaterialCreator

Diese Klasse stellt zwei statische Methoden **createMaterial** und **createUnlitMaterial** zur Verfügung. Ein JMonkey-Objekt **Material** speichert dabei die Farbgebung eines JMonkey-Elements auf dem Spielfeld und ist einem model zugeordnet. **createMaterial** liefert dabei ein **Material**, dass auf Beleuchtung reagiert, kann also glänzen und so weiter. **createUnlitMaterial** liefert dagegen ein **Material**, dass unabhängig von der Beleuchtung immer dieselbe Farbe besitzt.

RobotAnimController

Diese Klasse dient dazu, die Animationen eines Roboters zu kontrollieren. Jeder **Visual-Robot** besitzt demnach einen RobotAnimController. Ein RobotAnimController bekommt im Konstruktor ein **AnimControl** übergeben - eine Klasse, die schon einige Standard-JMonkey-Animationen bereithält. Neben dem Konstruktor hat die Klasse die Methode **setAnim**, die immer von **VisualGameBoard** mit dem entsprechenden **AnimState** aufgerufen wird, wenn ein **RoundResultEvent** ausgewertet wurde. **AnimState** ist dabei eine Enumeration, das alle unterschiedlichen Fälle von Animationen aufführt.

ConnectToServerPopupController

Diese Klasse implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Dieser Controller ist zuständig für das Popup, das erscheint, wenn man sich als Beobachter mit einem Server verbinden möchte. Es gibt zwei Buttons in diesem Popup und dementsprechend auch die beiden Methoden **onConnectButtonClick** und **onAbortButtonClick**, die die Funktionalität hinter diesen Buttons implementieren und von Nifty zu gegebener Zeit aufgerufen werden. Ebenfalls von Nifty aufgerufen werden die Methoden **onIpChanged**, **onPortChanged** und **onNameChanged**. Diese drei Methoden beziehen sich auf die Textfelder im Connect-Dialog, in die der Nutzer seinen Namen und IP und Port des Servers einzutragen hat. Werden diese Einträge verändert werden entsprechend diese drei Methoden ausgeführt. Sie alle rufen jeweils die private Methode **enableConnectButtonIfInputValid** auf, die festlegt, ob der Button „Verbinden“ aktiviert, also klickbar ist. Dies sollte nämlich nur der Fall sein, wenn alle drei Einträge in den Textfeldern valide erscheinen, also einen bestimmten regulären Ausdruck erfüllen. Diese Validitätsprüfung nimmt **enableConnectButtonIfInputValid** also bei jeder Änderung in den Textfeldern vor und disabled oder enabled den Connect-Button entsprechend. Die sonstigen Methoden der Klasse entsprechen dem Interface **Controller**.

ErrorPopupController

Diese Klasse implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Dieser Controller ist zuständig für das Popup, das auftaucht, wenn ein Verbindungsfehler auftritt. Die Funktionalität dieses Popups ist sehr beschränkt, es besitzt nur einen einzigen „Abbrechen“-Button und demnach auch nur eine Methode **onAbortButtonClick**, die das Popup wieder schließt und den Nutzer zum Hauptmenü zurück bringt. Die sonstigen Methoden der Klasse entsprechen dem Interface **Controller**.

PopupMenuController

Diese Klasse implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Dieser Controller ist zuständig für das kleine Menü, das der Nutzer aus dem Spiel heraus aufrufen kann (standardmäßig mit dem Shortcut esc), um von dort aus das Hauptmenü, die Einstellungen oder den Hilfebereich zu erreichen. Für jede dieser Funktionen gibt es einen Button sowie einen weiteren Button, der das Popup schlicht wieder schließt und den Nutzer im Spiel belässt. Die Methoden, die die Funktionalitäten dieser Buttons implementieren, heißen **onClosePopupButtonClick**, **onSettingsButtonClick**, **onHelpButtonClick** und **onBackToMainMenuButtonClick**. Die sonstigen Methoden der Klasse entsprechen dem Interface **Controller**.

QuitPopupController

Diese Klasse implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Dieser Controller ist zuständig für das Popup, das auftaucht, wenn ein Nutzer das Spiel verlassen will und aufgefordert wird, diese Entscheidung zu bestätigen. Dieses Popup besitzt zwei Buttons, mit denen man das Beenden entweder bestätigen oder den Vorgang abbrechen kann. Entsprechend diesen gibt es die beiden Methoden **onAbortButtonClick** und **onQuitButtonClick**, die diese Funktionalitäten implementieren und zur gegeben Zeit von Nifty aufgerufen werden. Die sonstigen Methoden der Klasse entsprechen dem Interface **Controller**.

CustomScreenController

Das CostumScreenController-Interface ist eine Zusammenführung und Teilimplementierung der Nifty-Interfaces **ScreenController** und **KeyInputHandler**. ScreenController dienen im Allgemeinen dazu, die Funktionalität hinter einem **Screen** zu steuern, so muss es für jeden unserer Dialoge eine ScreenController-Implementierung geben. Das Interface **KeyInputHandler** muss dann implementiert werden, wenn man Tastatureingaben lesen will.

Das Interface implementiert die von **KeyInputHandler** geerbte Methode **keyEvent** per default. Dabei wird ausgewertet, ob die Taste, mit der die Methode aufgerufen wurde, zu einem Shortcut gehört (Shortcuts können unter Einstellungen festgelegt werden). Dann wird die Funktion, die der Shortcut steuert, ausgeführt, also meist die Sichtbarkeit eines Dialogs getoggled. Dabei benutzt es ebenfalls im Interface per default implementierte Methoden, die angeben, ob ein bestimmter Dialog, der auf einen Shortcut hört, in diesem Controller überhaupt benutzbar ist (heßen z.B. `isChatEnabled`). Im Default geben diese Methoden alle true zurück.

LobbyScreenController, HelpScreenController & GameScreenController

Diese Klassen gehören zum Lobby-Dialog bzw. zum Help-Dialog. Sie implementieren das Interface **CustomScreenController** korrekt, aber abgesehen davon sind sie leer, denn in der Lobby, im Hilfebereich und während des Spiels gibt es keine Funktionalität, die es zu implementieren gäbe, abgesehen von den global funktionierenden Shortcuts.

MainMenuScreenController

Der MainMenuScreenController implementiert **CustomScreenController** und ist für die Funktionalität hinter dem Hauptmenü zuständig, muss als Menü also in andere Dialoge weiterleiten. Es besitzt für jeden Button im Hauptmenü eine Methode **on...ButtonClick**, die dann jeweils den Dialog aufrufen, der zum entsprechenden Button gehört. Außerdem überschreibt es die vom Interface per default implementierten Methoden **isChatEnabled**, **isPopupMenuEnabled** und **isEventListEnabled** und lässt sie false zurückgeben, da diese Dialoge im Hauptmenü nicht verfügbar sein sollen.

SettingsScreenController

Statt CustomScreenController zu implementieren, implementiert diese Klasse die Interfaces **ScreenController** und **KeyInputHandler** direkt, da Tastatureingaben im Setting-Dialog eine ganz andere Funktion haben als in den bisher beschriebenen Dialogen. Schließlich sollen hier Tastaturbelegungen definiert werden. Die Klasse implementiert also die **keyEvent** von **KeyInputHandler** selbst. Die Methode ignoriert Key-Events, wenn die Objektvariable `listening` nicht true ist. `listening` wird nur von der **onShortcutButtonClick**-Methode der Klasse auf true gesetzt, also immer dann, wenn der Benutzer einen bestimmten shortcut zum Ändern ausgewählt hat. Dann nimmt **keyEvents** die als nächstes gedrückte Taste an, setzt sie als neues Shortcut für das entsprechende Feld und setzt `listening` wieder auf false.

