NeXT Generation Transport Tycoon

Team 0

Software-Entwicklungspraktikum (SEP)
Sommersemester 2013

Systemspezifikation

Auftraggeber:
Technische Universität Braunschweig
Institut für Programmierung und Reaktive Systeme
Prof. Dr. Ursula Goltz
Mühlenpfordtstr. 23
38106 Braunschweig

Betreuer: Benjamin Mensing

Auftragnehmer:

Name	E-Mail-Adresse
Dennis Stelter	d.stelter@tu-bs.de
Henrik Lange	henrik.lange@tu-bs.de
Jochen Steiner	jochen.steiner@tu-bs.de
Markus-Björn Meißner	m-b.meissner@tu-bs.de
Patricia-Tatjana Kasulke	p.kasulke@tu-bs.de
Tessa Fabian	tessa.fabian@tu-bs.de

Braunschweig, 26. Juni 2013

Versionsübersicht

Version	Datum	Autor	Status	Kommentar
0.1	05.05.2013	Tessa Fabian, Henrik Lange	i.B.	Analyse von Produktfunktionen 40, 50, 110, 130 hinzugefügt
0.2	06.05.2013	Markus-Björn Meißner	i.B.	Einleitung hinzugefügt
0.3	06.05.2013	Markus-Björn Meißner	i.B.	Analyse von Produktfunktionen 10, 20, 30, 120 hinzugefügt
0.4	06.05.2013	Tessa Fabian, Jochen Steiner, Dennis Stelter	i.B.	Komponentenspezifikation hinzugefügt
0.5	06.05.2013	Henrik Lange	i.B.	Glossar hinzugefügt
0.6	07.05.2013	Patricia-Tatjana Kasulke, Jochen Steiner	i.B.	Analyse von Produktfunktionen 60, 70, 80, 100 hinzugefügt
0.7	08.05.2013	Tessa Fabian, Jochen Steiner	i.B.	Softwarearchitektur hinzugefügt
0.8	09.05.2013	Dennis Stelter	i.B.	Analyse von Produktfunktionen 90, 140 hinzugefügt
0.9	09.05.2013	Henrik Lange	i.B.	Layout: Anpassungen und Korrekturen
0.10	09.05.2013	Markus-Björn Meißner	i.B.	Verteilungsentwurf hinzugefügt
0.11	10.05.2013	Patricia-Tatjana Kasulke	i.B.	Kriterienerfüllung hinzugefügt
0.12	13.05.2013	Dennis Stelter, Tessa Fabian, Jochen Steiner, Henrik Lange	i.B.	Korrekturen der Sequenzdiagramme
0.13	14.05.2013	Dennis Stelter, Tessa Fabian, Jochen Steiner, Markus-Björn Meißner	i.B.	Korrektur von Kapitel 3.2
0.14	14.05.2013	Dennis Stelter, Jochen Steiner, Henrik Lange	i.B.	Korrektur von Kapitel 3.1

0.15	14.05.2013	Dennis Stelter, Tessa Fabian, Jochen Steiner, Markus-Björn Meißner	i.B.	Korrektur von Kapitel 3.3
0.16	15.05.2013	Patricia-Tatjana Kasulke, Henrik Lange	i.B.	Korrektur von Kapitel 8.0
0.17	15.05.2013	Dennis Stelter, Tessa Fabian, Jochen Steiner, Markus-Björn Meißner, Patricia-Tatjana Kasulke, Henrik Lange	i.B.	Korrektur der Texte von Kapitel 2.0
1.0	15.05.2013	Henrik Lange	abg.	Abgabe ISF
1.1	12.06.2013	Henrik Lange	i.B.	Korrektur von Produktfunktion 130 und Kapitel 8
1.2	12.06.2013	Jochen Steiner	i.B.	Korrektur des Dokumentes
1.3	20.06.2013	Henrik Lange	i.B.	Klassendiagramm C120
1.4	20.06.2013	Dennis Stelter	i.B.	Beschreibung des Datenmodells für die Karte hinzugefügt
1.5	20.06.2013	Jochen Steiner	i.B.	Beschreibung zu Komponente C70
1.6	21.06.2013	Henrik Lange	i.B.	Paketdiagramm C120 und Text hinzugefügt
1.7	21.06.2013	Henrik Lange	i.B.	Layout Anpassung
1.8	21.06.2013	Dennis Stelter	i.B.	Diagramme und Erläuterung zu Komponente C90 hinzugefügt
1.9	21.06.2013	Tessa Fabian, Patrica-Tatjana Kasulke, Markus-Björn Meißner	i.B.	Klassendiagramme, Paketdiagramme und Texte hinzugefügt
1.10	22.06.2013	Henrik Lange	i.B.	Rechtschreibfehler korrigiert und Kapitel 7 ergänzt
1.11	24.06.2013	Jochen Steiner	i.B.	Korrektur der Komponente C70 in Kapitel 5
1.12	24.06.2013	Henrik Lange	i.B.	Korrektur der Komponente C120 in Kapitel 5

1.13	25.06.2013	Dennis Stelter	i.B.	Korrektur der Kompontente C90 in Kapitel 5 und der Erläuterung in Kapitel 6.1
1.14	25.06.2013	Henrik Lange	i.B.	Strukturierung des Dokumentes
1.15	25.06.2013	Patrica-Tatjana Kasulke	i.B.	Texte und Diagramme zum Kapitel 6 Datenmodell hinzugefügt
2.0	26.06.2013	Henrik Lange	abg.	Abgabe ISF

i.B.: in Bearbeitung, **abg.**: abgeschlossen

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das vorliegende Dokument stellt den Systementwurf des Projekts „NeXT Generation Transport Tycoon“ dar.
In dieser Phase wird primär auf Entwürfe und erste Planungen mit Hauptaugenmerk auf die Systemarchitektur eingegangen. Es werden also zwingend benötigte Funktionalitäten untersucht und es wird festgelegt, welche Architektur für ihre Realisierung benötigt wird. Anschließend erfolgt ein erster Entwurf einer entsprechenden Systemarchitektur (sowohl in Form von Software als auch in Form eines verteilten Systems), der die gefundenen Bedingungen erfüllt (Übersichtsdiagramm siehe Anhang A, zum Betrachten in DIN A3 ausdrucken).

1.1 Projektdetails

Besonders interessante oder komplizierte Sachverhalte werden hier noch weiter vertieft.

1.1.1 Kollisionsvermeidung

Ein komplizierter Sachverhalt des Projektes „NeXT Generation Transport Tycoon“ ist die Vermeidung von Kollisionen der Roboter. Hierfür muss der Server in Echtzeit Informationen über die geplante Route aller aktiven Roboter und ihren Standort erhalten. Mithilfe dieser Informationen müssen „besetzte“ Pfade temporär gesperrt werden und eventuell ganze Routen neu geplant werden. Die Komplexität dieser Aufgabe steigt mit einer größeren Zahl von aktiven Robotern stark an, da es ab einer bestimmten Zahl nur noch schwer möglich ist, für jeden NXT komplette Routen zwischen zwei Punkten herzustellen.

1.1.2 Vermeidung von Deadlocks

Ein weiteres Problem sind die sogenannten Deadlocks, also Situationen, in denen alle beteiligten Roboter handlungsunfähig sind, da alle möglichen Routen durch andere Roboter versperrt sind. Solche Deadlocks können auftreten, wenn Roboter in eine Sackgasse fahren und ein anderer die einzige Zufahrt versperrt. In diesem Fall müssen die Roboter eine Möglichkeit besitzen, solche Situationen einerseits zu erkennen und andererseits entsprechende Lösungsverfahren zu starten, den Robotern evtl. sogar ein Verhalten aufzwingen, dass zwar gegen ihren aktuellen Auftrag

verstößt, aber für das Beheben des Deadlocks unabdingbar ist.
Sobald jede Partei im Spiel mehr als einen Roboter besitzt, kann das bewusste Herbeiführen von Deadlocks jedoch auch ein Bestandteil einer Strategie sein, um den Gegner durch Blockieren zum Ausweichen zu zwingen.

1.1.3 Generieren und Zuweisen von Transportaufträgen

Der Server soll allen Spielern Transportaufträge anbieten können, die zufällig generiert werden und den Transport von Waren zwischen zwei Punkten A und B erfordern.
Diese Aufträge müssen assoziierte Werte wie Entlohnung, Dauer und Kapazität haben. Weiterhin müssen alle Parteien auf diese Aufträge „bieten“, d.h. sie versuchen, sich gegenseitig zu unterbieten. Die Partei, die das günstigste Angebot macht, erhält den Zuschlag und kann den Auftrag ausführen.
Hierbei ist es wichtig, eine entsprechende KI zu implementieren, die dieses Bieten korrekt und strategisch ausführt. Weiterhin muss das Zuweisen von Robotern zu neuen Aufträgen möglich sein. Im Idealfall kann ein derzeit „besetzter“ Roboter seine aktuelle Route anpassen, um mehrere Aufträge zur gleichen Zeit ausführen zu können.

2 Analyse der Produktfunktionen

In diesem Kapitel werden die Zusammenhänge der einzelnen Funktionen über Sequenzdiagramme erläutert. Dadurch soll eine geeignete Architektur auf Basis der im Pflichtenheft gefundenen Produktfunktionen und nicht-funktionalen Anforderungen gefunden werden.

2.1 Analyse von Funktionalität $\langle F10 \rangle$: Bluetooth-Datenverbindung

Roboter und Server kommunizieren miteinander über eine Bluetooth-Verbindung. Die jeweiligen Kommunikations-Komponenten bauen dazu vor Spielbeginn die Verbindung auf. Während des Spiels werden Daten dann in einem Integer-Stream übertragen. Dabei können Daten sowohl vom Server zum Roboter als auch vom Roboter zum Server gesendet werden.

Abbildung 2.1 stellt die Funktionalität in einem Sequenzdiagramm dar. Der Roboter wartet auf eine Verbindung mit dem Server. Der Server baut mit der Verbindungsanfrage zu jedem der Roboter einen eigenen Ein-/Ausgabestrom auf. Während des Spiels sendet der Server den Robotern Auktionsanfragen, Streckenfreigaben und Wegenetzdaten zu. Der Roboter nutzt die Bluetooth-Verbindung zum Bieten auf Auktionen sowie für Freigabeanfragen bezüglich einzelner Streckenabschnitte. Wenn das Spiel beendet wurde, werden die Verbindungskanäle wieder geschlossen.

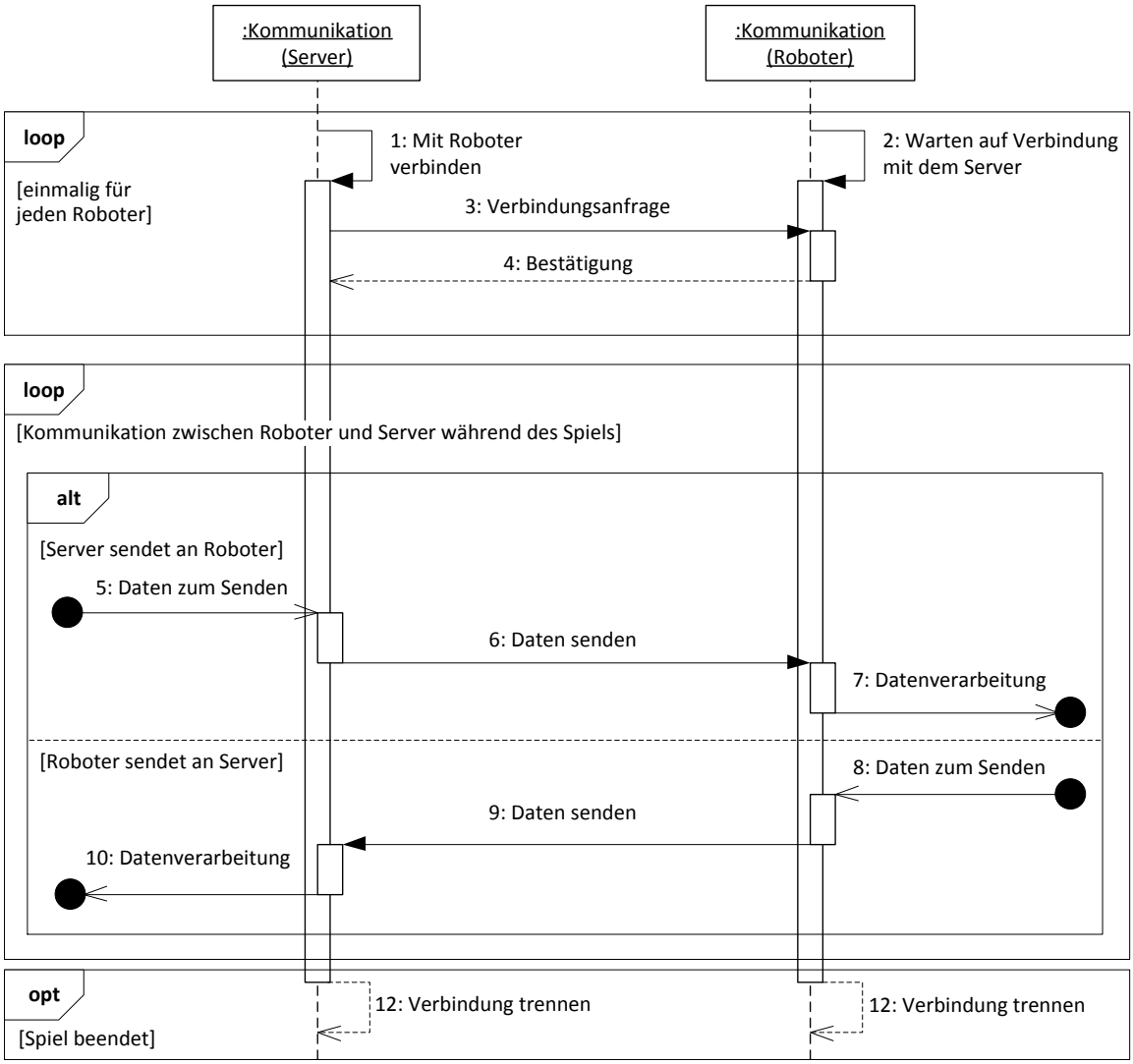

Abbildung 2.1: Sequenzdiagramm $\langle F10 \rangle$ „Bluetooth-Datenverbindung“

2.2 Analyse von Funktionalität $\langle F20 \rangle$: Datenverwaltung

Vor Spielbeginn müssen die notwendigen Daten (Wegenetz, Ressourcen, Produktionszyklus, Roboter-Startpositionen) auf den Server geladen werden. Erst wenn dieser Vorgang abgeschlossen ist, kann ein Spiel gestartet werden.
Während des Spiels verwaltet der Server die durch ausstehende oder geleistete Aufträge anfallenden Daten und aktualisiert die sich daraus ergebenden Statistiken (Preisbildung, Gewinne). Diese Statistiken werden später sowohl in der Spieler-GUI $\langle F110 \rangle$, als auch in der Benutzer-GUI angezeigt.

Abbildung 2.2 stellt die Initialisierung in einem Sequenzdiagramm dar. Der Benutzer startet das Programm mit einer ausführbaren Datei. Daraufhin hat er die Möglichkeit, alle benötigten Daten über eine spezielle GUI einzugeben. Die Korrektheit wird nach Eingabe geprüft oder es wird eine bestimmte Auswahl vorgegeben. Beispielsweise dürfen sich an einem Standort keine zwei Roboter befinden. Der Server soll nur korrekte Daten speichern und gibt dem Benutzer Rückmeldungen bezüglich dieser. Da sich Benutzer-GUI sowie Verwaltung (Server) auf dem selben Rechner befinden, wird keine besondere Kommunikationskomponente verwendet.

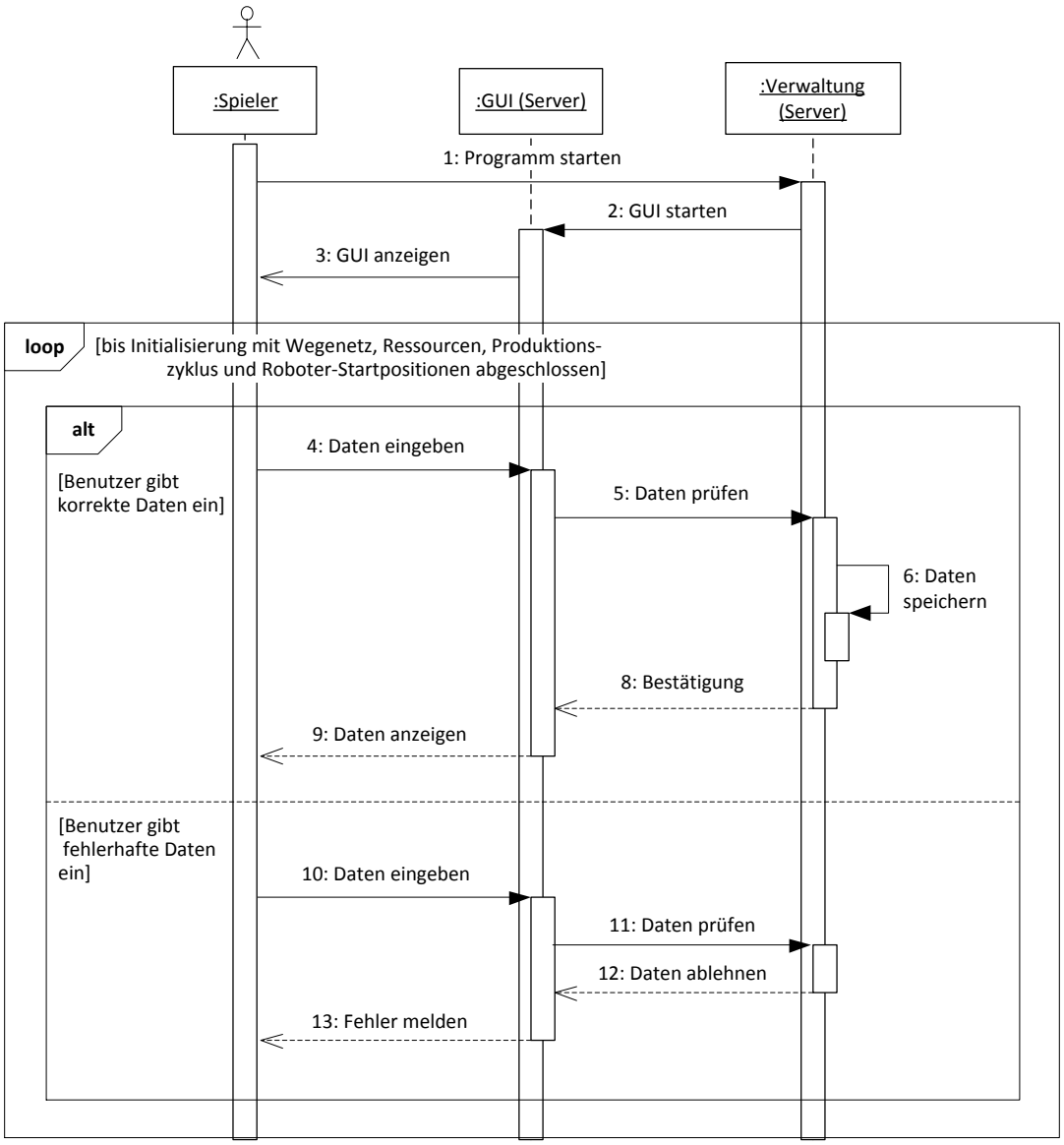

Abbildung 2.2: Sequenzdiagramm $\langle F20 \rangle$ „Initialisierung“

Die Funktionalität Verwaltung wird in Abbildung 2.3 dargestellt. Nach jedem erfolgreich abgeschlossenen Auftrag muss die Gewinnstatistik aktualisiert werden. Mit Veränderungen der Auftragslage werden Nachfrage und Preise eines Produkts angepasst. Der Benutzer hat die Möglichkeit, Aufträge zu erstellen. Die GUI greift dabei direkt auf Methoden der Serververwaltung zu.

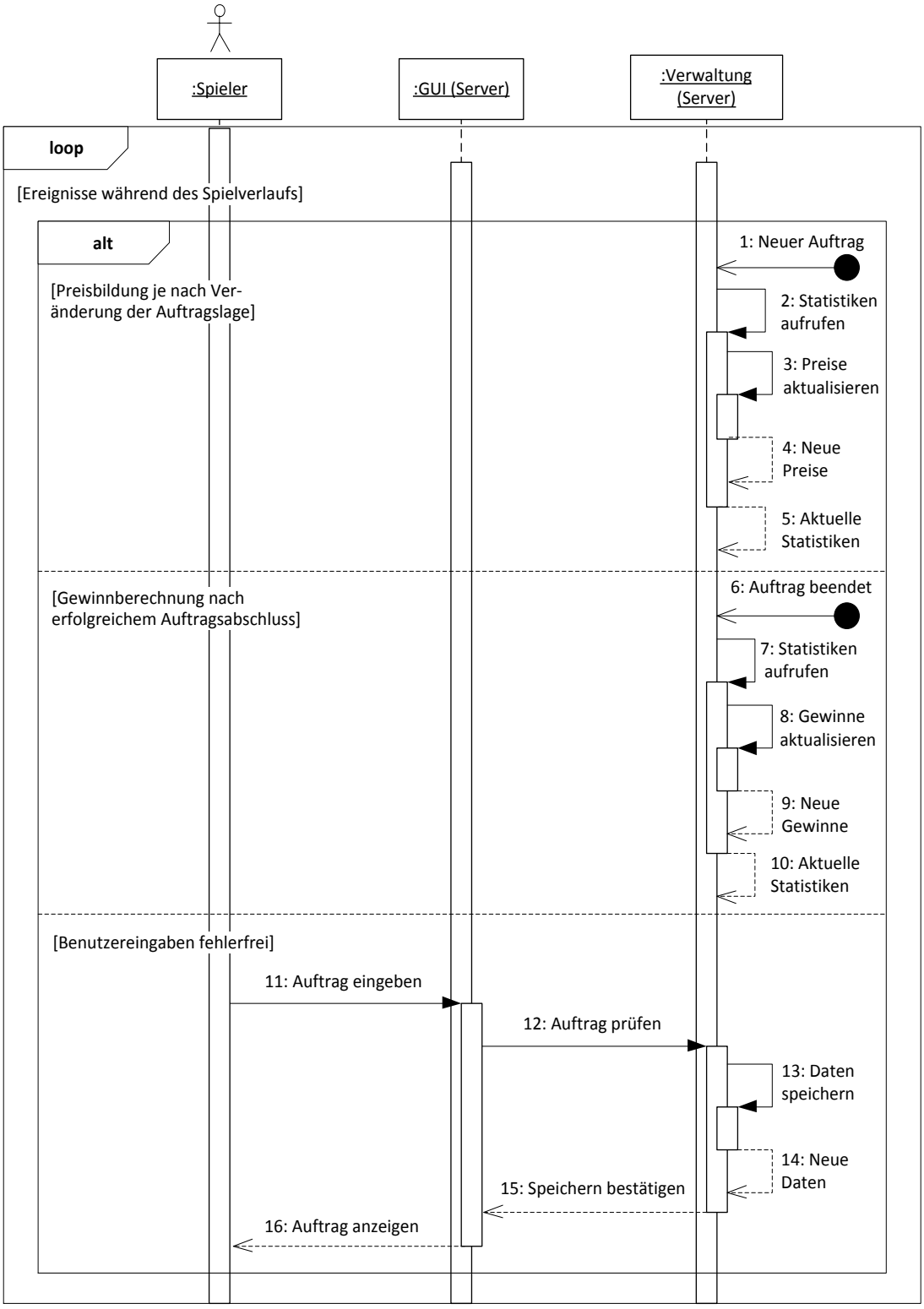

Abbildung 2.3: Sequenzdiagramm $\langle F20 \rangle$ „Verwaltung“

2.3 Analyse von Funktionalität $\langle F30 \rangle$: Initialisierung der Roboter

Die Roboter werden über Bluetooth-Verbindungen zuerst registriert und später initialisiert.

Abbildung 2.4 stellt die Funktionalität in einem Sequenzdiagramm dar. Sowohl Registrierung als auch Initialisierung nutzen eine Bluetooth-Verbindung $\langle F10 \rangle$. Dabei werden die Daten zwischen den jeweiligen Kommunikationskomponenten als Integer-Stream übertragen.
Jede Anfrage bzw. Übertragung der Verwaltungskomponente des Servers wird von der KI des Roboters bestätigt. Die Initialisierung der Roboter kann erst erfolgen, wenn die Initialisierung des Servers beendet wurde und ist erst dann erfolgreich abgeschlossen, wenn alle registrierten Roboter die Daten des Wegenetzes erhalten haben.

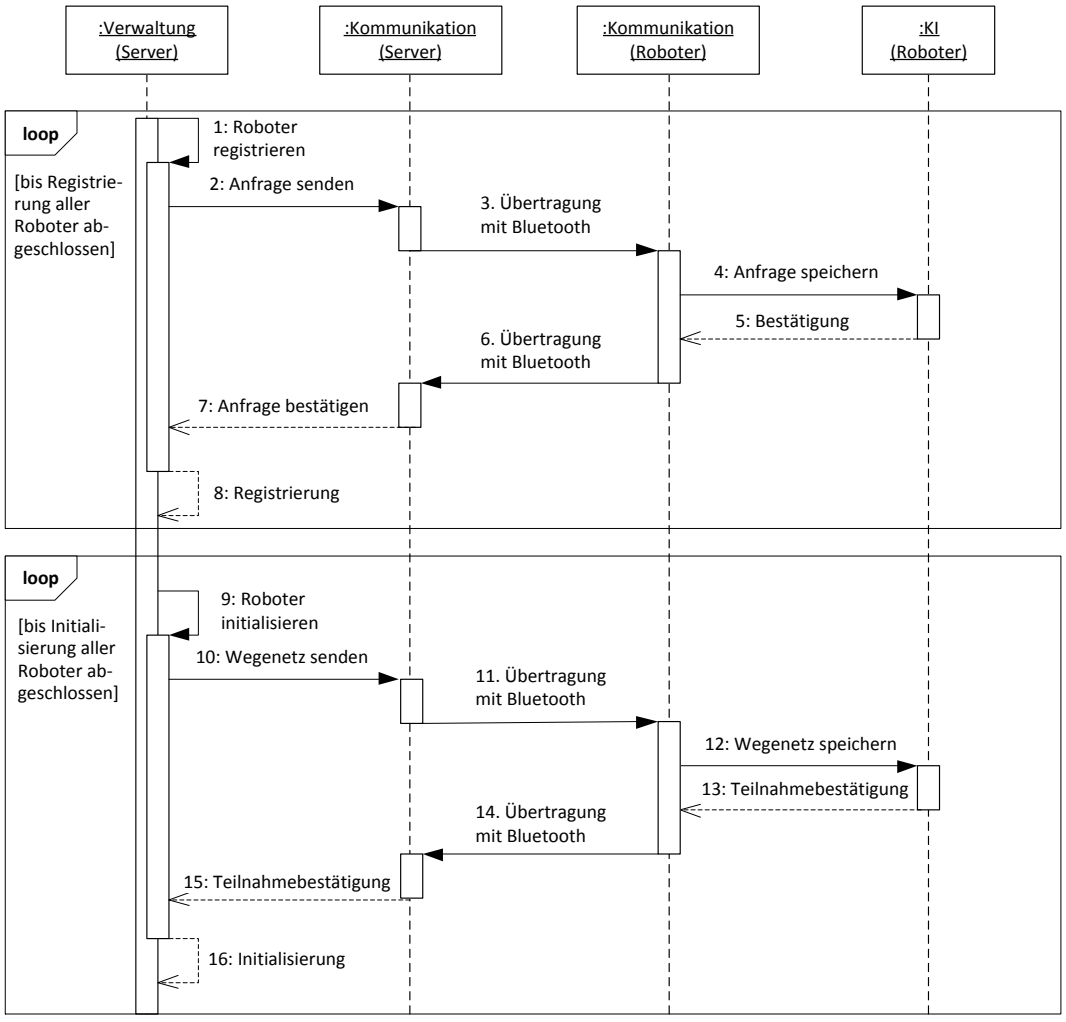

Abbildung 2.4: Sequenzdiagramm $\langle F30 \rangle$ „Initialisierung der Roboter“

2.4 Analyse von Funktionalität $\langle F40 \rangle$: Straßenerkennung durch Roboter

Der Roboter soll in der Lage sein, dem Straßenverlauf folgen zu können. Um dies zu erreichen, wertet er seine Sensoren aus und anhand deren Daten korrigiert er die Geschwindigkeit seiner Räder.
Als Entscheidungsgrundlage dienen die Helligkeitswerte, die die Sensoren mithilfe einer geeigneten Methode zurückgeben. Es gibt vier mögliche Szenarien:

- Beide Sensoren liefern helle Farbstufen zurück
- Der linke Sensor liefert eine dunkle Farbstufe zurück
- Der rechte Sensor liefert eine dunkle Farbstufe zurück
- Beide Sensoren liefern dunkle Farbstufen zurück

Die Abbildung 2.5 stellt die Funktion als Sequenzdiagramm dar. Der Roboter fragt fortlaufend seine Sensoren ab. Solange beide helle Farben messen, folgt der Roboter der schwarzen Linie. Wenn links oder rechts eine dunkle Farbstufe gemessen wird, wird dementsprechend die Geschwindigkeit des linken oder rechten Rades heruntergeregelt, um eine dementsprechende Drehung vorzunehmen. Sobald der linke und der rechte Sensor eine dunkle Farbstufe melden, ist das Straßenende erreicht. Eine Drehung um 180° ist erforderlich.

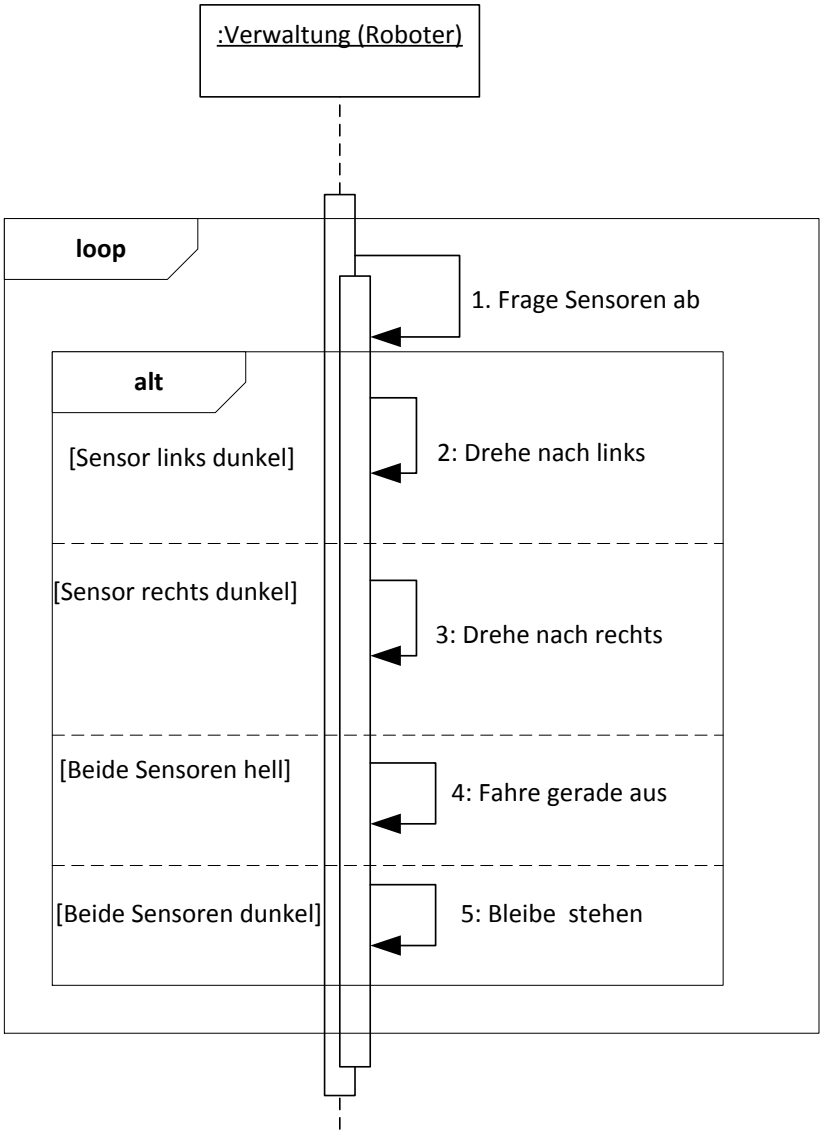

Abbildung 2.5: Sequenzdiagramm $\langle F40 \rangle$ „Straßenerkennung durch Roboter“

2.5 Analyse von Funktionalität $\langle F50 \rangle$: Transportvolumen der Roboter

Damit ein Roboter nicht beliebig viele Waren zur gleichen Zeit transportieren kann, muss implementiert werden, dass er vor der Abgabe des Auktionsgebotes prüft, ob er genügend Kapazität zur Verfügung hat.

Diese Problematik wird in dem Sequenzdiagramm in Abbildung 2.6 behandelt. Der Roboter überprüft nach Erhalt des Auftragsangebotes, ob genügend Kapazitäten vorhanden sind, um diesen Auftrag durchzuführen. Hat er genügend Volumen, so kann er ein Gebot für den Auftrag abgeben, wenn nicht, muss er den Auftrag ablehnen.

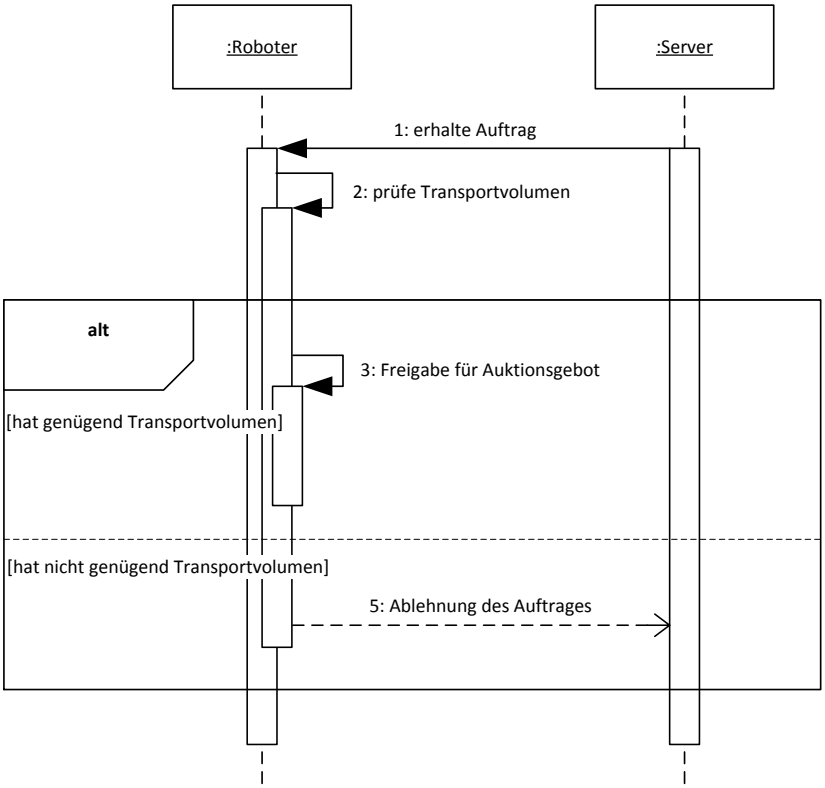

Abbildung 2.6: Sequenzdiagramm $\langle F50 \rangle$ „Transportvolumen der Roboter“

2.6 Analyse von Funktionalität $\langle F60 \rangle$: Treibstoffverbrauch der Roboter

Der Roboter braucht zur Bearbeitung seiner Transporte einen virtuellen Treibstoff. Er kennt die Restmenge seines Treibstoffvorrats und plant seine Routen unter Berücksichtigung der Treibstoffmenge.
Der Treibstoffverbrauch ist zeitabhängig und ohne Treibstoff fährt der Roboter nicht weiter. Der Roboter erledigt Aufträge und erhält nach erfolgreicher Ausführung eine Betankung.

