Thema:

Erstellung eines Multi-Agenten-Systems (MAS) mit Simulationsumgebung zur Robo-Minensuche

im Rahmen des Seminars „Landminenerkennung durch Robotereinsatz“

im Fachgebiet Informatik
am Lehrstuhl für Informatik

Themensteller: Dr. Dietmar Lammers
Betreuer: Dr. Dietmar Lammers

vorgelegt von: Kolja Sauerberg
Christian Sonnenberg
Dominik Eibl

Abgabetermin: 05.01.2004

Inhaltsverzeichnis

1 Einleitung

Die Aufgabenstellung „Erstellung eines Multi-Agenten-Systems (MAS) mit Simulationsumgebung zur Robo-Minensuche“ impliziert, dass die Kernaufgabe dieses Themas in der praktischen Erstellung eines solchen Softwaresystems liegt. Gleichwohl ist es wichtig, sich mit den theoretischen Grundlagen der (un-) möglichen Technologien, die dazu verwendet werden, vertraut zu machen. Diese erst erklären und rechtfertigen den Einsatz der verwendeten Technologien in Bezug auf eine effiziente Erstellung, Nutzung und Erweiterbarkeit des Produktes. Das Ziel dieser Arbeit ist daher, dem Leser zunächst einen theoretischen Überblick über unsere verwendeten Technologien sowie Begriffsdefinitionen zu verschaffen. Anschließend befasst sich die Arbeit konkreter mit der Simulationsumgebung. Zusammen mit dem Handbuch soll dem Anwender/Weiterentwickler unseres MAS der Einstieg erleichtert werden.

2 Theorie

2.1 Der Agent – Was bin ich?

Ein Agent in dem hier gebrauchten Sinne ist ein Stück Software, von der sich derzeit noch keine einheitliche Definition durchgesetzt hat. Agenten besitzen allerdings Eigenschaften, die von nahezu allen Experten zur Beschreibung eines Agenten herangezogen werden. In Anlehnung an menschliche Agenten suchen und sammeln sie Informationen und erledigen komplexe Aufgaben, d.h. sie haben eine Vertreterfunktion, sei es für Personen, Systeme oder Teilsysteme. Dies geschieht meist in Hinblick auf ein Gesamtziel, muss aber nicht zwingend der Fall sein, wenn z.B. zwei Agenten um knappe Ressourcen konkurrieren. Agenten besitzen einen gewissen Grad an Intelligenz, um ihre Aufgaben autonom durchführen zu können. Eine wichtige Voraussetzung dafür ist die Fähigkeit, ihr Wissen zu vergrößern; dies kann anhand von Beispielen oder Anweisungen geschehen, oder durch die Übernahme des Wissens anderer Agenten. Die Intelligenz zeigt sich durch ihre Fähigkeit, den Erfolg einer vorher ausgeführten Aktion zu bewerten. Die Bewertung kann intern durch implementierte Regeln und Funktionen geschehen sowie extern durch den Benutzer oder ein System.
Agenten existieren in einer Umgebung, die sie wahrnehmen und verändern können und mit der sie kommunizieren. Diese Umgebung setzt sich aus anderen Agenten, Agentensystemen oder Diensten zusammen. Die Fähigkeit, auf Umweltänderungen reagieren zu können, setzt die Existenz von Sensoren bzw. Regeln voraus. Ein Agent kann selbst die Umwelt verändern, z.B. aufgrund eines internen Parameters oder eines aktuellen Zustandes der Umwelt. Diese Eigenschaft wird häufig als proaktiv bezeichnet. Voraussetzung des aktiven Handelns ist die Existenz eines Ziels jedes Agenten. Wie bereits angedeutet, teilt sich ein Agent seine Umwelt mit anderen Agenten – es entsteht ein MultiAgentenSystem.

2.2 Multi-Agenten-System (MAS) – Wer weiß was?

Ein Multi-Agenten-System, ein System, das aus mehreren autonomen Komponenten besteht, soll hier wie folgt charakterisiert werden: Jeder einzelne Agent ist kein Alles-Könner. Jeder Agent besitzt individuelle Fähigkeiten und

individuelles Wissen. Es gibt keine globale Systemkontrolle, d.h. die Autonomie der einzelnen Agenten wird nicht beschnitten. Das Wissen ist dezentral und kann über Kommunikation erweitert werden, es existieren (implizite) übergeordnete Ziele. Als Schlussfolgerung aus diesen Prämissen ist zu ziehen, dass das unterschiedliche Wissen und die unterschiedlichen Fähigkeiten koordiniert werden müssen. Dies geschieht durch Interaktion zwischen dem Agenten und seiner Umwelt. Zur sinnvollen Interaktion bedarf es einer gemeinsamen Agentensprache, eines gemeinsamen Kommunikationsprotokoll, eines gemeinsamen Interaktionsprotokoll sowie einer gemeinsamen Weltsicht. Für all diese Schlüsselelemente gibt es derzeit unterschiedliche Bestrebungen, Standards zu setzen. Eine Organisation, die sich mit Standardisierungen in Bezug auf Agentensysteme beschäftigt, ist die Foundation of Physical Intelligent Agents, kurz FIPA. Da diese Organisation offene und hoffnungsvolle Ansätze bietet, soll sie kurz dargestellt werden und die wichtigsten Standards erläutert werden.

2.3 Interaktion mit FIPA-Standards

FIPA ist eine 1996 gegründete Non-Profit-Organisation , die ihre Aufgabe darin sieht, Standards für heterogene interaktive Agentensysteme zu schaffen. Da die Interaktion eine der wichtigsten Wesensmerkmale von Agenten ist, wird zunächst die FIPA- Agent Communication Language (ACL), die oben erwähnte gemeinsame Agentensprache, vorgestellt. Der Nachrichtenaustausch nach diesem Standard geschieht über Low Level Protokolle. Eine FIPA-ACL Nachricht enthält im Wesentlichen den Typ der Nachricht, die Teilnehmer wie z.B. Sender und Empfänger, die Inhaltsbeschreibung der Nachricht sowie Interaktionskontrollen, die das Interaktionsprotokoll sowie Nachrichtenkennungen festlegen. Der Typ der Nachricht kann z.B. ein request, answer oder ein refuse sein. Durch eine Typangabe wird bereits eine gewisse Semantik impliziert. Diese lässt sich am besten anhand eines einfachen Beispiels erklären: Die Frage „Wie spät ist es?“ oder die Aussage „Es zieht im Raum“ implizieren bereits die erwarteten Antworten bzw. Aktionen. Man bekommt die Antwort „Es ist 12 Uhr“ bzw. die Tür wird geschlossen, damit es nicht mehr zieht.