Auch sonst gibt es sehr viele interaktive Elemente im Setting-Dialog, für deren Funktionalität es jeweils eine eigene Methode gibt. Ansonsten implementiert SettingScreenController noch korrekt das **ScreenController**-Interface.

OutputFormatter

Diese Klasse stellt Hilfsmethoden für die Lobby her. In der Lobby sollen schließlich alle Spieler aufgelistet werden. Um dies zu vereinfachen bietet OutputFormatter die statische Methode **clientListToString** an, die eine Liste aus **Clients** entgegennimmt und aus einem einen String zusammenstellt. Diese Methode wird vom **ExchangePoint** aus aufgerufen. Eine weitere statische Methode **clientToString** übernimmt die Aufgabe für einen einzelnen **Client**. Da OutputFormatter nicht instanziert werden soll, wird der implizite Standardkonstruktor von einem leeren, privaten versteckt.

Update

Das Update-Interface kommt dann zum Einsatz, wenn Elemente in der GUI dargestellt werden sollen, die sich über die Zeit verändern können. Es enthält eine **update**-Methode die dann über den **UpdateManager** vom JME/Nifty-Thread aus aufgerufen werden kann. Im Gegensatz zum **IntervalUpdate** werden die **update**-Methoden von einfachen Updates nur bei Bedarf mit dem nächsten Frame aufgerufen. Da Update ein functional Interface ist (beachte Annotation FunctionalInterface), wird es bei uns übrigens nirgends direkt implementiert, sondern nur anonym zusammen mit lamda expressions verwendet.

IntervalUpdate

Wie das Interface **Update** ist das Interface IntervalUpdate für Elemente der GUI wichtig, die sich über die Zeit verändern können. Auch es besitzt eine **update**-Methode, die allerdings ein float „interval“ entgegennimmt (**update** aus **Update** nimmt nichts entgegen). Im Gegensatz zu **Update** wird diese **update**-Methode regelmäßig mit jedem Frame ausgeführt. Der interval-Parameter gibt dabei den Zeitpunkt des Aufrufs mit an, sodass er in der Methode verwendet werden kann. Für das **CountdownUpdate** z.B. ist es wichtig, zu wissen, wie viel Zeit seit dem letzten Aufruf vergangen ist, um die Zeit richtig anzeigen zu können. Abgesehen vom **CountdownUpdate** wird IntervalUpdate bei uns übrigens nirgends direkt implementiert, sondern nur anonym zusammen mit lamda expressions verwendet, da es ein functional Interface ist (beachte Annotation FunctionalInterface).

IntervalUpdater

Der IntervalUpdater ist dazu da, alle **IntervalUpdates** zu verwalten und regelmäßig auszuführen. Er soll nicht instanziert werden, deshalb wird der implizite Konstruktor von einem privaten überdeckt. Alle **IntervalUpdates**, die vom IntervalUpdater verwaltet werden sollen, stehen in der Klassenvariable updateMap, Eine Multimap, die ein Nifty-**Element** mit einem **IntervalUpdate** assoziiert. Um Werte in diese Map zu schreiben beziehungsweise daraus entfernen, gibt es die Methoden **queueUpdate** und **removeUpdate**. Eine weitere statische Methode **update** ist schließlich dazu da, die **update**-Methoden aller **IntervalUpdates** auszuführen. Diese Methode wird gegebenenfalls vom **UpdateManager** aufgerufen.

UpdateManager

Damit sich Elemente in der GUI verändern und bewegen können, müssen sie regelmäßig geupdatet werden. Dafür ist der **UpdateManager** da. Er soll nicht instanziert werden,

deshalb wird der implizite Konstruktor von einem privaten überdeckt. Die Klasse arbeitet auf der statischen Variable `updates`, eine **BlockingQueue** aus **Updates** und stellt drei statische Methoden zur Verfügung: **update**, die von der **Engine** aus aufgerufen wird und die **update**-Methoden aller **Update**-Objekte aus `updates` ausführt, **queueUpdate**, die der der `updates`-Queue ein **Update**-Objekt hinzufügt und **queueUpdateForElement**, die ebenfalls ein **Update** in die Queue schreibt, aber eines, das ein bestimmtes Element betrifft, und dabei alle anderen Updates, die das Element ebenfalls betreffen, aus der Queue schmeißt, damit aktuelle Ereignisse zeitnah sichtbar gemacht werden können.

DimensionUpdater

Manche Elemente der GUI müssen aktualisiert werden, wenn die Auflösung des Spiels geändert wird, also auch die Fensterbreite und -höhe sich ändert. Für diese Elemente ist DimensionUpdater da. Die Updates, die er verwalten muss, werden in der Hash-Map `updates` gespeichert. Um dieser Map Updates hinzuzufügen, gibt es die statische **queueUpdate**-Methode. Um die Updates auszuführen, gibt es die statische **updateDimensions**-Methode, die bei einer Veränderung der Auflösung von außen aufgerufen wird. Der DimensionUpdater soll nicht instanziert werden, deshalb wird der implizite Konstruktor von einem privaten überdeckt.

PlayerStatusViewConverter

Diese Klasse implementiert das Nifty-Interface **ListBoxViewConverter**. Dieses Interface muss implementiert werden, wenn in einer **ListBox** mehr angezeigt werden soll als reiner Text. Da im Player-Status (eine kleine Anzeige im Spielbildschirm, wo für jeden Spieler eine Übersicht mit Namen, Leben und so weiter aufgelistet wird) für jeden Spieler auch ein Icon angezeigt werden soll, das seinen Roboter darstellt, muss für ihn **ListBoxViewConverter** implementiert werden, um zu regeln, wie die Elemente in der **ListBox** angezeigt werden sollen. Dafür implementiert die Klasse die Methoden **display** and **getWidth**.

SpacerController

Ein Spacer ist ein Element in Nifty, das als Platzhalter dient. Es schafft also Abstand zwischen anderen Elementen auf der GUI. SpacerController implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Nun besitzen Spacers natürlich keine wirkliche Funktionalität. Die Klasse setzt schlicht die Höhe dieses Abstandhalters.

VerticalTabController

Diese Klasse implementiert **Controller**, ist also für die Funktionalität hinter einem Objekt auf einem **Screen** zuständig. Dieser Controller ist zuständig für das Funktionieren von Tabs, die in manchen Dialogen verwendet werden (Hilfe, Einstellungen), also Reitern, auf die der Inhalt eines Dialoges zur besseren Übersicht aufgeteilt ist. Jeder Tab wird dabei durch einen Button repräsentiert, mit dessen Hilfe er angewählt werden kann. Für dieses Verhalten gibt es die Methode **onButtonClick**, die von nifty mit einem bestimmten Schlüssel aufgerufen wird, der definiert, welcher Tab angesprochen werden soll. Dieser Tab wird dann unter Hilfenahme einer weiteren Methode **displayTab** sichtbar geschaltet. Die sonstigen Methoden der Klasse entsprechen dem Interface **Controller**.