In dem Sequenzdiagramm in Abbildung 2.7 wird die Funktion näher erörtert. Der Roboter verbraucht seinen Treibstoff. Falls er einen Auftrag erfolgreich erledigt hat, wird er betankt. Ist sein Treibstoffvorrat leer, scheidet er aus.

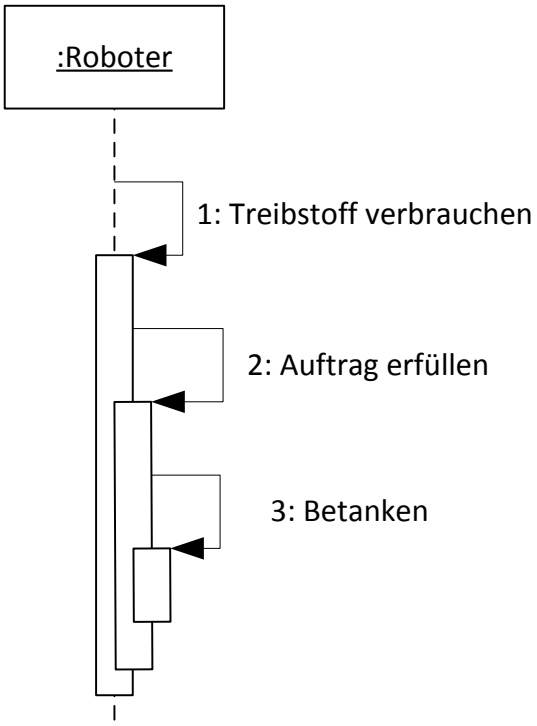

Abbildung 2.7: Sequenzdiagramm $\langle F60 \rangle$ „Treibstoffverbrauch der Roboter“

2.7 Analyse von Funktionalität $\langle F70 \rangle$: Streckenfreigabe durch Server

Der Roboter bearbeitet einen Auftrag und hat zuvor eine Route berechnet. Danach bittet er um Freigabe eines Streckenabschnitts. Der Server verwaltet die Strecken durch eine Adjazenzmatrix. Falls der Roboter eine Freigabe erhält, darf er in den freigegeben Streckenabschnitt fahren. Für alle anderen Roboter ist dieser Streckenabschnitt dann gesperrt. Der Streckenabschnitt wird freigegeben, wenn der Roboter signalisiert hat, dass er den Abschnitt verlassen hat.

Die genaue Funktionsweise dieser Funktion, wird im Sequenzdiagramm in Abbildung 2.8 erklärt. Der Roboter stellt eine Anfrage, ob ein Streckenabschnitt, den er befahren will, frei ist. Diese Anfrage wird durch die Kommunikationsmodule von Roboter und Server an das Spielkoordinationsmodul weitergegeben. Die Spielkoordination leitet die Anfrage an das Wegenetzverwaltungsmodul weiter. Dort wird geprüft, ob der Streckenabschnitt frei ist.
Falls der Streckenabschnitt frei ist, wird eine Freigabe an den Roboter gesendet und der Roboter fährt in den Streckenabschnitt. Wenn der Roboter den alten Streckenabschnitt verlassen hat, wird dieser in der Wegenetzverwaltung freigegeben.
Falls der Streckenabschnitt nicht frei ist, erhält der Roboter keine Freigabe und wartet solange, bis er eine Freigabe erhält. Danach verfährt der Roboter nach demselben Muster wie bei sofortiger Freigabe.

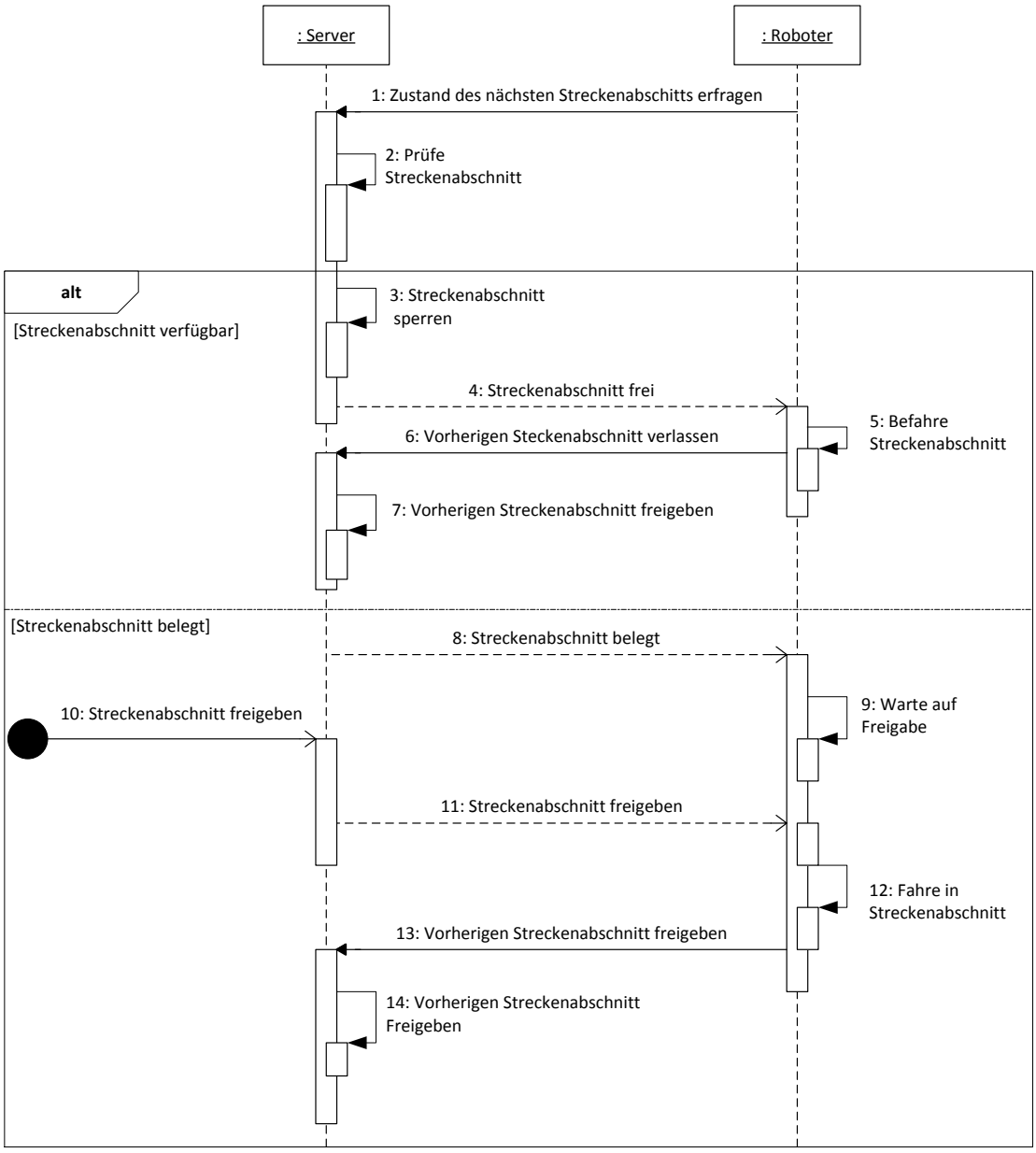

Abbildung 2.8: Sequenzdiagramm $\langle F70 \rangle$ „Streckenfreigabe durch Server“

2.8 Analyse von Funktionalität $\langle F80 \rangle$: Routenberechnung durch Roboter

Selbständige Wegewahl des Roboters. Der Roboter sucht sich selbständig den Weg durch das Wegenetz und kommuniziert stetig mit dem Server. Will der Roboter einen Abschnitt befahren, muss dieser eine Reservierungsanfrage an den Server senden. Dieser schaut, ob bereits eine Reservierung vorliegt oder der Abschnitt frei ist. Beim zweiten Fall reserviert der Server diesen Streckenabschnitt für diesen Roboter. Bis er dem Server das Verlassen des Abschnitts mitteilt. Nun kann der Abschnitt wieder freigegeben werden.

In dem Sequenzdiagramm, welches in der Abbildung 2.9 zu sehen ist, wird der zeitliche Ablauf der Kommunikation während der Routenberechnung durch den Roboter dargestellt.
Der Roboter fordert die Auftragsdaten vom Server an. Nach dem Erhalt der Auftragsdaten werden die Wegenetzdaten ausgewertet, damit der Roboter mit der Hilfe eines Algorithmus die jeweilige Strecke berechnen kann.
Darauf folgt eine Berechnung der für den Auftrag notwendigen Ressourcen, wie zum Beispiel der Treibstoffverbrauch. Nachdem die Berechnung abgeschlossen ist, sendet der Roboter die Daten an dem Server, der die Route auf Inkonsistenz überprüft. Wurde keine Inkonsistenz gefunden, sendet der Server eine positive Rückmeldung. Der Roboter bestätigt den Empfang der Nachricht und fährt die Route ab.
Bei einer negativen Rückmeldung, behebt der Roboter die Inkonsistenz und sendet dem Server die veränderten Daten zu. Der Server führt eine erneute Überprüfung durch.

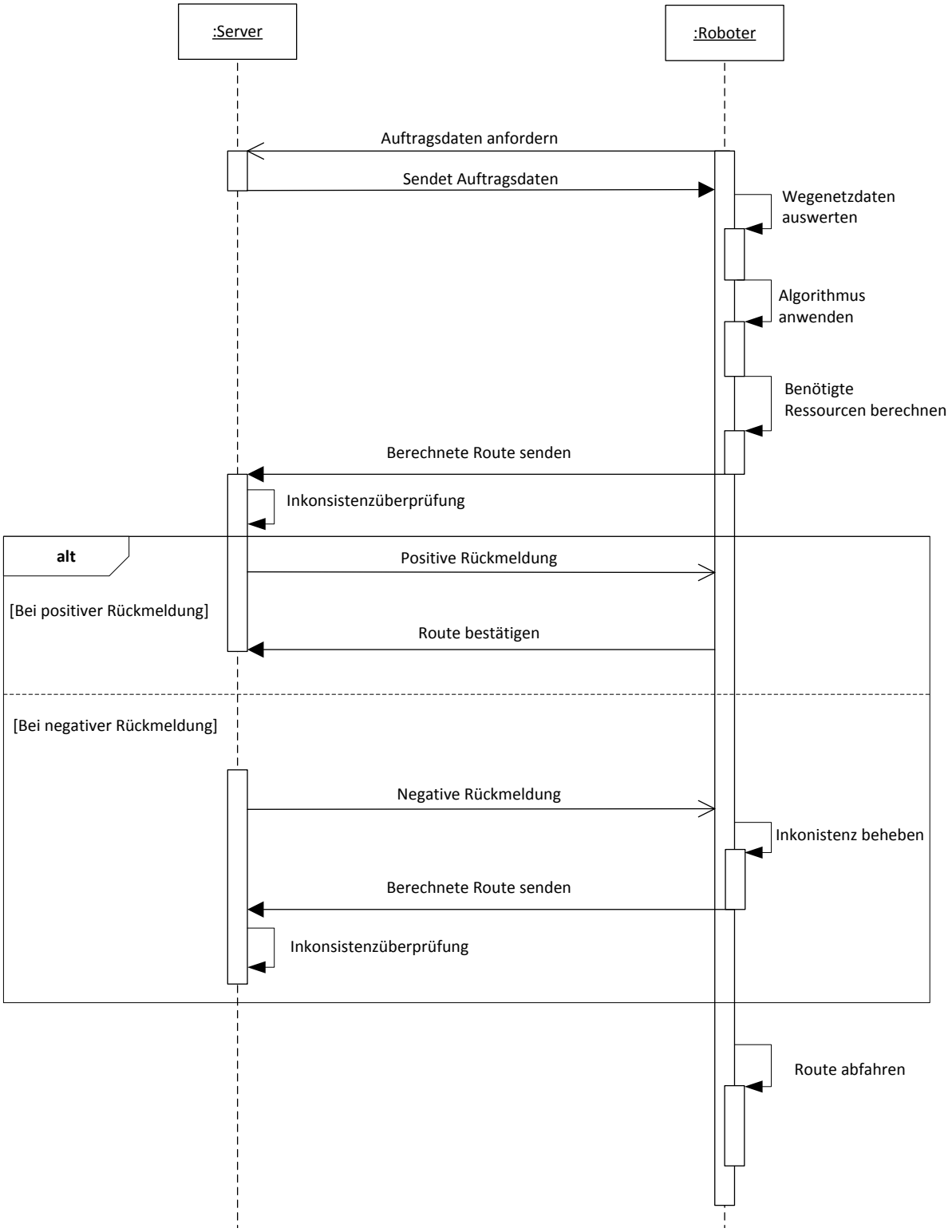

Abbildung 2.9: Sequenzdiagramm $\langle F80 \rangle$ „Routenberechnung durch Roboter“

2.9 Analyse von Funktionalität $\langle F90 \rangle$: Spielstart durch Benutzer

Das Spiel wird durch den Benutzer gestartet.

Im Sequenzdiagramm, in der Abbildung 2.10, wird näher auf die Funktion eingegangen. Nachdem der Benutzer den Startknopf gedrückt hat, erfolgt die Initialisierung ($\langle F20 \rangle$). Wenn diese erfolgreich verläuft, wird das Spiel gestartet. Schlägt die Initialisierung fehl, wird das Spiel nicht gestartet.

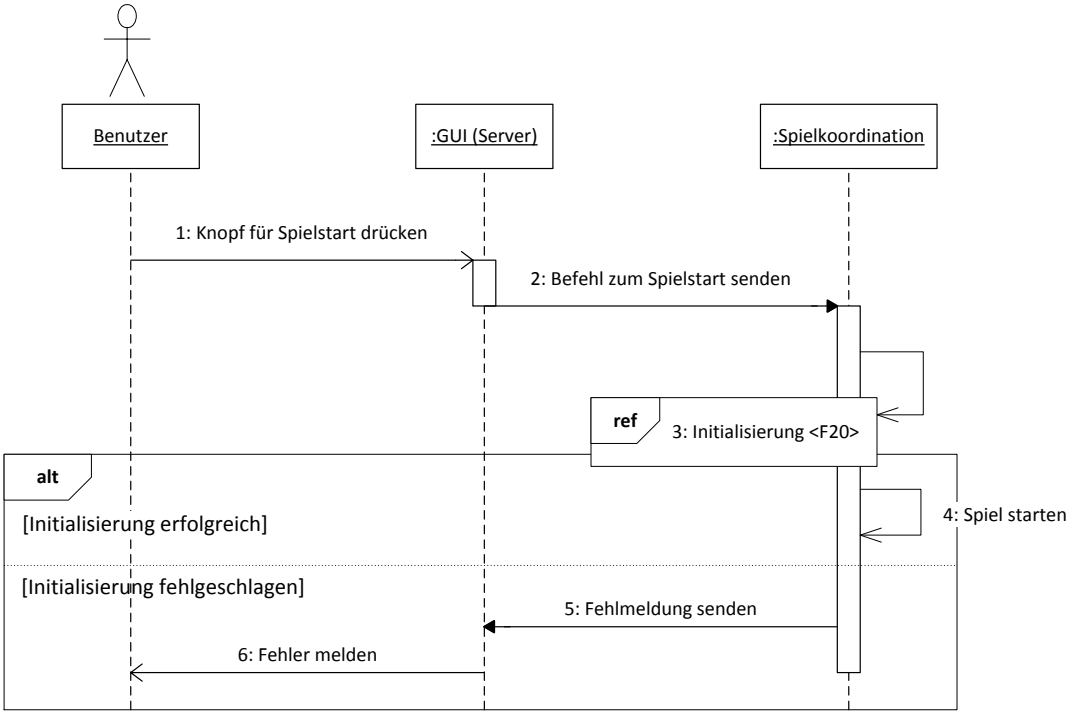

Abbildung 2.10: Sequenzdiagramm $\langle F90 \rangle$ „Spielstart durch Benutzer“

2.10 Analyse von Funktionalität $\langle F100 \rangle$: Spieldarstellung der Spieleroberfläche

Die Spieleroberfläche zeigt das Wegenetz mit den Standorten von Industrien und Robotern an, dabei werden zusätzlich Statistiken geladen, die den Spielern über den derzeitigen Stand der jeweiligen Objekte informiert.

In der Abbildung 2.11 wird die Spieldarstellung wie folgt dargestellt.
Wenn das Spiel vom Spieler gestartet wurde, wird die Spieleroberfläche geladen. Auf der Spieleroberfläche sieht er das aktuelle Wegenetz, die Standorte und Status der Roboter sowie die Industriestandorte, die visualisiert in Echtzeit dargestellt werden.
Dabei zeigt sie auch die Details, wie zum Beispiel vorhandene Waren zum Transport und gelagerte Waren zur Weiterverarbeitung, der einzelnen Industrien an.
Um das Wegenetz anzeigen zu können, werden Wegenetzdaten (Standorte der Industrie usw.) benötigt, die der Server bereithält und der GUI zur Verfügung stellt. Diese werden auf der Spieleroberfläche angezeigt.
Die Darstellung der Roboterstandorte wird durch die ständige Kommunikation zwischen Robotern und Server aktuell gehalten.

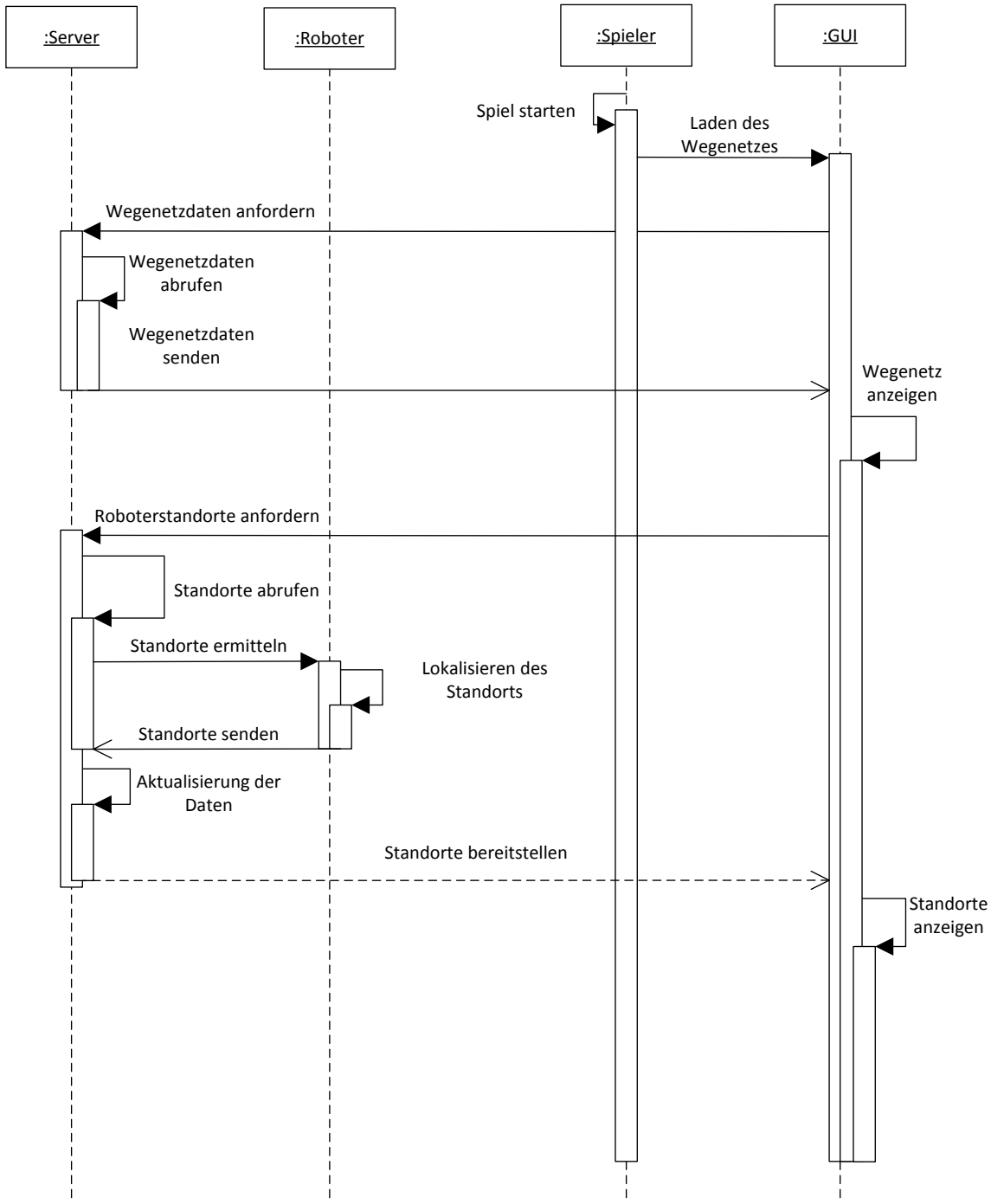

Abbildung 2.11: Sequenzdiagramm $\langle F100 \rangle$ „Spieldarstellung der Spieleroberfläche“

2.11 Analyse von Funktionalität $\langle F110 \rangle$: Statistikanzeige der Spieleroberfläche

Die Spieleroberfläche gibt Informationen über laufende Aufträge aus und zeigt zusätzlich Statistiken der Roboter an. Nachdem der Roboter seine Statistiken generiert und diese zur weiteren Verarbeitung an den Server übermittelt hat, werden dort Berechnungen durchgeführt. Das Ergebnis wird anschließend an den Client übertragen. Der Spieler kann die Ergebnisse dort aufrufen.

Das Sequenzdiagramm in der Abbildung 2.12, zeigt die Interaktion des Spielers beim Abruf von Statistiken der Roboter. Er kann diese am Client erst dann abrufen, wenn sie zuvor vom Server berechnet und an den Client übermittelt wurden. Dabei kommt es zwischen den Akteuren Server, Client und Roboter auf eine funktionierende Netzwerkverbindung an. Der Spieler arbeitet direkt am Client.

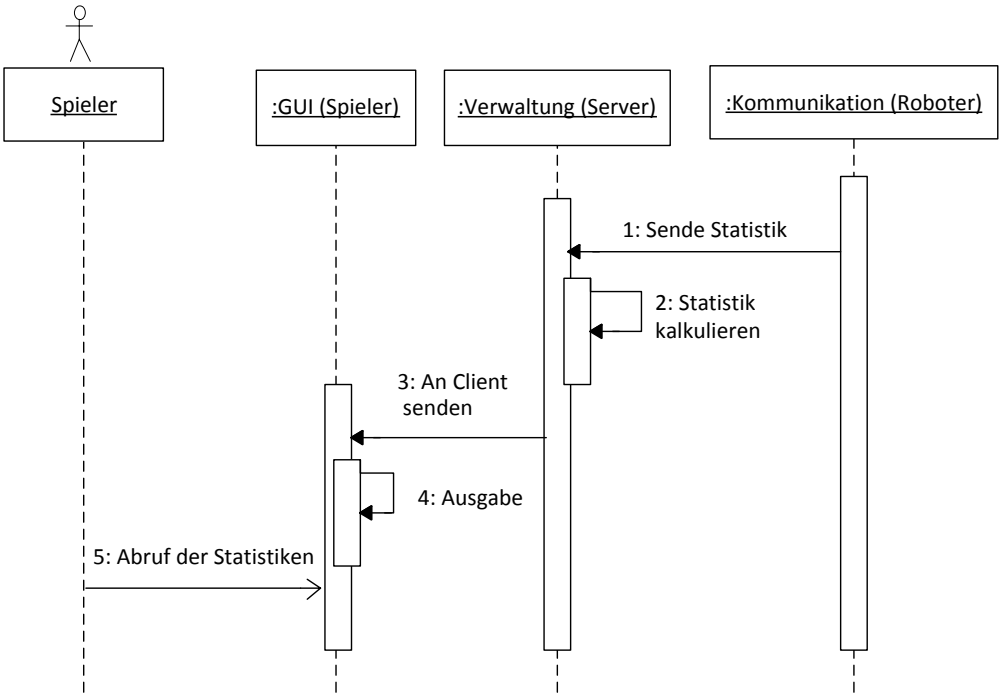

Abbildung 2.12: Sequenzdiagramm $\langle F110 \rangle$ „Statistikanzeige der Spieleroberfläche“

2.12 Analyse von Funktionalität $\langle F120 \rangle$: Auktionen für Transportaufträge

Der Server veranstaltet in bestimmten zeitlichen Abständen Auftrags-Auktionen. Solche Auktionen sind die einzige Möglichkeit, an neue Aufträge zu gelangen. Dabei müssen alle Teilnehmer entweder ihr Gebot abgeben oder den Auftrag ablehnen. Der Teilnehmer mit dem niedrigsten Gebot gewinnt die Auktion.

Abbildung 2 im Anhang stellt die Funktionalität in einem Sequenzdiagramm dar. Die Verwaltungkomponente des Servers steuert die Vergabe der Aufträge. Ein Auftrag wird entweder aus der Warteliste gewählt oder vom Benutzer vorgegeben. Eine neue Auktion wird gestartet und alle teilnehmenden Roboter bzw. Spieler werden mittels Bluetooth $\langle F10 \rangle$ bzw. TCP-Verbindung informiert. Bei eigenständigen Robotern berechnet die KI das bestmögliche Gebot. Im Falle eines Spielerroboters wird das Gebot vom Spieler selbst über die Spieler-GUI abgegeben. Nachdem alle Teilnehmer ein Gebot bzw. eine Ablehnung, beispielsweise im Falle von zu wenig Transportvolumen, abgegeben haben, ermittelt die Serververwaltung den Auktionsgewinner. Wurde der Gewinner gefunden, endet die Auktion. Spieler sollen über den Ausgang der Auktion benachrichtigt werden. Dem Roboter des Gewinners muss der Auftrag zugeteilt werden. Mit der Bestätigung, dass dieser in der Bearbeitungsliste des Roboters gespeichert wurde, wird der Auftrag daraufhin aus der Liste des Servers gelöscht.

Das Sequenzdiagramm befindet sich im Anhang im Abschnitt B und sollte in DIN A3 ausgedruckt werden.

2.13 Analyse von Funktionalität $\langle F130 \rangle$: Vermeidung von Deadlocks

Damit es einen reibungslosen Spielverlauf gibt, muss es eine Möglichkeit geben wie Deadlocks vermieden werden können, dazu wurde ein Algorithmus gefunden der dies tut.

Im Sequenzdiagramm in der Abbildung 3, die sich im Anhang befindet, wird dieser näher dargestellt. Um Deadlocks zu vermeiden, müssen verschiedene Dinge geprüft werden: so muss geschaut werden, ob der Roboter eine Alternativ-Route zum Ziel finden kann, ohne die blockierte Kante zu verwenden. Ist dies möglich, so wird sie als aktuelle Route verwendet und die alte Route zum Ziel ersetzt. Da es nicht immer möglich ist, eine alternative Route zu finden, muss der Roboter für die Situation gesagt bekommen, was zu tun ist.
So wartet der Roboter eine definierte Zeit ab, bevor er nochmals nachfragt, ob die Route freigegeben wurde. Dies muss getan werden, da es gut sein kann, dass der andere Roboter einfach vorbei fährt, da dieser nicht diesen Streckenabschnitt befahren wollte oder schon eine alternative Route berechnet hat. Ist dies aber nicht der Fall und der benötigte Streckenabschnitt ist immer noch belegt, so wird geprüft, ob es eine anliegende Kante gibt, die derzeitig frei ist und zum Ausweichen benutzt werden kann. Beim Ausweichen fährt der Roboter eine gewisse Länge auf die freie Kante, wendet dort und fragt den Server, ob der benötige Streckenabschnitt nun zum Befahren freigegeben ist.
Falls aber keine Kante frei zum Ausweichen war, muss der Roboter, falls er sich nicht in einer Sackgasse befindet, zum vorherigen Knoten zurück fahren und dort ausweichen. Dadurch wird der Deadlock aufgelöst, da die Auflösung auch bei mehreren Robotern schrittweise erfolgt.

Das Sequenzdiagramm befindet sich im Anhang im Abschnitt C und sollte in DIN A3 ausgedruckt werden.

2.14 Analyse von Funktionalität $\langle F140 \rangle$: Spieldarstellung über Benutzeroberfläche

Die Benutzeroberfläche zeigt das Wegenetz, Aufträge und Statistiken an.
In der Spieldarstellung werden Aufträge über die GUI an den Server gesendet, Roboter können aus dem Spiel entfernt werden und das Wegenetz wird vom Server aktualisiert und angezeigt.
Aufträge: Der Benutzer erstellt über die GUI des Servers einen Auftrag. Dieser wird von der GUI an die Spielkoordination übermittelt. Die Spielkoordination leitet den Auftrag anschließend an die Auktionsverwaltung weiter, welche diesen Auftrag dann zur Auktion anbietet ($\langle F120 \rangle$).

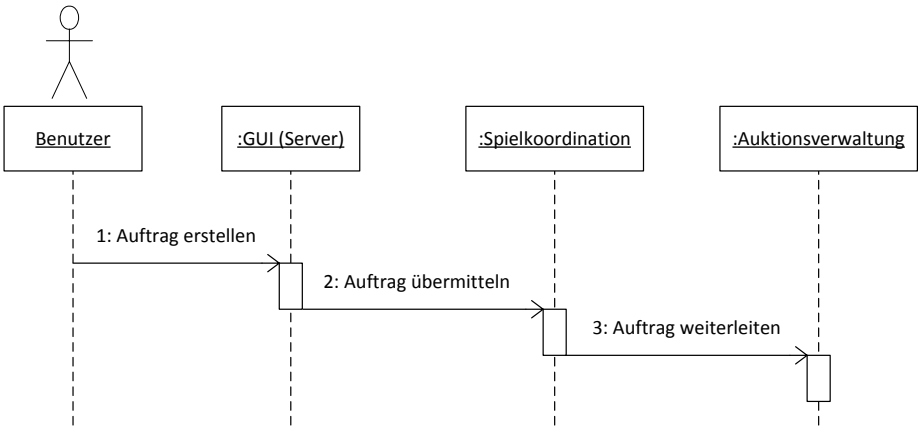

Abbildung 2.13: Sequenzdiagramm $\langle F140 \rangle$ zum Senden eines Auftrages

Roboter: Wenn der Benutzer einen Roboter aus dem Spiel entfernen möchte, wählt er diesen über die GUI des Servers aus. Dieser übermittelt anschließend den Befehl zur Entfernung des Roboters an die Spielkoordination.

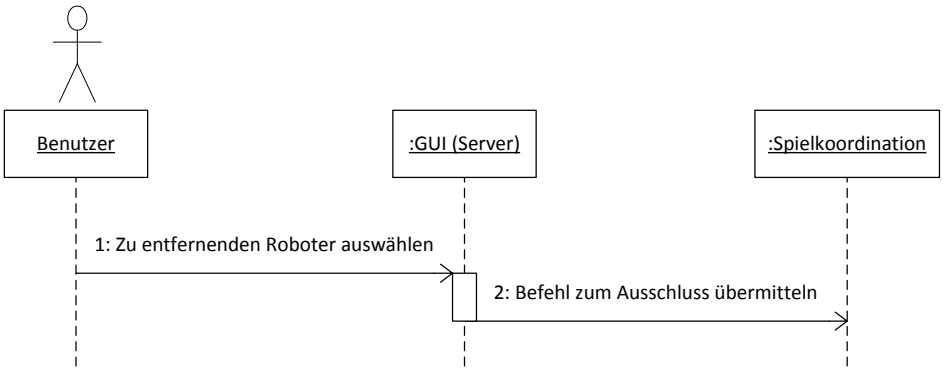

Abbildung 2.14: Sequenzdiagramm $\langle F140 \rangle$ zum Entfernen eines Roboters

Wegenetz: Damit der Benutzer einen aktuellen Überblick über das Wegenetz mit belegten und freien Streckenabschnitten und den Positionen der Roboter hat, wird die Visualisierung in der GUI in regelmäßigen Abständen aktualisiert. Dazu schickt die Spielkoordination eine Anforderung zur Aktualisierung an die Wegenetzverwaltung. Diese ruft das aktuelle Wegenetz ab und sendet die Informationen an die Spielkoordination, damit sie an die GUI übertragen werden können.

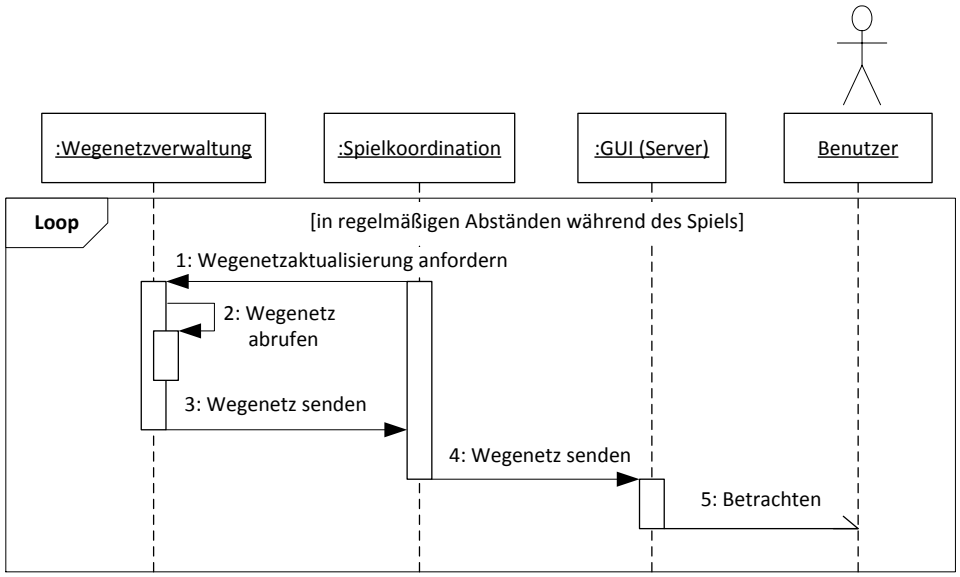

Abbildung 2.15: Sequenzdiagramm $\langle F140 \rangle$ zur Visualisierung des Wegenetzes

3 Resultierende Softwarearchitektur

Eine gute Softwarearchitektur ist grundlegend für eine gelungene Umsetzung der im Pflichtenheft formulierten Produktfunktionen. In diesem Kapitel wird die aus ihnen resultierende Softwarearchitektur vorgestellt.

3.1 Komponentenspezifikation

Zunächst werden die Komponenten der Software mit Hilfe eines Diagramms und einer näheren Beschreibung vorgestellt.