Die Inhaltsbeschreibung einer FIPA-ACL Nachricht umfasst nicht nur den eigentlichen Inhalt der Nachricht, sondern legt auch die verwendete Sprache, die

Kodierung sowie die Ontologie der Nachricht fest. Die Ontologie beschreibt die gemeinsame Weltsicht der Agenten und ist als eine Art offenes Wörterbuch zu verstehen, welches den Agenten in derselben Umwelt bekannt ist.
Neben den Spezifikationen zur Agentenkommunikation gibt es etliche weitere Spezifikationen der FIPA, insbesondere zur abstrakten Agentenarchitektur, zum Agentenmanagement und zum Nachrichtentransport über heterogene Netzwerke hinweg. Diese Standards werden hier aber nur der Vollständigkeit halber erwähnt.
Im folgenden Abschnitt geht es um MAS-Entwicklungsumgebungen; hier wird Jade als ein Vertreter vorgestellt, da JADE den Anspruch hat, den FIPA-Spezifikationen zu entsprechen.

2.4 Jade als MAS-Entwicklungsumgebung

Eine MAS-Entwicklungsumgebung soll, wie der Name bereits sagt, die Entwicklung von Multi-Agenten-Systemen mit all seinen Anforderungen vereinfachen. Die Hauptanforderungen sind das gegenseitige Finden von Agenten, die Kommunikation zwischen diesen und ein einfaches Agentenmanagement. Auch die Bereitstellung von Bibliotheken oder Diensten stellen ein Gütekriterium für eine solche Umgebung dar, wenn es z.B. darum geht, eine verteilte Anwendung effizient zu entwickeln. Im vorherigen Abschnitt wurde bereits die Organisation FIPA erwähnt, die sich all diese Anforderungen zur Aufgabe gemacht hat. Die Entwicklungsumgebung JADE folgt diesen Standards und wird im Folgenden kurz eingeführt, bevor im späteren Kapitel genauer darauf eingegangen wird.
JADE, Java Agent Development Framework ausgeschrieben, ist ein vollständig in Java implementierte Open-Source-Projekt der Tilab aus Italien, das ständig weiterentwickelt wird. Diese vereinfacht die Entwicklung von MAS neben der Einhaltung der FIPA-Standards durch eine Reihe von grafischen Tools zur Unterstützung der Entwicklungsphase und des Debuggens.
Insgesamt besteht JADE aus einer Bibliothek von Klassen zur Erzeugung von Agenten, einer Agentenplattform, die einen umfangreichen Satz an Systemdiensten und Agenten zur Verfügung stellt, den oben erwähnten graphischen Werkzeugen und GUI's zur Fernwartung bzw. Konfiguration.

2.5 Zwischen Theorie und Praxis

Dieser Absatz soll dieses Kapitel mit ein paar Daten aus der realen Welt und den daraus resultierenden Anforderungen, die an die Minensuche verknüpft sind, beenden.

Weltweit gibt es in nahezu allen Krisengebieten Landminen, deren Anzahl nur zu schätzen ist, da einfache Minen sehr günstig zu bauen sind. Es gibt über 600 bekannte Minentypen, deren Suche mit einem Metalldetektor wegen eines stetig sinkenden Metallgehalts immer schwieriger wird. Fest steht, dass nach derzeitigem Stand der Technik nur der gebündelte Einsatz von mehreren Sensoren sinnvoll ist. Manuelles Suchen (1 Mann/1 Handgerät) ist mit 20-60 qm am Tag Leistung relativ langsam; zudem ereignet sich auf 1000 geräumte Minen ein schwerer Unfall.

Drehschrauben wie Minen, Landschaften, Detektoren, Wetter und sonstige Hindernisse lassen den Einsatz einer Simulationsumgebung, in der Roboter testen können, als sehr sinnvoll erscheinen.

Agenten scheinen vor diesem Hintergrund also durchaus eine geeignete Wahl zu sein, gemeinsam ein Minenfeld zu bearbeiten, mit dem impliziten Oberziel, dieses zu „entschärfen“.

3 Technologien

3.1 Swing

Die grafische Benutzerschnittstelle der Simulationsumgebung ist unter Benutzung von Java Swing programmiert. Im Vergleich zum Abstract Window Toolkit (AWT) von Java bietet Swing einige gewichtige Vorteile. Das AWT hat ein sehr schlechtes Skalierungsverhalten, ein vererbungsbasiertes Ereignismodell und gründet auf einer Peer-Architektur. Peers sind native Komponenten von Benutzerschnittstellen, also zum System gehörig. Sie bekommen ihre Aufgaben von den AWT-Komponenten zugewiesen. Peers übernehmen somit die ganze Arbeit der AWT-Objekte, deren Aufgabe nur noch darin besteht, zum richtigen Zeitpunkt mit den Peers zu interagieren. Dadurch besitzt jede AWT-Komponente ihren eigenen Peer und wird auf den meisten Plattformen in einem nativen Fenster dargestellt. Dies führt zu einer sehr schlechten Leistungsfähigkeit, falls mehrere AWT-Komponenten erzeugt werden. Auch das Einfügen nativer Peers verschiedener Plattformen in eine Java-Umgebung bereitet große Probleme, so dass Inkonsistenzen über Plattformgrenzen hinaus auftreten können.