SetKeyInputMapping & ShortcutInputMapping

In unserem Spiel gibt es für Beobachter zwei verschiedene Arten von Tastatureingaben: Die eine wird nur im Settings-Dialog verwendet, um neue Tastaturbelegungen für Shortcuts zu vergeben, die andere kommt in allen anderen Dialogen zum Einsatz, um ebendiese Shortcuts zu verwenden. Da diese Tastatureingaben völlig unterschiedliche Aktionen auslösen, benötigen wir auch zwei verschiedene Implementierungen des Interfaces **NiftyInputMapping**, das eine `convert`-Methode vorlegt, die dazu da ist, ein **KeyInputEvent** in ein neutrales **NiftyInputEvent** umzuwandeln, das zur weiteren Verarbeitung der Tastatureingabe benötigt wird. **SetKeyInputMapping** ist dabei zuständig für die Tastaturbelegung im Settings-Dialog und **ShortcutInputMapping** konvertiert Tastatureingaben aus den übrigen Dialogen.

Events

Wenn in einer Nifty-Anwendung irgendeine Benutzereingabe erfolgt, wird sie in der Anwendung als **NiftyInputEvent** verarbeitet. Dies ist ein Interface, dass von Nifty mit dem Enum **NiftyStandardInputEvent** implementiert wird. Wir nutzen insgesamt neun eigene Implementierungen von **NiftyInputEvent**.

Ein **SetKeyEvent** wird dabei nur ausgelöst, wenn der Nutzer in den Einstellungen ein shortcut zum Ändern ausgewählt hat und dann eine Taste drückt. Dieses Event ist also dazu da, Shortcuts zu belegen. Das Event besitzt auch einen key - also eine Taste - in Form eines Integers als Objektvariable und dazugehörige Getter und Setter.

Die anderen acht Events stehen alle für je einen dieser Shortcuts und werden ausgelöst, wenn der Nutzer irgendwo sonst eine Taste drückt, mit der ein Shortcut derzeit belegt ist. Sie führen dann zur entsprechenden Reaktion. Die Events **ToggleChatEvent**, **ToggleConsoleEvent**, **ToggleEventListEvent**, **ToggleFpsEvent**, **ToggleHelpEvent**, **TogglePopupMenuEvent** und **ToggleStatsEvent**, sagen dabei aus, das die Sichtbarkeit des jeweiligen Dialogs, nach dem das Event benannt ist, hin- und hergeschaltet werden soll. **Toggle3dEvent** soll dagegen direkt ein Umschalten zwischen 3D-Darstellung und 2D-Darstellung des Spiels zur Folge haben. Die Einzige Information, die all diese Events enthalten, ist der Klassenname an sich und somit sind diese Klassen alle leer.

3.4.2 Klassendiagramme

Siehe folgende Seiten.

de

robantly

beobachter

contants

util

io

tupels

log

preferences

core

nifty_interface

game

util

chat

console

events

player

gui

jme

util

model

environment

nifty

input

util

update

controls

events

constants

Constants

```
+ PROTOCOL_VERSION: String  
+ CONSOLE_LINE_HEIGTH: int  
+ VISUALIZATION_TIME_PER_PLAYER: int  
+ SEC_PER_MIN: int  
+ NONE: String  
+ ROTATE_LEFT: int  
+ ROTATE_RIGHT: int  
+ ROTATE_FULL: int  
+ ROBOT_MOVE_SPEED: float  
+ SCENE_FORWARD_VECTOR: Vector3f  
+ TURN_SPEED: int  
  
- Constants()
```

<<Enumeration>>

AnimState

```
WALK_FORWARD  
WALK_BACKWARD  
COLLISION_WITH_WALL_FORWARD  
COLLISION_WITH_WALL_BACKWARD  
COLLISION_WITH_WALL_RIGHT  
COLLISION_WITH_WALL_LEFT  
CHECKPOINT_REACHED  
SPAWN_LAST_CHECKPOINT  
PUSHED_FORWARD  
PUSHED_BACKWARD  
PUSHED_RIGHT  
PUSHED_LEFT  
TURN_RIGHT  
TURN_LEFT  
WALK_AND_PUSH_FORWARD  
WALK_AND_PUSH_BACKWARD  
WAIT  
FALL_DOWN  
DESTROY  
WIN  
DISQUALIFY  
DEAD
```

<<Enumeration>>

RobotType

```
TV  
OVEN  
FLAT_IRON  
FRIDGE  
VACUUM_CLEANER  
CAMERA  
MIXER  
GAMEBOARD_MODEL  
WALL_MODEL  
  
- paths: Paths  
  
- RobotType(PATHS: Paths)
```

util

HardReset

```
- LOG: Logger
+ main(String[] ARGS): void
```

preferences

CustomDisplayMode

```
- DISPLAYMODE: DisplayMode
+ CustomDisplayMode(DisplayMode DISPLAYMODE)
```

NumberSetting

```
- NUMBER: int
- PREFIX: String
- SUFFIX: String
# NumberSetting(PREFIX: String, NUMBER: int,
SUFFIX: String)
# NumberSetting(PREFIX: String, NUMBER: int)
# NumberSetting(NUMBER: int, SUFFIX: String)
# NumberSetting(NUMBER: int)
```

Preferences

```
- LOG: Logger
+ prefs: java.util.prefs.Preferences
+ TOGGLE_POPUP_MENU: String
+ TOGGLE_HELP: String
+ TOGGLE_EVENT_LIST: String
+ TOGGLE_CHAT: String
+ TOGGLE_CONSOLE: String
+ TOGGLE_STATS: String
+ TOGGLE_FPS: String

+ FULLSCREEN: String
+ DISPLAYMODE: String
+ FRAMERATE: String
+ VSYNC: String
+ MSAA: String
+ FXAA: String
+ SSAO: String
+ SHADOW_MODE: String
+ SHADOW_MODE_OFF: String
+ SHADOW_MODE_RENDER: String
+ SHADOW_MODE_FILTER: String
+ SHADOW_MAP_SIZE: String
+ SHADOW_MAP_COUNT: String
+ UI_SCALING: String
+ RESTART: String
- currentUiScaling: int
+ DEFAULT_NODE: String
+ JME_NODE: String

- Preferences()
- fullscreen(Element: Element): void
- displayMode(Element: Element): void
- frameRate(Element: Element): void
- vsync(Element: Element): void
- msaa(Element: Element): void
- fxaas(Element: Element): void
- ssao(Element: Element): void
- shadowMode(Element: Element): void
- shadowMapSize(Element: Element): void
- shadowMapCount(Element: Element): void
- uiScaling(Element: Element): void
- restart(Element: Element): void
```

log

NiftyConsoleAppender

EXTENDS
AppenderSkeleton

io

IoUtil

```
+ saveRecordedGameToFile(RECORDEDGAME: RecordedGame): void
```

RecordedGame

```
- roundResultMessages: List<RoundResultMessage>
+ addRoundResultMessage(ROUNDRESULTMESSAGE: RoundResultMessage): void
```

tupels

ScreenElementCommandsTuple

```
- screenElementPair: ScreenElementPair
- commands: ConsoleCommands
+ ScreenElementCommandsTuple(SCREEN: Screen, ELEMENT: Element,
COMMANDS: ConsoleCommands)
```