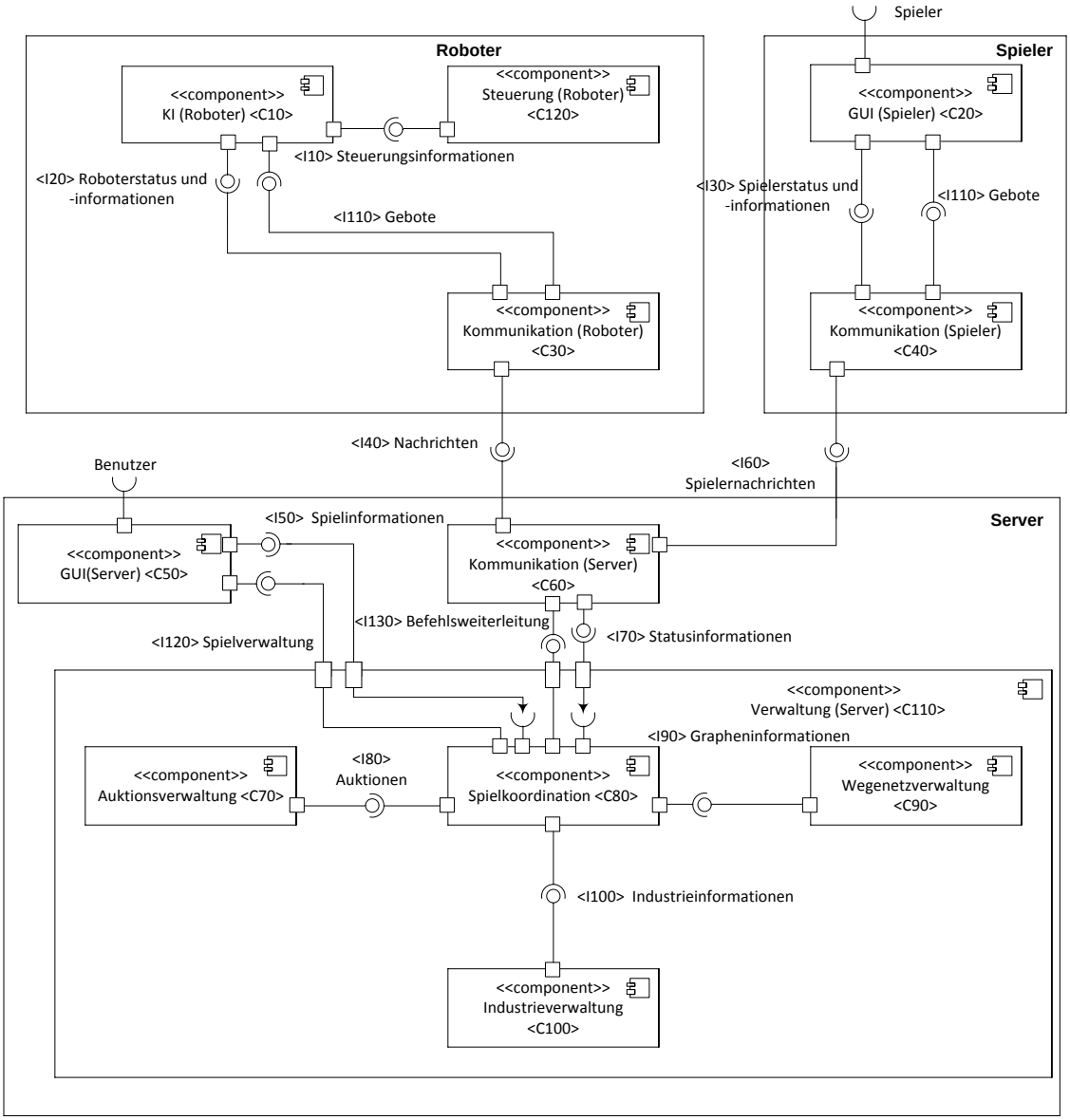

Abbildung 3.1: Komponentendiagramm

Es haben sich insgesamt zwölf Komponenten herauskristallisiert, die die komplette Funktionalität des Programms abdecken. Sie werden nachfolgend vorgestellt.

Komponente $\langle C10 \rangle$**: <KI (Roboter)>**
Die KI ist für die Entscheidungen zuständig, die der Roboter als Teilnehmer treffen muss.

Komponente $\langle C20 \rangle$**: <GUI (Spieler)>**
Die GUI visualisiert für den Spieler den Ablauf des Spiels und ermöglicht Eingaben wie z.B Gebote und Routenänderungen.

Komponente $\langle C30 \rangle$**: <Kommunikation (Roboter)>**
Das Kommunikationsmodul des Roboters ermöglicht die Kommunikation zum Server via Bluetooth.

Komponente $\langle C40 \rangle$**: <Kommunikation (Spieler)>**
Das Kommunikationsmodul des Spielers ermöglicht die Kommunikation zum Server via TCP/IP.

Komponente $\langle C50 \rangle$**: <GUI (Server)>**
Die GUI erlaubt dem Benutzer die Administration des Spiels.

Komponente $\langle C60 \rangle$**: <Kommunikation (Server)>**
Das Kommunikationsmodul des Servers ermöglicht die Kommunikation zum Roboter sowie zum Spieler über das jeweilige Protokoll.

Komponente $\langle C70 \rangle$**: <Auktionsverwaltung>**
Die Auktionsverwaltung nimmt Auftragsauktionen und Gebote entgegen und ermittelt einen Gewinner von den Auktionen.

Komponente $\langle C80 \rangle$**: <Spielkoordination>**
Die Spielkoordination ist für den reibungslosen Ablauf des Spiels zuständig.

Komponente $\langle C90 \rangle$**: <Wegenetzverwaltung>**
Die Wegenetzverwaltung speichert das zugrunde liegende Wegenetz. Sie ist für die Standorterfassung der Roboter sowie für die Streckenfreigaben und -sperrungen zuständig.

Komponente $\langle C100 \rangle$**: <Industrieverwaltung>**
Die Industrieverwaltung administriert die Industrien. Dies bedeutet, dass die hergestellten Produkte durch Aufträge wegtransportiert werden können. Zur Herstellung von Produkten werden Waren, die angeliefert werden, benötigt.

Komponente $\langle C110 \rangle$**: <Verwaltung (Server)>**
Die Verwaltung des Servers vereinigt die Komponenten $\langle C80 \rangle$, $\langle C90 \rangle$, $\langle C70 \rangle$ und $\langle C100 \rangle$. Damit dient sie als Verwaltungskomponente des Spiels.

Komponente $\langle C120 \rangle$**: <Steuerung (Roboter)>**
Die Steuerung des Roboters sorgt dafür, dass der Roboter seine Strecke gemäß der Befehle der KI abfährt.

3.2 Schnittstellenspezifikation

Innerhalb des Komponentendiagramms kristallisierten sich nachfolgende Schnittstellen sowie deren Operationen heraus. Sie werden nachfolgend erläutert.

Schnittstelle $\langle I10 \rangle$: <Steuerunginformationen>

Operation	Beschreibung
Steuerungsinformationen senden	Die KI sendet Befehle zur Manövrierung an die Steuerung.
Aktionspunkte zurückgeben	Die Steuerung übermittelt die Ankunft an Aktionspunkten, an denen die KI neue Befehle übermitteln muss.

Schnittstelle $\langle I20 \rangle$: <Roboterstatus und -informationen>

Operation	Beschreibung
Standort übermitteln	Die KI gibt über das Kommunikationsmodul ihre Position an.
Grunddaten senden	Die KI erhält vom Kommunikationsmodul zu Beginn Grunddaten.
Auftragsangebot weiterleiten	Das Kommunikationsmodul des Roboters leitet Auftragsangebote weiter.
Auftragsbestätigungen weiterleiten	Das Kommunikationsmodul des Roboters leitet Auftragsbestätigungen weiter.

Schnittstelle $\langle I30 \rangle$: <Spielerstatus und -informationen>

Operation	Beschreibung
Spielerstatus und -informationen übertragen	Die Kommunikation aktualisiert Spielstatus und -informationen (z.B. Wegenetzdaten).

Schnittstelle $\langle I40 \rangle$: <Nachrichten>

Operation	Beschreibung
Nachricht senden	Operationen von $\langle I20 \rangle$ und $\langle I110 \rangle$ werden zwischen den beiden Kommunikationsmodulen von Roboter und Server weitergeleitet.

Schnittstelle $\langle I50 \rangle$: <Spielinformationen>

Operation	Beschreibung
Spielinformationen übermitteln	Die Verwaltung bzw. Spielkoordination aktualisiert Spielstatus und -informationen (z.B. Wegenetzdaten).

Schnittstelle $\langle I60 \rangle$: <Spielernachrichten>

Operation	Beschreibung
Spielernachrichten übertragen	Operationen von $\langle I30 \rangle$ und $\langle I110 \rangle$ werden zwischen den beiden Kommunikationsmodulen von Spieler und Server weitergeleitet.

Schnittstelle $\langle I70 \rangle$: <Statusinformationen>

Operation	Beschreibung
Statusinformationen anfragen	Die Spielkoordination stellt Anfragen zur Ermittlung von Statusinformationen an das Kommunikationsmodul.

Schnittstelle $\langle I80 \rangle$: <Auktionen>

Operation	Beschreibung
Erstellte Auktionen weiterleiten	Die Spielkoordination sendet vom Benutzer oder von der Industrieverwaltung eingegebene Aufträge an die Auktionsverwaltung.
Auktionen an Teilnehmer empfangen	Die Auktionsverwaltung übergibt zu versteigernde Aufträge an die Spielkoordination, die diese an die Roboter bzw. Spieler weiterleitet.
Gebote von Teilnehmern senden	Die Spielkoordination sendet Gebote an die Auktionsverwaltung.
Auktionsgewinner empfangen	Die Spielkoordination empfängt den Auktionsgewinner von der Auktionsverwaltung.

Schnittstelle $\langle I90 \rangle$: <Grapheninformationen>

Operation	Beschreibung
Roboterposition senden	Die Spielkoordination sendet die Standorte der Roboter an die Wegenetzverwaltung.
Streckensperrung senden	Die Spielkoordination sendet die Sperrung einer Strecke an die Wegenetzverwaltung.
Streckenfreigabe senden	Die Spielkoordination sendet die Freigabe einer Strecke an die Wegenetzverwaltung.
Grunddaten senden	Die Spielkoordination sendet zu Beginn des Spiels die grundlegenden Daten des Wegenetztes an die Wegenetzverwaltung.
Status einer Strecke anfragen	Die Spielkoordination fragt den Status einer Strecke bei der Wegenetzverwaltung an.

Schnittstelle $\langle I100 \rangle$: <Industrieinformationen>

Operation	Beschreibung
Grunddaten senden	Die Spielkoordination sendet zu Beginn des Spiels die grundlegenden Daten der Industrien an die Industrieverwaltung.
Warenankunft übermitteln	Die Spielkoordination sendet nach erfolgreicher Abarbeitung eines Auftrags die Ankunft der transportierten Waren an die Industrieverwaltung.
Aufträge weiterleiten	Die Spielkoordination leitet Aufträge von der Industrieverwaltung an die Auktionsverwaltung weiter.

Schnittstelle $\langle I110 \rangle$: <Gebote>

Operation	Beschreibung
Gebote senden	Die GUI des Spielers bzw. die KI des Roboters sendet Gebote an das jeweilige Kommunikationsmodul zur Weiterleitung an den Server.

Schnittstelle $\langle I120 \rangle$: <Spielverwaltung>

Operation	Beschreibung
Aufträge erstellen	Die GUI gibt neu erstellte Aufträge an die Spielkoordination weiter.
Roboter entfernen	Die GUI gibt an die Spielkoordination den passenden Befehl weiter, dass ein Roboter entfernt werden soll.
Spiel starten	Die GUI gibt den Befehl zum Spielstart an die Spielkoordination weiter.
Streckensperrungen senden	Die GUI sendet zu sperrende Strecken an die Spielkoordination.

Schnittstelle $\langle I130 \rangle$: <Befehlsweiterleitung>

Operation	Beschreibung
Befehle weiterleiten	Befehle von Roboter bzw. Spieler, die durch das Kommunikationsmodul empfangen werden, werden an die Spielkoordination weitergeleitet$\langle I110 \rangle$.

3.3 Protokolle für die Benutzung der Komponenten

Grundsätzlich sind viele Softwarekomponenten dieses Projekts entweder spezifisch auf die entsprechende Anwendungssituation bezogen (z.B. Spielkoordination) oder so gestaltet, dass eine Adaptation auf einen neuen Anwendungsfall einer Neuentwicklung der Komponenten gleichkäme (z.B. GUI). Zwei Komponenten fallen hierbei jedoch heraus und lassen sich relativ einfach aus dem vorliegenden Projekt isolieren und könnten dann anderweitig verwendet werden.
Zunächst ist dies mit der Auktionsverwaltung möglich, die das Erschaffen von Auktionen, das Überwachen von Geboten und das Feststellen das „Auktionssiegers“ zum Ziel hat.

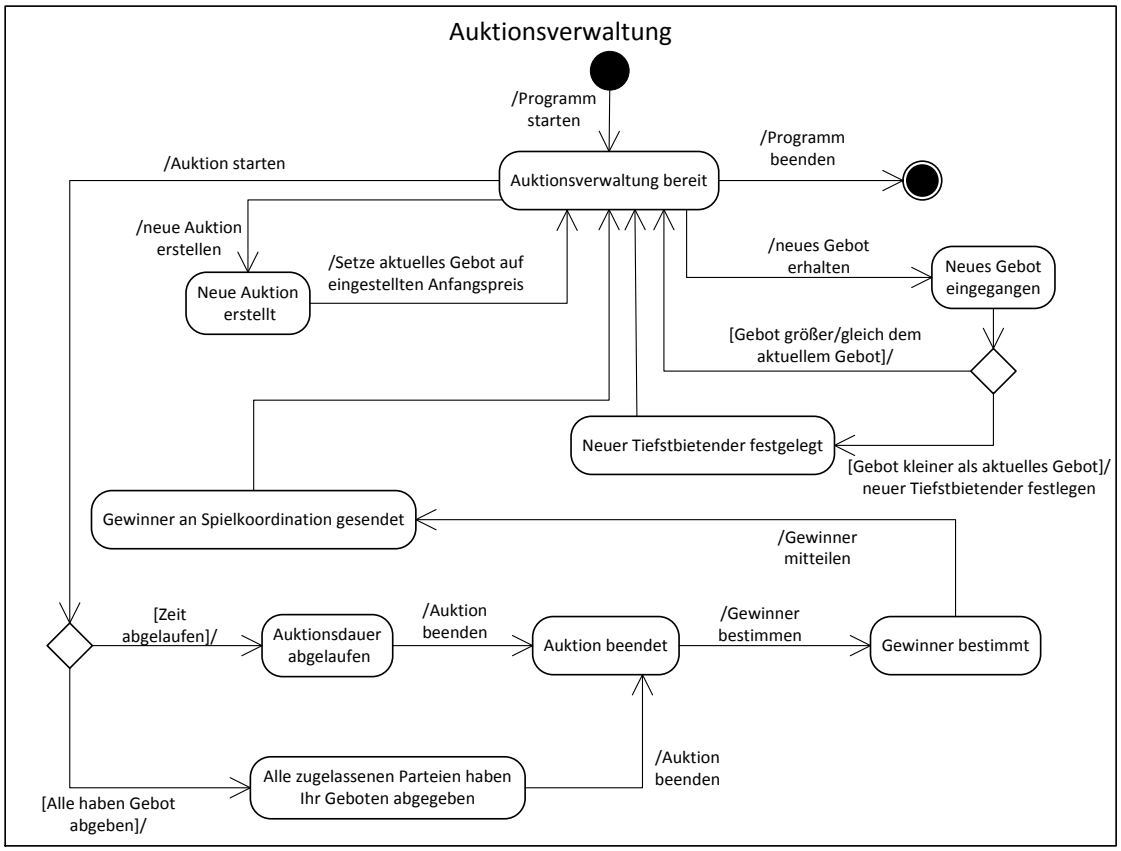

Abbildung 3.2: Auktionsverwaltung

Diese Komponente könnte zur generellen Auktionserstellung für z.B. Programme die ein Angebot-Nachfrage-System implementieren wollen (z.B Wirtschaftssimulation).

Die zweite Komponente ist das grundlegende Kommunikationsinterface, das in folgender (oder ähnlicher) Form in den drei Komponenten Kommunikation (Server), (Roboter) und (Spieler) vorkommen muss.

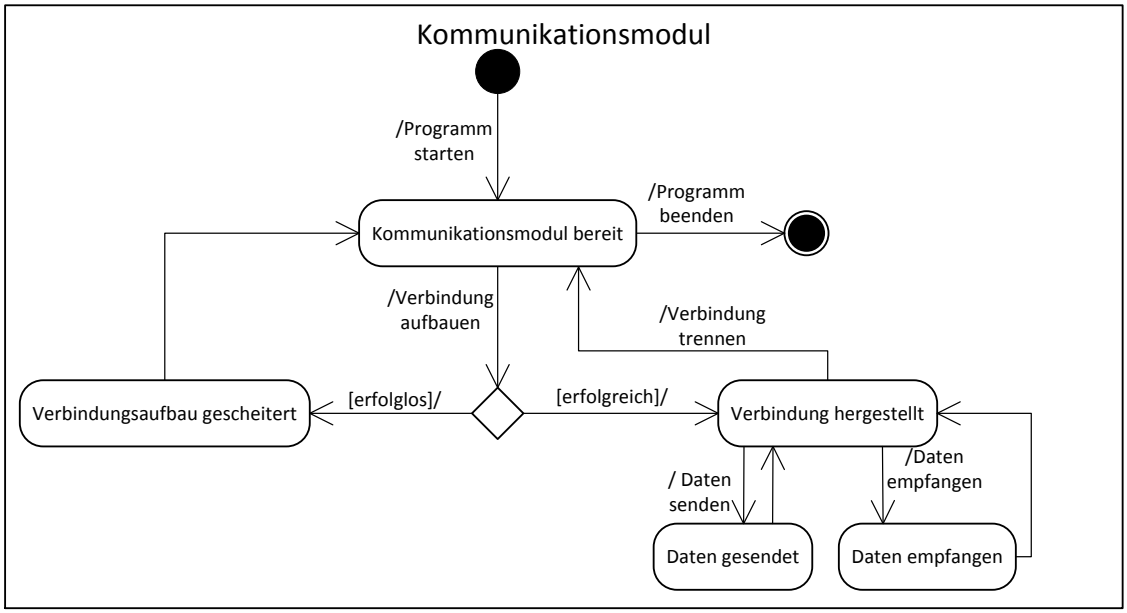

Abbildung 3.3: Kommunikationsmodul

Mithilfe dieser Komponente kann die Kommunikation zwischen verschiedenen Systemen (NXT-Roboter ⇔ Server) gewährleistet werden.

Eine weitere Komponente ist die Spielkoordination $\langle C80 \rangle$, dargestellt durch folgendes Statechartdiagramm (Abbildung 3.4):

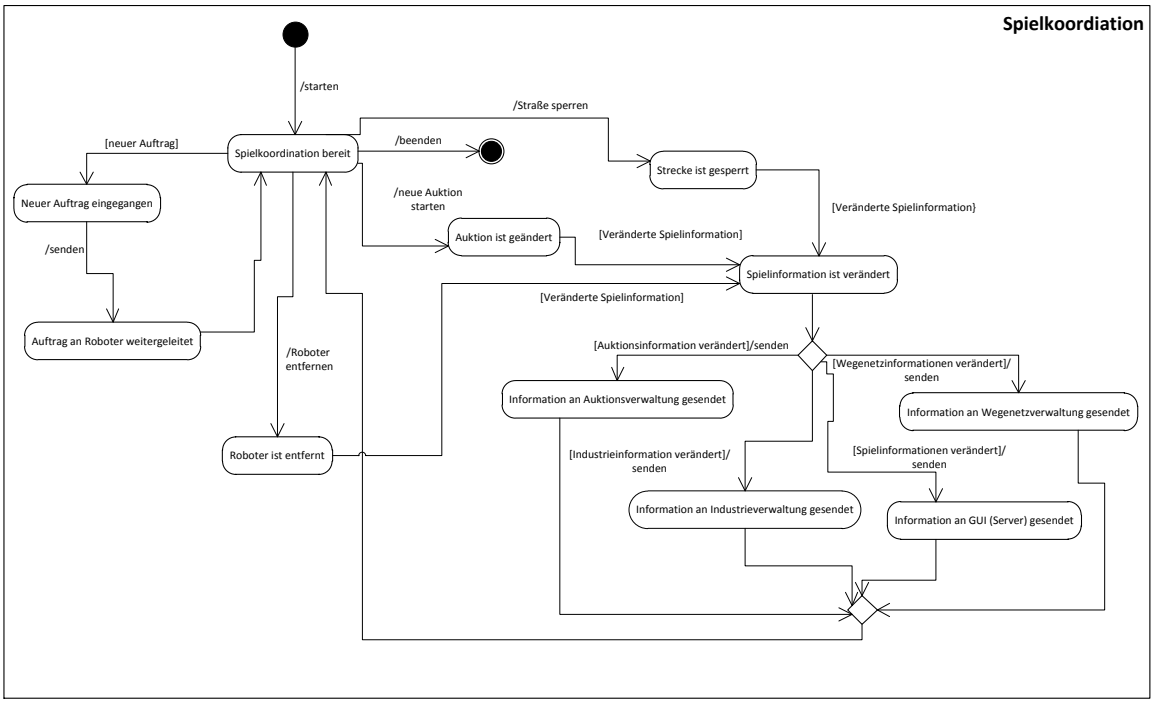

Abbildung 3.4: Spielkoordination

Sobald ein neuer Auftrag existiert, wird er durch die Spielkoordination an den entsprechenden Roboter übermittelt. Eine Änderung der Spielinformation kann durch drei Aktionen ausgelöst werden:

- Ein Roboter wird entfernt
- Eine Straße wird gesperrt
- Eine Auktion findet statt

Die Resultate dieser Aktionen müssen zur GUI des Servers übermittelt werden. Zusätzlich wird jede Information zur jeweilig zuständigen Verwaltungskomponente übermittelt. Neue Aufträge werden von der Spielkoordination an die betreffenden Roboter weitergeleitet. Sobald ein Roboter entfernt oder eine Strecke gesperrt wurde, werden die Verwaltungskomponenten der Industrie, des Wegenetzes, der Auktion sowie die GUI des Servers aktualisiert.

Eine weitere Komponente ist die Industrieverwaltung $\langle C100 \rangle$, dargestellt durch folgendes Statechartdiagramm (Abbildung 3.5):

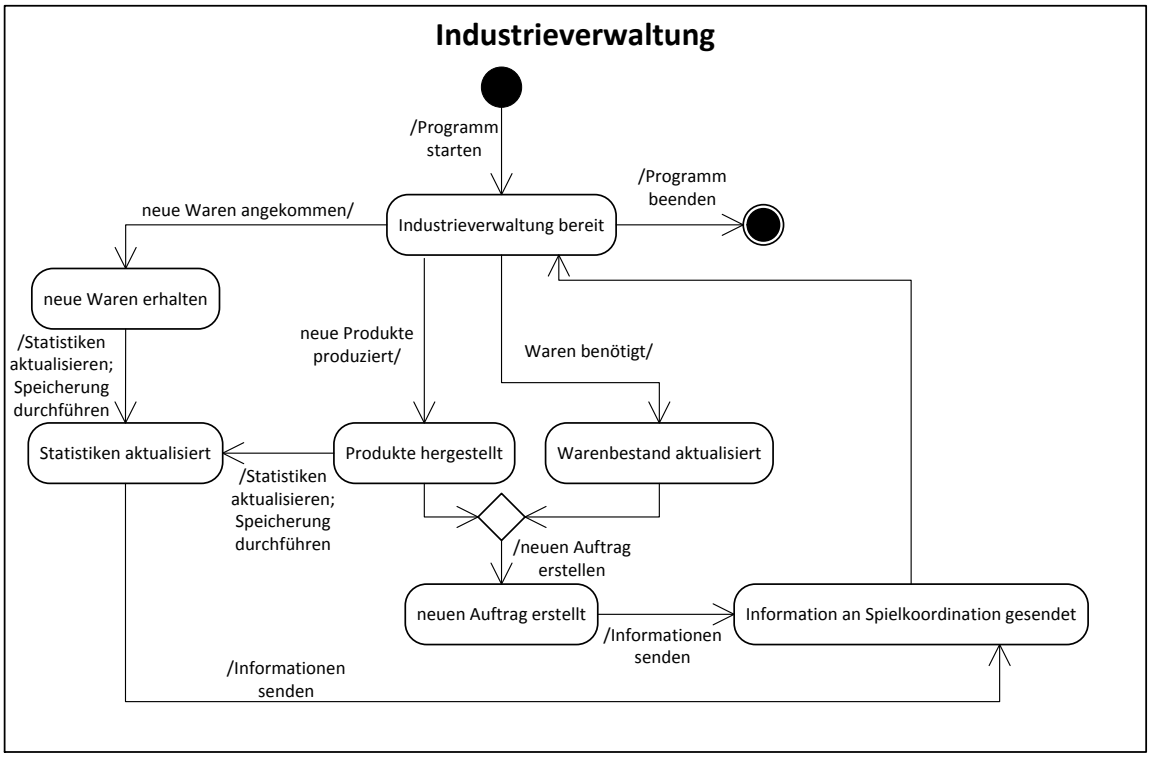

Abbildung 3.5: Industrieverwaltung

Diese verwaltet die Industrien und benötigt Waren zu Erzeugung von Produkten. Falls Produkte hergestellt wurden oder Waren benötigt werden erstellt sie neue Aufträge, welche sie an die Spielkoordination weitergibt. Zusätzlich aktualisiert sie die Statistiken, wenn neue Waren eintreffen oder Produkte hergestellt wurden.

Die im folgenden Zustandsdiagramm (Abbildung 3.6) darstellte Komponente ist die GUI(Spieler) $\langle C20 \rangle$:

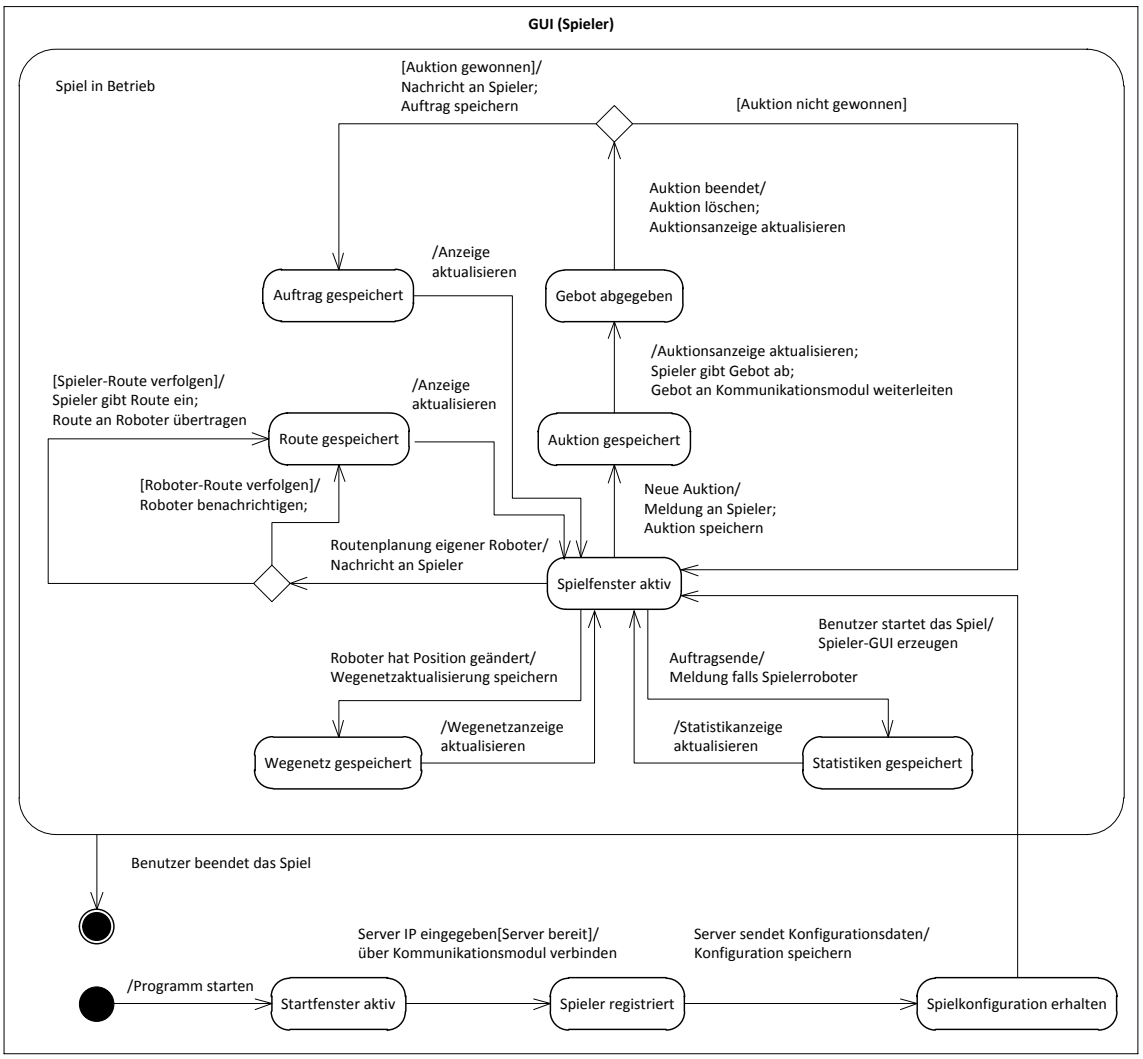

Abbildung 3.6: Spieleroberfläche

Es wird dargestellt, wie der Spieler auf bestimmte Aktionen anderer Komponenten im Spiels reagieren kann. Neue Auktionen verlangen das Handeln des Spielers, indem er Gebote abgibt. Im Fall einer Routenberechnung hat er aber auch die Möglichkeit dem Roboter die Routenwahl zu überlassen. Die Spieler-GUI liefert eine ähnliche Darstellung wie die Benutzer-GUI. Jedoch besitzt der Spieler nur einen eingeschränkten Zugriff auf Informationen beispielsweise andere Roboter betreffend. Die GUI(Spieler) beinhaltet zudem noch einen verwaltenden Teil, der sich um das Erkennen und Verarbeiten der empfangenen und zu sendenden Objekte kümmert.

Das nachfolgende Zustandsdiagramm (Abbildung 3.7) stellte die Komponente GUI(Benutzer) $\langle C50 \rangle$ dar:

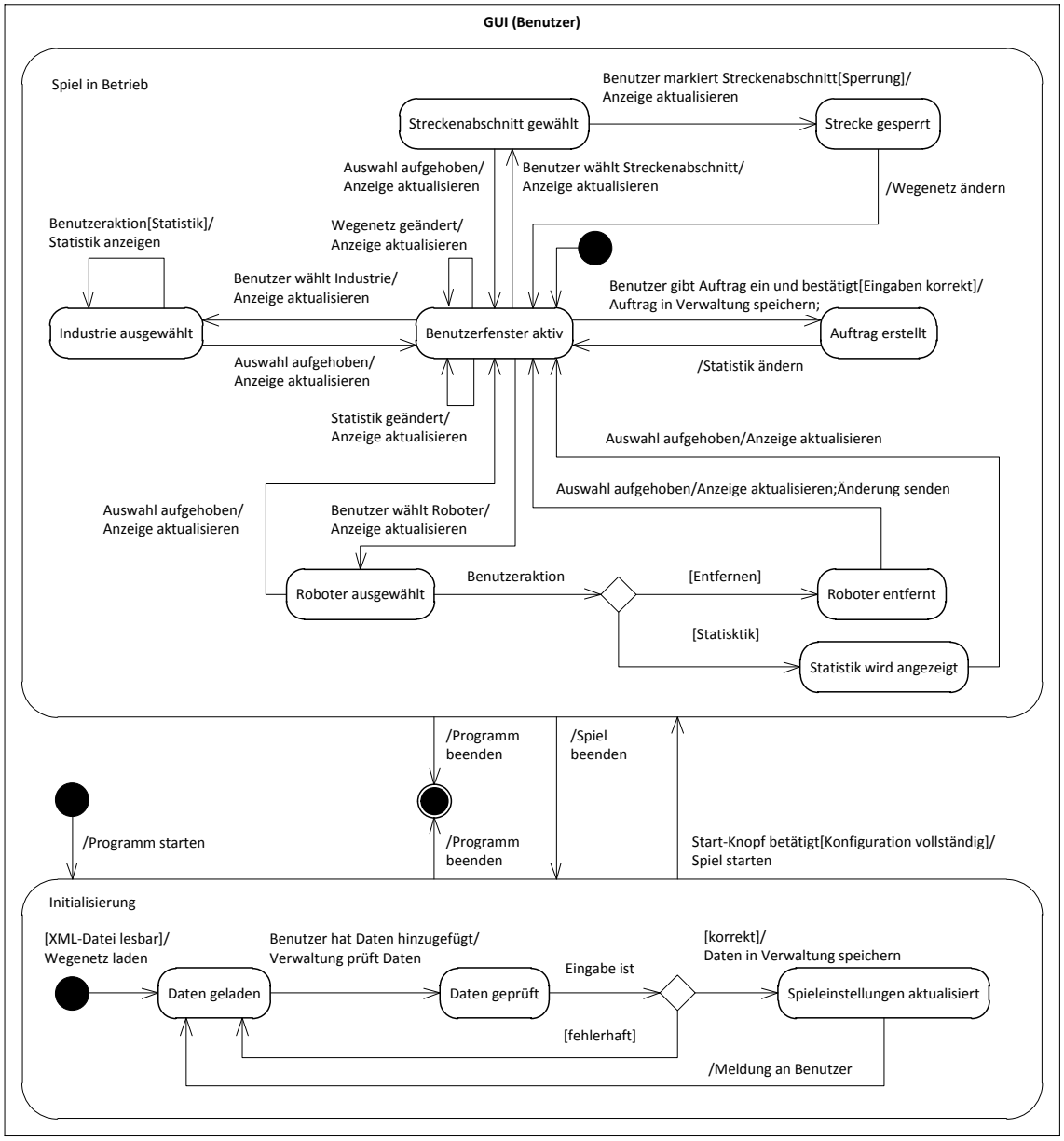

Abbildung 3.7: Benutzeroberfläche

Der Benutzer kann während des Spiels jederzeit Aktionen durchführen, die anderen Komponenten als Ereignis dienen, in einen anderen Zustand zu wechseln. So kann er z. B. Aufträge erstellen oder Strecken sperren, wodurch zum einen die Auktionsverwaltung und zum anderen die Wegenetzverwaltung aktiv werden.