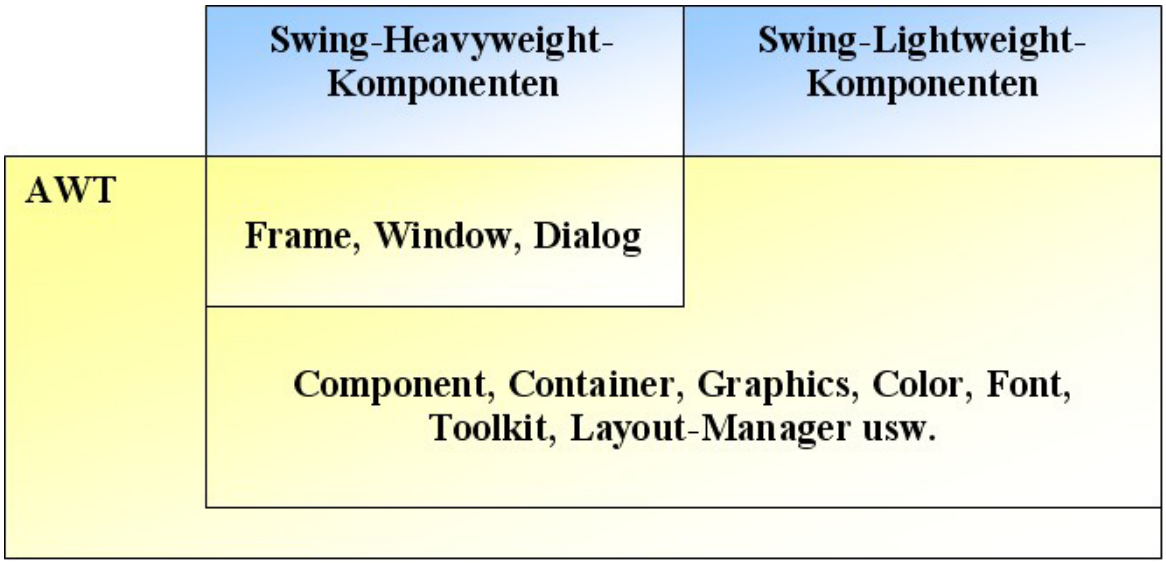

Abbildung 1: Swing vs. AWT

Swing verwendet die Infrastruktur des AWT (z.B. Grafiken, Farben, Schriftarten, Toolkit, Layoutmanager), aber es verwendet keine AWT-Komponenten. Die einzigen für Swing relevanten Klassen des AWT sind Frame, Window und Dialog, die in erweiterter Form als Heavyweight-Komponenten in Swing wieder zu finden sind. Heavyweight-Komponenten haben einen nativen Peer und werden in ihrem

eigenen, undurchsichtigen Fenster dargestellt. Im Gegensatz dazu haben Lightweight-Komponenten keinen nativen Peer, sondern befinden sich immer im Fenster einer Heavyweight-Komponente. Somit kann eine Lightweight-Komponente auch einen undurchsichtigen Hintergrund haben. Im Prinzip ersetzt Swing die meisten Heavyweight-Komponenten (bis auf JFrame, JDialog und JWindow) durch Lightweight-Komponenten. Dadurch werden weniger native Peers verwendet und das Laufzeitverhalten verbessert sich vehement.

Swing basiert auf einer Model-View-Controller Architektur. Genau wie AWT-Objekte delegieren Swing-Objekte Aufgaben an andere Objekte. Bei Swing-Objekten erfolgt die Delegation an eine Erweiterung der Klasse ComponentUI (und nicht an native Peers). Der Delegate der Swing-Komponenten ist in Java implementiert und kann daher - im Gegensatz zu AWT-Komponenten - einfach erweitert werden, um das Verhalten zu ändern. Durch die Model-View-Controller Architektur ist die Funktionalität der Swing-Komponenten losgelöst von der Darstellung und der Ein- und Ausgabe. Der UI-Delegate einer Swing-Komponente entspricht dem View-Controller (Darstellung, Ein- und Ausgabe), die in Java implementierte Funktionalität ist das Model. Dadurch können Aspekte der visuellen Darstellung der Komponente oder Ereignisbehandlung verändert werden. Außerdem ist es möglich, auf verschiedenen Plattformen die gleiche Darstellung (z.B. durch die Wahl eines Java Look & Feel) zu erreichen.

Für diese Simulationsumgebung wird das Spielfeld der Feldagenten-GUI durch JButtons dargestellt. Diese Wahl ist nur unter Verwendung von Swing möglich, da es sich bei JButtons um Lightweight-Komponenten handelt. Auch die Zuweisung von Icons ist nur bei Swing-Schaltflächen möglich. Da die Simulationsumgebung ein plattformübergreifendes, verteiltes Multi-Agenten-System ist, werden durch die Verwendung von Swing Inkonsistenzen in der Darstellung vermieden.

3.2 Jade

Die Simulationsumgebung Basiert im Kern auf einem *Multi-Agenten-System (*kurz *MAS).* Das MAS wurde hierbei mit Hilfe des *JADE* (Java Agent Development) – Frameworks (Version 3.0b1) realisiert. JADE wird von *Telecom Italia Lab* entwickelt und ist unter der Gnu Public License (GPL) kostenlos erhältlich. JADE weist einige bemerkenswerte Eigenschaften auf, die die Entwicklung eines MAS und somit auch die Entwicklung der Simulationsumgebung stark vereinfachen.

3.2.1 JADE - Features

JADE ist vollständig in der Programmiersprache *Java* implementiert. Die durch das JADE – Framework verwaltete *Agentenplattform* ist *verteilt* und als Folge der vollständigen Implementierung in Java *betriebssystemübergreifend*. Zudem stellt JADE eine graphische Benutzerschnittstelle zur Verfügung, um Agentenplattformen verwalten zu können.
Innerhalb einer Agentenplattform ist eine uneingeschränkte Mobilität der Agenten gegeben, der Transfer von Zustand und Code eingeschlossen.
Das Ausführen von verschieden Agentenaktivitäten (in JADE: *Behaviours*) erfolgt parallel bzw. nebenläufig.
Darüber hinaus ist eine JADE – Agentenplattform stets konform zur *FIPA* – Spezifikation. JADE unterstützt die Bildung von *Agenten – Domänen*, in welchen eine Menge von Agenten zusammengefasst werden können, um gemeinsam einen Dienst für andere Agenten (bzw. Agentendomänen) derselben Agentenplattform zur Verfügung zu stellen.
Die zwischen den Agenten ausgetauschten Nachrichten (*ACL - Messages*) werden *effizient* innerhalb einer Agentenplattform transportiert. Um die Interaktion unter den Agenten zu vereinheitlichen und zu regulieren, stellt JADE bereits eine *Bibliothek* von verschiedenen *FIPA – Interaktionsprotokollen* bereit.
Nicht zuletzt erlaubt das JADE – Framework auch, autonome Agenten aus externen Anwendungen heraus in einer konkreten Agentenplattform zu erzeugen.