ScreenElementPair

```
- SCREEN: Screen
- ELEMENT: Element
+ ScreenElementPair(Screen SCREEN, Element ELEMENT)
```

jme

Engine

- LOG: Logger
- filterPostProcessor: FilterPostProcessor
- shadowMapSize: int
- shadowMapCount: int
- shadows: boolean
- event: RoundResultEvent
- EXCHANGEPOINT: ExchangePoint
- currentEvent: RoundResultEvent
- nifty: Nifty
- classLoader: ClassLoader
- EVENTS: BlockingQueue<RoundResultMessage>
- statVisible: AtomicBoolean
- fpsVisible: AtomicBoolean
- niftyJmeDisplay: NiftyJmeDisplay
- finalStateTimer: float
- eventCounter: int
- + Engine(EXCHANGEPOINT: ExchangePoint)
- + init(): void
- + addToEventQueue(EVENT: RoundResultMessage): void
- + toggleStats(): void
- + toggleFPS(): void
- + isFullscreenSupported(): boolean
- + reshapeNiftyJmeDisplay(): void

CameraController

- chaseCamera: ChaseCamera
- cam: Camera
- target: Node
- is2D: boolean
- isInverted: boolean
- + CameraController(Camera CAMERA)
- + createChaseCamera(Vector3f TARGETPOS): ChaseCamera
- + setChaseCameraTarget(Spatial TARGET): void
- + setChaseCameraStaticTarget(Vector3f TARGETPOS): void
- setCameraTo2D(): void
- setCameraTo3D(): void
- + toggleInvertAxis(): void
- + toggle2D3D(): void

EXTENDS
SimpleApplication

util**Paths**

- pathToGlossMap: String
- pathToModel: String
- pathToMaterial: String
- pathToDiffuseMap: String
- pathToNormalMap: String
- pathToSpecularMap: String
- pathToTolcon: String
- + Paths(String pathToModel, String pathToMaterial, String pathToTolcon)
- + Paths(String pathToModel, String pathToMaterial, String pathToDiffuseMap, String pathToNormalMap, String pathToSpecularMap, String pathToTolcon)

//ResourcePaths

- PATH_MODEL_TV: String
- PATH_MATERIAL_TV: String
- PATH_MODEL_OVEN: String
- PATH_MATERIAL_OVEN: String

model**environment****Environment**

- intendedSize: float
- + Environment(INDEX: int, INDEX: int)
- randomizeModel(): String

EnvironmentFactory**MaterialCreator**

- + createMaterial(PATHS: Paths): Material

MeshProcessing

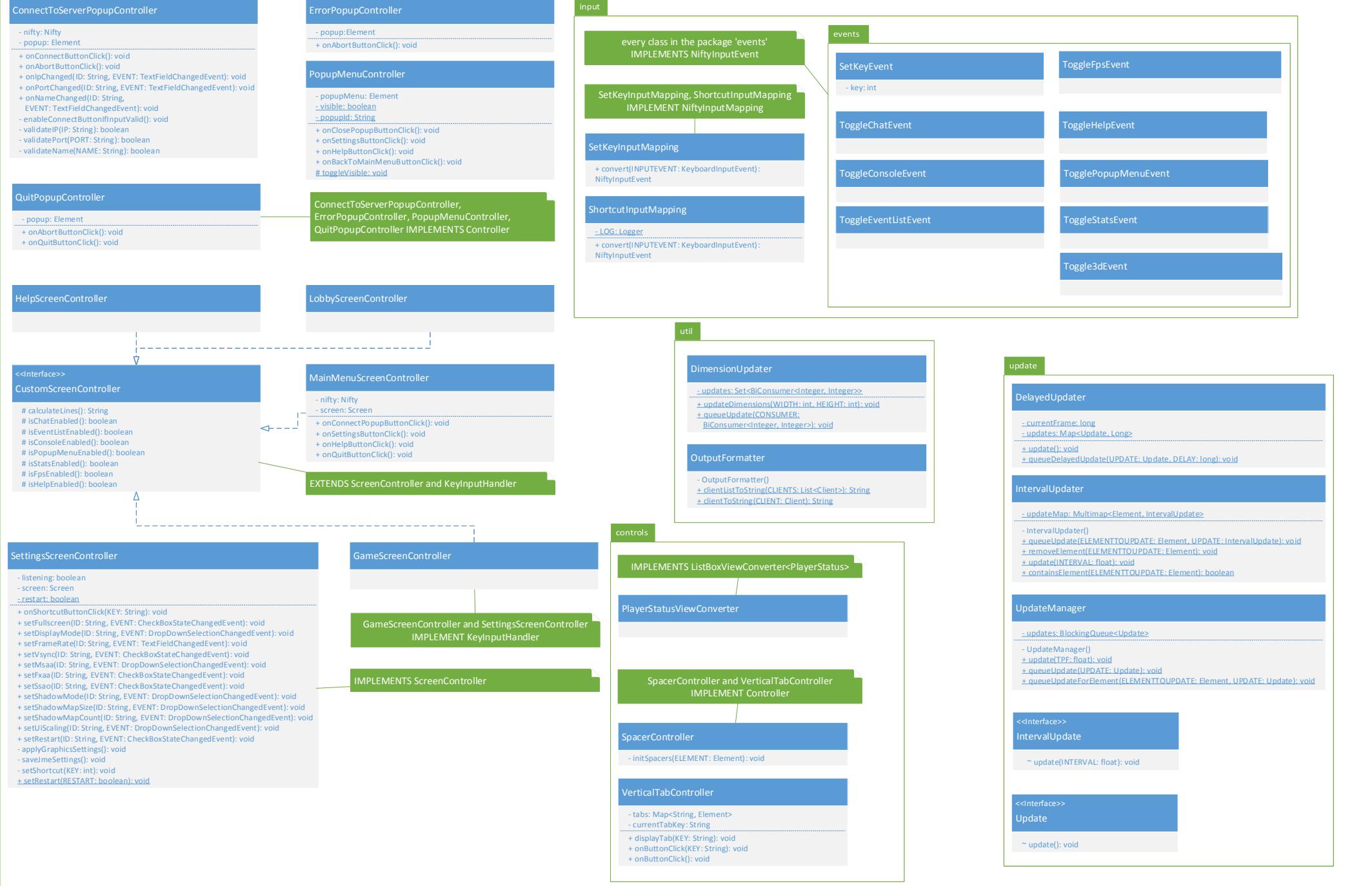
- + getMyGeometry(SPATIAL: Spatial): Geometry

ModelLoader

- + loadModelByName(ROBOTTYPE: RobotType): Node
- + getTileSize(): double
- + loadGameBoardModel(WIDTH: int, HEIGHT: int): Spatial

RobotAnimController

- ANIMCONTROL: AnimControl
- animChannel: AnimChannel
- + RobotAnimController(ANIMCONTROL: AnimControl)
- + setAnim(ANIMSTATE: AnimState): void