Eine weitere Komponente ist die Wegenetzverwaltung $\langle C90 \rangle$, dargestellt durch folgendes Statechartdiagramm (Abbildung 3.8):

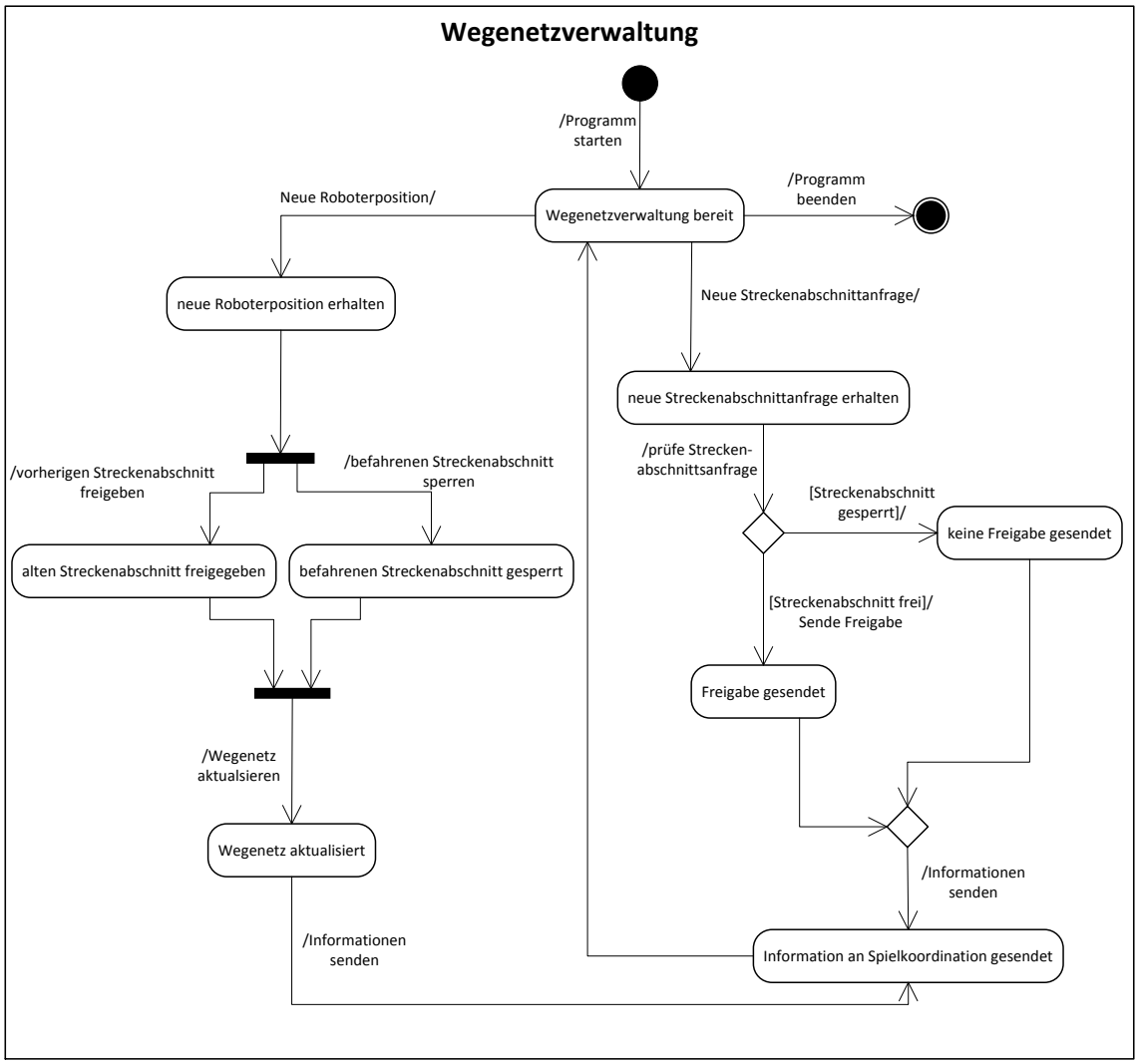

Abbildung 3.8: Wegenetzverwaltung

Diese verwaltet das Wegenetz und gibt Freigaben zum befahren einer Strecke an die Roboter. Sie erhält die neuen Positionen der Roboter und aktualisiert dann das Wegenetz und gibt die aktuellen Informationen an die Spielkoordination weiter. Außerdem erhält sie Anfragen von den Robotern, ob ein Streckenabschnitt befahrbar ist. Falls dieser frei ist wird eine Freigabe an den Roboter über die Spielkoordination gesendet. Falls der Streckenabschnitt gesperrt ist wird keine Freigabe gesendet.

Die primäre Komponente des Roboters ist die KI $\langle C10 \rangle$, dargestellt durch folgendes Statechartdiagramm (Abbildung 3.9):

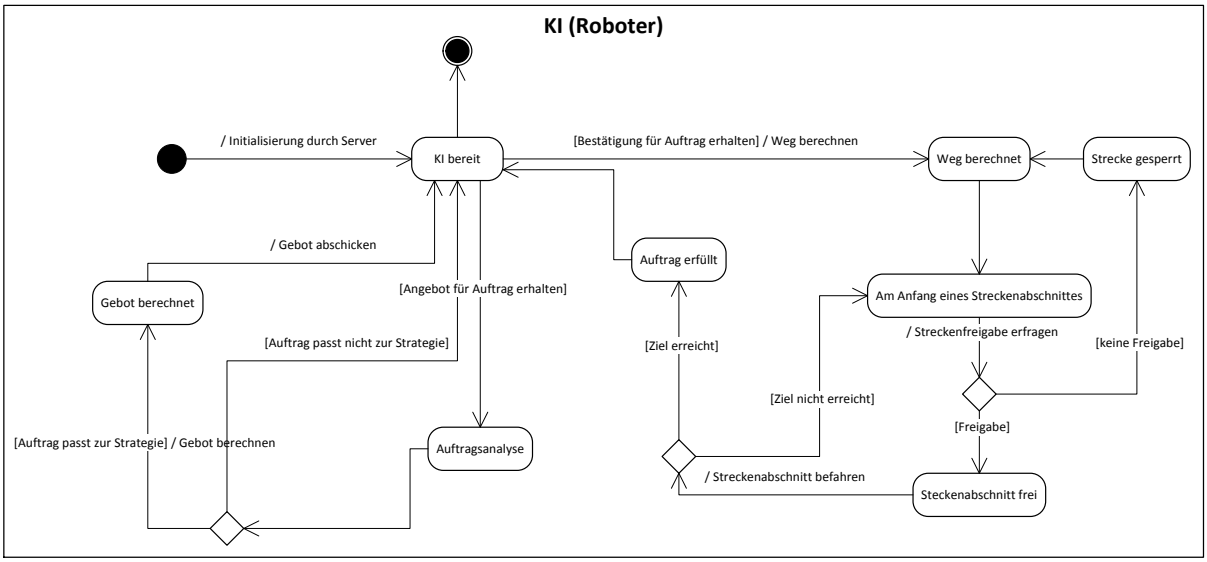

Abbildung 3.9: KI (Roboter)

Sie betrachtet Angebote zu Aufträgen und entscheidet gemäß ihrer Strategie, ob sie auf diesen Auftrag bieten möchte. Wenn sie die Bestätigung für einen Auftrag erhält, berechnet sie die zu fahrende Strecke, um ihren Auftrag abzuarbeiten. Für jede neue Strecke erfragt sie eine Freigabe, damit es nicht zur Kollision mit anderen Robotern kommt. Sie hat einen Auftrag erfolgreich abgeschlossen, wenn sie am Ziel angekommen ist und die transportierten Waren abliefern kann.

Die letzte Komponente ist die Steuerng des Roboters $\langle C120 \rangle$, dargestellt durch folgendes Statechart-diagramm (Abbildung 3.10):

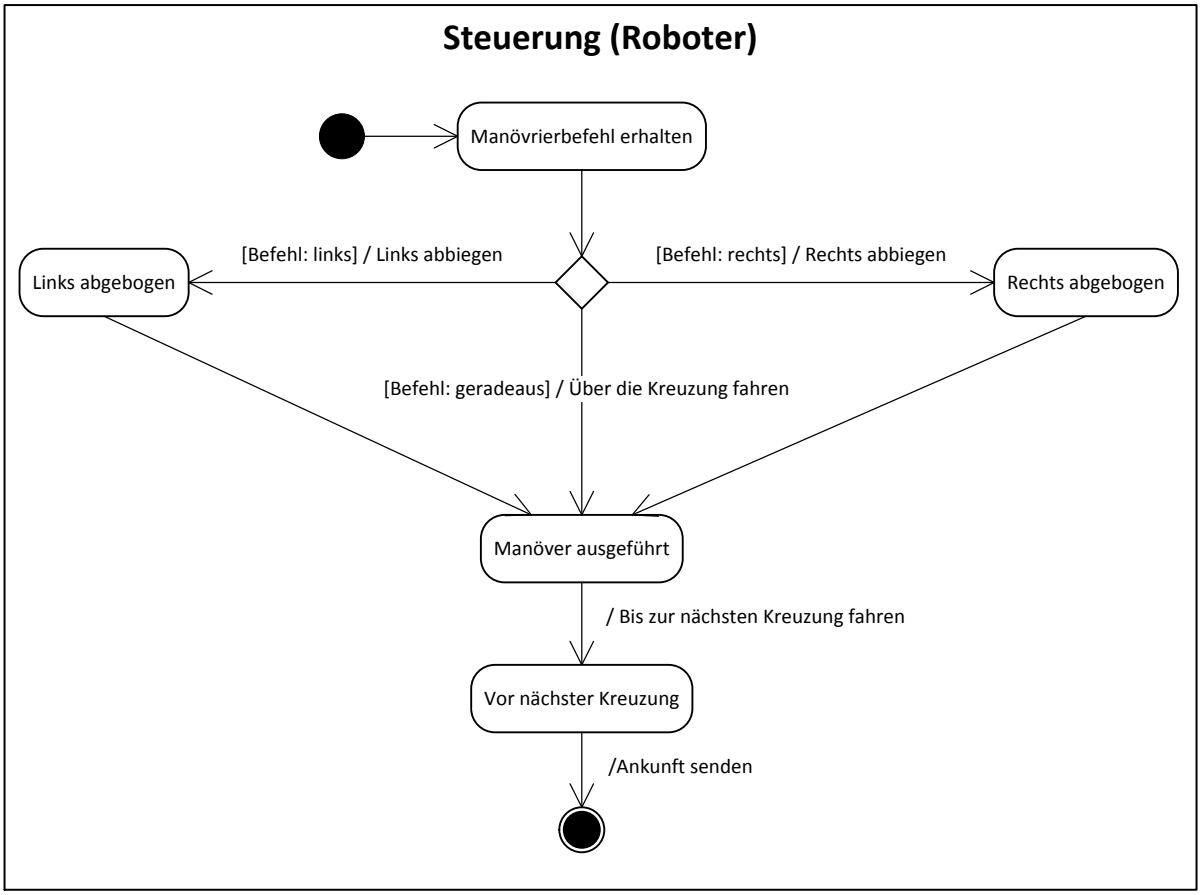

Abbildung 3.10: Steuerung des Roboters

Diese dient lediglich zur Manövrierung des Roboters auf dem Wegenetz. Wenn die Steuerung vor einer neuen Kreuzung mit einem Manövrierbefehl aufgerufen wird, prüft sie zuerst die Richtung, die sie einschlagen muss und biegt ab bzw. fährt über die Kreuzung geradeaus. Danach fährt sie über den nächsten Abschnitt bis vor die nächste Kreuzung und übermittelt die Ankunft an die KI.

4 Verteilungsentwurf

In der Abbildung 4 sieht man, dass das System in 3 Teile eingeteilt ist. Die Verbindung zwischen Server und NXT-Roboter geschieht über Bluetooth, hierbei werden Statuswerte und Kommandos übertragen. Die Kommunikation zwischen Spieler PC und Server PC, erfolgt über TCP/IP, dabei werden hauptsächlich Statistiken und Einstellungen übertragen.

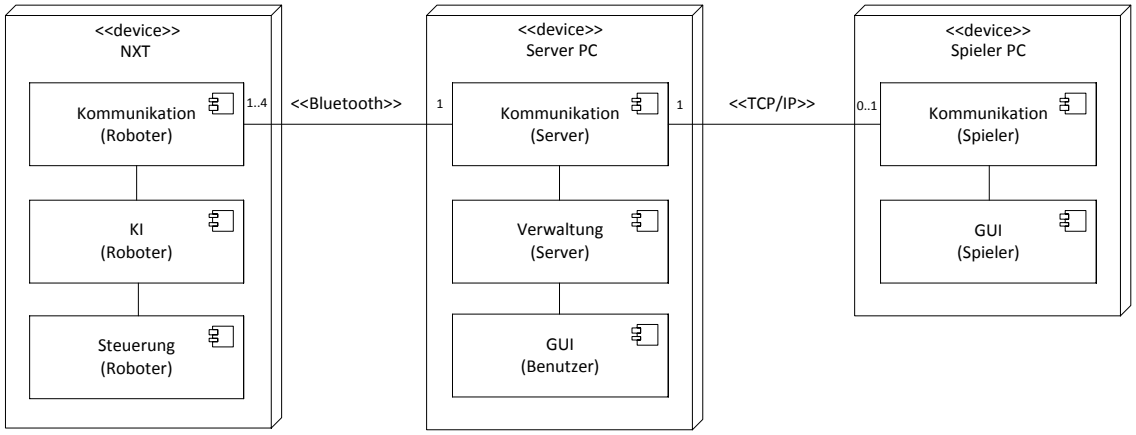

Abbildung 4.1: Verteilungsdiagramm

5 Implementierungsentwurf

In diesem Kapitel wird die Implementierung der in Kapitel 3 vorgestellten Komponenten vorgestellt. Insbesondere rücken dabei die verwendeten Klassen und Bibliotheken in den Mittelpunkt.

Das zugehörige Komponentendiagramm befindet sich in Abschnitt D des Anhanges.

5.1 Implementierung von Komponente $\langle C10 \rangle$: KI(Roboter)

Die KI des Roboters ist in der Lage eine Route zu berechnen. Des Weiteren ist sie in der Lage, Deadlocks zu verhindern. Es können somit keine Situationen entstehen, bei denen das Spiel stehen bleibt.

5.1.1 Paket-/Klassendiagramm

Für die Komponente $\langle C10 \rangle$ ergeben sich nachfolgende Diagramme.

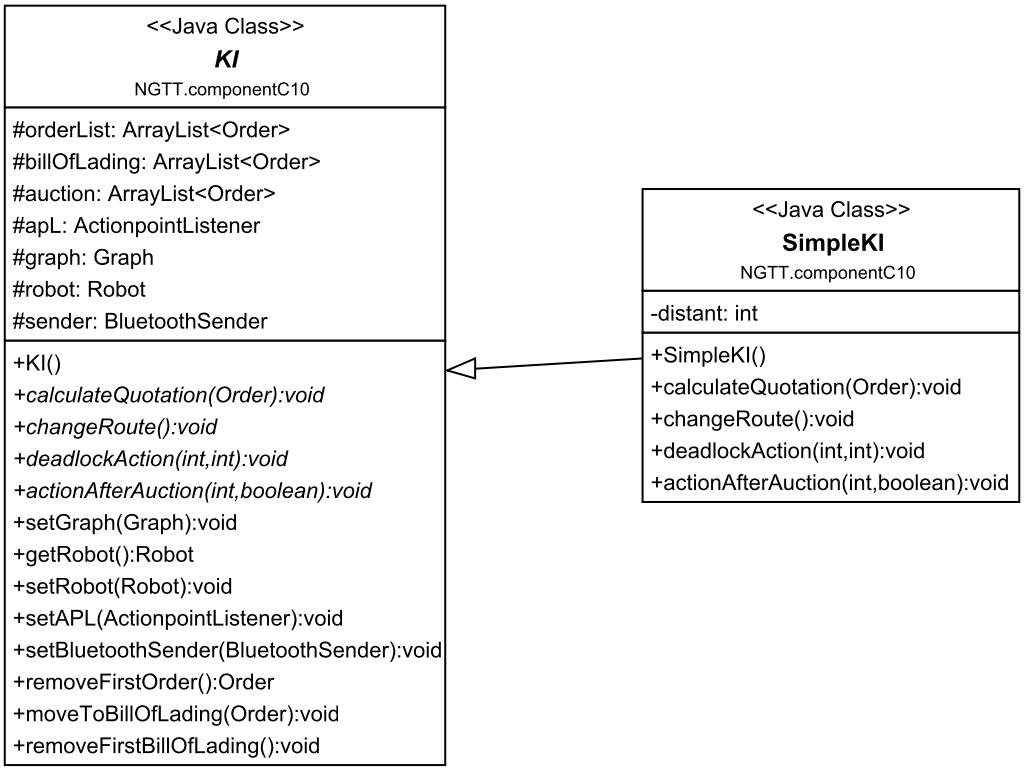

Abbildung 5.1: Klassendiagramm für Komponente $\langle C10 \rangle$

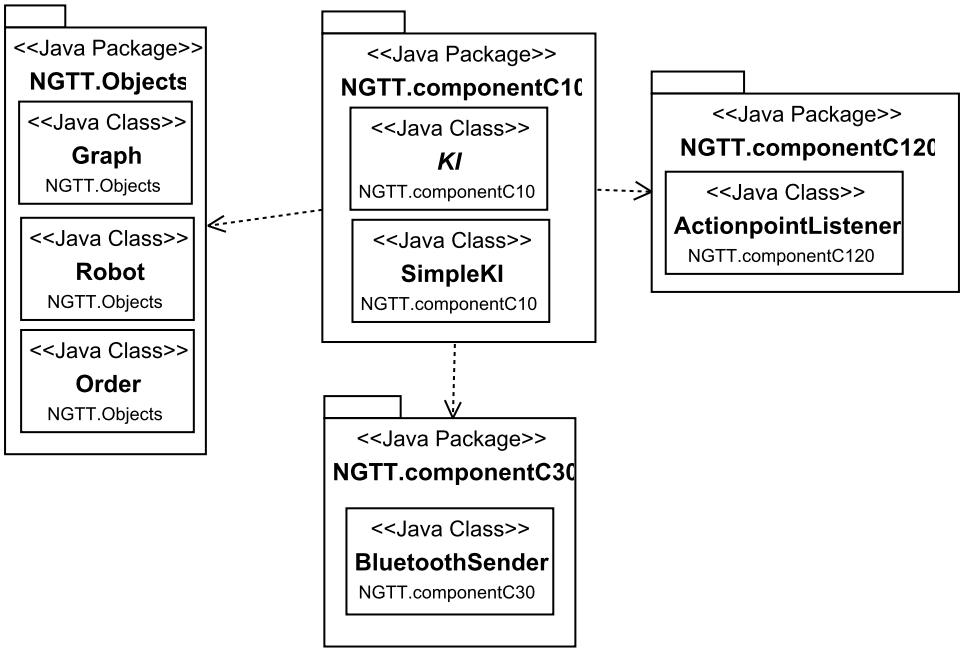

Abbildung 5.2: Paketdiagramm für Komponente $\langle C10 \rangle$

5.1.2 Erläuterung

Die Komponente KI ist wie folgt aufgebaut:

KI$\langle CL10 \rangle$

Aufgabe

Die abstrakte Klasse KI beschreibt allgemein die KI des Roboters, welche in der Klasse SimpleKI genauer ausgeprägt wird. Die Methoden removeOrder sowie moveToBillOfLading sind keine abstrakte Methoden, da sie von der Implementierung einer KI unabhängig und somit allgemeingültig sind.

Attribute

Attributname	Datentyp	Beschreibung
orderList	ArrayList<Order>	Speichert angenommende Aufträge ab.
billOfLading	ArrayList<Order>	Speichert die ID der Aufträge, die schon angenommen wurden und nur noch zu ihrem Ziel gebracht werden müssen.
auction	ArrayList<Order>	Speichert die Aufträge der Auktionen, diese werden dann später in die OrderList eingefügt, wenn sie erhalten wurden. Wenn nicht, werden sie gelöscht.
apL	ActionpointListener	ActionpointListener.
graph	Graph	Der Graph.
robot	Robot	Die Werte des Roboters.
sender	BluetoothSender	Der Stream, der nötig ist, um Daten zu senden.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
KI	-	KI	Konstruktor zum Erstellen einer Instanz.

calculateQuotation	Order	void	Berechnung eines Angebotes und das Abspeichern, falls der Auftrag erhalten wird, sowie das Senden an den Server.
changeRoute	-	void	Legt fest, wie der Roboter fahren wird.
deadlockAction	-	void	Legt das Verhalten bei Deadlocks fest.
actionAfterAuction	Integer, Boolean	void	Fügt den gewonnenen Auftrag in die Auftragsliste ein und ändert je nach Strategie die Route.
removeOrder	-	Order	Entfernt den ersten Auftrag aus der Auftragsliste, wenn die Liste mindestens ein Objekt enthält.
moveToBillOfLanding	Order	void	Fügt den als Parameter übergebenen Auftrag zum Lieferschein hinzu.

Kommunikationspartner

Komponente $\langle C120 \rangle$, Komponente $\langle C30 \rangle$

SimpleKI$\langle CL20 \rangle$

Aufgabe

Die Klasse SimpleKI ist eine genaue Ausprägung der Klasse KI und definiert somit konkret das Verhalten des Roboters.

Attribute

Attributname	Datentyp	Beschreibung
distant	Integer	Gibt die Entfernung des Zieles des aktuellen Auftrags zum Zielpunkt des letzten Auftrages in der Auftragsliste an.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
SimpleKI	-		Konstruktor zum Erstellen einer Instanz.

calculateQuotation	Order	void	Die Methode nimmt einen Auftrag als Parameter entgegen, berechnet ein Angebot und speichert einen Auftrag bei Annahme ab. Das Angebot wird ebenso an den Server gesendet.
changeRoute	-	void	Wenn der Roboter keine Ladung beherbergt und keine Arbeit hat, so wird eine derzeitige Startposition gewählt und zu dem Anfang und anschliessend dem Ende eines Auftrages eine Route berechnet.
deadlockAction	Integer	void	Bei der Behandlung von Deadlocks werden an den übergebenen Knoten geprüft, ob es noch an beiden freie Kanten gibt, die befahren werden und somit als Umleitung dienen.
actionAfterAuction	Integer, Boolean	void	Speichert den Auftrag, der in der Auktionsliste war, in die Auftragsliste oder löscht ihn, wenn er ihn nicht bekommen hat.

Kommunikationspartner

Komponente $\langle C120 \rangle$, Komponente $\langle C30 \rangle$

5.2 Implementierung von Komponente$\langle C20 \rangle$: GUI(Spieler)

Die GUI visualisiert für den Spieler den Ablauf des Spiels und ermöglicht Eingaben wie z.B Gebote und Routenänderungen.

5.2.1 Paket-/Klassendiagramm

Für die Komponente $\langle C20 \rangle$ ergeben sich nachfolgende Diagramme.

Das Klassendiagramm befindet sich im Anhang im Abschnitt E, es sollte in DIN A3 ausgedruckt werden.

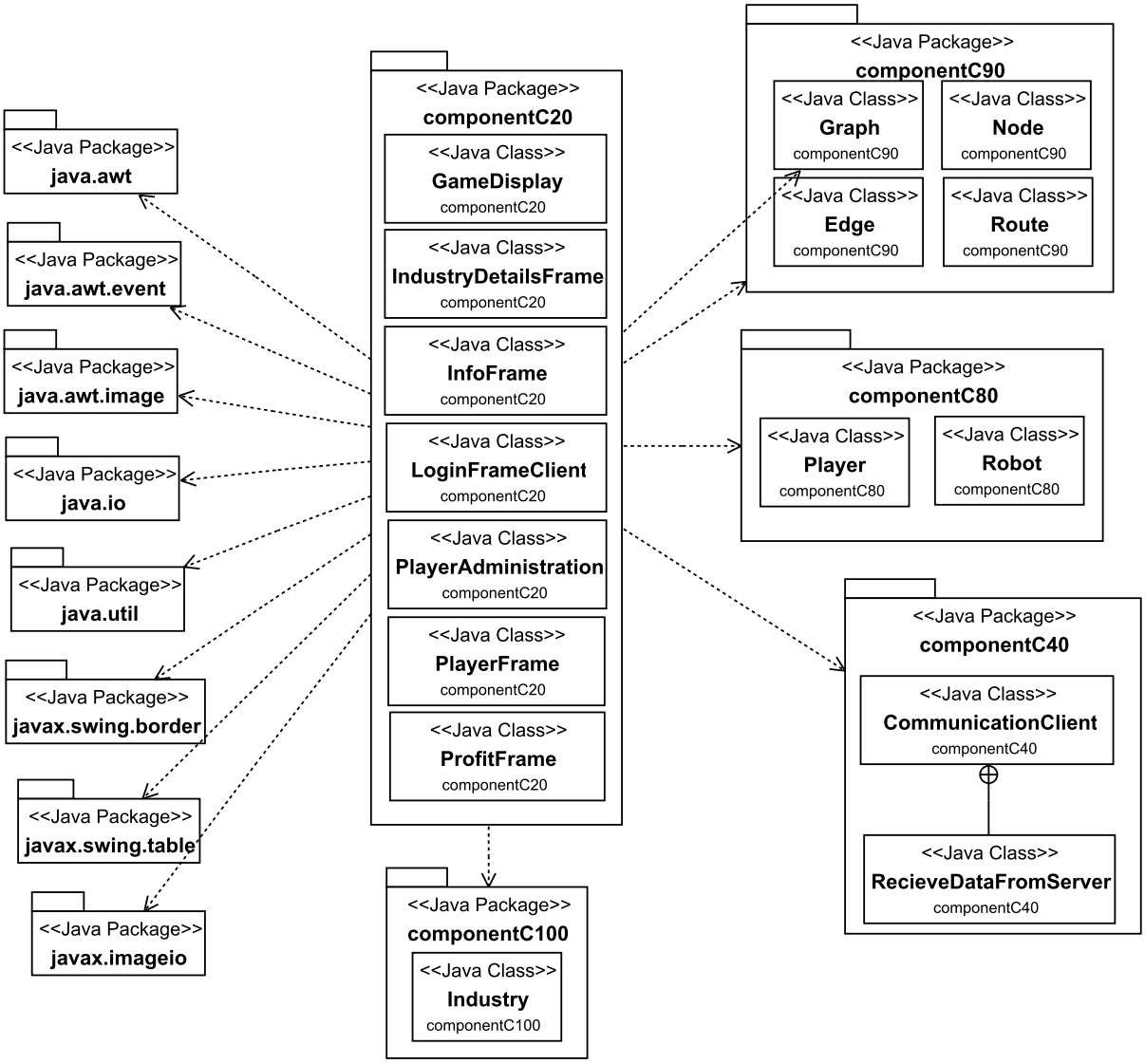

Abbildung 5.3: Paketdiagramm für Komponente $\langle C20 \rangle$

5.2.2 Erläuterung

Die Klassen der Komponente gestalten sich wie folgt:
InfoFrame⟨$CL30$⟩

Aufgabe

Info-Fenster zur Ausgabe von Informationen über das Projekt.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Dient zum Setzen des Layouts und zur Aufnahme der restlichen Komponenten.
iconPanel	JPanel	Panel zur Aufnahme von Icons.
versionNamePanel	JPanel	Panel zur Darstellung der Versionsnummer.
icon	JLabel	Label für das Icon.
projectName	JLabel	Label für den Projektnamen.
version	JLabel	Label für die Version.
dennisStelter	JLabel	Label für Namen.
henrikLange	JLabel	Label für Namen.
markusBjoernMeissner	JLabel	Label für Namen.
patriciaTatjanaKasulke	JLabel	Label für Namen.
jochenSteiner	JLabel	Label für Namen.
tessaFabian	JLabel	Label für Namen.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
main	args	void	Start der Anwendung.
start	JFrame	void	Öffnen des Fensters.
infoFrame	-		Konstruktor zum Erzeugen einer Instanz.

Kommunikationspartner

Komponente ⟨$C40$⟩

playerFrame⟨$CL40$⟩

Aufgabe

Frame der Spieler-GUI.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Setzen des Layouts und zur Aufnahme der restlichen Komponenten.
gameDataRouteLabelPanel	JPanel	Panel für das Label des Wegenetzes.
gameDataRoutePanel	JPanel	Panel für das Wegenetz.
orderPanel	JPanel	Panel für die Aufträge.
bidPanel	JPanel	Panel für die Gebote.
givenRoute	JTextField	Textfeld für übertragene Route.
ownRoute	JTextField	Textfeld für die Eingabe einer eigenen Route.
profit	JTextField	Textfeld für den Gewinn.
averageProfit	JTextField	Textfeld für den durchschnittlichen Gewinn.
cargo	JTextField	Textfeld für Ware.
roboColor	JTextField	Textfeld für die Farbe eines Roboters.
bid	JTextField	Textfeld für ein Gebot.
orderChange	JComboBox	Combobox zur Abänderung eines Auftrages.
routeLabel	JLabel	Label für die Route.
gameDataLabel	JLabel	Label für Spieldaten.
givenRouteLabel	JLabel	Label für die gegebene Route.
ownRouteLabel	JLabel	Label für die eigene, gewählte Route.
profitLabel	JLabel	Label für den Gewinn.
averageProfitLabel	JLabel	Label für den Durchschnittsgewinn.
cargoLabel	JLabel	Label für die Ware.
roboColorLabel	JLabel	Label für die Farbe des Roboters.
changeRouteButton	JButton	Button zum Ändern einer Route.
giveBid	JButton	Button zum Abgeben eines Gebots.
industryDetailButton	JButton	Button zum Starten der Industrieübersicht.
orderTable	JTable	Tabelle für Aufträge.
menuBar	JMenuBar	MenuBar für das Menü.
menuName	JMenu	Name des Menüs.
menuHelp	JMenuItem	Menüitem für das Hilfe-Fenster.
menuCloseGame	JMenuItem	Menüpunkt zum Beenden des Programms.
menuInfo	JMenuItem	Menüpunkt für das Info-Fenster.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
main	String[]	void	Start der Anwendung.
start	JFrame	void	Öffnen des Fensters.
playerFrame	-		Konstruktor.
actionPerformed	ActionEvent	void	Ausführen von Actionevents.

Kommunikationspartner

Komponente ⟨$C40$⟩

IndustryDetailsFrame⟨$CL50$⟩

Aufgabe

Frame für Industrieinformationen.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Dient zum Setzen des Layouts und zur Aufnahme der restlichen Komponenten.
industryTable	JTable	Tabelle für die Industrie.
closeFrame	JButton	Button zum Schließen des Fensters.
industryTablePane	JScrollPanel	Scrollbalken für die Tabelle.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
main	String[]	void	Start der Anwendung.
start	JFrame	void	Öffnen des Fensters.
IndustryDetailsFrame	-		Konstruktor zum Erstellen einer Instanz.
actionPerformed	ActionEvent	void	Ausführen von Actionevents anhand des übergebenen Events.

Kommunikationspartner

Komponente ⟨$C40$⟩

ProfitFrame⟨$CL60$⟩

Aufgabe

Frame für Profittabelle.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Dient zum Setzen des Layouts und zur Aufnahme der restlichen Komponenten.
dataset	CategoryDataset	Datenmenge für das Diagramm.
series1	String	Name der Größe 1 des Diagramms.
series2	String	Name der Größe 2 des Diagramms.
avProfit	int	Durchschnittlicher Gewinn.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
ProfitFrame	-		Konstruktor zum Erstellen einer Instanz.
createDatabase	-	CategoryDataset	Datenmenge erstellen.
createChart	CategoryDataset	JFreeChart	Anhand der übergebenen Datenmenge wird ein Diagramm erstellt.
addNewProfit	Integer	void	Neuen Profit, der zu einem bestimmten Zeitpunkt erreicht wurde, hinzufügen.
actionPerformed	ActionEvent	void	ActionListener, der anhand des übergebenen Events die entsprechenden Aktionen auslöst.

Kommunikationspartner

Komponente $\langle C40 \rangle$

PlayerAdministration$\langle CL70 \rangle$

Aufgabe

Verwaltungsklasse für den Spieler.

Attribute

Attributname	Datentyp	Beschreibung
com	CommunicationClient	Kommunikationsobjekt für den Spieler.
ordersInAuction	ArrayList<Order>	Listen für Aufträge zum Errechnen von Statistiken.
ordersToDo	ArrayList<Order>	Listen für Aufträge, die abzuarbeiten sind.
ordersDone	ArrayList<Order>	Listen für abgearbeitete Aufträge.
playerList	ArrayList<Player>	Liste der teilnehmenden Spieler.
industryList	ArrayList<Industry>	Liste der Industrie.
playername	String	Name des Spielers.
playerId	short	ID des Spielers.
teamId	short	ID des Teams.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
main	String[]	void	main-Methode
PlayerAdministration	-		Konstruktor zum Erstellen einer Instanz.
distributeData	Object	void	Wertet die empfangenen Objekte aus und startet die entsprechenden Aktionen.

Kommunikationspartner

Komponente $\langle C40 \rangle$

LoginFrameClient$\langle CL80 \rangle$

Aufgabe

GUI für den Client zum Eingeben von Verbindungsdaten.

Attribute

Attributname	Datentyp	Beschreibung
count	Integer	Zählervariable.
serialVersionUID	Long	ID.
contentPane	JPanel	Dient zum Setzen des Layouts und zur Aufnahme der restlichen Komponenten.
textField	JTextField	Textfeld zur Eingabe.
textField_1	JTextField	Textfeld zur Eingabe.
btnNewButton	JButton	Button, mit dem eine Verbindung erstellt werden kann.
lblGamestart	JLabel	Label gibt aus, ob und wann ein Spiel schon gestartet ist.
panel_1	JPanel	Panel für Layout.
panel_2	JPanel	Panel für Layout.
panel_3	JPanel	Panel für Layout.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
LoginFrameClient	-		Konstruktor zum Erstellen einer Instanz.
actionPerformed	ActionEvent	void	ActionListener-Methode, der anhand des übergebenen Events entsprechende Aktionen auslöst.

Kommunikationspartner

Komponente $\langle C40 \rangle$

5.3 Implementierung von Komponente $\langle C30 \rangle$: Kommunikation (Roboter)

Die Komponente $\langle C30 \rangle$ ist für die Kommunikation mit dem Roboter zuständig und gibt die erhaltenen Nachrichten an die Komponente $\langle C10 \rangle$ weiter.

5.3.1 Paket-/Klassendiagramm

Die Komponente $\langle C30 \rangle$ wird mit Hilfe folgender Klassen und Pakete implementiert.

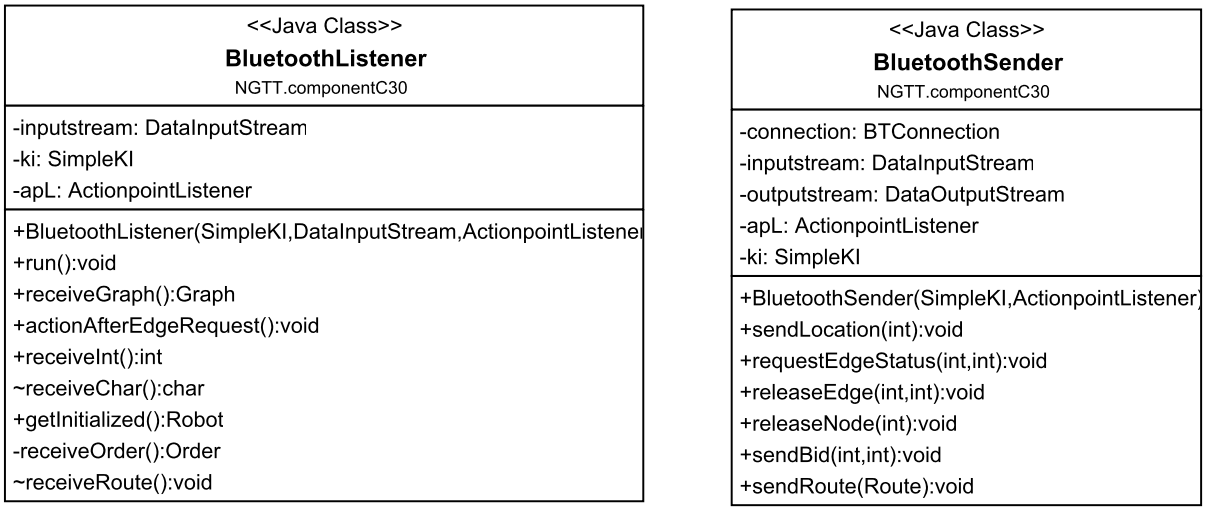

Abbildung 5.4: Klassendiagramm für Komponente $\langle C30 \rangle$

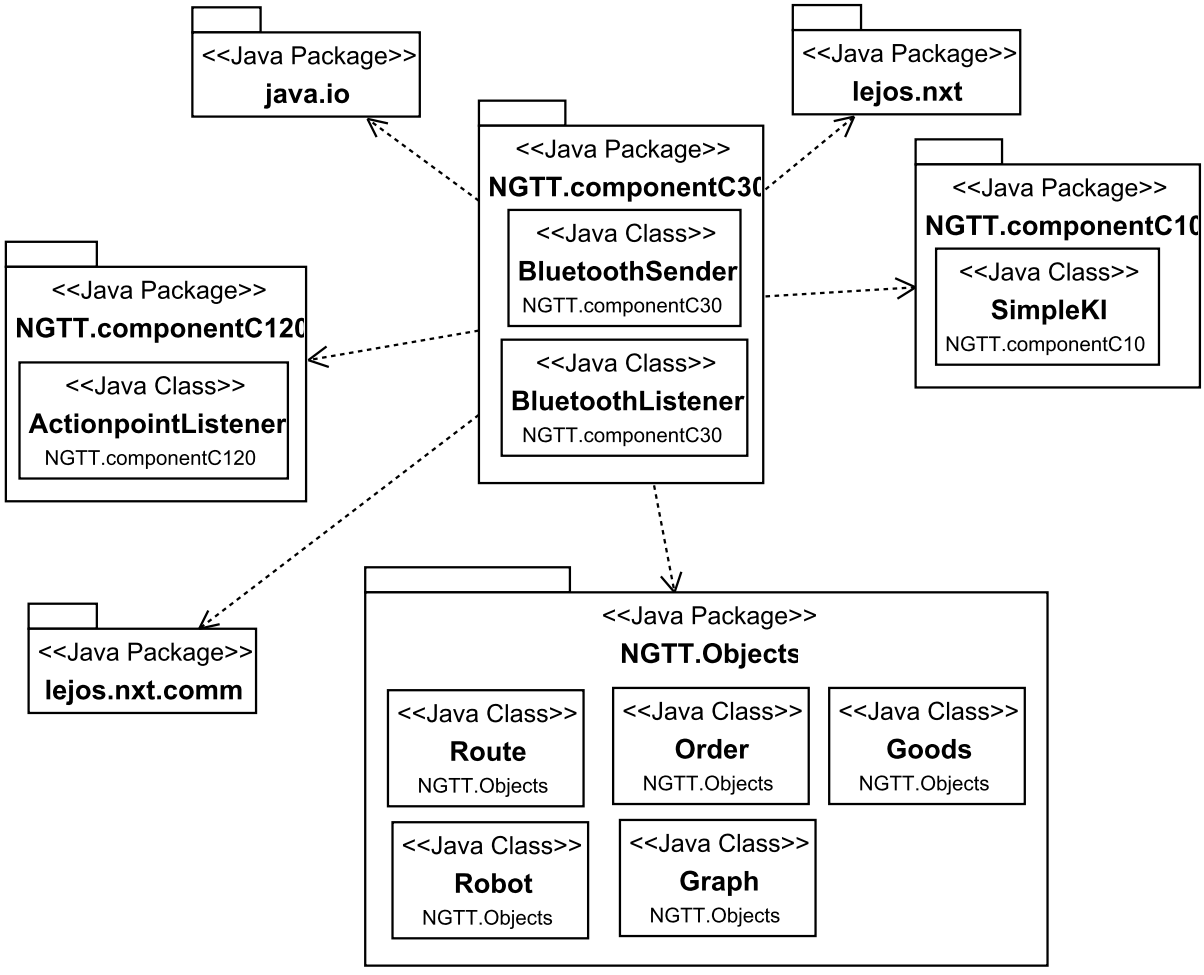

Abbildung 5.5: Paketdiagramm für Komponente $\langle C30 \rangle$

5.3.2 Erläuterung

Die Klassen der Komponente ergeben sich wie folgt:
BluetoothSender⟨$CL90$⟩

Aufgabe

Klasse sendet Daten.