3.2.2 Jade – Agentenplattform

Eine JADE – Agentenplattform besteht aus vier zentralen Elementen. Dazu zählen das **`Agent – Management – System`** (kurz AMS), der **`Directory Facilitator`** (kurz DF), das **`Nachrichtentransport – System`** und schließlich die Agenten selbst.
Das AMS beaufsichtigt den Zugang und die Benutzung der Agentenplattform. Darüber hinaus verwaltet das AMS eine Liste aller Agenten der Plattform (white page service). Pro Plattform existiert nur ein AMS.
Der DF stellt den so genannten yellow page service der Plattform bereit. Er enthält eine Liste von Diensten, welche die Agenten der jeweiligen Plattform anbieten.

Benötigt ein Agent einen bestimmten Dienst, so wendet sich dieser an den DF. Der DF sowie das AMS sind als Agenten realisiert.

Das Nachrichtentransport – System, auch als **`Agent Communication Channel`** (ACC) bezeichnet, stellt die Softwarekomponente dar, welche den gesamten Austausch von Nachrichten innerhalb der Plattform und zu anderen Plattformen kontrolliert.

Eine Agentenplattform kann über mehrere Rechner (Hosts) verteilt sein. Dazu muss auf dem Host eine JAVA – Virtual – Machine (JVM) vorhanden sein. Der Host, auf dem eine Agentenplattform gestartet worden ist, beherbergt den **`Main – Container`** (front end). In einem Container leben mehrere Agenten und er stellt eine komplette Laufzeitumgebung für die Ausführung von Agenten zu Verfügung. Auf demselben oder auf anderen Hosts können zusätzliche Container erzeugt werden, welche sich dann mit dem Main – Container verbinden. Allerdings enthält nur der Main – Container die Agenten des AMS und DF. In folgender Abbildung werden die Zusammenhänge noch einmal dargestellt.

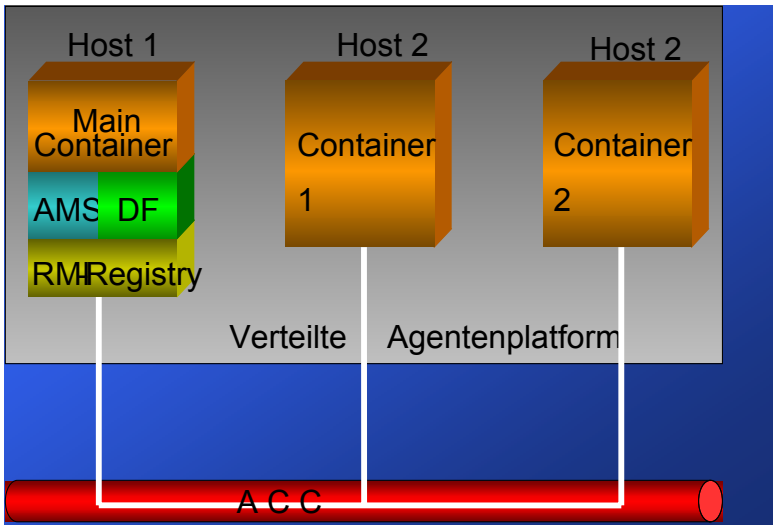

Abbildung 2: JADE Agentenplattform mit mehreren Containern

3.2.3 JADE – Agenten

Die Roboter der Simulationsumgebung werden als Agenten realisiert. Diese Vorgehensweise ist berechtigt, weisen Roboter doch ähnliche Eigenschaften auf, wie Software – Agenten. Roboter arbeiten zu einem gewissen Grad intelligent (Problemlösungswissen steckt in Algorithmen und Daten) und können autonom agieren. Zusätzlich interagieren Roboter mit ihrer Umwelt und bei bestimmten Problemstellungen agieren sie kooperativ mit anderen Robotern. Aus dieser

Erkenntnis heraus werden in dieser Arbeit diejenigen JADE - Agenten, welche Roboter repräsentieren, auch als **`Robot - Agents`** bezeichnet.

JADE – Agenten sind Instanzen einer benutzerdefinierten Java – Klasse, welche von der Klasse **`Agent`** des Paketes **`jade.core`** abgeleitet ist. Agenten können in ihrem Lebenszyklus verschiedene Zustände durchlaufen: **`initiated`** (Agentenobjekt erzeugt, jedoch noch nicht beim AMS angemeldet und somit noch nicht in der Agentenplattform sichtbar), **`active`** (Agent ist angemeldet und besitzt einen regulären Namen und eine Adresse), **`waiting`** (Agent ist im Wartezustand, alle seine Aktivitäten sind gestoppt), **`suspended`** (Agent ist gestoppt), **`transit`** (wird erreicht, wenn ein mobiler Agent zu einem neuen Container migriert, z.B. vom Rechner auf ein Handy), **`deleted`** (Agent ist endgültig tot und nicht mehr beim AMS registriert).

Die Aktivitäten eines Agenten (in JADE: **`Behaviours`**) werden gleichzeitig ausgeführt. Jeder Dienst oder jede Funktionalität eines Agenten sollte als ein Behaviour implementiert sein. Ein interner Scheduler der Agent - Klasse, welcher vor dem Programmierer verborgen ist, verwaltet automatisch die Ausführung der Agenten – Behaviours. Das Scheduling erfolgt nach dem **`round robin - Algorithmus`**. Die Behaviours können sich in der Art ihrer Ausführung unterscheiden. Einige Behaviours werden immer wieder zur Ausführung gebracht (deren **`done()`** – Methode gibt immer den Wert **`false`** zurück), z.B. ein aktives Warten beim Nachrichtenempfang. Andere Behaviours werden nur genau einmal ausgeführt (deren **`done()`** – Methode gibt den Wert **`true`** zurück), z.B. eine Bewegen – Aktion eine Robot – Agents.