3.4.3 Kommunikationsdiagramme

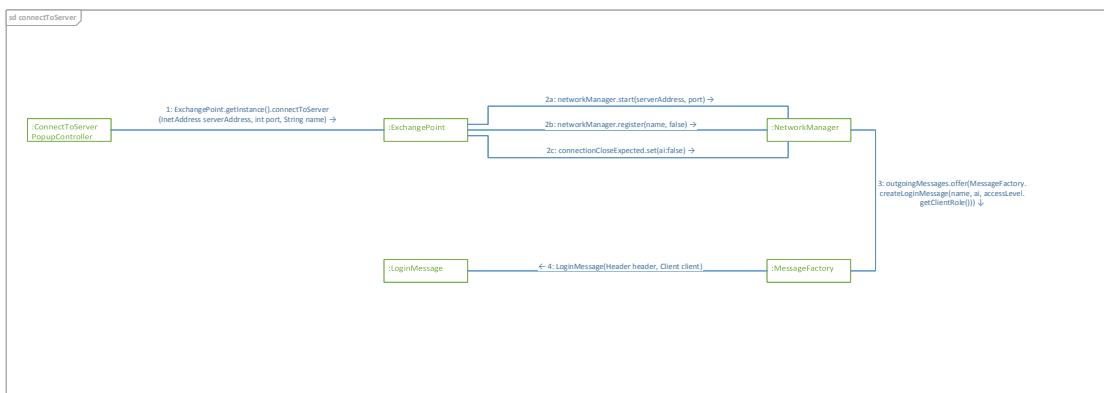


Abbildung 3.9: Beobachter - Kommunikationsdiagramm: ConnectToServer

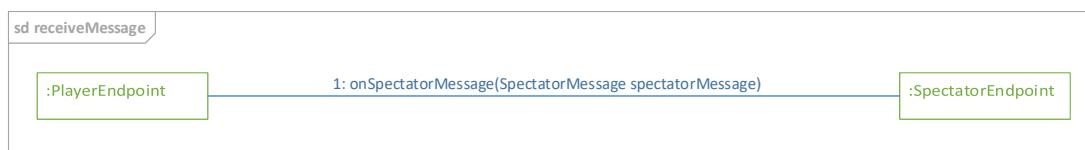


Abbildung 3.10: Server - Kommunikationsdiagramm: ReceiveMessage

Dieses Diagramm zeigt, was passiert, wenn der Network-Manager eine Nachricht empfängt. Es repräsentiert den Ablauf für die folgenden Methoden:

- `onChatMessage(ChatMessage chatMessage)`
- `onErrorMessage(ErrorMessage errorMessage)`
- `onGameCfgMessage(GameCfgMessage gameCfgMessage)`
- `onGameInitMessage(GameInitMessage gameInitMessage)`
- `onLobbyStatusMessage(LobbyStatusMessage lobbyStatusMessage)`
- `onLoginResponseMessage(LoginResponseMessage loginResponseMessage)`
- `onPauseMessage(PauseMessage pauseMessage)`
- `onResumeMessage(ResumeMessage resumeMessage)`
- `onRoundResultMessage(RoundResultMessage roundResultMessage)`

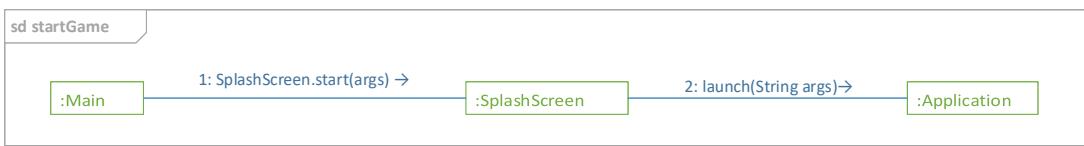


Abbildung 3.11: Server - Kommunikationsdiagramm: StartGame

3.5 Spieler

3.5.1 Beschreibung der Methoden

Da der Spieler alle Funktionalität enthalten muss, die der Beobachter besitzt, werden nur wenige zusätzliche Klassen benötigt. Diese befinden sich in der Komponente Spieler. Außerdem werden auch Klassen aus Util genutzt, die im Beobachter noch keine Verwendung gefunden haben.

Main

Die Spielerkomponente besitzt eine eigene Main-Klasse mit **main-Methode**, um den Spieler ausführbar zu machen. Die Main in Spieler unterscheidet sich von der Main im Beobachter, weil dieser einen abgewandelten ExchangePoint benutzt, der ja in der Main-Klasse gestartet wird.

ExchangePointImpl

Die ExchangePoints in unserer Anwendung sind die Stellen, an denen ein Client organisatorisch zusammengeführt wird, also die Vereinigung von GUI, Logik und Netzwerk. Es ist offensichtlich, dass diese Klasse für den Spieler nicht identisch zum Beobachter sein kann. Um aber möglichst wenig redundanten Code zu verwalten, ist das, was beiden ExchangePoints gemein ist, in der abstrakten Klasse **BasicExchangePointImpl** zusammengefasst. Zusätzlicher Code für die beiden Komponenten ist dann in einer jeweils eigenen Klasse **ExchangePointImpl** enthalten, die von **BasicExchangePointImpl** erben. Die BasicExchangePointImpl des Spielers besitzt im wesentlichen die Methoden **onPlayerHandMessage**, da ein Beobachter diese Nachrichten natürlich nicht empfangen kann, und **onServerMessage**, da der Spieler gegenüber Servernachrichten mehr Zugriffsrechte hat als der Beobachter. Außerdem besitzt die Klasse eine eigene **reset**-Methode.

GameScreenControllerAdditions

Diese Klasse verwaltet alle GUI-Elemente, die auf dem Spielbidschirm im Spieler zusätzlich zu den im Beobachter bereits sichtbaren Elementen angezeigt werden müssen inklusive der Funktionalität dahinter. Diese Elemente sind jene, die zur Programmauswahl notwendig sind - also die Visualisierung der jeweils vom Server gesendeten PlayerHand, Programm Slots, in denen das Programm per Drag n Drop zusammengestellt werden kann und einen Kompass aus RadioButtons, mit dem bei Bedarf die Blickrichtung beim Spawnen des Roboters festgelegt werden kann.

Die Klasse enthält folgende öffentliche Methoden: **accept** wird zu Beginn eines Spiels einmalig aufgerufen. Sie bereitet den Kartenbereich vor, erstellt also Droppables als Programm Slots und RadioButtons für den Kompass. Außerdem gibt sie das Layout für den Kartenbereich vor. **updateCards** muss von außen aufgerufen werden, um die angezeigte PlayerHand zu aktualisieren, wenn eine neue Runde beginnt. Sie erstellt also die Programmkarten als **Draggables** und legt fest, ob der Kompass für die folgende Runde aktiviert oder deaktiviert sein muss. Mit **getProgram** kann man sich von außen das aktuell auf den Programm Slots liegende Programm geben lassen, was am Ende einer Runde benötigt wird, da es dann an den Server versandt werden muss. Ebenso