Attribute

Attributname	Datentyp	Beschreibung
connection	BTConnection	Objekt zum Aufbau einer Bluetoothverbindung.
inputStream	DataInputStream	Stream zum Einlesen von Daten.
outputStrean	DataOutputStream	Stream zum Auslesen von Daten.
apL	ActionPointListener	ActionPointListener-Objekt.
ki	SimpleKI	Künstliche Intelligenz.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
BluetoothSender	SimpleKI, Actionpoint-Listener		Der Konstruktor erzeugt aus den Parametern eine Instanz dieser Klasse.
sendLocaction	Integer	void	Sendet die übergebene Position an den Server.
requestEdgeStatus	Integer	void	Erfragt den Status einer Kante. Diese Kante wird gesperrt, wenn sie frei ist.
releaseEdge	Integer	void	Identifiziert eine Kante anhand der übergebenen Knoten und gibt sie wieder frei.
releaseNode	Integer	void	Gibt den übergebenen Knoten wieder frei.
sendBid	Integer	void	Sendet ein Gebot an den Server.
sendRoute	Route	void	Sendet das übergebene Route-Objekt an den Server.

Kommunikationspartner

Komponente ⟨$C40$⟩

BluetoothListener⟨$CL100$⟩

Aufgabe

Diese Klasse nimmt Daten entgegen.

Attribute

Attributname	Datentyp	Beschreibung
inputStream	DataInputStream	Stream zu Einlesen von Daten.
ki	SimpleKI	Künstliche Intelligenz.
apL	ActionPointListener	ActionPointListener-Objekt.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
BluetoothListener	SimpleKI, DataInputStream, Actionpoint-Listener		Konstruktor erstellt aus den Parametern eine Instanz der Klasse.
run	-	void	Startet das Einlesen von Daten von der Schnittstelle.
receiveGraph	-	Graph	Erstellt ein Graph-Objekt aus den eingelesenen Graph-Daten.
actionAfterEdgeRequest	-	void	Nach dem Abfragen einer Kante werden entsprechend des Ergebnisses die entsprechenden Parameter gesetzt.
receiveInt	-	Integer	Empfängt eine ganze Zahl.
receiveChar	-	Character	Empfängt einen Buchtaben.
getInitialized	-	Robot	Erstellt ein Roboterobjekt und gibt dieses zurück.
receiveOrder	-	Order	Liest die benötigten Parameter für ein Order-Objekt aus dem Inputstream und erstellt ein solches Objekt.
receiveRoute	-	void	Empfängt eine Route.

Kommunikationspartner

Komponente $\langle C40 \rangle$

5.4 Implementierung von Komponente $\langle C40 \rangle$: Kommunikation (Spieler)

Die Komponente Kommunikation(Spieler) dient der Datenübertragung zwischen Spieler und Server.

5.4.1 Paket-/Klassendiagramm

Für die Komponente $\langle C40 \rangle$ ergeben sich nachfolgende Diagramme.

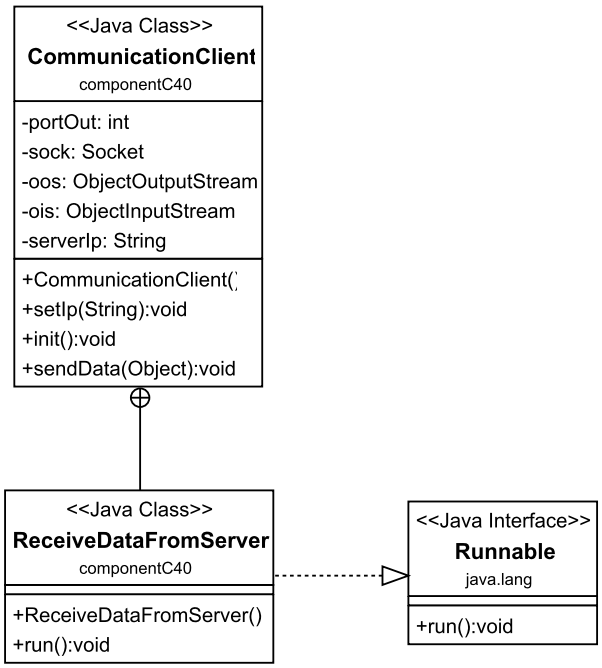

Abbildung 5.6: Klassendiagramm für Komponente $\langle C40 \rangle$

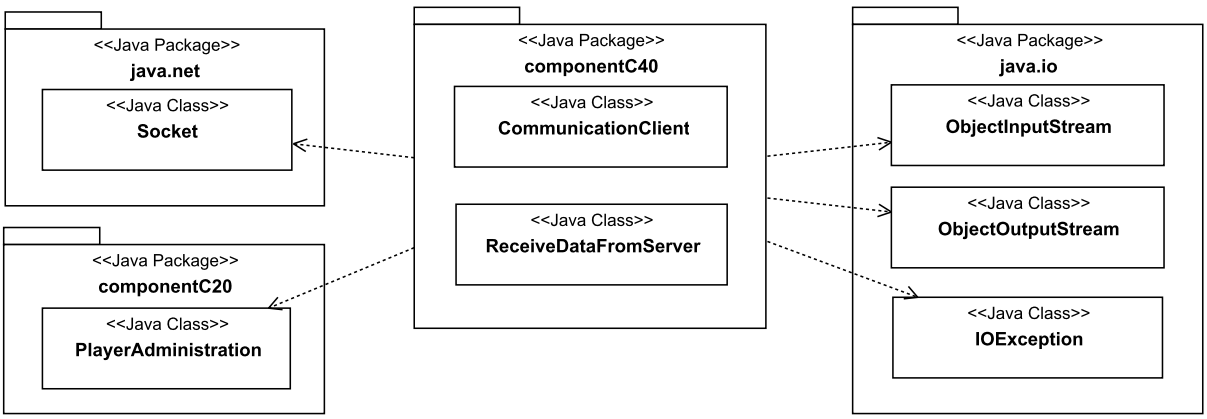

Abbildung 5.7: Paketdiagramm für Komponente $\langle C40 \rangle$

5.4.2 Erläuterung

CommunicationClient$\langle CL110 \rangle$

Aufgabe

Verbinden zu dem vom Server bereitgestellten Socket. Erstellt des Weiteren einen Inputstream und einen Outputstream für den Client. Der Spieler kann mit dem Server nur über diese Klasse kommunizieren.

Attribute

Attributname	Datentyp	Beschreibung
portOut	int	Port, mit dem der Server auf Verbindungen wartet.
sock	Socket	Socket, den der Client zum Verbinden nutzt.
oos	ObjectOutputStream	ObjectOutputStream der Verbindung.
ois	ObjectInputStream	ObjectInputStream der Verbindung.
serverIp	String	Die IP des Servers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
init	-	void	Verbindung zum Server herstellen und erstellen des Input- und Outputstreams.
sendData	Object	void	Senden eines Datenobjekts an den Server.

Kommunikationspartner

Komponente $\langle C20 \rangle$

ReceiveDataFromServer$\langle CL120 \rangle$

Aufgabe

Innere Klasse von CommunicationClient. Wird als Thread gestartet und wartet auf ankommende Daten des InputStreams.

Attribute

Keine eigenen Attribute vorhanden. Nutzt als Innere Klasse den Inputstream von ConnectionClient.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
run	-	void	Wenn gestartet, werden Daten des Inputstreams gelesen und an die GUI(Spieler) weitergereicht.

Kommunikationspartner

Komponente $\langle C60 \rangle$

5.5 Implementierung von Komponente $\langle C50 \rangle$: GUI (Server)

5.5.1 Paket-/Klassendiagramm

Für die Komponente $\langle C50 \rangle$ ergeben sich nachfolgende Diagramme.

Das Klassendiagramm befindet sich im Anhang im Abschnitt F, es sollte in DIN A3 ausgedruckt werden.

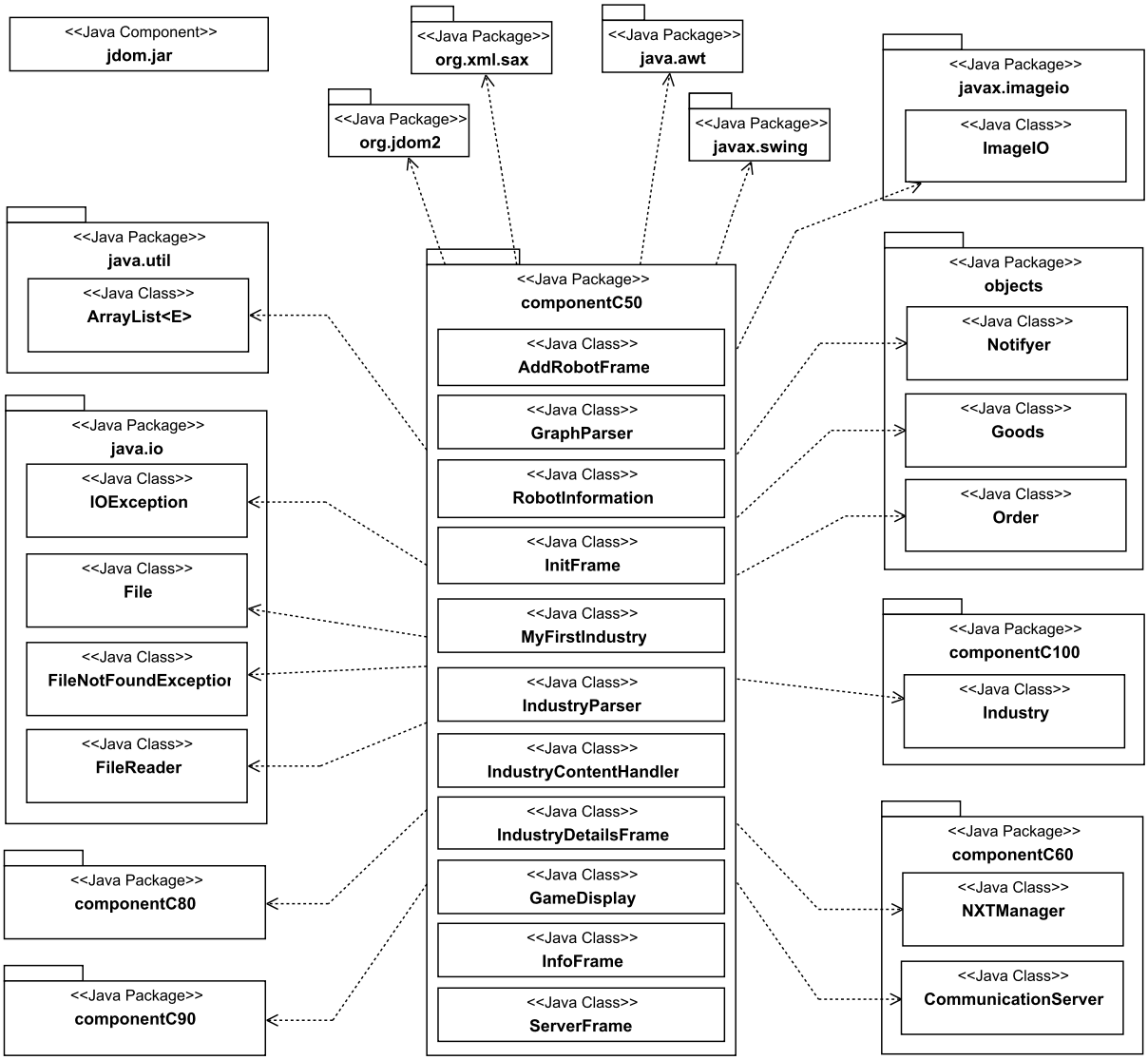

Abbildung 5.8: Paketdiagramm für Komponente $\langle C50 \rangle$

5.5.2 Erläuterung

InitFrame$\langle CL130 \rangle$

Aufgabe

Der InitFrame dient zum Erstellen der Spielkonfiguration und zum Starten des Spiels. XML-Dateien für den Graphen und die Industrien werden eingelesen. Die Verbindung zu Spielern und Robotern wird aufgebaut. Zudem erfolgt die Einteilung der Teilnehmer in Teams. Erst nachdem alle Eingaben auf Korrektheit geprüft worden sind, kann das Spiel gestartet werden.

Attribute

Attributname	Datentyp	Beschreibung
parser	GraphParser	XML-Parser für die Map.
ri	RobotInformation	Objekt für die Roboterdaten beim Verbinden.
arf	AddRobotFrame	Frame zum Verbinden mit einem Roboter.
contenPane	JPanel	Top-Level-Container.
textField	JTextField	Eingabefeld für den Pfad der XML-Industriedatei.
textField_1	JTextField	Eingabefeld für den Pfad der XML-Kartendatei.
txtTest	JTextField	Eingabefeld für das Transportvolumen der Roboter.
textField_3	JTextField	Eingabefeld für den Treibstoffvorrat der Roboter.
spinner	JSpinner	Auswahl für die maximale Anzahl der Aufträge eines Roboters.
table_1	JTable	Table für die verbundenen Roboter.
model	DefaultTableModel	Enthält die Roboterdaten für table_1.
model_2	DefaultTableModel	Model für Spieler.
model_3	DefaultTableModel	Model für Industrien.
model_4	DefaultTableModel	Model für Team 0.
model_5	DefaultTableModel	Model für Team 1.
fuel	int	Treibstoffvorrat der Roboter.
maxcapacity	int	Maximales Transportvolumen der Roboter.
map	File	Kartendatei.
techtree	File	Industriedatei.
industryList	ArrayList<Industry>	Liste der Industrien der XML-Datei.
com	NXTManager	Kommunikationsobjekt eines Roboters.
table	JTable	Tabelle für die verbundenen Spieler.
btnNewButton_2	JButton	Button zum Aktivieren des ServerSockets.
table_2	JTable	Tabelle für Industrien.
table_4	JTable	Tabelle für Team 0.
table_5	JTable	Tabelle für Team 1.
panel_11	JPanel	Zeigt Status der XML-Kartendatei.
panel_13	JPanel	Zeigt Status der XML-Industriedatei.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
InitFrame	-	-	Konstruktor.
init	-	void	Erstellt den JFrame.
green_init	JTextField, JLabel	void	Genutzt bei der Visualisierung einer zulässigen XML Datei.
red_init	JTextField, JLabel	void	Genutzt bei der Visualisierung einer nicht zulässigen oder fehlenden XML-Datei.
updateTeamTables	-	void	Löscht die Teamtabellen und füllt sie wieder mit aktuellen Informationen aus der GameRegistry.
actionPerformed	ActionEvent	void	Methode des ActionListener Interfaces. Wird aktiv, sobald sich der ActionListener eines Objektes meldet. Führt die für den betätigten Button vorgesehenen Aktionen aus.
keyPressed	KeyEvent	void	Methode des KeyListener Interfaces.
keyReleased	KeyEvent	void	Methode des KeyListener Interfaces.
keyTyped	KeyEvent	void	Methode des KeyListener Interfaces. Wird genutzt, um Tabelleneinträge zu entfernen.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C60 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$, Komponente $\langle C100 \rangle$

GraphParser$\langle CL140 \rangle$

Aufgabe

Der GraphParser hat die Aufgabe, die Daten einer XML Datei auszulesen und daraus ein Graph-Objekt zu erzeugen. Dabei wird die Bibliothek jdom.jar verwendet.

Attribute

Attributname	Datentyp	Beschreibung
file	File	Auszulesende Datei.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
GraphParser	File	-	Konstruktor mit Angabe der XML-Datei.
GraphParser	String	-	Konstruktor mit Angabe des XML-Dateipfades.
parse	-	void	Methode startet das Auslesen einer XML-Datei.

Kommunikationspartner

Komponente $\langle C50 \rangle$

RobotInformation$\langle CL150 \rangle$

Aufgabe

Ein beim Verbindungsaufbau zu einem Roboter genutztes Objekt, in dem die Roboterdaten vorläufig gespeichert werden, bevor Daten an die GameRegistry weitergegeben werden.

Attribute

Attributname	Datentyp	Beschreibung
name	String	Name des Roboters.
location	int	Position des Roboters.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
RobotInformation	-	-	Konstruktor.

Kommunikationspartner

Komponente $\langle C50 \rangle$

AddRobotFrame$\langle CL160 \rangle$

Aufgabe

Ein kleiner JFrame zum Eingeben von Robotername und Position. Enthält einen Button zum Verbinden mit dem Roboter.

Attribute

Attributname	Datentyp	Beschreibung
conentPane	JPanel	Top-Level-Container.
textField_1	JTextField	Eingabefeld für die Position des Roboters.
textField	JTextField	Eingabefeld für den Namen des Roboters.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
AddRobotFrame	RobotInformation	-	Konstruktor, benötigt Referenz auf ein Objekt für die Roboterdaten.
main	String[]	-	Main-Methode für Testzwecke.

Kommunikationspartner

Komponente $\langle C50 \rangle$

IndustryParser$\langle CL170 \rangle$

Aufgabe

Steuerungklasse für den IndustryContentHandler. Benutzt das Paket org.xml.sax.*.

Attribute

Attributname	Datentyp	Beschreibung
ich	IndustryContentHandler	IndustyContentHandler-Objekt.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
IndustryParser	-	-	Konstruktor.
getFile	String	void	Nach Übergabe des Dateipfades erstellt die Methode ein Objekt zum Auslesen der XML-Datei und startet dessen Methode parse.

Kommunikationspartner

Komponente $\langle C50 \rangle$

IndustryContentHandler$\langle CL180 \rangle$

Aufgabe

Klasse zum Auslesen von Industriedaten aus einer XML-Datei. Implementiert ContentHandler. Benutzt das Paket org.xml.sax.*.

Attribute

Attributname	Datentyp	Beschreibung
industry	MyFirstIndustry	Objekt für aus der XML gelesene Industriedaten.
allIndustries	ArrayList<MyFirstIndustry>	Liste von Objekten mit Industriedaten.
currentValue	String	Variable für den aktuell ausgelesenen Wert.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
IndustryParser	-	-	Konstruktor.
character	char[], int, int	void	Aktuelle Zeichen, die gelesen werden, werden in eine Zwischenvariable gespeichert.
startElement	String, String, String, Attributes	void	Methode wird aufgerufen, wenn der Parser zu einem Start-Tag kommt.
endElement	String, String, String	void	Methode wird aufgerufen, wenn der Parser zu einem End-Tag kommt.
endDocument	-	void	Methode des Interface ContentHandler.
endPrefixMapping	String	void	Methode des Interface ContentHandler.
ignorableWhitespace	char[], int, int	void	Methode des Interface ContentHandler.
processingInstructions	String, String	void	Methode des Interface ContentHandler.
setDocumentLocator	Locator	void	Methode des Interface ContentHandler.
skippedEntity	String	void	Methode des Interface ContentHandler.
startDocument	-	void	Methode des Interface ContentHandler.

startPrefixMapping	String	void	Methode des Interface Content-Handler.

Kommunikationspartner

Komponente ⟨$C50$⟩

MyFirstIndustry⟨$CL190$⟩

Aufgabe

In die Attribute der Objekte dieser Klasse werden von dem IndustryContentHandler die aus der XML gelesenen Industriedaten gespeichert.

Attribute

Attributname	Datentyp	Beschreibung
id	int	Industrie-ID.
name	String	Name der Industrie.
position	int	Position der Industrie.
inputtype	ArrayList<String>	Namen der Waren am Wareneingang.
outputtype	ArrayList<String>	Namen der Produkte am Warenausgang.
inputamount	ArrayList<Integer>	Warenmengen am Wareneingang.
outputamount	ArrayList<Integer>	Warenmengen am Warenausgang.
inputprice	ArrayList<Double>	Preise am Wareneingang.
outputprice	ArrayList<Double>	Preise am Warenausgang.
factor	double	Produktionsfaktor.
fuel	int	Treibstoffvorrat.
productionTime	int	Produktionszeit.
stock	int	Größe des Warenlagers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
MyFirstIndustry	-	-	Konstruktor.

Kommunikationspartner

Komponente ⟨$C50$⟩

ServerFrame⟨$CL200$⟩

Aufgabe

Der ServerFrame dient dem Benutzer auf Serverseite zur Visualisierung des Spiels. Spielinformationen und Statistiken werden dargestellt.

Attribute

Attributname	Datentyp	Beschreibung
info	InfoFrame	Projektinformationsfenster.
displayPanel	GameDisplay	Interaktive Anzeige des Wegenetzes.
industrie	IndustryDetailsFrame	Industriedetailsfenster.
conentPane	JPanel	Top-Level-Container.
roboDataTable	JTable	Tabelle zur Anzeige der Roboterdaten.
orderTable	JTable	Tabelle zur Anzeige der Aufträge.
readDataOrderField	JTextField	Eingabefeld für den Pfad einer XML-Auftragsdatei.
startOrderField	JTextField	Eingabefeld zur Auftragserstellung.
endOrderField	JTextField	Eingabefeld zur Auftragserstellung.
weightField	JTextField	Eingabefeld zur Auftragserstellung.
minBidField	JTextField	Eingabefeld zur Auftragserstellung.
industryDetailsButton	JButton	Button für die Industriedetails.
roboRemoveButton	JButton	Button zum Entfernen eines Roboters.
createOrderButton	JButton	Button zum Senden eines Auftrags.
orderChange	JComboBox	Auswahl der Tabellenansicht.
menuBar	JMenuBar	Fenstermenü.
menuName	JMenu	Menüname.
menuHelp	JMenuItem	Menüauswahl Hilfe.
menuCloseGame	JMenuItem	Menüauswahl Spiel beenden.
menuInfo	JMenuItem	Menüauswahl Info.
routeDataPanel	JPanel	JPanel für das GameDisplay.
roboRemoveOrdercreatePanel	JPanel	JPanel.
createOrderPanel	JPanel	JPanel zur Auftragserstellung.
autoCreateOrderPanel	JPanel	JPanel zur Auftragserstellung.
createManuellOrderPanel	JPanel	JPanel zur Auftragserstellung.
orderChangePanel	JPanel	JPanel zur Auftragserstellung.
detailsFramePanel	JPanel	Enthält Button für Industriedetails.
newOrderLabel	JLabel	Enthält Überschrift für Auftragserstellung.
roboDeatailsLabel	JLabel	Enthält Überschrift für Roboterdetails.
group	ButtonGroup	RadioButtonGroup.
createOrderRadioButton	JRadioButton	RadioButton zur Auftragserstellung.
autoCreateOrderRadioButton	JRadioButton	RadioButton zur Auftragserstellung.
readOrderdataRadioButton	JRadioButton	RadioButton zur Auftragserstellung.

model	DefaultTableModel	Model für Aufträge in Auktionen.
model2	DefaultTableModel	Model für zu bearbeitende Aufträge.
model3	DefaultTableModel	Model für geleistete Aufträge.
modelRobo	DefaultTableModel	Model für Roboterdetails.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
ServerFrame	-	-	Konstruktor.
actionPerformed	ActionEvent	void	Methode des ActionListener Interfaces. Wird aktiv, sobald sich der ActionListener eines Objektes meldet. Führt die für den betätigten Button vorgesehenen Aktionen aus.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C50 \rangle$, Komponente $\langle C50 \rangle$

GameDisplay$\langle CL210 \rangle$

Aufgabe

Klasse zum Darstellen der Karte in einem JPanel zur Interaktion mit dem Benutzer. Implementiert ActionListener und MouseListener, um auf die Aktionen des Benutzers entsprechend reagieren zu können.

Attribute

Attributname	Datentyp	Beschreibung
graph	Graph	Karte
scale	int	Größe eines Knoten.
distX	int	Horizontale Distanz zwischen zwei Knoten.
distY	int	Vertikale Distanz zwischen zwei Knoten.
frameX	int	Abstand zum Rand.
frameY	int	Abstand zum Rand.
makeRoute	boolean	Schalter für Routenwahlmodus.
popup	JPopupMenu	Popup-Fenster des GameDisplay.
menu_1	JMenu	Popup-Fenster-Menü.
menu_1b	JMenuItem	Popup-Fenster-Menü.
menu_1c	JMenuItem	Popup-Fenster-Menü.
menu_2	JMenuItem	Popup-Fenster-Menü.
menu_3	JMenu	Popup-Fenster-Menü.

pop	JMenuItem[]	Popup-Fenster-Menü.
pop1	JLabel	Enthält Details zu angewähltem Objekt.
pop2	JLabel	Enthält Details zu angewähltem Objekt.
pop3	JLabel	Enthält Details zu angewähltem Objekt.
pop4	JLabel	Enthält Details zu angewähltem Objekt.
pop5	JLabel	Enthält Details zu angewähltem Objekt.
referenceObj	Object	Angewähltes Objekt beim Routenwahlmodus.
referenceOrder	Order	Order für die die Route geplant wird.
routeForRobot	Route	Zu planende Route.
robotWithoutRoute	Robot	Roboter, für den die Route geplant wird.
imgBuffer	Image	Zwischenspeicher für den Bildaufbau.
closed	Image	PNG für Streckensperrung.
robot	Image	PNG für Roboter.
industry	Image	PNG für Industrien.
teamColors	Color[]	Farben der Teams.
suggestionColor	Color	Farbe der vorgeschlagenen Route.
playerList	ArrayList<Player>	Liste der Teilnehmer.
industryList	ArrayList<Industry>	Liste der Industrien.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
GameDisplay	int, int	-	Konstruktor erwartet Breite und Höhe.
loadImages	-	void	Lädt kleine Grafikdateien.
scaling	Graphics2D	void	Berechnet das Koordinaten-Raster für die angezeigten Objekte anhand der momentanen Fenstergröße.
paintComponent	Graphics	void	Zeichnet den Inhalt des GameDisplays neu und blendet ihn anschließend ein.
drawEdge	Edge, Egde, Node, Graphics2D	void	Zeichnet abhängig von Markierungen und Positionen Kanten bzw. Objekte auf das GameDisplay.

drawNode	Node, Graphics2D, int, int	void	Zeichnet abhängig von Markierungen und Positionen Knoten bzw. Objekte auf das GameDisplay.
actionPerformed	ActionEvent	void	Methode des ActionListener Interfaces. Wird aktiv, sobald sich der ActionListener eines Objektes meldet. Führt die für den betätigten Button vorgesehenen Aktionen aus.
mouseReleased	MouseEvent	void	Sobald der Benutzer den gedrückten Mausknopf wieder loslässt, wird anhand der ermittelten Mausposition eine Aktion ausgeführt, wie beispielsweise das Öffnen eines Menüs. Jede Interaktion mit dem GameDisplay läuft über diese Methode.
mouseClicked	MouseEvent	void	MouseListener Interface.
mouseEntered	MouseEvent	void	MouseListener Interface.
mouseExited	MouseEvent	void	MouseListener Interface.
mousePressed	MouseEvent	void	MouseListener Interface.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C80 \rangle$

InfoFrame$\langle CL220 \rangle$

Aufgabe

Ein kleiner Frame, welcher die Spielversion und die Namen der SEP-Team Mitglieder anzeigt.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Top-Level-Container.
iconPanel	JPanel	Panel für Icon.
versionNamePanel	JPanel	Panel für Projektname und Version.
icon	JLabel	Label für Icon.
projectName	JLabel	Label für Projektname.

version	JLabel	Label für Version.
dennisStelter	JLabel	Label für Name.
henrikLange	JLabel	Label für Name.
markusBjoernMeissner	JLabel	Label für Name.
patriciaTatjanaKasulke	JLabel	Label für Name.
jochenSteiner	JLabel	Label für Name.
tessaFabian	JLabel	Label für Name.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
InfoFrame	-	-	Konstruktor.
start	InfoFrame	void	Startet den InfoFrame.

Kommunikationspartner

Komponente $\langle C50 \rangle$

IndustryDetailsFrame$\langle CL230 \rangle$

Aufgabe

Ein kleiner Frame, welcher während des Spiels die Industriedetails anzeigen soll.

Attribute

Attributname	Datentyp	Beschreibung
contentPane	JPanel	Top-Level-Container.
industryTable	JTable	Tabelle für Industrieinformationen.
closeFrame	JButton	Button zum Schließen der Industriedeatails.
industryTablePane	JScrollPane	Enthält die Tabelle für Industrieinformationen.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
IndustryDetailsFrame	-	-	Konstruktor.
actionPerformed	ActionEvent	void	Methode wird beim Schließen des Frames aufgerufen.

Kommunikationspartner

Komponente $\langle C50 \rangle$

5.6 Implementierung von Komponente $\langle C60 \rangle$: Kommunikation(Server)

Die Komponente Kommunikation(Server) dient der Datenübertragung zwischen Spieler und Server über ein TCP Netzwerk und der Datenübertragung zwischen Roboter und Server über eine Bluetoothverbindung.

5.6.1 Paket-/Klassendiagramm

Für die Komponente $\langle C60 \rangle$ ergeben sich nachfolgende Diagramme.

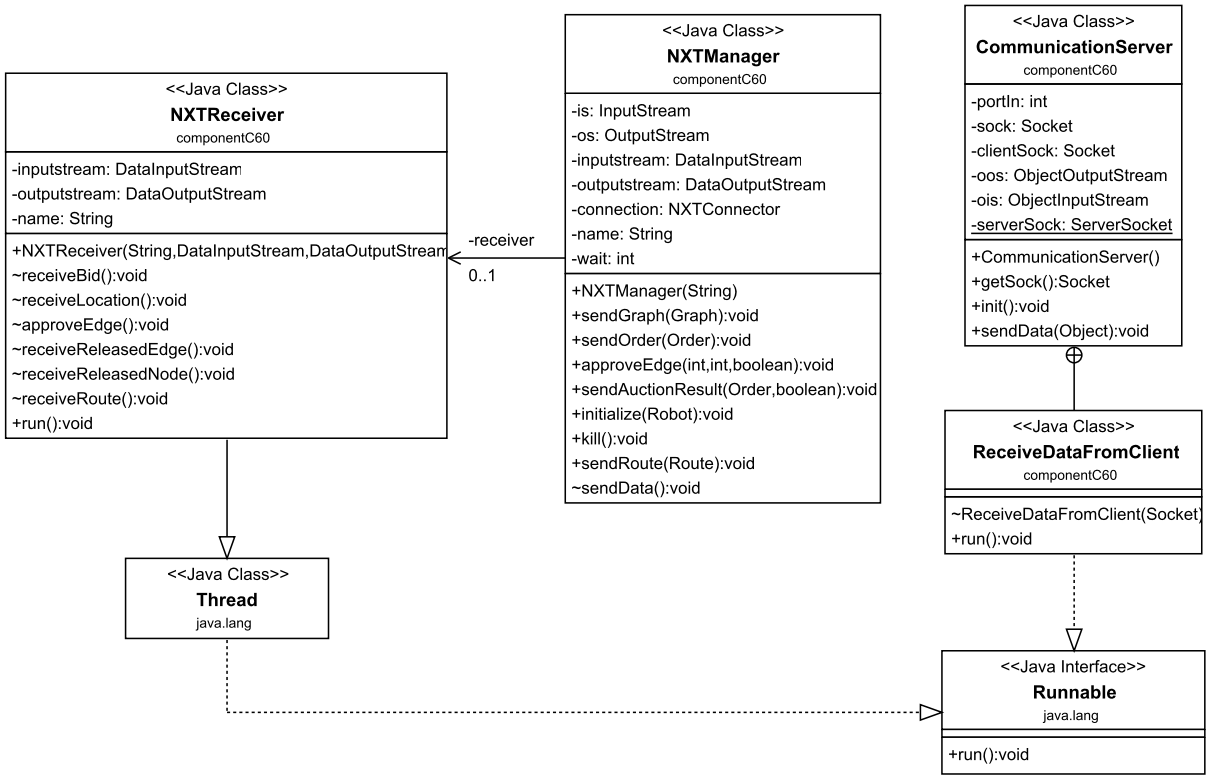

Abbildung 5.9: Klassendiagramm für Komponente $\langle C60 \rangle$

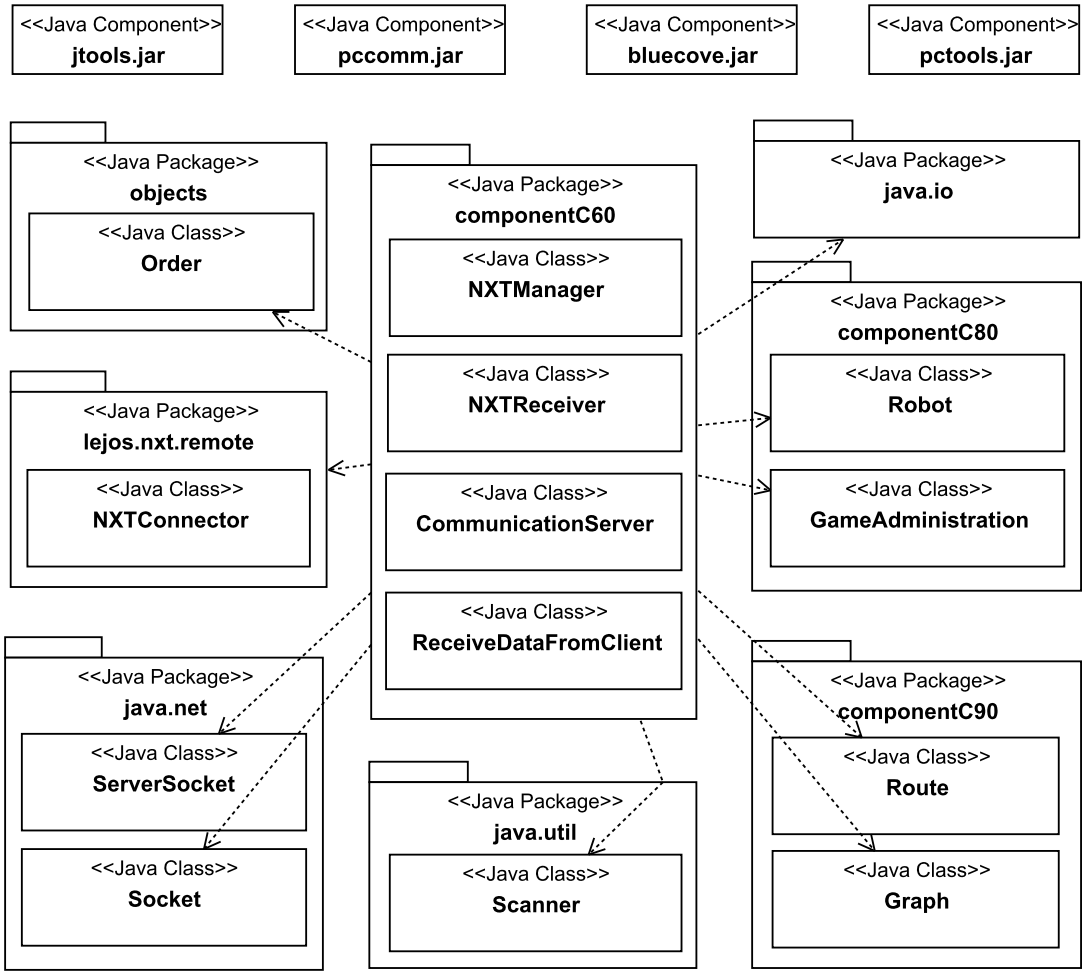

Abbildung 5.10: Paketdiagramm für Komponente $\langle C60 \rangle$

5.6.2 Erläuterung

CommunicationServer$\langle CL240 \rangle$

Aufgabe

Erzeugt einen Serversocket und wartet auf sich verbindende Spieler. Stellt ein Spieler eine Verbindungsanfrage, werden Inputstream und Outputstream über einen neuen Socket erstellt. Jede TCP-Kommunikation im Spiel verläuft über diese Klasse.