Jeder Agent hat eine Nachrichtenwarteschlange („Posteingang“), welche auf verschiede Arten abgefragt werden kann: **`polling`** (aktives Warten), **`blockierend`** (solange keine Nachricht eingetroffen ist, ist Agent im Zustand **`waiting`**), **`time – out – basiert`** (blockiert für eine bestimmte Zeit), **`Mustervergleiche`** (Nachrichten werden über Filter, so genannte **`Message Templates`**, abgefragt).

Nachfolgende Grafik verdeutlicht das Konzept des JADE – Agenten noch einmal.

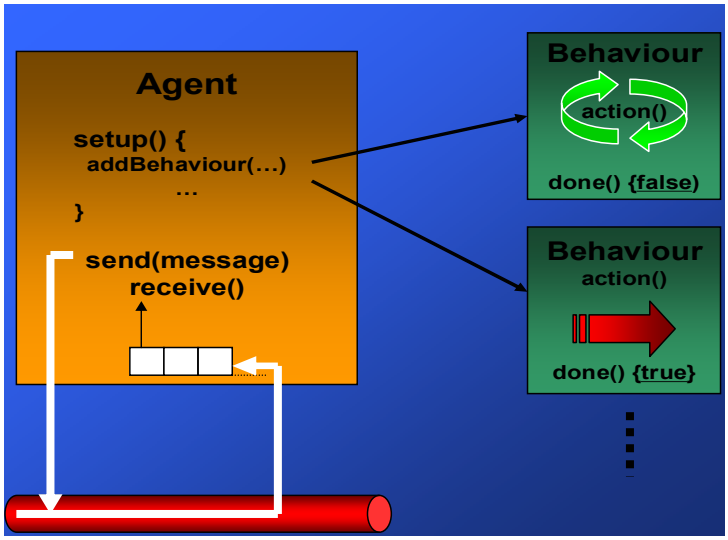

Abbildung 3: Konzept des JADE – Agent

3.2.4 JADE – Nachrichten

Die Kommunikation zwischen den Agenten bzw. der Roboter – Agenten erfolgt auf Basis von so genannten **`ACL – Messages`** (ACL -> **`Agent Communication Language`**). Diese Nachrichten sind jedoch nicht als Folge von Strings (Zeichenketten) implementiert, vielmehr wird eine Nachricht als ein Objekt der Klasse **`jade.lang.ACLMessage`** repräsentiert.

Die Besonderheit bei der Abfrage der Nachrichtenwarteschlange eines Agenten besteht darin, dass Nachrichten gezielt auf Basis von Mustervergleichen selektiert werden können. Die Muster lassen sich aus verschieden Eigenschaften einer JADE - Nachricht bilden. So besitzt eine ACL – Message ein *Performative* - Attribut (legt Nachrichtentyp fest, z.B. **`REQUEST`**, **`INFORM`**), ein **`Language`** - Attribut (vereinbart eine Nachrichtensprache), einen *Sender* (Absender), einen *Receiver* (Empfänger) oder eine **`ConversationID`**. Eine Nachrichtabfrage kann jetzt gezielt nach Nachrichten eines bestimmten Absenders, eines Nachrichtentyps oder einer **`ConversationID`** erfolgen. Auch Kombinationen sind möglich. So können zusammengesetzte „Filter“ beispielsweise Nachrichten selektieren, die von einem bestimmten Typ und gleichzeitig einen bestimmten Absender haben. Derartige Filter werden in JADE durch die **`Message Templates`** (Mustervorlagen) realisiert. Der Vorteil der Nachrichtenabfrage via **`Message Templates`** liegt darin begründet, dass die Nachrichtenwarteschlange jetzt nicht

mehr nach dem FIFO – Prinzip abgefragt werden muss, sondern gezielt nach bestimmten Nachrichten „gefahndet“ werden kann.
Innerhalb der hier vorgestellten Simulationsumgebung werden sehr viele ACL Messages zwischen den beteiligten Agenten verschickt. Um diese Nachrichtenmenge koordiniert handhaben zu können, wurde insbesondere von den vorgestellten **`Message Templates`** intensiv Gebrauch gemacht.

.

4 Architektur

4.1 Architektur der Simulationsumgebung

Die Simulationsumgebung stütz sich, wie schon erwähnt, im Kern auf einem Multi – Agenten- System (MAS) ab. Dabei ergibt sich nun die Frage, was eigentlich Agenten einer Landminenerkennung bzw. dessen Simulation zu tun haben.
Prinzipiell ist der Unterschied zwischen Roboter und Agent nicht sehr groß (siehe Kapitel 1.1), insbesondere wenn ein Roboter weitestgehend autonom agieren soll. In der Simulationsumgebung wird vornehmlich der Einsatz „selbstständig“ agierender, programmierbarer Roboter nachgebildet.
Eine zusätzliche Rechtfertigung für den Einsatz eines MAS ergibt sich, wenn nicht nur Roboter, sondern auch alle aktiv an der Minensuche beteiligten Objekte als Agenten – Analogon betrachtet werden. Diese Sichtweise erweitert die Simulation um eine ganzheitliche Komponente, nämlich die Administration. In der Realität agieren ja nicht nur die Roboter autonom und unter umständen räumlich getrennt („verteilt“), sondern auch das gesamte Minensuchteam an sich. Dieser Aspekt wurde bei der Entwicklung der vorliegenden Simulationsumgebung aufgegriffen und konsequent umgesetzt. Alle aktiv an der Minensuche beteiligten Objekte werden demnach als Agenten modelliert. Nicht zuletzt hat uns die JADE – Architektur zu dieser Sicht der Dinge animiert.
Die Architektur der Simulationsumgebung ist demnach zweigeteilt. Auf der einen Seite befinden sich die Roboter – Agenten. Die andere Seite stellt die Administration mit *Server-*, *Feld-*, *Simulationsagent* und *Remote-Robot-Management-Agent* (kurz rrm - Agent) dar. Beiden Seiten unterliegt das *JADE – Plattform*. Der Server – Agent ist die einzige nach außen sichtbare Schnittstelle für einen Roboter – Agenten. Die oberste Schicht der Architektur manifestiert sich in der graphischen Benutzerschnittstelle (kurz *GUI*). Das GUI ist hierbei als Sammelbegriff für die speziellen GUIs insbesondere der Roboteragenten, und des Feldagenten.
Dieser Zusammenhang wird in folgender Graphik noch einmal dargestellt. Die sichtbaren Pfeile repräsentieren Kommunikationskanäle bzw. –beziehungen. Der rrm - Agent unterstützt das Erstellen von Robot – Agents außerhalb des Main – Containers (z.B. von einem anderen Rechner aus).