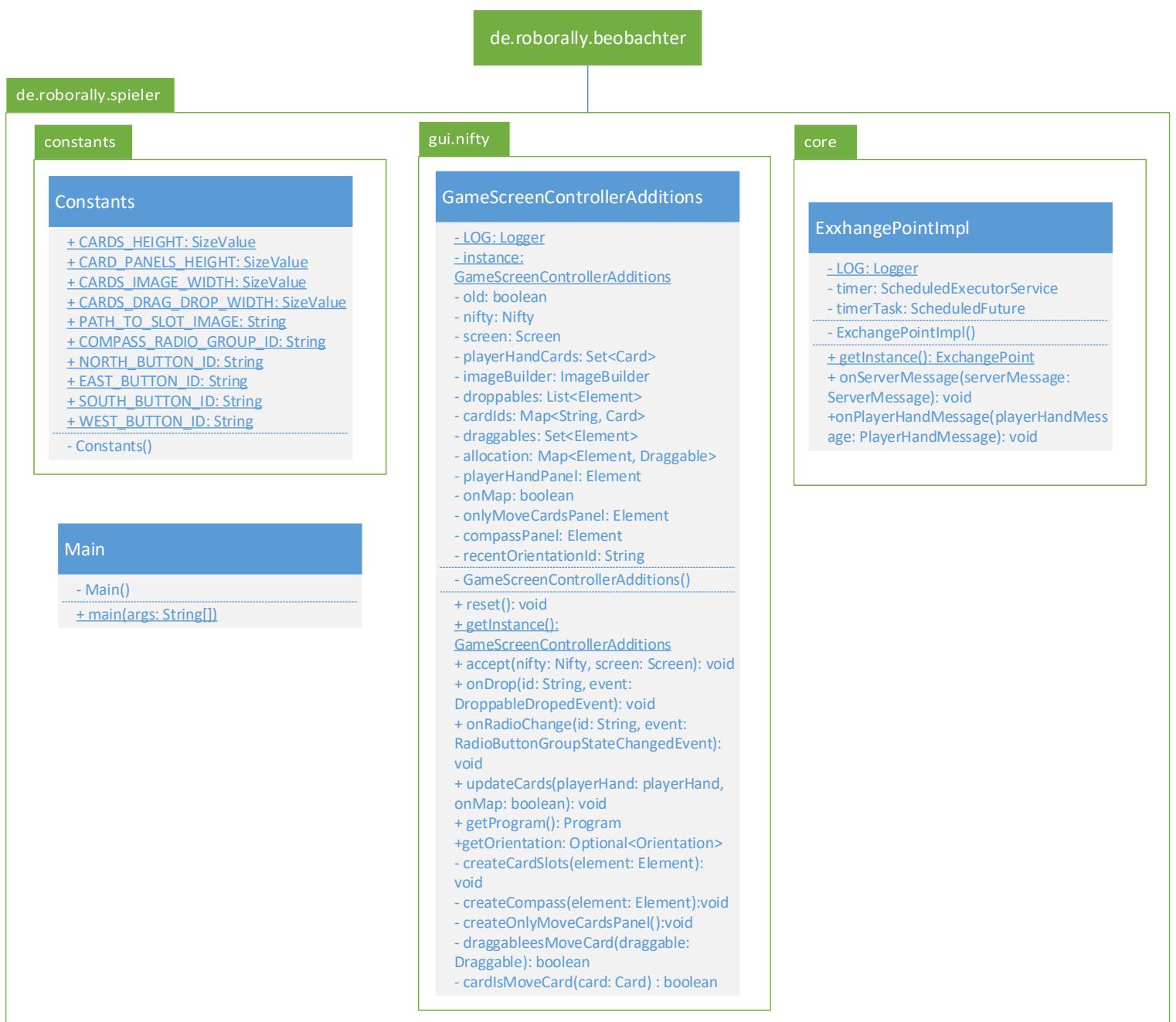
gibt **getOrientation** die aktuell im Kompass ausgewählte Richtung zurück. Weiterhin gibt es die beiden Methoden **onDrop** und **onRadioChange**, die von Nifty ausgelöst werden, wenn eine Karte auf einem Program Slot abgelegt wird bzw wenn im Kompass eine Änderung vorgenommen wird. Diese Methoden speichern den Zustand von aktuellem Programm und Kompass ab und regeln das Verhalten der Karten, falls eine verbotene Eingabe gemacht wird. Schließlich besitzt die Klasse noch eine **reset**-Methode, die nach einem Spiel aufräumt und den Ursprungszustand wiederherstellt, um ein weiteres Spiel beginnen zu können.

3.5.2 Kommunikationsdiagramm

Beim Spieler gibt es keine weiteren speziellen Abläufe, die allgemeinen Kommunikationsdiagramme 3.1 und die Kommunikationsdiagramme aus dem Beobachter decken alle Fälle ab.

3.5.3 Klassendiagramm

Siehe folgende Seite.



3.6 KI

3.6.1 Beschreibung der Methoden

Für die Realisierung der KI, benötigt man natürlich alle Klassen aus der Komponente KI und noch einige Klassen aus der Komponente Util.

Main

Innerhalb der KI-Komponente gibt es die Klasse Main, die eine **main-Methode** enthält und lediglich dazu dient, das Programm ausführbar zu machen. In der main-Methode werden Name, Server-IP und Port abgefragt und dann mit den entsprechenden Daten ein eine Instanz der Klasse ExchangePoint erstellt, welche eine Verbindung zum Server aufbaut.

AiResult

Die Klasse AiResult, ist eine Klasse, in welcher die Ergebnisse der KI gespeichert werden. Aus dieser werden später Programm und Orientierung ausgelesen und dem Server gesendet.

Logic

In der Klasse Logic wird der beste Weg ermittelt, den die KI laufen kann. Hierzu wird die Klasse BFS verwendet. Nach der Ermittlung des besten Programms, wird diese mit Hilfe der AiResult Klasse an die ExchangePoint-Klasse weitergeleitet.

BFS

Diese Klasse führt eine Breitensuche über das Spielbrett (LogicGameBoard aus der Util-Komponente) aus. Dabei wird die Distanz einer Tile zum nächsten Checkpoint berechnet.

ExchangePoint

Diese Klasse ist für die Kommunikation zwischen der KI und dem Server zuständig. In dieser wird geregelt, welche Aktionen bei der Ankunft von bestimmten Messages ausgeführt werden sollen.

3.6.2 Kommunikationsdiagramm

Bei der KI gibt es auch keine weiteren speziellen Abläufe, die allgemeinen Kommunikationsdiagramme 3.1 decken alle Fälle ab.

3.6.3 Klassendiagramm

Siehe folgende Seite.

de.roborally.ki

Main

- LOG: Logger
+ main(args: String[]): void

AiResult

- program: Program
- orientation: Orientation
~ AiResult(program: Program,
orientation: Orientation)
~ AiResult(program: Program)
~ getProgram(): Program
~ getOrientation():
Optional<Orientation>

Logic

- LOG: Logger
- eval: Map<Tile, Integer>
- oldTile: Tile
- oldTileCount: int
- currEvalCheckpoint: Tile
~ solveBruteForce(playerHand:
playerHand, gameBoard:
LogicGameBoard)

ExchangePoint

de.roborally.util

BFS

- path(gameBoard: LogicGameBoard):
Map<Tile, Integer>

4 Anhang

Hier wird das Glossar aus dem Pflichtenheft eingebunden, damit man schnell Begrifflichkeiten nachschauen kann.

Glossar

BEGRIFF Abbruch

Siehe Spielabbruch

BEGRIFF Ausführung eines Programms

BESCHR. Nachdem alle Spieler ihr Programm zusammengestellt haben, geht es an die Ausführung. Die Befehle werden nacheinander ausgeführt, dabei wird erst von allen Spielern der erste Befehl ausgeführt, danach von allen Spielern der zweite usw. Wessen Befehl dabei jeweils zuerst ausgeführt wird, wird durch die Prioritäten bestimmt, die auf den Befehlskarten angegeben sind. Während die Befehle ausgeführt werden, können die Spieler nicht mehr auf die Bewegungsvorgänge der Roboter Einfluss nehmen.

SYNONY. Programmausführung

BEGRIFF Ausrichtung

BESCHR. Der Roboter hat eine Ausrichtung, die bestimmt, wo bei ihm vorne und hinten ist, also seine aktuelle Bewegungsrichtung festlegt. Damit die Ausrichtung immer klar ist, legt der Spieler beim Einsetzen seines Roboters ins Spiel die Ausrichtung mit dem ersten Bewegungsbefehl fest.