Attribute

Attributname	Datentyp	Beschreibung
portIn	int	Port auf dem der Server auf Verbindungen mit Spielern wartet.
sock	Socket	Socket der inneren Klasse für die Verbindung zu einem Spieler.
clientSock	Socket	Socket für die Verbindung zu einem Spieler.
oos	ObjectOutputStream	ObjectOutputStream einer Verbindung.
ois	ObjectInputStream	ObjectInputStream einer Verbindung.
serverSock	ServerSocket	ServerSocket des Servers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
init	-	void	Erstellt den ServerSocket und stellt die Verbindung zu einem Client her. Danach werden Input- und Outputstreams geöffnet.
sendData	Object	void	Senden eines Datenobjekts an den Client.

Kommunikationspartner

Komponente $\langle C40 \rangle$

ReceiveDataFromClient$\langle CL250 \rangle$

Aufgabe

Innere Klasse von CommunicationServer. Wird als Thread gestartet und wartet auf ankommende Daten des InputStreams.

Attribute

Keine eigenen Attribute vorhanden. Nutzt als innere Klasse den Inputstream von ConnectionServer.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
run	-	void	Wenn gestartet, werden Daten des Inputstreams gelesen und an die Spielkoordination weitergereicht.

Kommunikationspartner

Komponente $\langle C80 \rangle$

NXTManager⟨*CL*260⟩

Aufgabe

Nachdem der Name eines NXT-Roboters als String an den Konstruktor übergeben wurde, stellt die Klasse eine Bluetoothverbindung zum NXT-Roboter her. Dazu werden ein Inputstream und ein Outputstream erstellt. Dabei werden die Bibliotheken jtools.jar, pccomm.jar, bluecove.jar und pctools.jar verwendet. Weiter enthält die Klasse Methoden, die die für den Roboter wichtigen Daten der auf Serverseite genutzten Objekte auslesen und über ein festgelegtes Protokoll an den Roboter als eine Abfolge einzelner Integer-Werte übertragen. Die Bluetooth Kommunikation vom Server zum Roboter verläuft über diese Klasse.

Attribute

Attributname	Datentyp	Beschreibung
receiver	NXTReceiver	Speicherung des Empfängers.
is	InputStream	InputStream.
os	OutputStream	OutputStream.
inputstream	DataInputStream	Eingangsdatenstrom.
outputstream	DataOutputStream	Ausgangsdatenstrom.
connection	NXTConnector	Bluetoothverbindungsobjekt.
receiver	NXTReceiver	Empfänger der Bluetoothverbindung.
name	String	Name des Roboters.
wait	int	Wartezeit zwischen den Nachrichten.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
NXTManager	String	-	Konstruktor.
sendGraph	Graph	void	Sendet den Graphen an den NXT.
sendOrder	Order	void	Sendet einen Auftrag an den NXT.
approveEdge	int, int, boolean	void	Sendet den Status einer Kante zwischen zwei bestimmen Knoten an den Roboter.
sendAuctionResult	Order, boolean	void	Sendet das Ergebnis einer Auktion an den NXT.
initialize	Robot	void	Übertragen von Location, MaxCapacity, MaxOrders und Fuel an den NXT.

kill	-	void	Befehl zum Stoppen eines Roboters, um ihn aus dem Spiel zu entfernen.
sendRoute	Route	void	Überträgt eine vom Spieler geplante Route an einen Spieler-Roboter.
sendData	-	void	Methode überträgt einzelne Integer zum Testen und Erweitern des Protokolls.

Kommunikationspartner

Komponente $\langle C30 \rangle$, Komponente $\langle C80 \rangle$

NXTReceiver$\langle CL270 \rangle$

Aufgabe

Die Klasse erzeugt einen Thread, in dem der InputStream der Verbindung zum NXT permanent nach ankommenden Daten abgefragt wird. Wurde ein bestimmter Integer-Wert empfangen, wird die dem Protokoll entsprechende Methode der Klasse aufgerufen, um die weiteren Integer-Daten zu sammeln und dann zur GameAdministration weiterzuleiten.

Attribute

Attributname	Datentyp	Beschreibung
inputstream	DataInputStream	Dateneingabestrom.
outputstream	DataOutputStream	Datenausgabestrom.
name	String	Robotername.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
NXTReceiver	String, DataInputStream, DataOutput-Stream	-	Konstruktor
receiveBid	-	void	Empfängt ein Gebot vom NXT.
receiveLocation	-	void	Empfängt die Position des NXT. Meldet das Erreichen eines Knotens.

approveEdge	-	void	Empfängt eine Freigabeanfrage des NXT. Angefragt werden Knoten und die dahinterliegende Kante.
receiveReleasedEdge	-	void	Wird benötigt, wenn der Roboter das Verlassen einer Kante meldet.
receiveReleasedNode	Robot	void	Wird empfangen, wenn der NXT das Verlassen eines Knotens sendet.
receiveRoute	-	void	Eine von einem Spieler-NXT an den Spieler vorgeschlagene Route wird empfangen.
run	-	void	In einer Schleife wird der erste empfangene Integer-Wert ausgewertet und dann eine der anderen Methoden der Klasse aufgerufen.

Kommunikationspartner

Komponente $\langle C30 \rangle$, Komponente $\langle C80 \rangle$

5.7 Implementierung von Komponente $\langle C70 \rangle$: Auktionsverwaltung

Die Komponente Auktionsverwaltung dient der Verwaltung von Auktionen. Sie nimmt Gebote von Robotern bzw. Spielern entgegen und ermittelt einen Gewinner für die Aufträge.

5.7.1 Paket-/Klassendiagramm

Für die Komponente $\langle C70 \rangle$ ergeben sich nachfolgende Diagramme.

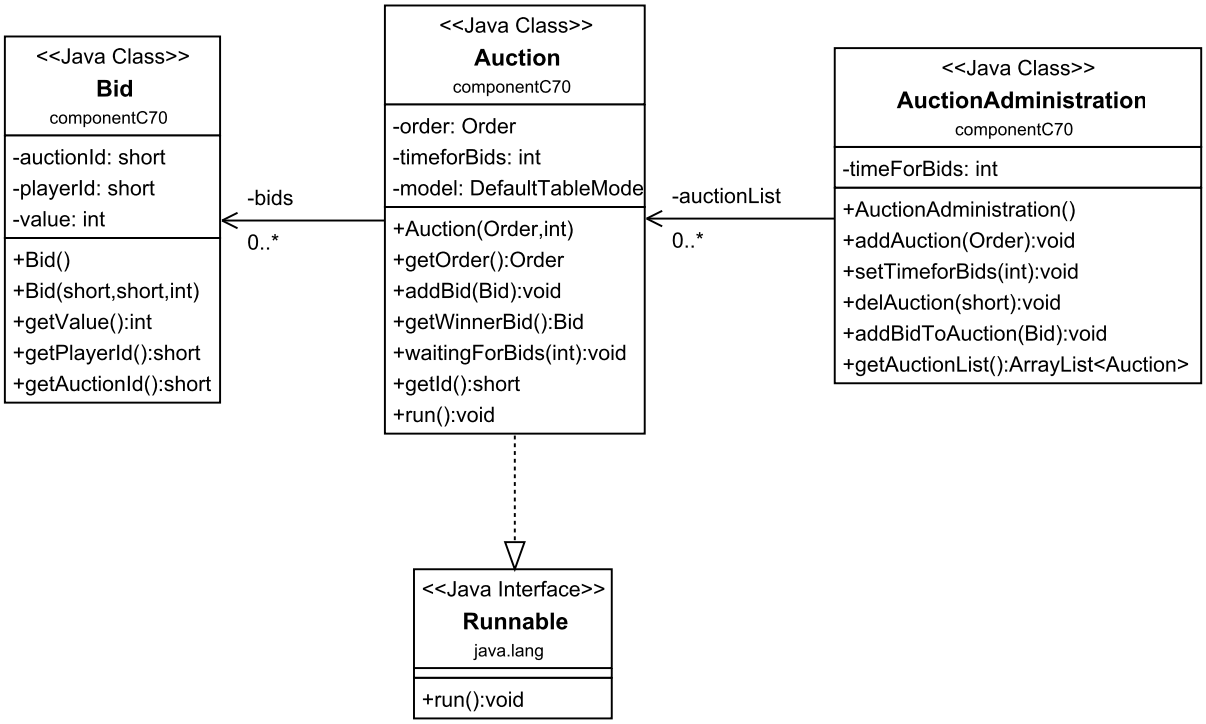

Abbildung 5.11: Klassendiagramm für Komponente $\langle C70 \rangle$

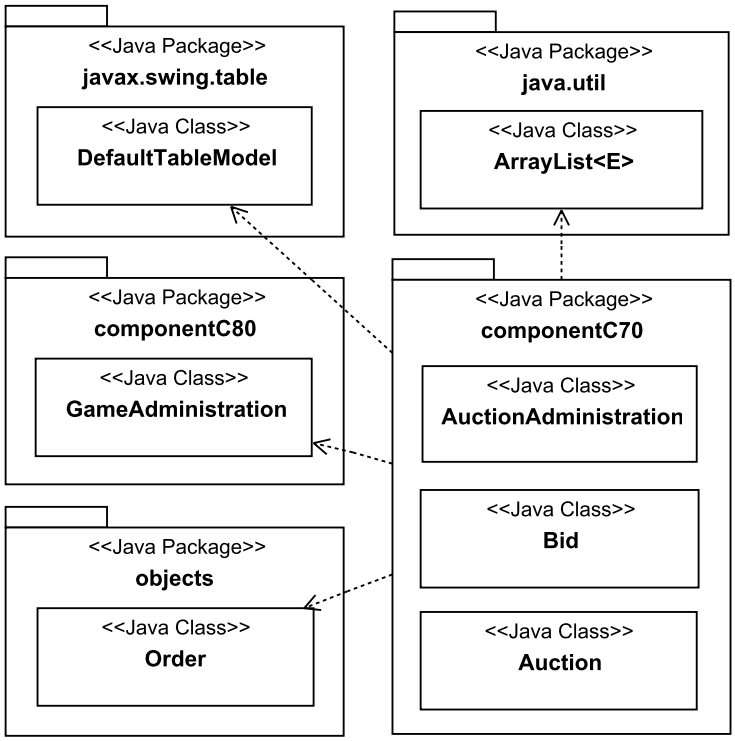

Abbildung 5.12: Paketdiagramm für Komponente $\langle C70 \rangle$

5.7.2 Erläuterung

Auction⟨$CL280$⟩

Aufgabe

Verwaltung von Auktionen und Ermittlung eines Gewinners.

Attribute

Attributname	Datentyp	Beschreibung
bids	ArrayList<Bid>	Liste zum Speichern der Gebote.
order	Order	Order zum Speichern eines Auftrags.
timeforBids	int	Zeit, die die Roboter/Spieler zum Bieten haben.
model	DefaultTableModel	Dient zur korrekten Darstellung von Tabellen.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Auction	Order, int	-	Konstruktor zum Erstellen einer Instanz Auction.
addBid	Bid	void	Fügt ein neues Gebot hinzu.
waitingForBids	int	void	Wartet auf eingehende Gebote.
run	-	void	run-Methode zum starten der Auktion als Thread.
getWinnerBid	-	-	Ermittelt das Gebot des Gewinners aus der Liste der Gebote.

Kommunikationspartner

Komponente ⟨$C80$⟩

Bid⟨$CL290$⟩

Aufgabe

Verwaltung von Geboten.

Attribute

Attributname	Datentyp	Beschreibung
auctionId	short	Eindeutige ID für Auktionen/Order.
playerId	short	Eindeutige ID für Spieler/Roboter als Teilnehmer.
value	int	Wert eines Gebots.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Bid	-	-	Standard-Konstruktor zum Erstellen einer Instanz Bid.
Bid	short,short,int	-	Konstruktor zum Erstellen einer Instanz von Bid, diesem wird die Auktions-ID,Teilnehmer-ID und der Wert eines Gebots übergeben.

Kommunikationspartner

Komponente $\langle C80 \rangle$

AuctionAdministration$\langle CL300 \rangle$

Aufgabe

Verwaltung von Auktionen und Ermittlung eines Gewinners.

Attribute

Attributname	Datentyp	Beschreibung
auctionList	ArrayList<Auction>	Liste zum Speichern von Auktionen.
timeforBids	int	Zeit, die die Roboter/Spieler zum Bieten haben.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
AuctionAdministration	-	-	Standard-Konstruktor zum Erstellen einer Instanz AuctionAdministration.
addAuction	Order	void	Fügt Auktion zur Liste hinzu und startet Auktion.
delAuction	short	void	Entfernt eine Auktion aus der Liste.
addBidToAuction	Bid	void	Fügt zu einer Auktion ein Gebot hinzu.

Kommunikationspartner

Komponente $\langle C80 \rangle$

5.8 Implementierung von Komponente $\langle C80 \rangle$: Spielkoordination

Die Spielkoordination vermittelt mit der Klasse GameAdministration die Daten zwischen allen anderen Komponenten des Servers. Zudem hält sie die Objekte der registrierten Roboter und Spieler in der Klasse GameRegistry bereit.

5.8.1 Paket-/Klassendiagramm

Für die Komponente $\langle C80 \rangle$ ergeben sich nachfolgende Diagramme.

Das Klassendiagramm befindet sich im Anhang in Abschnitt G, es sollte in DIN A3 ausgedruckt werden.

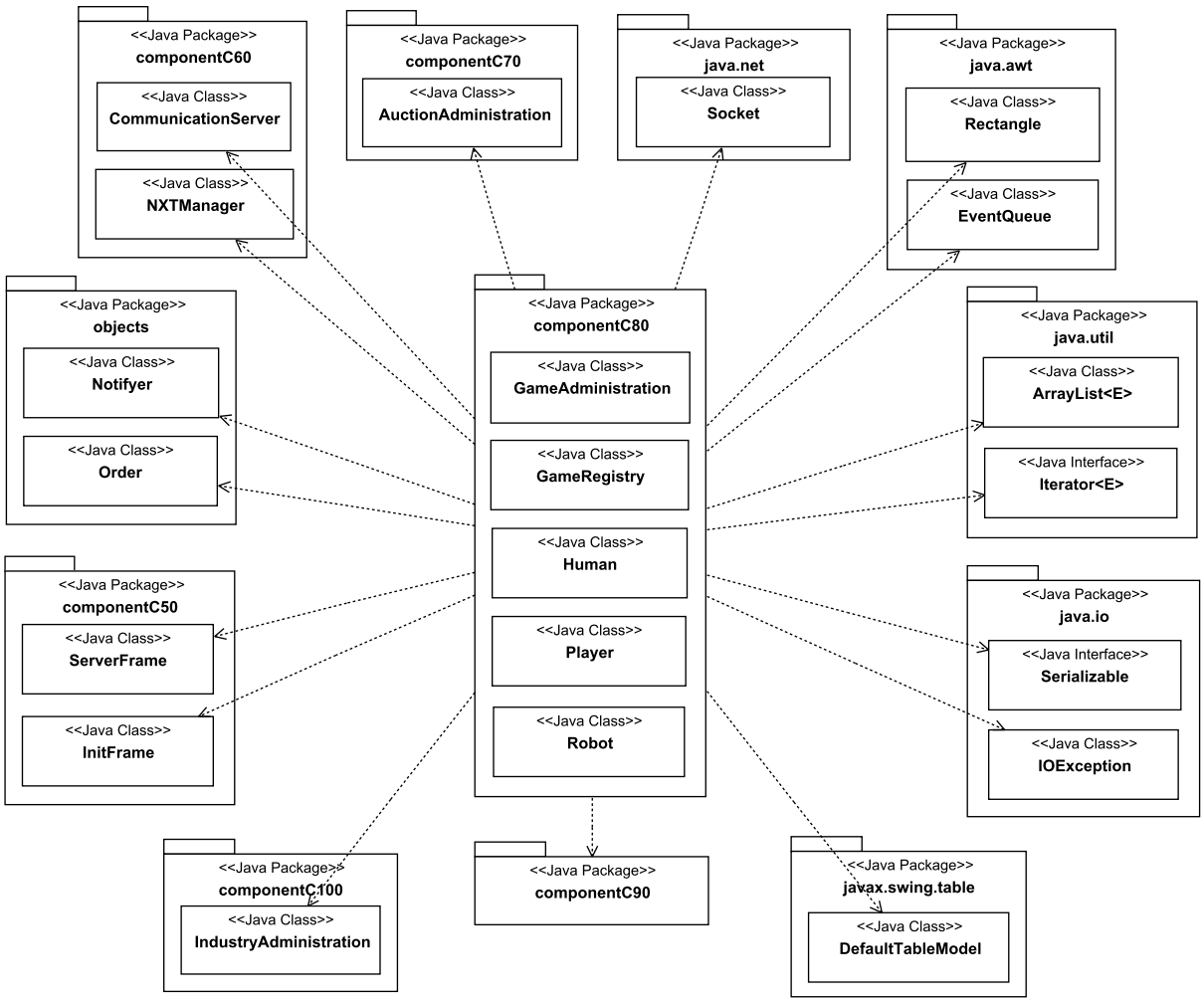

Abbildung 5.13: Paketdiagramm für Komponente $\langle C80 \rangle$

5.8.2 Erläuterung

GameAdministration$\langle CL310 \rangle$

Aufgabe

Die GameAdministration ist die ausführbare Klasse des Servers. Über sie wird anfangs der zur Initialisierung dienende InitFrame gestartet und nach abgeschlossener Initialisierung der ServerFrame. Die Klasse selbst ist als Singleton implementiert. So greifen die anderen Komponenten über die einzige Instanz der GameAdministration auf deren Methoden und Objekte zu. Sämtlicher Netzwerkverkehr läuft in der Klassenmethode distributeData zusammen. Dort werden Objekte mit einem dem Protokoll entsprechenden String-Header ausgelesen, Aktionen wie das Übergeben von Daten an andere Komponenten durchgeführt und eine eventuelle Rückgabe mit neuem Header an das jeweilige Ziel zurückgesendet.

Attribute

Attributname	Datentyp	Beschreibung
instance	GameAdministration	Instanz der GameAdministration.
orderList	ArrayList<Order>	Liste der erstellten Aufträge.
indyadmin	IndustryAdministration	Objekt der Industrieverwaltung.
auctadmin	AuctionAdministration	Objekt der Auktionsverwaltung.
mapadmin	MapAdministration	Objekt der Wegenetzverwaltung
iframe	InitFrame	Initialisierungfenster.
sframe	ServerFrame	Grafische Benutzeroberfläche des Servers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
main	String[]	void	Ausführbare main-Methode startet die Server-Applikation.
GameAdministration	-	-	Konstruktor.
getInstance	-	GameAdministration	Rückgabe der einzigen Instanz der Klasse.
init	-	void	Initialisierung der Attribute und starten des InitFrame.
startServerFrame	-	void	Initialisieren und Starten des ServerFrame.
industryToMap	-	void	Leitet die bei der Initialisierung geladenen Industriedaten an die Wegenetzverwaltung weiter.
robotsToGui	-	void	Weiterleitung der Roboterdaten der Initialisierung an die Tabelle des ServerFrame.
robotsToMap	-	boolean	Leitet die bei der Initialisierung geladenen Roboterdaten an die Wegenetzverwaltung weiter.
addOrder	Order	void	Speichert einen erstellten Auftrag in orderList und gibt ihn weiter an die Auktionsverwaltung.
sendToClients	Object	void	Ein Objekt wird an alle Spieler gesendet.
sendToAClient	Object, short	void	Ein Objekt wird an einen bestimmten Spieler gesendet.

sendToRobots	Object	void	Ein Objekt wird an alle Roboter gesendet.
sendToARobot	Object, short	void	Ein Objekt wird an einen bestimmten Roboter gesendet.
issueOrder	Order, short	void	Einen Auftrag an einen bestimmten Teilnehmer senden.
distributeData	Object	void	Empfängt ein Daten-Objekt, liest den Header aus und führt eine entsprechende Aktion durch.
getRowIndex	DefaultTableModel, int, short	int	Gibt die Zeilennummer zu einer ID in einem DefaultTableModel zurück.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C60 \rangle$, Komponente $\langle C70 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$, Komponente $\langle C100 \rangle$

GameRegistry$\langle CL320 \rangle$

Aufgabe

Die GameRegistry enthält die Objekte der Teilnehmer in speziellen Teilnehmerlisten. Die Klasse ist als Singleton implementiert, da sie nur einmal existieren soll. Alle Komponenten des Servers haben Zugriff auf die Objekte der Teilnehmer über diese einzige Instanz der Klasse.

Attribute

Attributname	Datentyp	Beschreibung
instance	GameRegistry	Instanz der GameRegistry.
playerList	ArrayList<Player>	Liste der teilnehmenden Roboter und Spieler.
humanList	ArrayList<Human>	Liste der teilnehmenden Spieler.
robotList	ArrayList<Robot>	Liste der teilnehmenden Roboter.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
GameRegistry	-	-	Konstruktor.
getInstance	-	GameRegistry	Rückgabe der einzigen Instanz der Klasse.

addHuman	String, CommunicationServer	-	Erstellt und fügt einen Spieler zur Registry hinzu.
addHuman	String	-	Fügt einen Spieler zur Registry hinzu.
addRobot	String, int, NXTManager	-	Fügt einen Roboter zur Registry hinzu.
getRobotIdByName	String	short	Gibt die ID eines Roboters anhand seines Namens zurück.
deactivateRobot	short	-	Methode zum Deaktivieren eines Roboters in der Registry.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C60 \rangle$, Komponente $\langle C70 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$

Player$\langle CL330 \rangle$

Aufgabe

Die Klasse Player ist die Oberklasse für die beiden Teilnehmerklassen Human und Robot. Da Objekte dieser Klasse über das TCP-Netzwerk versendet werden, implementiert sie das Interface Serializable.

Attribute

Attributname	Datentyp	Beschreibung
id	short	Fortlaufende ID für Teilnehmer.
playerId	short	Einzigartige ID eines Teilnehmers.
name	String	Name des Teilnehmers.
cash	double	Konto des Teilnehmers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Player	-	-	Konstruktor.
Player	short, String	-	Konstruktor erstellt einen Teilnehmer mit TeamID und Name.
addCash	double	void	Fügt dem Spielerkonto einen Betrag hinzu.

Kommunikationspartner

Komponente ⟨$C50$⟩, Komponente ⟨$C80$⟩

Human⟨$CL340$⟩

Aufgabe

Objekte dieser Klasse enthalten besondere Informationen eines Spielers. Dazu gehören das Objekt für die TCP- Verbindung und eine Liste, in der die zu steuernden Roboter vermerkt sind.

Attribute

Attributname	Datentyp	Beschreibung
ownsRobots	ArrayList<short>	Liste der IDs der zu steuernden Roboter.
com	CommunicationServer	TCP-Verbindungsobjekt des Spielers.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Human	String, CommunicationServer	-	Konstruktor.
Human	String, short	-	Konstruktor mit Name und ID.
addRobot	short	void	Weist dem Spieler einen Roboter zu.

Kommunikationspartner

Komponente ⟨$C50$⟩, Komponente ⟨$C80$⟩

Robot⟨$CL350$⟩

Aufgabe

In dieser Klasse werden die zusätzlichen Attribute eines Roboters gespeichert. Dazu gehören spezielle Markierungen und Flächen für die Darstellung und Interaktion auf der Spieloberfläche, sowie die Eigenschaften bzw. Position eines Roboters im Spiel.

Attribute

Attributname	Datentyp	Beschreibung
bounding	Rectangle	Feld des Roboters auf dem GameDisplay.
location	int	Position des Roboter auf dem Wegenetz.
maxcapacity	int	Maximale Transportmenge des Roboters.
fuel	int	Treibstoff des Roboters.
maxorders	int	Größe der Auftragsliste eines Roboters.

marked	boolean	Ist der Roboter durch Benutzer/Spieler auf dem GameDisplay ausgewählt oder nicht.
orders	ArrayList<Order>	Zu bearbeitende Aufträge.
manager	NXTManager	Bluetooth-Verbindungsobjekt des Roboters.
belongsToHuman	short	Wird der Roboter durch einen Spieler gesteuert oder nicht.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Robot	String, int, NXTManager	-	Konstruktor mit Name, Position und Verbindungsobjekt.
addOrder	Order	void	Fügt der Auftragsliste des Roboters einen Auftrag hinzu.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$

5.9 Implementierung von Komponente $\langle C90 \rangle$: Wegenetzverwaltung

Die Wegenetzverwaltung kümmert sich um alle Aufgaben rund um das Wegenetz. Sie verwaltet die Kartendaten und sperrt Strecken bzw. gibt sie wieder frei.

5.9.1 Paket-/Klassendiagramm

Für die Komponente $\langle C90 \rangle$ ergeben sich nachfolgende Diagramme.

Das Klassendiagramm befindet sich im Anhang in Abschnitt H, es sollte in DIN A3 ausgedruckt werden.

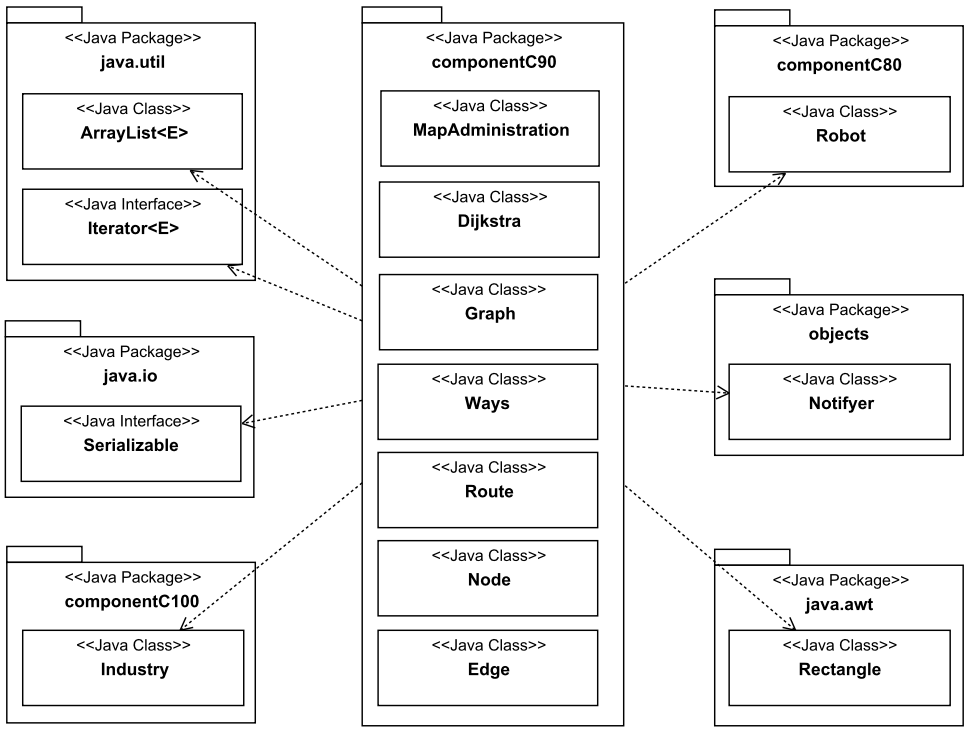

Abbildung 5.14: Paketdiagramm für Komponente $\langle C90 \rangle$

5.9.2 Erläuterung

Die Komponente Wegenetzverwaltung ist wie folgt aufgebaut:

MapAdministration$\langle CL360 \rangle$

Aufgabe

Diese Klasse ist der Kern der Komponente, von hier aus werden alle Operationen (innerhalb der Komponente) ausgeführt.

Attribute

Attributname	Datentyp	Beschreibung
graph	Graph	Instanz des Graphen, die zur Verwaltung genutzt wird.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
MapAdministration	-	-	Konstruktor zum Erzeugen einer neuen Wegenetzverwaltung.
MapAdministration	Graph	-	Konstruktor zum Erzeugen einer neuen Wegenetzverwaltung, bekommt den zu verwaltenden Graphen übergeben.
getPermission	Node, Edge, short	boolean	Methode zum Erfragen der Freigabe einer Kante. Übergeben werden ein Knoten zur schnelleren Identifikation, eine Kante und die ID des Roboters. Gibt true zurück, wenn die Kante frei ist.
getLocation	Object	int	Gibt den Standort eines übergebenen Roboters/Objektes in Form der Knotennummer zurück.
setLocation	Object, int	boolean	Setzt Objekte auf der Karte an einen bestimmten Ort, der in Form der Knotennummer übergeben werden muss. Gibt true zurück, wenn die Positionierung erfolgreich verlaufen ist, und false, wenn die Position schon besetzt ist.
releaseSelection	Object	void	Gibt einen Streckenabschnitt frei. Dabei kann sowohl ein Knoten, als auch eine Kante übergeben werden.
makeCoordinates	Graph	Graph	Erzeugt Koordinaten für Knoten innerhalb eines Graphen zur späteren Darstellung. Ein Graph ohne Koordinaten muss dazu übergeben werden.
alignGraphList	Graph	Graph	Verschiebt den übergebenen Graphen in den Bildbereich, so dass keine negativen Knoten mehr existieren, und gibt ihn zurück.

Kommunikationspartner

Komponente $\langle C80 \rangle$

Graph⟨$CL370$⟩

Aufgabe

Objektklasse zur Speicherung und Verwaltung eines Graphen.

Attribute

Attributname	Datentyp	Beschreibung
graph	ArrayList<Node>	Array-Liste zur Speicherung der Knoten.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Graph	-	-	Konstruktor zur Erzeugung eines Graph-Objekts.
Graph	ArrayList<Node>	-	Konstruktor zur Erzeugung eines Graph-Objekts aus einer übergebenen Knotenliste.
getTempGraphList	-	ArrayList <Node>	Liefert temporäre Kopie des Graphen für Bearbeitungszwecke.
getNodeByNumber	int	Node	Gibt das Knotenobjekt zu einer bestimmten Position zurück.
getEdgeByNumber	short	Edge	Gibt das Kantenobjekt zu einer bestimmten ID zurück.
getIndexOfNode	short	int	Sucht den angegebenen Knoten im Graphen und liefert seine Speicherposition.
size	-	int	Gibt die Größe des Graphen (Anzahl der Knoten) aus.
addNode	int	void	Fügt einen Knoten zum Graphen hinzu.
addEdge	int, int, char, int	void	Fügt eine Kante hinzu. Dabei müssen die Nummer des Knotens, von dem die Kante ausgeht, die Nummer des erreichten Knotens, die Richtung, in der die Kante den Ursprungsknoten verlässt und die Länge der Kante angegeben werden.

addEdge	int, Edge	void	Fügt eine Kante hinzu. Übergeben werden müssen die Nummer des Ursprungsknotens und ein Kantenobjekt.
nodesToGraph	ArrayList<Node>	void	Fügt Knoten in Form einer Array-Liste in den Graphen ein.
shortestWay	int	Ways	Errechnet kürzeste Wege vom gegebenen Anfangspunkt.
getDirection	int, int	char	Liefert die Ausrichtung einer Kante zu einem Knoten. Übergeben werden der Zielknoten und der Startknoten.
route	int, int, char	Route	Liefert eine zu fahrende Route. Dazu müssen der Start- und Zielknoten und ein „s“, wenn die Route zu einem Auftragsanfang, bzw. ein „f“, wenn sie zu einem Auftragsziel führen soll, übergeben werden.
getEdgesBetween	short, short	Edge[]	Gibt die Kanten aus, die zwischen zwei übergebenen Knoten liegen.
edgeIsBetween	short	Node[]	Gibt die Knoten zurück, zwischen denen die angegebene Kante liegt.

Kommunikationspartner

Keine.

Node⟨$CL380$⟩

Aufgabe

Objektklasse zur Speicherung und Verwaltung eines Knotens.

Attribute

Attributname	Datentyp	Beschreibung
bounding	Rectangle	Bounding Box um den Knoten zur späteren Darstellung.
number	short	Nummer des Knotens für den Zugriff.
edges	ArrayList<Edge>	Anliegende Kanten des Knotens gespeichert in einer Array-Liste.
usedBy	short	Variable zur Speicherung des benutzenden Roboters.

hasIndustry	short	Speichert die ID einer Industrie, wenn keine Industrie vorhanden ist, wird -1 gespeichert.
marked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
rmarked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
smarked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
posX	double	Rasterkoordinate für die Graphen-Visualisierung.
posY	double	Rasterkoordinate für die Graphen-Visualisierung.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
inUse	-	boolean	Gibt aus, ob der Knoten in Benutzung ist.
setUnused	-	void	Markiert den Status des Knotens als in Benutzung.
Node	-	-	Konstruktor zum Erstellen eines Knoten-Objekts.
Node	short, ArrayList<Edge>	-	Konstruktor zum Erstellen eines Knoten-Objekts mit übergebener Knotennummer und anliegenden Kanten.
getEdge	int	Edge	Gibt das Kantenobjekt zu einem bestimmten Zielknoten aus.
getNodeByNumber	short, ArrayList<Node>	Node	Liefert den zur Knotennummer passenden Knoten im übergebenen Graphen.
addEdge	Edge	void	Fügt eine Kante zu dem Knoten hinzu.

Kommunikationspartner

Keine.

Edge⟨$CL390$⟩

Aufgabe

Objektklasse zur Speicherung und Verwaltung einer Kante.

Attribute

Attributname	Datentyp	Beschreibung
id	short	ID der Kante zur Visualisierung.
number	short	Nummer der Kante.
bounding	Rectangle	Bounding Box um die Kante zur späteren Darstellung.
destination	short	Zielknoten der Kante.
direction	char	Himmerlsrichtung der Kante.
weight	short	Gewichtung bzw. Länge der Kante.
marked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
rmarked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
smarked	boolean	Hilfsariable zum Aufbau des Graphen bei der Visualisierung.
usedBy	short	Variable zur Speicherung des benutzenden Roboters.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Edge	short, char, short	-	Konstruktor zum Erzeugen eines Kanten-Objekts.
resetId	-	void	Setzt die ID der Kante zurück.