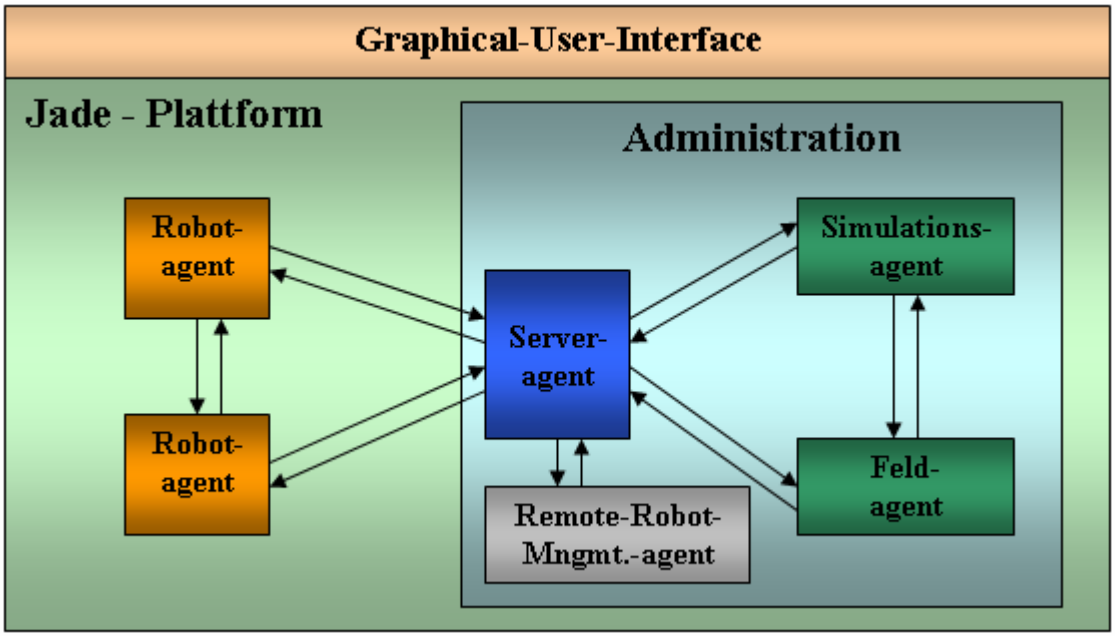

Abbildung 4: Simulationsumgebung

4.2 Rolle der Agenten

In der Simulationssoftware kommunizieren mehrere Agenten miteinander. Jeder Agent führt bestimmte Aufgaben aus. In diesem Abschnitt werden die Aufgaben der einzelnen Komponenten erläutert.

4.2.1 Serveragent

Der Serveragent ist die Kommunikationsschnittstelle zwischen den Robotern und der simulierten Umwelt. Er startet alle für die Simulation wichtigen Komponenten und registriert die Roboter im System. Dem System kann jederzeit ein Roboter hinzugefügt werden. Eine weitere Aufgabe ist der Empfang und das Weiterleiten von Nachrichten, die Roboter bzw. die Umweltkomponenten an ihn versenden. Solche Nachrichten können beispielsweise Handlungsbefehle der Roboter (Move, Turn etc.) oder Antworten auf Sensoranfragen (Rückgabewert des Metallsensors eines Roboters für ein bestimmtes Feld, …) sein. Der Serveragent ist in der Lage, den jeweils richtigen Agenten mit Informationen zu versorgen.
Zeitintervalle werden in der aktuellen Version nicht betrachtet. Die Software kann aber um diese erweitert werden, damit Effizienz und Effektivität der zu prüfenden Detektionsalgorithmen besser gemessen werden können.

Der Serveragent kann sowohl einfache Nachrichten als auch komplexere Objekte übertragen. Die komplexeren Objekte, wie z.B. Bilder, werden vorher zu Strings mittels des Verschlüsselungsalgorithmus Base64 codiert.

4.2.2 Simulationsagent

Der Simulationsagent erhält Aufträge vom Serveragenten, wenn ein Roboter einen Sensor benutzt, sich bewegt oder dreht. Daraufhin kommuniziert der Simulationsagent mit dem Feldagenten, um sich die für die Durchführung der Aufträge benötigten Informationen zu besorgen. Die Informationen werden aber oftmals zwecks einer realistischeren Simulation verfälscht an den Serveragenten weitergegeben. Je nach Sensor und Qualität entscheidet der Simulationsagent über den Grad der Verfälschung.
Des Weiteren berechnet der Simulationsagent die Energieabzüge bei Durchführung von bestimmten Roboteraktionen. Jeder Roboter hat einen bestimmten Energievorrat, der durch Bewegungen und Sensoreinsatz stetig aufgebraucht wird. Ist der Energievorrat verbraucht, so kann der Roboter keine weitere Handlung mehr vollziehen.