IstEin Nord, Ost, Süd, West

KannSein Eigenschaft des Roboters

ASPEKT Bewegung des Roboters

BEGRIFF Befehl

Siehe Programmbebefhl

BEGRIFF Blockierung

BESCHR. Entsteht, wenn ein Roboter in eine Richtung bewegt werden soll, in der vom Roboter aus gesehen eine Mauer steht. Dies kann geschehen, wenn der Roboter selbst einen dahingehenden Laufbefehl erhält, oder aber wenn er durch einen anderen Roboter geschoben wird. Im Fall einer Blockierung wird die Bewegung, durch die sie verursacht ist, einfach abgebrochen und das Spiel wird fortgesetzt.

ASPEKT Bewegung des Roboters

BEISPIEL *Roboter A soll drei Felder vorwärts laufen, stößt aber nach einem Feld gegen Roboter B. Roboter A schiebt Roboter B ein Feld weiter, wobei Roboter B gegen eine Wand prallt. Jetzt sind beide Roboter blockiert und Roboter A darf sein verbliebenes Feld nicht weiterziehen.*

BEGRIFF Checkpoint

BESCHR. Bestimmte Position auf dem Spielfeld. Es befinden sich mehrere (mindestens zwei) Checkpoints auf einem Spielfeld, die eine vorgegebene Reihenfolge besitzen. Alle Spieler starten am ersten Checkpoint. Dabei muss jeder Spieler den Roboter mit dem ersten Befehl vom Checkpoint wegbewegen. Aufgabe der Spieler ist es, die Checkpoints mit ihren Robotern in der richtigen Reihenfolge zu erreichen. Wird ein Roboter zerstört, wird er zudem an einem Checkpoint wieder eingesetzt.

IstEin Position

KannSein Zwischenziel, Ziel

ASPEKT Spielziel

BEGRIFF Client

BESCHR. Ein Client ist ein Computersystem, das entweder von einem Menschen bedient oder durch eine KI gesteuert wird.

IstEin Computer

KannSein Beobachter, Teilnehmer

ASPEKT System

BEGRIFF Disqualifikation

BESCHR. Ein Spieler wird disqualifiziert, sobald sein Roboter zum dritten Mal zerstört wird oder eine seiner Aktionen ungültig ist. Wurden alle Spieler disqualifiziert, wird das Spiel beendet.

ASPEKT Spielregeln

BEISPIEL *Spieler A braucht mehr Zeit als ihm zur Verfügung steht, um seinen Roboter zu programmieren, weshalb er aus dem Spiel ausscheidet.*

BEGRIFF Drehbefehl

BESCHR. Befehl, mit dem die Spieler den Roboter anweisen können, sich um die eigene Achse zu drehen.

IstEin Befehl

KannSein Vierteldrehung, halbe Drehung

ASPEKT Bewegung des Roboters

BEISPIEL „Vierteldrehung gegen den Uhrzeigersinn“

BEGRIFF Engine

Siehe Server

BEGRIFF Feld

Siehe Position

BEGRIFF Gewinner

BESCHR. Gewinner ist der Spieler, dessen Roboter alle Checkpoints in der richtigen Reihenfolge passiert und als erstes das Ziel erreicht. Er wird nach Ende des Spiels für alle Teilnehmer und Beobachter sichtbar angezeigt.

IstEin Spieler

KannSein KI, Mesch

ASPEKT Spielregeln

BEGRIFF Highscore

BESCHR. Im Zusammenhang des SOPRAs ist der Highscore eine globale Teambewertung. Unter anderem fließen Bewertung von Meilensteinen und Aufgabenblättern sowie die Anzahl der vertriebenen Lizenzen auf der Messe und erreichte Punkte auf dem Turnier mit ein. Der Highscore ist öffentlich.

IstEin Teambewertung

BEGRIFF Hindernis

Siehe Mauer

BEGRIFF Kachel

Siehe Position

BEGRIFF Karte

Siehe Programmbebefhl

BEGRIFF Konfiguration
Siehe Spielkonfiguration

BEGRIFF Künstliche Intelligenz (KI)
BESCHR. Eine KI simuliert einen Spieler und kann so am Spiel als virtueller Spieler teilnehmen.
IstEin Teilnehmer

BEGRIFF Laufbefehl
BESCHR. Befehl, mit dem die Spieler den Roboter anweisen können, eine bestimmte Anzahl von Feldern zu ziehen.
IstEin Befehl
KannSein Vorwärtsbewegung, Rückwärtsbewegung
ASPEKT Bewegung des Roboters
BEISPIEL „1 Feld rückwärts“

BEGRIFF Level
BESCHR. Ein Spielfeld mit einer bestimmten Größe und bestimmten Anzahl und Positionen von Mauern und Checkpoints. Verschiedene Levels können unterschiedlich schwierig sein. Levels können mit dem Leveleditor erstellt werden.
ASPEKT Spielregeln

BEGRIFF Leveleditor
BESCHR. Tool, um Level zu erstellen und zu konfigurieren.

BEGRIFF Mauer
BESCHR. Eine Mauer ist ein Hindernis, das auf dem Spielfeld platziert ist. Eine Mauer steht dabei auf der Grenze zwischen zwei Feldern. Roboter können eine Mauer nicht passieren. Steht der Bewegung eines Roboters eine Mauer im Weg, so wird der Roboter blockiert und bleibt stehen.
ASPEKT Bewegung des Roboters
SYNONY. Hindernis, Wand

BEGRIFF Messe
BESCHR. Treffen aller Teams im Rahmen des SOPRAs, an dem die Teams Lizenzen von Spielekomponenten voneinander erwerben können.

BEGRIFF Plattform

BESCHR. Betriebssystem, auf dem das Spiel ausgeführt wird.

KannSein Windows 10, Linux

ASPEKT System

SYNONY. Betriebssystem

BEGRIFF Position

BESCHR. Kästchen auf dem Spielfeld; durch zwei Koordinaten eindeutig bestimmt.

BEISPIEL A1

SYNONY. Feld, Kachel

BEGRIFF Priorität

BESCHR. Auf jeder Befehlskarte wird eine Priorität angegeben, die darüber entscheidet, in welcher Reihenfolge die Befehle der Spieler ausgeführt werden. Die Prioritäten sind so gesetzt, dass unter verschiedenen Befehlen insgesamt eine feste Rangfolge herrscht, und die Rangfolge unter Befehlen mit demselben Wortlaut zufällig ist.

IstEin zufällige natürliche Zahl

ASPEKT Spielregeln

BEISPIEL *Spieler1 und Spieler2 spielen gegeneinander. Spieler1 hat ein Programm aus Befehlen mit den Prioritäten (1), (5), (7), (2), (15) in dieser Reihenfolge zusammengestellt, Spieler2 eins mit den Prioritäten (12), (3), (4), (9), (16). Die Befehle werden abwechselnd abgearbeitet, aber immer so, dass der erste Befehl beider Spieler abgearbeitet wurde, bevor der zweite Befehl eines Spielers an die Reihe kommt. Also hier wird erst der Befehl mit der Priorität (12) von Spieler1 ausgeführt, dann der Befehl von Spieler2, da 12>1. Dann kommen nach dem selben Prinzip nacheinander die Befehle mit den Prioritäten (5),(3), (7),(4), (9),(2), (16),(15).*

BEGRIFF Programm

BESCHR. Ein Programm ist im Zusammenhang des Spiels eine Abfolge von simplen Befehlen an einen Roboter, um ihn über das Spielfeld zu bewegen; wird von einem Spieler zusammengestellt.