Kommunikationspartner

Keine.

Dijkstra⟨$CL400$⟩

Aufgabe

Klasse zur Ausführung des Dijkstra-Algorithmus zur Berechnung kürzester Wege.

Attribute

Diese Klasse besitzt keine Attribute.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Dijkstra	-	-	Konstruktor.
shortestway	short, ArrayList<Node>	Ways	Methode zur Berechnung der kürzesten Wege in einem Graphen von einem bestimmten Startknoten.

Kommunikationspartner

Keine.

Route⟨$CL410$⟩

Aufgabe

Objektklasse zur Speicherung und Verwaltung einer Route.

Attribute

Attributname	Datentyp	Beschreibung
toVisitNodes	ArrayList<Short>	Array-Liste zur Speicherung der Knoten, die noch besucht werden sollen.
orientation	ArrayList<Character>	Array-Liste zur Speicherung der Richtung, in die an den jeweils aufeinanderfolgenden Knoten gefahren werden soll.
distance	ArrayList<Short>	Array-Liste zur Speicherung der einzelnen Entfernungen zwischen den zu besuchenden Knoten.
orderId	short	ID des aktuellen Auftrags.
collect	boolean	Dient zur Bestimmung, ob Waren bereits abgeholt sind oder noch abgeholt werden müssen.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Route	-	-	Konstruktor zum Erzeugen einer neuen Route.
addRoute	Route	void	Hängt eine Route an das Ende der aktuellen.
addToVisitNode	int	void	Speichert eine Knotennummer als zu besuchen ab.
addOrientation	char	void	Fügt neues Manöver hinzu, das beim nächsten Knoten ausgeführt werden muss.
addDistance	short	void	Fügt Distanz zum Ziel hinzu.
getNextCommand	-	char	Liefert das nächste auszuführende Manöver.
size	-	int	Gibt die Größe der Route in Form der zu besuchenden Knoten aus.

Kommunikationspartner

Keine.

Ways$\langle CL420 \rangle$

Aufgabe

Klasse zur internen Verwendung bei der Berechnung kürzester Wege für eine einfache Übergabe der Werte

Attribute

Attributname	Datentyp	Beschreibung
distance	short[]	Array zur Speicherung von Distanzen zwischen Knoten.
parent	short[]	Array zur Speicherung des vorherigen Knotens.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Ways	int	-	Konstruktor setzt den Weg.

Kommunikationspartner

Keine.

5.10 Implementierung von Komponente $\langle C100 \rangle$: <Industrieverwaltung>

Diese Komponente dient zur Verwaltung der Industrien im Spiel.

5.10.1 Paket-/Klassendiagramm

Für die Komponente $\langle C100 \rangle$ ergeben sich nachfolgende Diagramme.

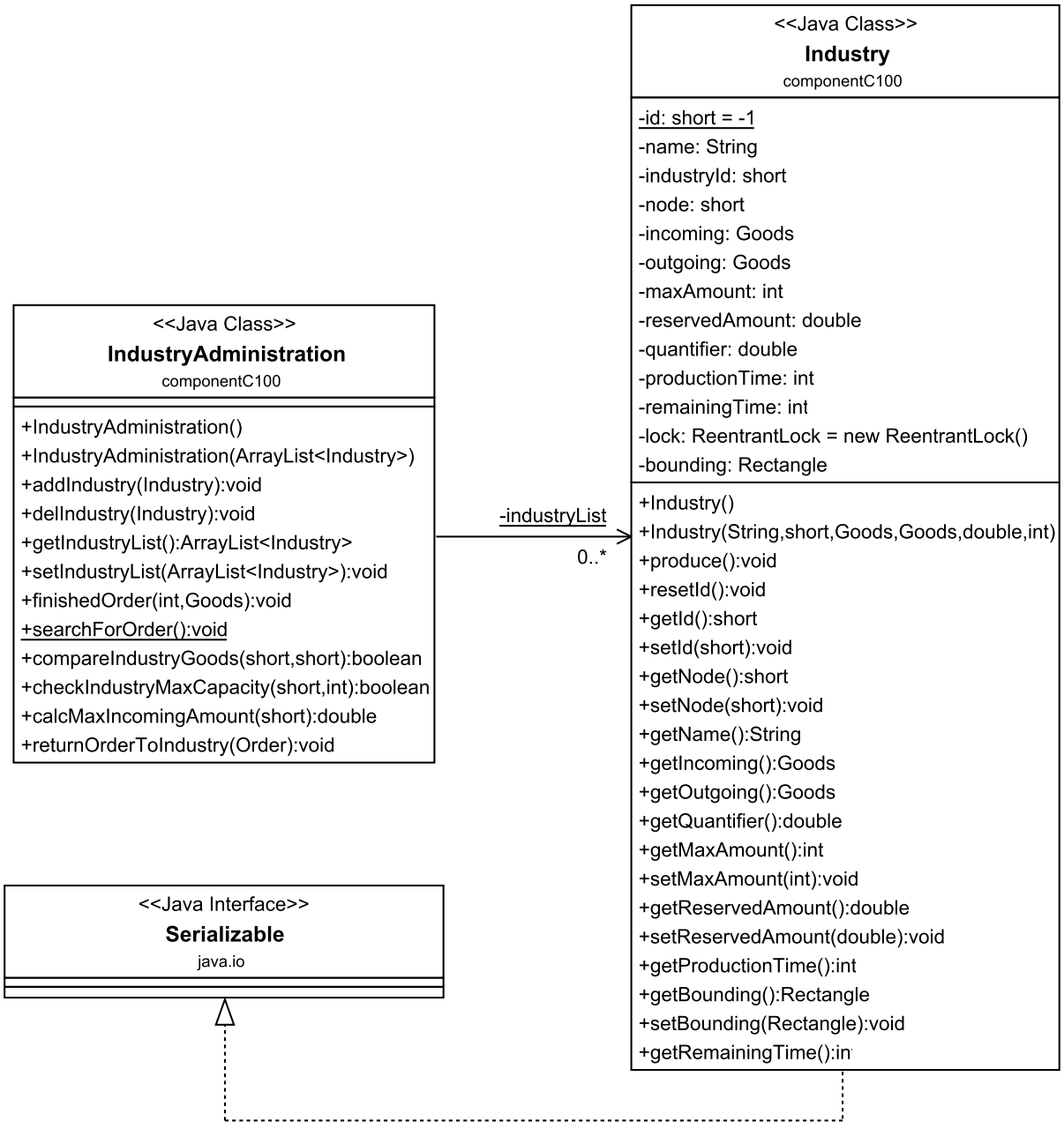

Abbildung 5.15: Klassendiagramm für Komponente $\langle C100 \rangle$

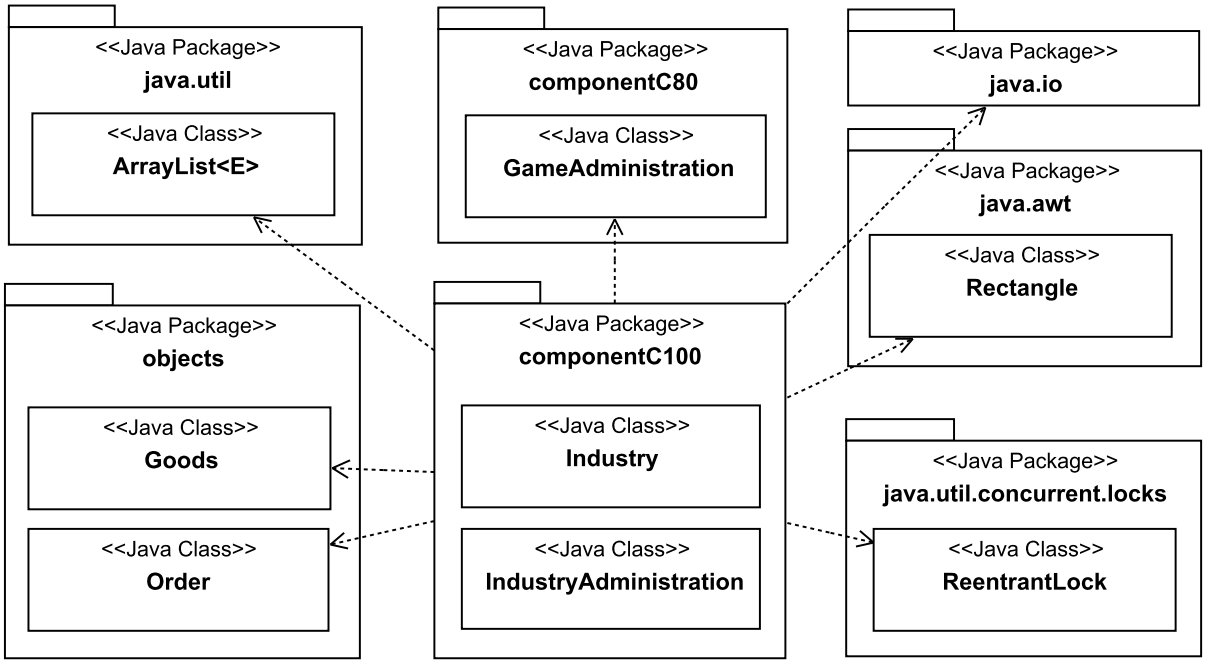

Abbildung 5.16: Klassendiagramm für Komponente $\langle C80 \rangle$

5.10.2 Erläuterung

IndustryAdministration$\langle CL430 \rangle$

Aufgabe

Die Klasse IndustryAdministration verwaltet die Industrien mit Hilfe einer Array-Liste. Sie beinhaltet unter anderem das Hinzufügen und Entfernen von Industrien.

Attribute

Attributname	Datentyp	Beschreibung
industryList	ArrayList<Industry>	Liste der Industrien.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
IndustryAdministration	-	-	Konstruktor.
IndustryAdministration	-	ArrayListe <Industry>	Ist Konstruktor und es wird die Array-Liste übergeben.
addIndustry	-	void	Fügt neue Industrie an die Liste an.
delIndustry	-	void	Löscht eine Industrie.

getIndustryList	-	ArrayListe <Industry>	Gibt Industrieliste zurück
setIndustryList	ArrayListe <Industry>	void	Übergibt eine Industrieliste.
finishedOrder	int, Goods	void	Überprüft eingehende Güter, verändert ggf. Lagerbestand und ruft die Methode produce auf.
searchForOrder	-	void	Vergleicht alle Gütereingänge mit Güterausgängen der Industrieren und generiert bei Übereinstimmung und positivem Güterausgangsbestand einen neuen Auftrag.
compareIndustryGoods	short, short	boolean	Methode vergleicht Wareneingang der Zielindustrie mit dem Warenausgang der liefernden Industrie.
checkIndustryMaxCapacity	short, int	boolean	Methode prüft ob, ein Auftrag an eine Industrie geliefert werden kann, um mit der Produktion zu beginnen, oder ob der Warenausgang zu voll ist.
calcMaxIncomingAmount	short	double	Methode ermittelt die maximale Warenmenge, die geliefert werden darf, um die Kapazität des Warenlagers nicht zu überschreiten.
returnOrderToIndustry	Order	void	Methode reicht die Menge der Ware, die für einen Auftrag reserviert wurde, an den Warenausgang zurück.

Kommunikationspartner

Komponente $\langle C80 \rangle$

Industry⟨*CL*440⟩

Aufgabe

Die Klasse Industry ist dafür da, dass die Industrien verwaltet werden können. Sie wird benötigt, um die Aufträge an den Industrien koordinieren zu können.

Attribute

Attributname	Datentyp	Beschreibung
id	short	Identifikation
name	String	Name der Industrien.
industryId	short	Identifikationsnummer der Industrien.
node	short	Knotenpunkt.
incoming	Goods	Wareneingang.
outgoing	Goods	Warenausgang.
maxCount	int	Allgemeine Größe des Warenlagers für produzierte Waren.
reservedAmount	double	Reservierte Menge.
quantifier	double	Produktionsmengenmultiplikator.
productionTime	int	Produktionszeit.
remainingTime	int	Verbleibende Zeit während der Produktion.
lock	ReentrantLock	Blockiert Wareneingang während der Produktion eines anderen Produkts.
bounding	Rectangle	Der Flächenbereich, über den man die Industrie auf dem GameDisplay auswählen kann.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Industry	-	-	Konstruktor.
Industry	String, short, Goods, Goods, double, int	-	Konstruktor.
produce	-	void	Umwandlung der Ware. Bearbeitung der Industrien. Warteliste bei Warenlieferung.
resetId	-	void	Setzt die Id zurück auf **-1**.
getId	-	short	Rückgabe der ID einer Industrie.

setId	-	void	Setzen der ID einer Industrie.
getNode	-	short	Rückgabe eines Knotens.
setNode	short	void	Setzen eines Knotens.
getIncoming	-	Goods	Rückgabe der Eingangsgüter einer Industrie.
getOutgoing	-	Goods	Rückgabe der Ausgangsgüter einer Industrie.
getName	-	string	Rückgabe des Industrienamens.
getQuantifier	-	double	Rückgabe des Produktionsmengenmultiplikators.
getMaxAmount	-	Gint	Rückgabe der maximalen Lagerkapazität.
setMaxAmount	int	void	Setzen der maximalen Lagerkapazität.
getReservedAmount	-	double	Rückgabe der reservierten Lagermenge.
setReservedAmount	double	void	Setzen der reservierten Lagermenge.
getProductionTime	-	int	Rückgabe der Produktionszeit.
getBounding	-	Rectangle	Rückgabe der Fläche, über die die Industrie auf dem GameDisplay anwählbar ist.
setBounding	Rectangle	void	Setzen der Fläche, über die die Industrie auf dem GameDisplay anwählbar ist.
getRemainingTime	-	in	Rückgabe der ablaufenden Zeit.

Kommunikationspartner

Komponente $\langle C80 \rangle$

5.11 Implementierung von Komponente $\langle C110 \rangle$: Verwaltung (Server)

Die Komponente Verwaltung (Server) umfasst die Komponenten $\langle C70 \rangle$,$\langle C80 \rangle$, $\langle C90 \rangle$ und $\langle C100 \rangle$. Zusätzlich enthält sie aber auch die weiteren Klassen Order, Goods und Notifyer, die in dem Paket objects untergebracht sind. Da die Sub-Komponenten bereits beschrieben worden sind, wird auf eine erneute Erläuterung an dieser Stelle verzichtet.

5.11.1 Paket-/Klassendiagramm

Für die zusätzlichen Klassen des Pakets objects der Komponente $\langle C110 \rangle$ ergeben sich nachfolgende Diagramme.

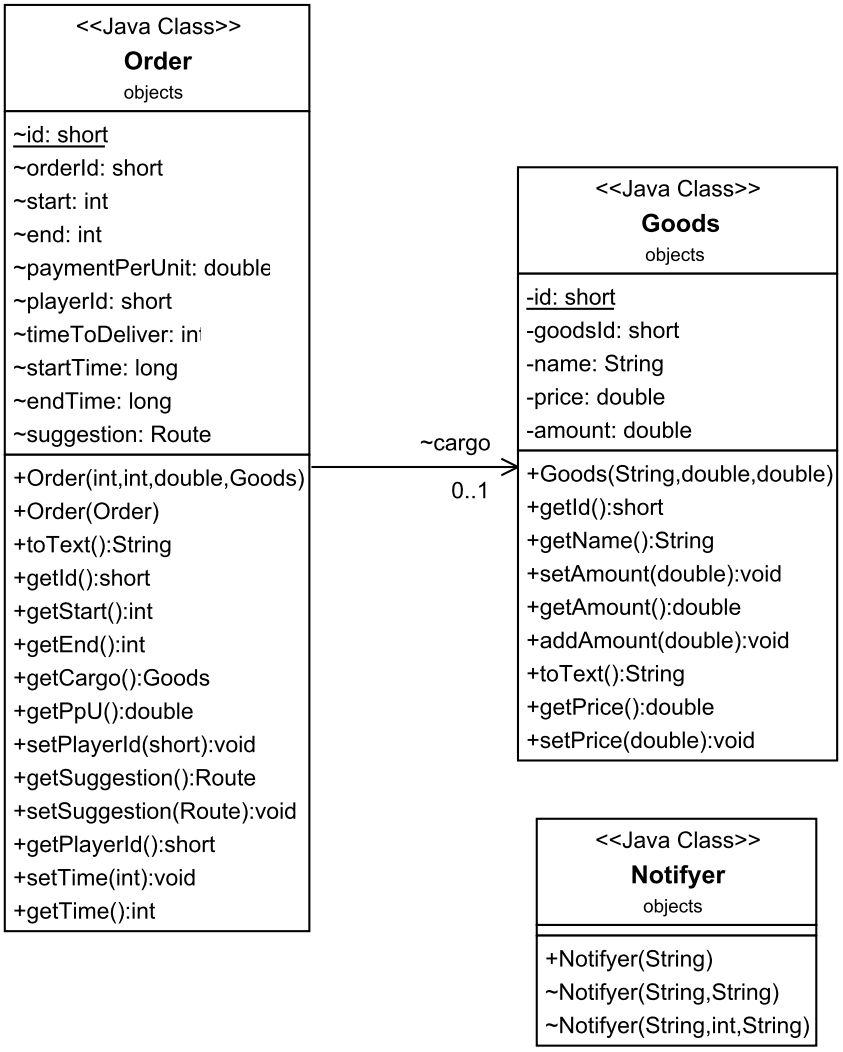

Abbildung 5.17: Klassendiagramm des Pakets objects der Komponente $\langle C110 \rangle$

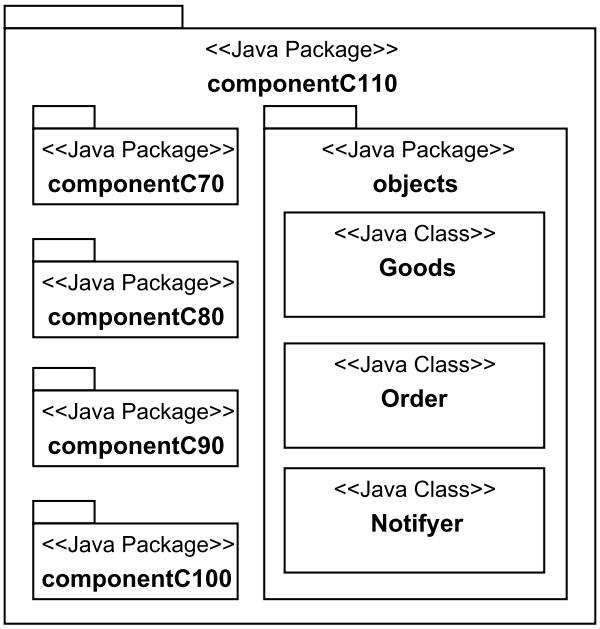

Abbildung 5.18: Paketdiagramm für Komponente $\langle C110 \rangle$

5.11.2 Erläuterung

Order$\langle CL450 \rangle$

Aufgabe

Mit der Klasse Order werden die in den Auktionen des Spiels zu ersteigernden Aufträge dargestellt.

Attribute

Attributname	Datentyp	Beschreibung
id	short	Fortlaufende ID für Aufträge.
cargo	Goods	Die zu transportierende Ware.
orderId	short	ID des Auftrags.
start	int	Startpunkt des Auftrags.
end	int	Zielpunkt des Auftrags.
paymentPerUnit	double	Bezahlung pro Einheit.
timeToDeliver	long	Zeitfenster für den Auftrag.
startTime	long	Zeit bei Auftragsannahme.
endTime	long	Zeit bei Auftragsende.
suggestion	Route	Vom Roboter vorgeschlagene Route.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Order	int, int, double, Goods	-	Konstruktor erstellt einen Auftrag mit allen Informationen.
Order	Order	-	Konstruktor zum Kopieren eines Order-Objekts.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C70 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$, Komponente $\langle C100 \rangle$

Goods$\langle CL460 \rangle$

Aufgabe

Die Objekte der Klasse Goods stellen die Waren des Spiels dar.

Attribute

Attributname	Datentyp	Beschreibung
id	short	Fortlaufende ID für Waren.
goodsId	short	ID der Ware.
name	String	Warenname.
price	double	Warenpreis.
amount	double	Warenmenge.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Goods	String, double, double	-	Konstruktor erstellt ein neues Warenpaket.
addAmount	double	void	Erhöht die Menge der Ware um den Wert des Parameters.

Kommunikationspartner

Komponente $\langle C50 \rangle$, Komponente $\langle C70 \rangle$, Komponente $\langle C80 \rangle$, Komponente $\langle C90 \rangle$, Komponente $\langle C100 \rangle$

Notifyer⟨$CL470$⟩

Aufgabe

Der Notifyer dient dazu, den Benutzer beim Initialisieren des Servers oder auch während des Spiels über Ereignisse (meistens Fehler) zu informieren. Dazu erstellt die Klasse einen kleinen Frame mit einer entsprechenden Textnachricht als Inhalt.

Attribute

Keine.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Notifyer	String	-	Konstruktor mit der anzuzeigenden Nachricht als String-Parameter.
Notifyer	String, String	-	Konstruktor mit Fenstertitel und Nachricht.
Notifyer	String, int, String	-	Konstruktor mit Fenstertitel, Typ und Nachricht.

Kommunikationspartner

Komponente ⟨$C50$⟩, Komponente ⟨$C60$⟩, Komponente ⟨$C70$⟩, Komponente ⟨$C80$⟩, Komponente ⟨$C90$⟩, Komponente ⟨$C100$⟩

5.12 Implementierung von Komponente $\langle C120 \rangle$: Steuerung(Roboter)

Die Steuerung ist dafür zuständig, dass der Roboter weiß, was er an welchem Ort zu tun hat. Dies beinhaltet unter anderem die Nachfrage nach Kantenfreigaben und das korrekte Befahren des Wegenetzes.

5.12.1 Paket-/Klassendiagramm

Für die Komponente $\langle C120 \rangle$ ergeben sich nachfolgende Diagramme.

Das Klassendiagramm befindet sich im Anhang in Abschnitt I, es sollte in DIN A3 ausgedruckt werden.

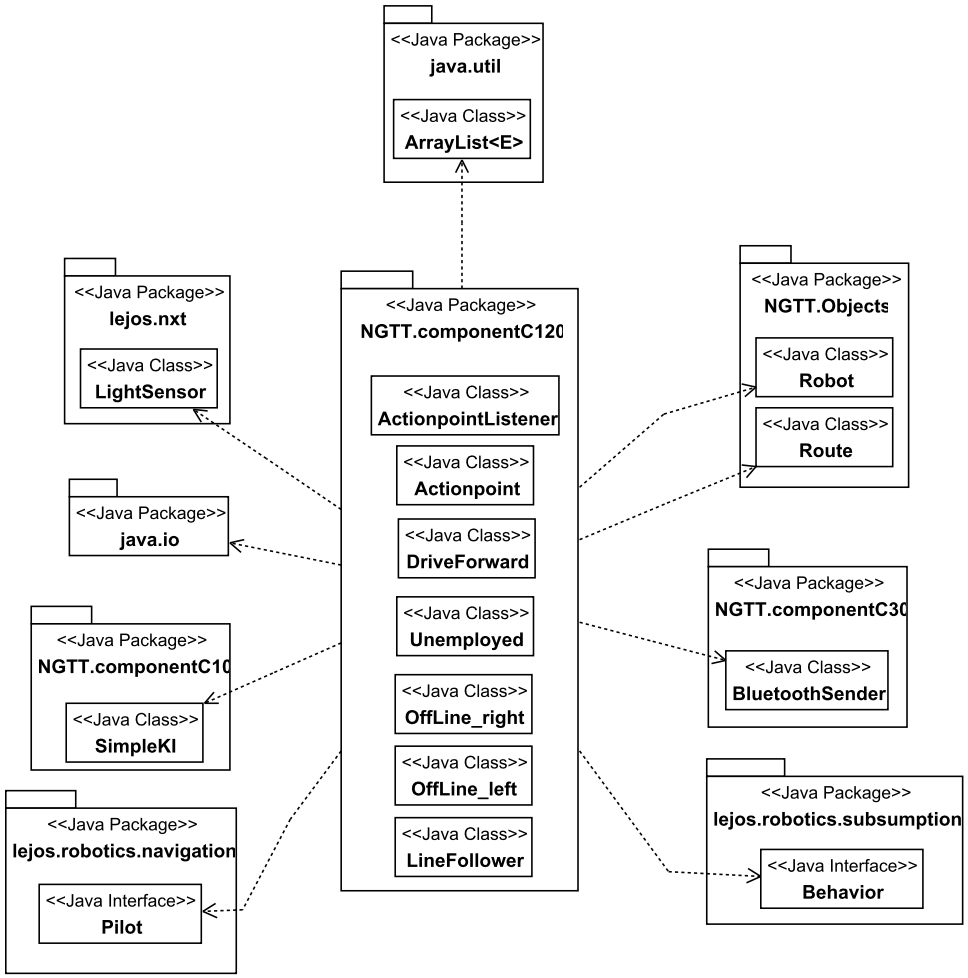

Abbildung 5.19: Paketdiagramm für Komponente $\langle C120 \rangle$

5.12.2 Erläuterung

Die Komponente Steuerung ist wie folgt aufgebaut:
ActionpointListener⟨$CL480$⟩

Aufgabe

Die Klasse ActionpointListener ist dafür da, dass der Roboter weiß, was er an welchem Ort machen muss. Sie speichert wichtige Daten die notwendig sind damit der Roboter an seinen Bestimmungsort ankommt.

Attribute

Attributname	Datentyp	Beschreibung
approveCount	int	Zählt die Anfragen für Streckenfreigaben.
freeEdge	boolean	Information, ob eine angefragte Kante frei ist.
route	ArrayList<Character>	Enthält die Befehle, die benötigt werden, damit der Roboter zum Ziel kommt.
toVisitNodes	ArrayList<Short>	Enthält die zu besuchenden Knoten.
distance	ArrayList<Short>	Enthält die Distanz zum Ziel.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
setCMD	ArrayList<Character>	void	Setzt die Kommandos in *route*.
setRoute	Route	void	Löscht die derzeitige Route und setzt dann die notwendigen Daten.
size	-	int	Liefert die Anzahl der noch zu fahrenden Kanten zurück.
getNextCommand	-	char	Löscht das oberste Kommando und liefert dieses zurück. Existiert kein Kommando, wird eine **0** zurückgeliefert.
getFirstCommand	-	char	Liefert das oberste Kommando zurück, ohne es zu löschen. Existiert kein Kommando, wird eine **0** zurückgeliefert.

isUnemployed	-	boolean	Prüft, ob der Roboter eine Route zum ausführen besitzt
incId	-	void	Wird aufgerufen, um die Actionpoints auf der Strecke zu zählen. Ist der Zähler bei **2** angekommen und incId wird ausgeführt, so wird *id* auf **0** gesetzt.
decDistance	-	void	Löscht den obersten Wert von *distance*, damit die Entfernung zum Ziel bekannt ist.
getDestination	-	short	Liefert den letzten Knoten der derzeitig gespeicherten Route.
getNextNode	-	short	Liefert den nächsten anzufahrenden Knoten, ohne ihn zu löschen.
getAfterNextNode	-	short	Liefert den übernächsten anzufahrenden Knoten, ohne ihn zu löschen.
isEdgeFree	-	boolean	Prüft ob, eine positive Antwort vom Server für eine Anfrage zur Streckenfreigabe vorliegt.
incApproveCount	-	void	Zählt, wie viele Anfragen für eine bestimmte Kante schon getätigt wurden und, ob diese Kante dabei blockiert war.
isBlocked	-	boolean	Prüft, ob der Roboter fahren darf, ist die Kante blockiert und er befindet sich beim 2. Actionpoint (Der Actionpoint kurz vor dem Knoten), so muss er stehen bleiben.

Kommunikationspartner

Komponente $\langle C10 \rangle$, Komponente $\langle C30 \rangle$

Actionpoint⟨$CL490$⟩

Aufgabe

Die Klasse Actionpoint wird jedes Mal aufgerufen, wenn der Roboter an einem Actionpoint **(Ap)** angekommen ist, der Roboter zählt die abgefahrenen Actionpoints. Dies ist dafür da, dass sich der Roboter orientieren kann. Außerdem werden Methoden aufgerufen, die nur an bestimmten Actionpoints ausgeführt werden müssen:

0. Ap: Der Roboter befindet sich an einem Knoten und muss dort eine Operation ausführen (Roboter ausrichten, beladen und entladen).

1. Ap: Der Roboter befindet sich hinter einem Knoten, er muss dann den Knoten wieder freigeben, damit spätere Roboter wissen, dass der Knoten nicht mehr belegt ist. Zudem erfragt der Roboter, ob der nächste Streckenabschnitt frei ist.

2. Ap: Der Roboter befindet sich nun kurz vor dem Knoten, er darf diesen Knoten nur befahren, wenn er eine Freigabe für den zu befahrenden Streckenabschnitt erhalten hat.

Attribute

Attributname	Datentyp	Beschreibung
apL	ActionpointListener	Enthält wichtige Informationen über die Strecke.
sender	BluetoothSender	Lässt den Roboter mit dem Server kommunizieren.
ki	SimpleKI	Legt fest, wie der Roboter in bestimmten Situationen handeln muss.
l_right	LightSensor	Der rechte initialisierte Lichtsensor.
l_left	LightSensor	Der linke initialisierte Lichtsensor.
r_glv	int	Kalibrierungswert für den rechten Lichtsensor.
l_glv	int	Kalibrierungswert für den linken Lichtsensor.
pilot	Pilot	Hilft bei der Motoransteuerung.
robot	Robot	Enthält Informationen über den Roboter.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Actionpoint	LightSensor, LightSensor, ActionpointListener, int, int, Robot, SimpleKI, BluetoothSender		Der Konstruktor setzt notwendige Informationen.
action	-	void	Führt seine Aufgabe je nach Actionpoint aus.
suppress	-	void	Setzt den *suppress*-Wert wieder auf **false**.
takeControl	-	boolean	Die Methode action wird nur ausgeführt, wenn die beiden Sensoren einen dunklen Untergrund messen.

Kommunikationspartner

Komponente $\langle C10 \rangle$, Komponente $\langle C30 \rangle$

DriveForward$\langle CL500 \rangle$

Aufgabe

Ist dafür da, dass der Roboter geradeaus fährt.

Attribute

Attributname	Datentyp	Beschreibung
l_right	LightSensor	Der rechte initialisierte Lichtsensor.
l_left	LightSensor	Der linke initialisierte Lichtsensor.
r_glv	int	Kalibrierungswert für den rechten Lichtsensor.
l_glv	int	Kalibrierungswert für den linken Lichtsensor.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
DriveForward	LightSensor, LightSensor, int, int	Konstruktor	Setzt notwendige Informationen.
action	-	void	Die Räder des Roboters fahren mit gleicher Geschwindigkeit.
suppress	-	void	Setzt den *suppress*-Wert wieder auf **false**.
takeControl	-	boolean	Der Roboter fährt nur geradeaus, wenn keiner der Sensoren einen bestimmten Wert überschreitet.

Kommunikationspartner

Keiner.

OffLine_left⟨$CL510$⟩

Aufgabe

Korrigiert den Roboter, falls er links von dem Wegenetz abkommt.

Attribute

Attributname	Datentyp	Beschreibung
apL	ActionpointListener	Enthält wichtige Informationen über die Strecke.
l_right	LightSensor	Der rechte initialisierte Lichtsensor.
l_left	LightSensor	Der linke initialisierte Lichtsensor.
r_glv	int	Kalibrierungswert für den rechten Lichtsensor.
l_glv	int	Kalibrierungswert für den linken Lichtsensor.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
OffLine_left	LightSensor, LightSensor, int, int, ActionpointListener	Konstruktor	Der Konstruktor setzt notwendige Informationen.
action	-	void	Lässt das rechte Rad langsamer werden, damit der Roboter nach rechts fährt.
suppress	-	void	Setzt den *suppress*-Wert wieder auf **false**.
takeControl	-	boolean	Der rechte Sensor erkennt eine dunkle Fläche.

Kommunikationspartner

Keiner

OffLine_right⟨$CL520$⟩

Aufgabe

Korrigiert den Roboter, falls er rechts von dem Wegenetz abkommt.

Attribute

Attributname	Datentyp	Beschreibung
apL	ActionpointListener	Enthält wichtige Informationen über die Strecke.
l_right	LightSensor	Der rechte initialisierte Lichtsensor.
l_left	LightSensor	Der linke initialisierte Lichtsensor.
r_glv	int	Kalibrierungswert für den rechten Lichtsensor.
l_glv	int	Kalibrierungswert für den linken Lichtsensor.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
OffLine_right	LightSensor, LightSensor, int, int, ActionpointListener	Konstruktor	Der Konstruktor setzt notwendige Informationen.
action	-	void	Lässt das linke Rad langsamer werden, damit der Roboter nach links fährt .
suppress	-	void	Setzt den *suppress*-Wert wieder auf **false**.
takeControl	-	boolean	Der linke Sensor erkennt eine dunkle Fläche.

Kommunikationspartner

Kein

Unemployed⟨$CL530$⟩

Aufgabe

Stoppt den Roboter falls er keine Route besitzt.

Attribute

Attributname	Datentyp	Beschreibung
apL	ActionpointListener	Enthält wichtige Informationen über die Strecke.
ki	SimpleKI	Legt fest, wie der Roboter in bestimmten Situationen handeln muss.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
Unemployed	ActionpointListener, SimpleKI	Konstruktor	Der Konstruktor setzt notwendige Informationen.
action	-	void	Hält den Roboter solange an bis er eine neue Route hat.
suppress	-	void	Setzt den *suppress*-Wert wieder auf **false**.
takeControl	-	boolean	Liefert ein **true**, wenn der Roboter derzeit keine Route besitzt.

Kommunikationspartner

Komponente $\langle C10 \rangle$, Komponente $\langle C30 \rangle$

LineFollower$\langle CL540 \rangle$

Aufgabe

Initialisiert die Sensoren und startet den Arbitrator mit den Behaviors: DriveForward, OffLine_right, OffLine_left, Actionpoint und Unemployed.

Attribute

Attributname	Datentyp	Beschreibung
apL	ActionpointListener	Enthält wichtige Informationen über die Strecke.
sender	BluetoothSender	Lässt den Roboter mit dem Server kommunizieren.
ki	SimpleKI	Legt fest wie er in bestimmten Situationen handeln muss.
l_right	LightSensor	Der rechte initialisierte Lichtsensor.
l_left	LightSensor	Der linke initialisierte Lichtsensor.
r_glv	int	Kalibrierungswert für den rechten Lichtsensor.
l_glv	int	Kalibrierungswert für den linken Lichtsensor.
pilot	Pilot	Hilft bei der Motoransteuerung.
robot	Robot	Enthält Informationen über den Roboter.
suppress	boolean	Nötig für die wiederholte Ausführung einer Aktion.

Operationen

Name	Parameter	Rückgabetyp	Beschreibung
LineFollower	ActionpointListener, Robot, SimpleKI, BluetoothSender		Der Konstruktor setzt notwendige Informationen und initialisiert die Sensoren.
run	-	void	Initialisiert die Behavior und startet den Arbitrator.

Kommunikationspartner

Komponente $\langle C10 \rangle$, Komponente $\langle C30 \rangle$

6 Datenmodell

Als persistent gespeicherte Daten werden in diesem Abschnitt die Struktur des Wegenetzes und die Beschaffenheit von Waren sowie die Standorte der Industrien auf der Karte erläutert. Diese Daten werden alle in Form von XML-Dateien gespeichert.

6.1 Karte

Im Folgenden wird die Struktur der Karte detailliert erläutert.