4.2.3 Feldagent

Der Feldagent ist für die Erstellung und Verwaltung des Spielfeldes zuständig. Bei der Erstellung werden die Spielfeldgröße und die zu jeder Kachel gehörigen Feldinformationen festgelegt. Zusätzlich werden während der Simulation die aktuellen Positionen und Zustände aller im System registrierten Roboters abgespeichert. Änderungen am Spielfeld erfolgen ausschließlich über den Feldagenten, z.B. wird eine Mine vom Spielfeld entfernt, wenn ein Roboter auf diese fährt und explodiert.
Der Feldagent übermittelt auf Anfrage die entsprechenden Feldinformationen an den Server- und Simulationsagenten. Auf dem Feldagenten setzt die Feldagenten-GUI auf, die entsprechend des Spielfeldaufbaus und der Änderungen während der Simulation aktualisiert wird.
(Ferner besteht die Möglichkeit, generierte Karten abzuspeichern und zu laden.)

4.2.4 Grafische Benutzerschnittstelle

Die grafische Benutzerschnittstelle erlaubt dem Benutzer, Einblicke in die Geschehnisse zu nehmen, die während einer Simulation ablaufen. Hierbei werden das GUI des Feldagenten und des Roboters unterschieden.

Das GUI des Feldagenten kann nur einmal beim Start des Serveragenten erzeugt werden. Sichtbar ist dieses GUI nur auf dem Rechner des Serveragenten. Neben der Anzeige des Spielfelds, den Feldinformationen für jede Kachel und dem Spielverlauf (z.B. Bewegungen und Blickrichtungen der Roboter) können gleichzeitig Roboter registriert und abgemeldet werden. Auch die Generierung von Spielfeldern (manuell oder automatisch) sowie das Speichern und Laden von Karten erfolgt über diesen Monitor.

Die Programmierung der Roboter-GUI ist eigentlich Aufgabe derjenigen, die die Roboter implementieren. Jeder Roboter hat eine andere Strategie beim Aufzeichnen seiner Karte. Durchsuchen mehrere Roboter simultan ein Spielfeld, so ist es nicht sinnvoll eine grafische Benutzerschnittstelle vorzuschreiben. Als Vorlage dient das GUI des **`ManualRobotAgent`** (in der Abgabe enthalten).

4.3 Use Case

Die Entwicklung der Simulationssoftware ist Use-Case getrieben erfolgt. Jeder Agent ist ein Akteur. Akteure können demnach Serveragent, Simulationsagent, Feldagent und alle registrierten Roboteragenten sein. Serveragent, Simulationsagent und Feldagent verkörpern zusammen die Simulationsumgebung, in welche sich beliebig viele Roboteragenten einloggen können.

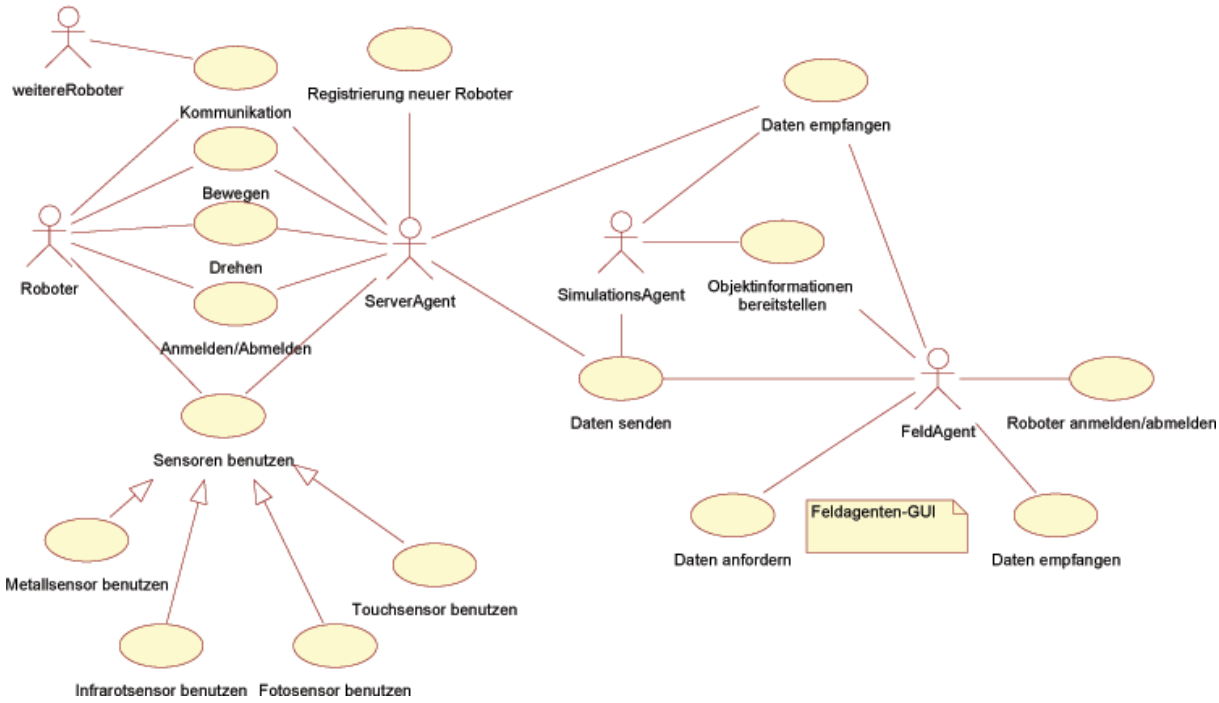

Abbildung 5: Use Case Diagram erstellt mit Rational Rose

Ein Roboteragent kann sich im System an- und abmelden, bewegen, drehen, kommunizieren oder einen Sensor benutzen. Benötigt der Roboter Daten über ein Feld, so hat er die Möglichkeit alle im System implementierten Sensoren zu nutzen. In der aktuellen Version 1.0 existieren ein Stoßsensor zum Aufspüren von Hindernissen, ein Fotosensor zur Bildauswertung (z.B. zur Minenerkennung oder zum Erkennen von Flüssen), ein Infrarotsensor zur Minenerkennung und ein Sensor, der Metallgegenstände erkennen kann.