IstEin Abfolge von 5 Befehlen

ASPEKT Programmierung des Roboters

BEISPIEL „1 Feld vorwärts - Vierteldrehung gegen den Uhrzeigersinn - 3 Felder vorwärts - 2 Felder vorwärts - halbe Drehung (180°)“

SYNONY. Programmierung

BEGRIFF Programmausführung
Siehe Ausführung

BEGRIFF Programmbebefhl
BESCHR. Spielkarte mit einfacher Bewegungsanweisung für einen Roboter.
KannSein Drehbefehl, Laufbefehl
ASPEKT Programmierung des Roboters
BEISPIEL „ein Feld vorwärts“
SYNONY. Befehl, Karte

BEGRIFF Programmierung
Siehe Programm

BEGRIFF Robo Rally
BESCHR. Brettspiel, nach dessen Vorlage das Computerspiel entwickelt wird und somit auch Name des Computerspiels.

BEGRIFF Roboter
BESCHR. Virtuelle Spielfigur, die von einem Spieler durch einfache Befehle programmiert werden kann.
ASPEKT Spielfigur

BEGRIFF Runde
BESCHR. Ein Spiel besteht aus mehreren Runden. Jede Runde hat folgenden Ablauf:
1. Spieler erhalten Befehlskarten
2. Spieler stellen Programm für ihre Roboter zusammen
3. Programme werden ausgeführt
ASPEKT Spielregeln
SYNONY. Rundenablauf

BEGRIFF Rundenablauf
Siehe Runde

BEGRIFF Server

BESCHR. Der Server ist im Zusammenhang des Spiels ein Computersystem, das die Leitung über ein Spiel trägt. Er muss das Spiel verwalten, also zum Beispiel Clients ermöglichen, sich für ein Spiel anzumelden, und den Spielverlauf moderieren, also die Clients auffordern, ihre Spielzüge zu machen und auf Gültigkeit überprüfen. Der Server ist somit für die Einhaltung der Spielregeln zuständig.

IstEin Computer

ASPEKT System

SYNONY. Spieleengine, Engine

BEGRIFF Spielabbruch

BESCHR. Liegt dann vor, wenn ein Spiel manuell beendet wird.

IstEin Spielende

ASPEKT Spiel

SYNONY. Abbruch

BEGRIFF Spielbrett

BESCHR. Zweidimensionales Raster, worauf Mauern und Checkpoints positioniert sind und die Roboter bewegt werden.

ASPEKT Spielregeln

SYNONY. Spielfeld

BEGRIFF Spieleengine

Siehe Server

BEGRIFF Spielende

BESCHR. Regulär ist das Spiel beendet, sobald ein Roboter das Ziel erreicht und zuvor alle Zwischenziele in der richtigen Reihenfolge besucht hat. Ein Spiel kann aber auch manuell abgebrochen werden. Auch wenn alle Spieler disqualifiziert wurden, wird das Spiel beendet.

KannSein jemand gewinnt, Abbruch, Disqualifikation aller Spieler

ASPEKT Spielregeln

BEGRIFF Spieler

Siehe Teilnehmer

BEGRIFF Spielername

BESCHR. Jeder Spieler besitzt einen Namen, anhand dessen man ihn eindeutig identifizieren kann.

KannSein Zeichenkette aus Buchstaben und Zahlen

BEGRIFF Spielfeld
Siehe Spielbrett

BEGRIFF Spielkonfiguration
BESCHR. Einstellung für ein Spiel
KannSein Spieleranzahl, Größe des Spielfelds, Anzahl Positionen von Mauern, Anzahl, Reihenfolge und Position von Checkpoints, Zeitlimit für Roboterprogrammierung pro Runde.
ASPEKT System
BEISPIEL Ein Spiel hat folgende Konfogurationen: 3 Spieler auf einem 25x25-Spielfeld, 3 Mauern auf den Grenzen der Felder B5-B6, J10-K10, W3-W4, 2 Checkpoints, wobei der erste auf A1 steht und der letzte auf Y25, Zeitlimit für eine Programmierung sind 16 Sekunden.
SYNONY. Einstellung, Konfiguration

BEGRIFF Spielregeln
BESCHR. Vorschriften, wie sich die Spieler verhalten müssen; Definition aller Spiel-elemente wie Spielablauf, Spielende usw. Die Einhaltung der Spielregeln wird vom Server überwacht.
ASPEKT Spielregeln

BEGRIFF Teilnehmer
BESCHR. Teilnehmer sind Menschen oder KIs, die mit dem Spiel interagieren und die darin befindlichen Roboter programmieren können.
IstEin Client
KannSein Mensch, KI
ASPEKT Spiel
SYNONY. Spieler

BEGRIFF Turnier
BESCHR. Als Turnier wird der Wettbewerb bezeichnet, der 2017 im Rahmen des SOPRAs am Ende des Projekts veranstaltet werden soll. Das Turnier dient dazu, die entwickelten Programme der einzelnen Teams spielerisch zu testen.
IstEin Wettbewerb

BEGRIFF unentschieden

BESCHR. Ein Spiel geht unentschieden aus, falls alle Spieler vor Spielende disqualifiziert wurden oder bei einem manuellen Abbruch.

IstEin Spielausgang

ASPEKT Spielregeln

BEGRIFF Verlierer

BESCHR. Spieler, die zum Spielende noch nicht alle Checkpoints einschließlich dem Ziel erreicht haben. Verlierer werden nach Spielende für alle Teilnehmer und Beobachter sichtbar angezeigt.

IstEin Spieler

KannSein Mensch, KI

ASPEKT Spielregeln

BEGRIFF Wand

Siehe Mauer

BEGRIFF Zerstörung

BESCHR. Es gibt zwei Möglichkeiten für einen Roboter, zerstört zu werden: Entweder, er fällt während der Befehlsausführungsphase vom Spielfeld, oder, er befindet sich dann auf einem Checkpoint, wenn ein anderer Roboter auf ebendiesem eingesetzt wird. Wird ein Roboter zerstört, ist er für die restliche Runde nicht mehr im Spiel und wird zu Beginn der nächsten Runde an jenem Checkpoint eingesetzt, den er zuletzt erreicht hat. Wird der Roboter eines Spielers zum dritten Mal zerstört, wird der Spieler disqualifiziert.

KannSein Roboter fällt von Spielfeld, Roboter sitzt auf Checkpoint, an dem ein anderer Roboter eingesetzt wird.

ASPEKT Spielregeln

BEISPIEL *Roboter A steht bei Checkpoint 2, der sich ein Feld vom Spielfeldrand entfernt befindet und erhält den Befehl, zwei Felder vorwärts (Richtung Rand) zu ziehen. Er fällt vom Spielfeld und wird zerstört. Danach zieht Roboter B auf Checkpoint 2. Die nächste Runde beginnt, und Roboter A wird an der Stelle von Checkpoint 2 wieder auf das Spielfeld gesetzt. Damit wird Roboter 2 zerstört.*

BEGRIFF Ziel

BESCHR. Letzter Checkpoint auf einem Spielfeld. Wer das Ziel mit seinem Roboter als erstes erreicht und zuvor alle anderen Checkpoints (Zwischenziele) in vorgegebener Reihenfolge besucht hat, gewinnt das Spiel.

IstEin Checkpoint

ASPEKT Spielziel

SYNONY. Letzter Checkpoint

BEGRIFF Zwischenziel

BESCHR. Checkpoint, der nicht das Ziel ist. Also sind alle Checkpoints außer dem letzten, Zwischenziele.

IstEin Checkpoint

ASPEKT Spielziel