6.1.1 Diagramm

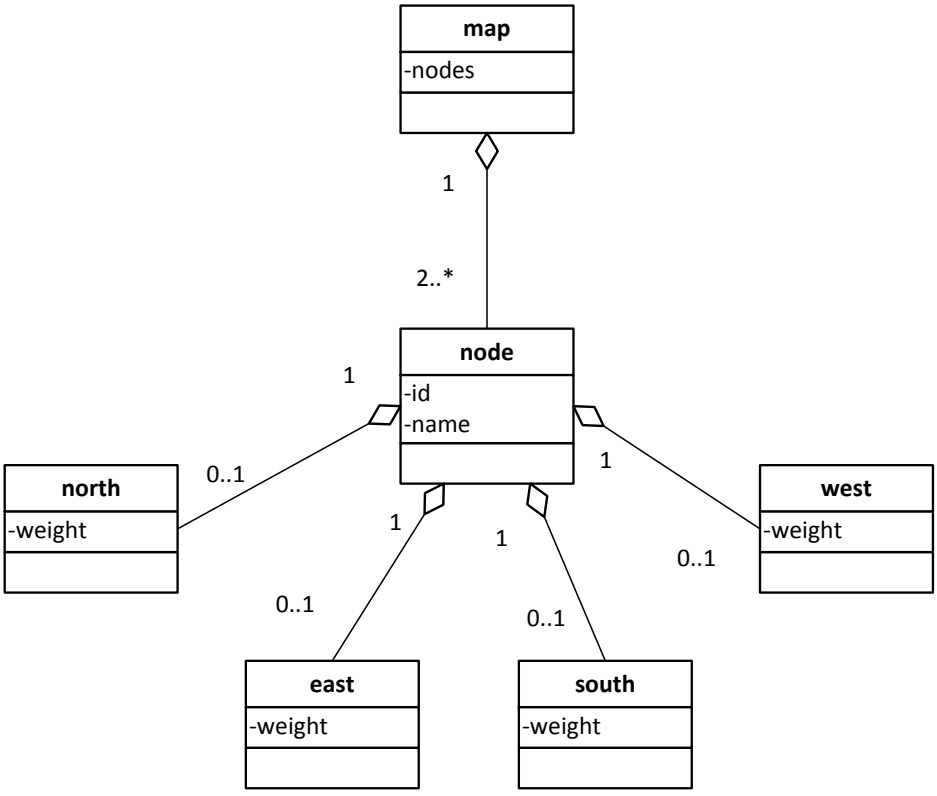

Abbildung 6.1: Struktur der Kartendaten

6.1.2 Erläuterung

Die Karte ist das zentrale Element des Projekts, deswegen ist ihre korrekte und effiziente Speicherung äußerst wichtig. Sie besteht im Wesentlichen aus Knoten und ihren in entsprechender Entfernung anliegenden Nachbarn.

map $\langle E10 \rangle$

Beziehung	**Kardinalität**
Es gibt genau ein Kartenobjekt pro Datei	1

node $\langle E20 \rangle$

Beziehung	**Kardinalität**
Ist Teil der Karte $\langle E10 \rangle$	2..*

north $\langle E30 \rangle$

Beziehung	**Kardinalität**
Beschreibt den Knoten im Norden des Knotens $\langle E20 \rangle$	0..1

east $\langle E40 \rangle$

Beziehung	**Kardinalität**
Beschreibt den Knoten im Osten des Knotens $\langle E20 \rangle$	0..1

south $\langle E50 \rangle$

Beziehung	**Kardinalität**
Beschreibt den Knoten im Süden des Knotens $\langle E20 \rangle$	0..1

west $\langle E60 \rangle$

Beziehung	**Kardinalität**
Beschreibt den Knoten im Westen des Knotens $\langle E20 \rangle$	0..1

6.2 Industrie

Im Folgenden wird die Struktur der XML-Datei für die Industriedaten detailliert dargestellt.

6.2.1 Diagramm

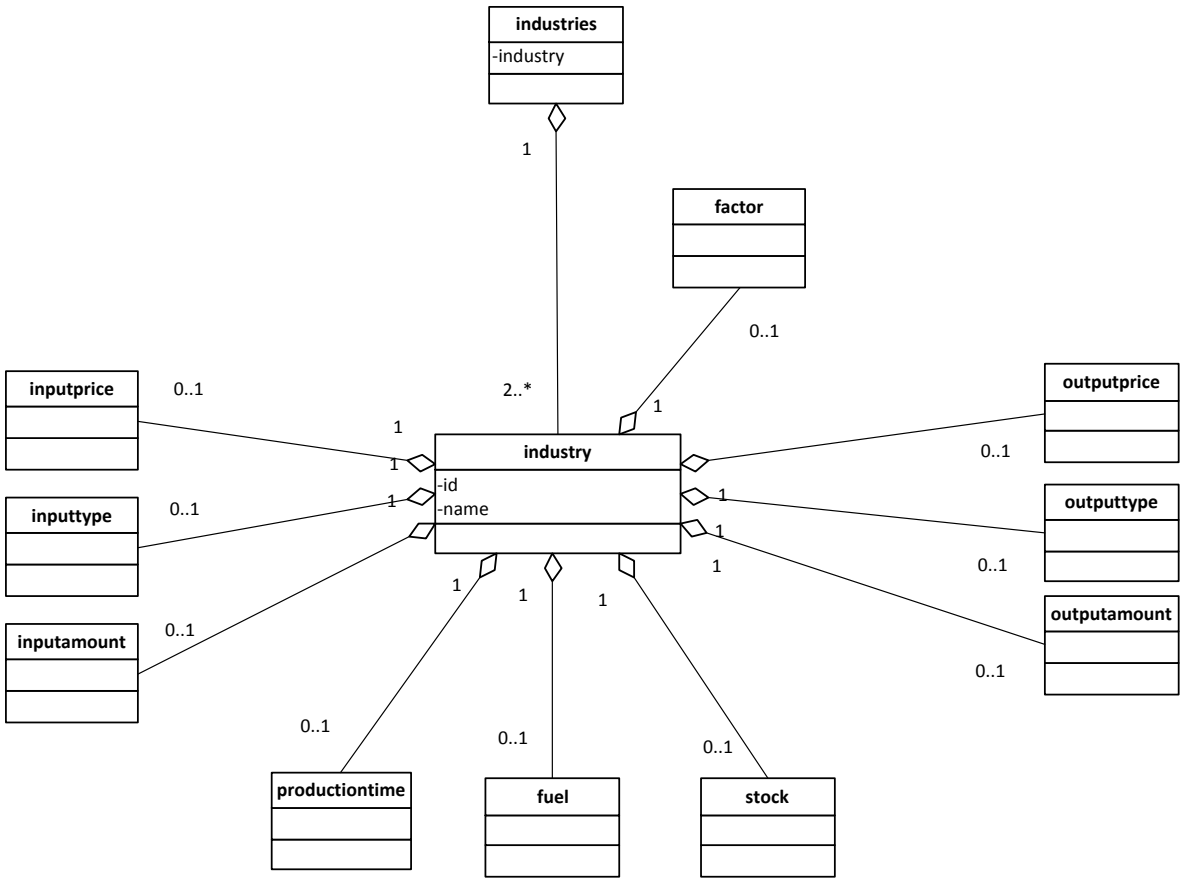

Abbildung 6.2: Struktur der Industriedaten

6.2.2 Erläuterung

Jede Industrie besitzt die Attribute id, name, fuel, position. Das Attribut id dient zur Identifikation der Industrie, name beschreibt die Art der Industrie, fuel gibt die Menge an Kraftstoff an, die der Roboter bei erfolgreicher Auftragsdurchführung erhält und position gibt den aktuellen Standort im Wegenetz an.
Zudem gibt es noch inputtype, -amount, -price sowie outputtype, -amount, -price und factor. Sie beschreiben die Art des Inputs, die Menge des Inputs, die benötigt wird, um eine Einheit Output zu produzieren, sowie den Preis für die Menge des Inputs. Der Faktor gibt an, welches Verhältnis zwischen Input- und Outputmenge besteht.
Der Zeitraum des Produktionsprozesses wird in productiontime gespeichert. Die Kapazität des Lagers wird unter stock abgelegt.

industries $\langle E70 \rangle$

Beziehung	**Kardinalität**
Es gibt genau ein Industries-Objekt pro Datei.	1

industry $\langle E80 \rangle$

Beziehung	**Kardinalität**
Ist Teil von industries $\langle E70 \rangle$.	0..1

stock $\langle E90 \rangle$

Beziehung	**Kardinalität**
Ist die Lagerkapazität von $\langle E80 \rangle$.	0..1

productiontime $\langle E100 \rangle$

Beziehung	**Kardinalität**
Ist die Produktionszeit von $\langle E80 \rangle$, die für ein Produkt benötigt wird.	0..1

inputtype $\langle E110 \rangle$

Beziehung	**Kardinalität**
Gibt an welche Waren benötigt werden zum Herstellen eines Produktes.	0..1

inputamount ⟨*E*120⟩

Beziehung	Kardinalität
Ist die Menge die benötigt wird um ein ein Produkt herzustellen.	0..1

inputprice ⟨*E*130⟩

Beziehung	Kardinalität
Ist der Startpreis für den Wareneingang.	0..1

outputtype ⟨*E*140⟩

Beziehung	Kardinalität
Gibt an was für ein Produkt hergestellt wird.	0..1

outputamount ⟨*E*150⟩

Beziehung	Kardinalität
Gibt an wie viel aus den erhaltenen Rohstoffen produziert wird.	0..1

outputprice ⟨*E*160⟩

Beziehung	Kardinalität
Ist der Startpreis für den Warenausgang.	0..1

fuel ⟨*E*170⟩

Beziehung	Kardinalität
Ist die Menge an Kraftstoff, die die Industrie vergibt.	0..1

factor ⟨*E*180⟩

Beziehung	Kardinalität
Ist das Verhältnis von Output- und Inputmenge.	0..1

6.3 Goods

Im Folgenden wird die Struktur der XML-Datei für die Güterdaten detailliert dargestellt.

6.3.1 Diagramm

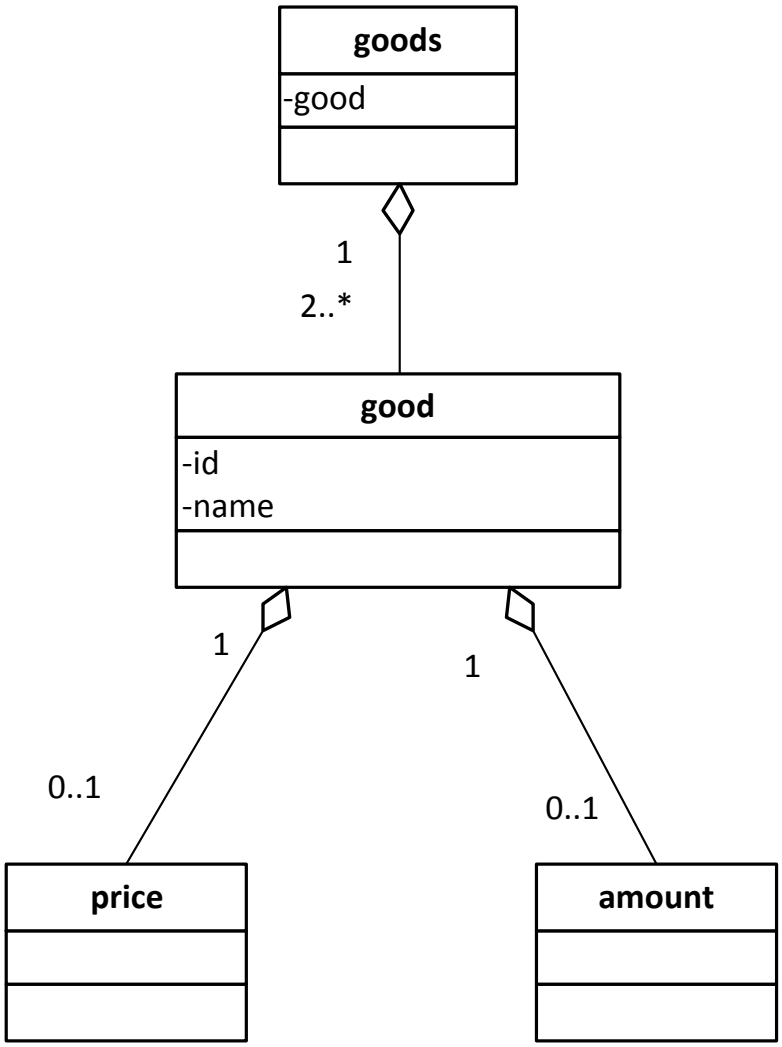

Abbildung 6.3: Struktur der Güterdaten

6.3.2 Erläuterung

Zu jedem Gut gibt es die Attribute id, name, amount und price.
Das Attribut id dient zur Identifikation des Gutes, name gibt den Name des Gutes und gleichzeitig den Typ mit an, amount gibt die Menge und price den Preis für die Menge des Gutes.

goods $\langle E190 \rangle$

Beziehung	**Kardinalität**
Es gibt genau ein goods-Objekt pro Datei	1

good $\langle E200 \rangle$

Beziehung	**Kardinalität**
Ist Teil von $\langle E190 \rangle$	0..1

price $\langle E210 \rangle$

Beziehung	**Kardinalität**
Ist der Preis von $\langle E200 \rangle$	0..1

amount $\langle E220 \rangle$

Beziehung	**Kardinalität**
Ist die Menge von $\langle E200 \rangle$	0..1

7 Serverkonfiguration

Für die Realisierung dieses Software-Produktes ist kein externer Server notwendig. Als Server dient ein selbst geschriebenes Programm, welches keine spezielle Konfiguration benötigt.

8 Erfüllung der Kriterien

In diesem Abschnitt wird beschrieben, wie die einzelnen Kriterien des Pflichtenheftes erfüllt werden und worauf geachtet wird. Es wird zunächst auf die Musskriterien, dann auf die Soll- und Kann-Kriterien und zuletzt auf die Abgrenzungskriterien eingegangen.

8.1 Musskriterien

Die folgenden Kriterien sind unverzichtbar und müssen durch das Produkt erfüllt werden:

$\langle RM1 \rangle$ „*Der Roboter erkennt den Straßenverlauf in Form von schwarzen Linien und folgt ihnen.*“
Dieses Kriterium wird folgendermaßen umgesetzt: Der Roboter besitzt zwei Lichtsensoren, die die Helligkeit messen. Der Straßenverlauf wird durch dunkle Linie dargestellt, die der Roboter von dem hellen Untergrund unterscheiden kann. Der weiße Untergrund wird als Helligkeitsstufe „0“ und der dunkle Untergrund misst einen Wert „>0“. Zur Markierung von Straßenenden und Kreuzungen wurden ebenfalls schwarze quer zur Straßenrichtung verlaufende Streifen geklebt. Kreuzungen oder Industriestandorte erkennt der Roboter, indem beide Sensoren einen dunklen Untergrund erkennen.

$\langle RM2 \rangle$ „*Der Roboter führt angenommene Aufträge selbständig durch.*“
Wenn der Server einen Auftrag zu vergeben hat, wird dieser für alle im Spiel vorhandenen Roboter freigegeben. Der Server startet eine Auktion an der alle Roboter, die genug Ressourcen haben (z.B. genügend Treibstoff) teilnehmen können. Mit Hilfe einer Methode, die die Gebote anfordert, werden alle getätigten Gebote gesammelt und ausgewertet. Bevor der Roboter ein Gebot abgibt, berechnet er selbständig mit einer Methode, ob er den Auftrag annehmen kann und welches das bestmögliche Gebot ist. Erst wenn er diese Berechnungen durchgeführt hat, nimmt er an der Auktion teil und sendet dem Server sein Gebot zu. Nach einem Zeitfenster wird die Auktion geschlossen und der Gewinner ermittelt. Der Server sendet ihm den Auftrag zu. Der Roboter führt den Auftrag aus.

$\langle RM3 \rangle$ „*Der Roboter kann zwischen zwei Punkten den Weg selbstständig finden.*“
Dieses Kriterium wird einer Methode, die den Weg zwischen zwei Knotenpunkten mit der Hilfe von einem implementierten Algorithmus findet, umgesetzt.

⟨*RM*4⟩ „*Der Roboter muss Aufträge ersteigern können.*“
Über eine Bluetooth-Verbindung kommuniziert der Roboter mit dem Server, der ihm mit Hilfe der Auktionsverwaltung die Auktion freigibt und ein Gebot anfordert. Dieser teilt dem Server nach seiner Berechnung, ebenfalls über die Bluetooth-Verbindung, seine Antwort mit. Das Gebot wird in einer „Liste“ gespeichert und ausgewertet. Erhält der Roboter den Auftrag und hat somit die Auktion gewonnen, speichert er den Auftrag und teilt den Erhalt dem Server mit.

⟨*RM*5⟩ „*Der Roboter muss mit dem Server kommunizieren können.*“
Die Kommunikation zwischen Server und Roboter wird via Bluetooth hergestellt. Beide Komponenten sind über ihre Kommunikationsmodule miteinander vernetzt.

⟨*RM*6⟩ „*Der Roboter hat eine maximale Ladekapazität.*“
Die maximale Ladekapazität wird durch den Administrator oder einem Standardwert festgelegt. Dieser wird während des Spiels nicht verändert. Die KI des Roboters überprüft bei jeder Gebotsberechnung, ob die Größe, der bereits angenommen Aufträge, addiert mit der Größe, des zu ersteigernden Auftrags, mehr als die maximale Ladekapazität ergibt. Nur wenn der Wert niedriger ist, kann der Roboter den neuen Auftrag annehmen und ersteigern. Wurde ein Gebot abgegeben, so überprüft der Server die Korrektheit des Transportvolumens, damit möglich Manipulationen ausgeschlossen sind.

⟨*RM*7⟩ „*Der Roboter muss Deadlock-Situationen erkennen und vermeiden können.*“
Diese Funktion wird wie folgt umgesetzt: ist der Weg blockiert, so versucht er eine Alternativroute zu berechnen, geht dies nicht und es vergeht eine gewisse Zeit, bemerkt der Roboter, dass es sich hierbei um einen Konflikt handeln muss. Er versucht dann auf einen anliegenden Weg auszuweichen und somit den anderen Roboter den Weg freizumachen. Ist ihm das nicht möglich, so muss er zurückfahren wenn er sich nicht an einem Endknoten befindet.
Eine Vermeidung der Deadlocks wird durch die stetige Kommunikation zwischen Server und allen auf dem Spielfeld vorhandenen Robotern umgesetzt.

⟨*RM*8⟩ „*Der Server muss das Wegenetz verwalten.*“
Der Graph ist in einer .xml-Datei gespeichert und wird vom Server importiert. Mit Hilfe von Methoden werden alle Graphendetails ausgelesen und Attributen zugeordnet.

⟨*RM*9⟩ „*Der Server muss Aufträge verwalten.*“
Die Verwaltungskomponente des Servers vergibt Aufträge, die sich entweder in einer Warteliste befinden oder vom Benutzer vorgegeben werden. Via Bluetooth bzw. TCP-Verbindung werden die Spielteilnehmer über eine Auktion informiert. Aufträge werden in einer Liste gespeichert, die die jeweiligen Attribute (Auftragsnummer, Größe, Art usw.) enthält.

⟨*RM*10⟩ „*Der Server muss den Roboter hinsichtlich der Kartendaten initialisieren.*“
Der Roboter wird zunächst über eine Bluetooth-Verbindung registriert. Die Daten zwischen den jeweiligen Kommunikationskomponenten werden als Stream versendet. Bei der Initialisierung des Roboters wird dem Roboter das Wegenetz zugesendet, der dieses speichert und seine Teilnahmebestätigung mitteilt.

⟨*RM*11⟩ „*Der Server muss die Preisbildung verwalten.*“
Die Serververwaltung bildet den Preis je nach Auftragslage. Dabei schaut er, wie hoch die Warenverfügbarkeit des Produktes ist. Dafür wird geschaut, auf welchem Verfügbarkeitslevel, sich die Ware befindet. Ist nur eine gewisse Menge verfügbar, gilt sie als knapp. Somit wird der Preis erhöht. Der neue Preis wird gespeichert und freigegeben.

⟨*RM*12⟩ „*Der Server verwaltet die Kommunikation zwischen Benutzeroberfläche und Roboter.*“
Während des Spiels verwaltet der Server die durch ausstehende oder geleistete Aufträge anfallenden Daten und aktualisiert die sich daraus ergebenden Statistiken (Preisbildung, Gewinne). Diese Statistiken werden später sowohl in der Spieler-GUI ⟨*F*110⟩, als auch in der Benutzer-GUI angezeigt. Die Verwaltung zwischen der Benutzeroberfläche und den Robotern wird via Bluetooth bzw. TCP/IP-Verbindung über das Kommunikationsmodul des Servers ermöglicht.

⟨*RM*13⟩ „*Der Server muss mit mehreren Robotern gleichzeitig kommunizieren.*“
Für die gleichzeitige Kommunikation mit allen teilnehmenden Robotern, stellt der Server zu Beginn mit jedem einzelnen Roboter eine Verbindung her, die bis zum Ende des Spieles aufrecht gehalten wird.

⟨*RM*14⟩ „*Der Server muss den Standort der Roboter kennen.*“
Durch die stetige Kommunikation mit dem Server, wird an jedem Abschnitte einer Kante die Position gesendet, dadurch ist die Verfolgung des Roboters möglich.

⟨*RM*15⟩ „*Der Server muss Auftragsauktionen anbieten.*“
Über die Kommunikationsmodule wird den Robotern bzw. Spieler mitgeteilt, ob eine neue Auftragsauktion freigegeben wurde. Die Verwaltungskomponenten des Servers steuert die Vergabe der Aufträge. Bei jedem Start einer Auktion werden alle mitspielenden Roboter informiert. Ist ein neuer Auftrag vorhanden, werden die Daten vom Server bereitgestellt und an das Kommunikationsmodul der Roboter versendet, damit sie an der Auktion teilnehmen können. Darauf folgt eine Bestätigung oder Ablehnung der Roboter, die an den Server gesendet wird. Der Server entfernt daraufhin die Kennung der Roboter, für diesen Auftrag. Wenn alle Roboter eine Antwort an den Server gesendet haben, werden die Gebote verglichen und der Roboter mit dem niedrigsten Gebot wird sein Gewinn mitgeteilt.

⟨*RM*16⟩ „*Der Server muss Geschäftsstatistiken für die Roboter führen.*“
Während des Spiels sendet der Roboter, bei Änderung, seine Daten an den Server, aus diesen Daten wird eine Geschäftsstatistik berechnet. Über die Kommunikationsmodule findet ein stetiger Datenabgleich statt.

⟨*RM*17⟩ „*Der Server muss es ermöglichen, dass Ressourcen und Produkte ergänzt werden können.*“
Das Ergänzen von Ressourcen und Produkten wird durch die Eingabemöglichkeit der Benutzeroberfläche ermöglicht. Der Benutzer erstellt über die GUI des Servers einen Auftrag, der an die Spielkoordination übermittelt wird. Diese leitet den Auftrag an die Auktionsverwaltung weiter, die den Auftrag in einer Auktion anbietet. Industrien können ebenfalls über die Benutzeroberfläche ergänzt werden. Dies ist allerdings nur während der Spielpause möglich. Wird eine neue Industrie angelegt, wird sie in die vorhandene Liste eingefügt.

⟨*RM*18⟩ „*Die Spieleroberfläche muss das Wegenetz visualisieren.*“
Der Server übermittelt die gespeicherten Wegenetzdaten an das Spielerinterface. Zudem übermittelt er die reservierten Abschnitte des Wegenetzes mit den Kennungen der Roboter. Die Darstellung erfolgt in der Form einer 2D-Ansicht.

⟨*RM*19⟩ „*Die Spieleroberfläche muss eine Übersicht über die Aufträge bieten.*“
Dies wird folgendermaßen umgesetzt. Mit Hilfe der Spieler GUI wird ein Fenster angezeigt, welches alle notwendigen Daten, z.B. das Wegenetz und die Auftragsinformationen des eigenen Roboters, beinhaltet. Der Spieler kann aussuchen, welche Aufträge angezeigt werden (zurzeit verfügbare, schon abgearbeitete oder zurzeit bearbeitete Aufträge). Anhand eines Filters, der gesetzt wird, werden die gewünschten Aufträge aufgelistet. Klickt man auf den Auftragsdetails-Button, wird eine Anfrage an den Server gestellt, der die aktuellen Daten heraussucht und an die Spieler GUI übergibt.

⟨*RM*20⟩ „*Die Spieleroberfläche muss eine Statistik über die Roboter anzeigen.*“
Der Server sammelt Spieldaten jedes mitspielenden Roboters und speichert diese zwischen. Anhand dieser Daten werden durchschnittliche Gewinne und der gesamte Gewinn errechnet und der GUI zur Verfügung gestellt. Die berechneten Werte werden mit der Kennung des Roboters auf der Spieleroberfläche angezeigt.

⟨*RM*21⟩ „*Die Spieleroberfläche muss die Teilnahme an Auftragsauktionen ermöglichen.*“
Durch eine Eingabemaske auf der Spieleroberfläche ist es möglich, an einer Auktion teilzunehmen. Dazu wird der gewünschte Auftrag ausgewählt. Bei einem Gebot wird die Auftragsnummer mit der Kennung des Roboters verknüpft und an den Server gesendet. Dieser registriert das Gebot des Roboters/Spielers und beendet die Auktion, wenn er von allen mitspielenden Robotern eine Antwort erhalten hat oder es zu einen Timeout gekommen ist und teilt dann dem Sieger den Gewinn der Auktion mit.

⟨*RM*22⟩ „*Die Visualisierung des Wegenetz muss den Standort und Status von Robotern und Industriestandorten enthalten.*“
Der Server stellt die Status des Roboters und der Industriestandorte bereit. Diese werden auf der Oberfläche dargestellt. Die Aktualität der Daten wird durch die stetige Kommunikation zwischen Server und Roboter gewährleistet.

⟨*RM*23⟩ „*Die Karte des Wegenetzes kann leicht vom Benutzer an die örtlichen Bedingungen (aufgeklebtes Straßennetz) angepasst werden.*“
Dies wird durch eine einfache Oberfläche gewährleistet, die die Möglichkeit bietet durch einfache Handhabung Knoten und Kanten hinzuzufügen oder diese zu entfernen. Dies ist nur vor dem Spielbeginn möglich, um eine Inkonsistenz des Spiel zu vermeiden.

⟨*RM*24⟩ „*Die Spieloberfläche muss Details (vorhandene Waren zum Transport, gelagerte Waren zur Weiterverarbeitung) einzelner Industrien anzeigen.*“
Dieses Kriterium wird mit der Hilfe des „Industriedetails anzeigen“-Button gewährleistet. Dieser führt eine Anfrage aus, sobald er gedrückt wurde, und listet alle Details der Industrien auf.

⟨*RM*25⟩ „*Die Anordnung und der Produktionszyklus von Industrien soll leicht änderbar bzw.anpassbar sein.*“
Die Anpassung der Industrien wird durch den Administrator durchgeführt. Über die Administratoroberfläche kann ein Industriestandort mit all seinen Details angeschaut werden und die Attribute abgeändert und an den Server gesendet werden. Dies ist nur vor dem Spielbeginn möglich.

8.2 Sollkriterien

Die Erfüllung folgender Kriterien für das abzugebende Produkt wird angestrebt:

⟨*RS*1⟩ „*Der Roboter verwaltet seinen (virtuellen) Benzinverbrauch.*“
Wird mit Hilfe eines Counters, der die Zählmenge dekrementiert, umgesetzt. Zum Start des Spiels wird die Treibstoffmenge aller Roboter auf den gleichen Wert gesetzt. Im Laufe des Spiels zählt der Counter zeitabhängig nach unten. Bei jedem erfolgreich abgeschlossenen Auftrag wird die Treibstoffmenge wieder erhöht. Ist der Counter bei Null angekommen, hat der Roboter/das Team verloren.

⟨*RS*2⟩ „*Der Roboter zeigt eine Statusmeldung mit Transportinformationen an.*“
Über die Displayausgabe des Roboters werden die Statusmeldungen angezeigt. Hier werden der aktuelle Gewinn, Ladekapazität, Anzahl der angenommenen Aufträge, das aktuelle Ziel sowie Treibstoffmenge angezeigt.

⟨*RS*3⟩ „*Der Roboter kann eigenständig lukrative Aufträge wählen.*“
Bei der Auftragsannahme entscheidet der Roboter je nach Ausgangslage, ob der Auftrag mehr oder weniger lukrativ ist. Dies wird folgendermaßen umgesetzt. Zur Berechnung des möglichen Gewinns bei erfolgreicher Auftragsdurchführung werden mehrere Faktoren betrachtet. Dabei spielen die Entfernung des Roboters vom aktuellen Standort zum Ziel, die Auftragsdauer sowie der Preis des Auftrags eine Rolle. Befindet sich ein Roboter in der Nähe des Auftrags und ist der Produktionszyklus kurz und dadurch dieser Auftrag schnell durchführbar ist, wird dieser Auftrag als sehr lukrativ betrachtet.

⟨*RS*4⟩ „*Der Server soll eine Administrator-Oberfläche haben.*“
Die Administratoroberfläche wird mit Java erzeugt und beim Starten des Spiels ausgewählt. Sie enthält wie die Spieleroberfläche die Statistiken und eine Übersicht der Roboter und Industrien. Im Gegensatz zur Spieleroberfläche kann der Administrator neue Roboter hinzufügen, die vom Server initialisiert werden. Er alleine kann zudem das Spiel starten, indem er den „Spiel-Starten“-Button drückt.

⟨*RS*5⟩ „*Der Server soll Benutzeraufträge annehmen können.*“
Über die Administratoroberfläche legt der Benutzer neue Aufträge an, die an den Server geschickt werden. Dieser überprüft die Richtigkeit der Daten und sendet eine Bestätigung. Sind die Daten korrekt fügt der Server den Auftrag an das Ende der Warteliste ein. Die Warteliste wird aktualisiert und an die Spieleroberfläche gesendet.

⟨*RS*6⟩ „*Der Spieler soll die vom Roboter vorgeschlagene Route verändern können.*“
Diese Kriterium wird über die Spieleroberfläche ermöglicht. Der Roboter sendet die Daten der vorgeschlagenen Route an den Server, der die Daten weiter an die Spieleroberfläche leitet. Die Route wird visualisiert dargestellt. Will der Benutzer die Route verändern, klickt er auf den Button „Bearbeiten-der-Route“. Die Route wird zur Manipulation freigegeben. Nun kann der Benutzer durch Klicken auf verschiedene Abschnitte die Route verändern. Hat er die endgültige Route gefunden, speichert er sie über den „Speichern“-Button. Die Route wird an den Server gesendet, der sie auf Konsistenz überprüft und danach an den Roboter weiterleitet. Der Roboter überschreibt die Route mit der neuen und führt den Auftrag aus.

8.3 Kannkriterien

Die Erfüllung folgender Kriterien für das abzugebende Produkt wird angestrebt:
⟨*RC*1⟩ „*Der Roboter soll die Verderblichkeit der Waren berücksichtigen.*“
Dies würde bei der Routenplanung in Betracht gezogen werden. Das bedeutet das verderbliche Waren eine höhere Priorität hätten.

⟨*RC*2⟩ „*Der Server kann eine Kalibrierung der Roboter vornehmen.*“
Dies würde durch ein Button in der Benutzer-GUI realisiert werden, welchen den Roboter während des Spiels zum Kalibrieren zwingt.

⟨*RC*3⟩ „*Die Spieleroberfläche muss von einem anderen PC aus aufrufbar sein*“
Dies würde über TCP/IP realisiert werden. Dann könnte die Spieler-GUI sich über die IP des Servers in das Spiel einklinken und bei einer neuen Runde dran teilnehmen.

8.4 Abgrenzungskriterien

Folgende Funktionalitäten werden nicht durch das Produkt, sondern wie folgt beschrieben anderweitig erfüllt:

⟨*RW*1⟩ „*Nach dem Spielstart darf das Wegenetz nicht geändert werden können.*“
Da das Wegenetz statisch während des Spielbetriebs existiert, darf das Wegenetz nicht nach dem Spielstart geändert werden.

⟨*RW*2⟩ „*Die Roboter transportieren Waren nicht physisch.*“
Da für die Aktion Be- und Entladen keine Vorrichtung existiert und die Entwicklung keinen Informationstechnischen Mehrwert hat, wird dies nicht umgesetzt.

⟨*RW*3⟩ „*Die grafische Darstellung erfolgt nicht in Echtzeit.*“
Da die dauernde Übertragung nicht viel mehr Information enthält, als eine abschnittsweise Ortsbestimmung, soll die grafische Darstellung nicht in Echtzeit erfolgen. Außerdem existiert die Darstellung des Spiels mit den Robotern.

⟨*RW*4⟩ „*Die Karte wird nicht dreidimensional visualisiert.*“
Da eine Visualisierung durch die Roboter existiert, soll diese nicht dreidimensional visualisiert werden.

⟨*RW*5⟩ „*Zur Spielzeit dürfen die Eigenschaften von Ressourcen und Produkte nicht verändert werden.*“
Dies ist notwendig da eine Änderung während der Spielzeit zu Inkonsistenzen führen könnte.

9 Glossar

Actionpoint

Actionpoints sind Knoten und die senkrecht auf dem Wegenetz aufgeklebten Markierungen.

Arbitrator

Der Arbitrator entscheidet in Abhängigkeit der Reihenfolge und der definierten Methode takeControll in jedem Behavior, wann welcher Behavior aktiv wird.

Auftrag

Der Auftrag setzt sich aus Auftragnehmer, Auftraggeber, der zu transportierenden Ware, Start- und Zielpunkt sowie dem zu erwartenden Lohn zusammen.

Behavior

Ein Behavior definiert Aktionen, die bei bestimmten Geschehnissen passieren sollen.

Benutzer

Menschlicher Anwender, der auf den Server zugreift. Der Benutzer (Admin) verwaltet den Server. Der Benutzer kann (muss aber nicht) Spieler sein.

Broadcast

Die identische Nachricht an alle teilnehmenden Roboter.

Deadlock

Beim Deadlock können die Roboter nicht mehr ihrer Aufgabe nachkommen, weil sie sich gegenseitig behindern. Ein Deadlock ist also ein zu vermeidender Spielzustand.

FCFS-KI

Diese KI arbeitet nach dem Prinzip first-come, first-served, nimmt also immer den zuerst zur Verfügung stehenden Auftrag an.

GUI

Die GUI ist die grafische Benutzeroberfläche für den Server und den Spielerclient.

Kartendaten

Mit den Kartendaten sind die Informationen über das Wegenetz gemeint, welches als Graph implementiert wird.

Konflikt

Ein Konflikt tritt auf, wenn eine Straße von einem Roboter versperrt ist. Andere Roboter können diese zur selben Zeit nicht befahren.

Künstliche Intelligenz (KI)

Das Programm auf den Robotern, welches das Handeln des Roboters festlegt.

NXT

NXT ist ein Roboterbausatz der Firma Lego.

Ressourcen

Unter Ressourcen fallen Industrien, Waren und der Kraftstoff für die Roboter.

Roboter

Die Roboter arbeiten jeweils mit einer NXT-2.0-Einheit der Firma Lego und stellen die Transportmittel für die Waren dar.

Server

Der Server dient hier als Koordinationsschnittstelle und übernimmt somit die Aufgabe des Spielleiters.

Spieler

Menschlicher Anwender, der auf die Spieleroberfläche (nicht den Server) zugreift.

Spieleroberfläche

Graphische Benutzeroberfläche für den Spieler, um einerseits das Spielgeschehen zu beobachten, andererseits aktiv am Spiel teilzunehmen.

Spielzeit

Die Spielzeit bezeichnet den Zeitraum, der nach Initialisierung des Spiels und dessen Start bis zum Erreichen des Spielziels abläuft.