Sobald ein Roboter eine Aktion ausführt, werden die entsprechenden Daten an den Serveragenten geschickt. Dieser bildet die einzige Schnittstelle zwischen Robotern und simulierter Umwelt. Der Serveragent speichert die Adressen aller im System vorhandenen Agenten und weiß, welche Daten er von den Agenten jeweils anfordern kann bzw. an welchen Agenten Daten geschickt werden müssen. Sobald der Serveragent also Daten von einem Roboteragenten erhält, weil dieser eine Aktion ausführen möchte, leitet der Serveragent die notwendigen Schritte ein. Für den Use Case „Bewegen“ und „Sensor benutzen“ wird der Feldagent kontaktiert, da für solche Handlungen Feldinformationen benötigt werden. Der Feldagent erstellt und aktualisiert die grafische Benutzerschnittstelle. Er fordert Daten von dem Feldagenten-GUI an (z.B. bestimmte Feldinformationen bei Sensoreinsatz) und sendet diese an den Serveragenten zurück. Der

Serveragent kontaktiert den Simulationsagenten. Der Simulationsagent entscheidet dann, ob die Daten zwecks einer realistischeren Simulation verfälscht werden. Denkbar wären Verfälschungen der von einem Fotosensor zurückgelieferten Bilder aufgrund von extremen Umweltbedingungen (z.B. starker Regen). Die bearbeiteten Informationen werden daraufhin über den Serveragenten an den entsprechenden Roboteragenten weitergeleitet, der seine Aktion damit abschließen kann.
Der Use Case „Drehen“ erfordert nur die Unterstützung des Simulationsagenten, der den Energieverbrauch der Drehung berechnet. Feldinformationen werden hierbei nicht benötigt.
Die An- und Abmeldung von Robotern wird vom Serveragenten verwaltet. Roboter können mittels des Use Case „An-/Abmelden“ eine An- und Abmeldeanfrage stellen. Ebenso ist es möglich, Roboter über die grafische Benutzerschnittstelle des Feldagenten zu registrieren bzw. zu entfernen. Die Anfragen werden dann an den Serveragenten weitergeleitet. Anfragen können abgelehnt werden, wenn z.B. ein Roboter sich mit einem im System bereits vergebenen Namen anmeldet.
Bei dem Use Case „Kommunikation“ kann ein Roboter Kontakt zu einem anderen Roboter aufnehmen. Über eine Anfrage beim Serveragenten erhält der Senderroboter einen Verweis auf den Empfänger und kann anschließend mit diesem Daten austauschen (z.B. zur Synchronisation von Karten).

4.4 Ausgewählte Klassendiagramme

Die Software besteht aus sehr vielen selbst entworfenen Klassen und referenziert auf diverse JADE- und Java-Bibliotheken. Die Darstellung des kompletten Klassendiagramms ist aufgrund der Komplexität nicht sinnvoll. Dennoch sollen einige wichtige Zusammenhänge anhand von Klassendiagrammausschnitten erläutert werden.

4.4.1 Implementierung eigener Roboter

Alle Agenten (Roboteragenten sowie Simulations-, Feld- und Serveragent) erben von der Klasse **`Agent`**, die in einer JADE-Bibliothek (**`jade.core`**) implementiert ist. Roboteragenten weisen allesamt besondere Eigenschaften auf, die in der Klasse **`RobotAgent`** – einer Unterklasse von **`Agent`** – verankert sind. Somit sind

alle im System vorhandenen Roboter Unterklassen von **`RobotAgent`**. Um einen eigenen Roboter (in diesem Beispiel wurde der Name **`Robbi`** gewählt) in das System zu integrieren, muss also die Klasse des neuen Roboters von **`RobotAgent`** erben. Der neue Roboter überschreibt die Methode **`runRobot()`** und erzeugt in dieser Methode eine Referenz auf eine weitere selbst geschriebene Befehlsklasse (hier: **`Operation1`**).

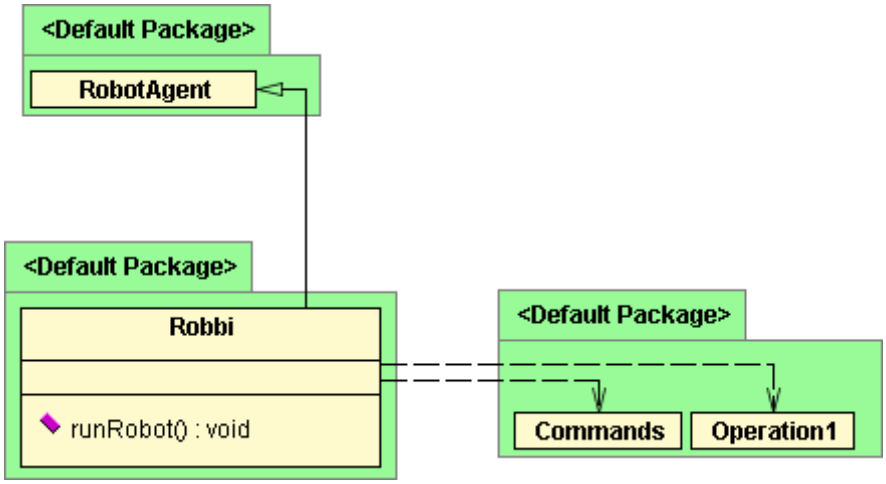

Abbildung 6: Ausschnitt des Klassendiagramms – eigener Roboter

`Operation1` ist eine Unterklasse von **`Commands`**, welche alle existierenden Roboterbefehle beinhaltet (**`move()`**, **`touch()`** etc.). In **`Operation1`** (genauer: in der **`run()`**-Methode) hat der Roboterprogrammierer alle Freiheiten, einen Algorithmus für den zu testenden Roboter zu implementieren. Weil JADE nicht threadsicher ist, läuft der vom Programmierer erzeugte Algorithmus in einem Thread ab, der in der **`execute()`**-Methode erzeugt wird. Der Thread führt nach seiner Erzeugung automatisch die **`run()`**-Methode aus. Um die Thread-Eigenschaften zu nutzen, bindet die Klasse Operation1 das Interface **`java.lang.Runnable`** ein.

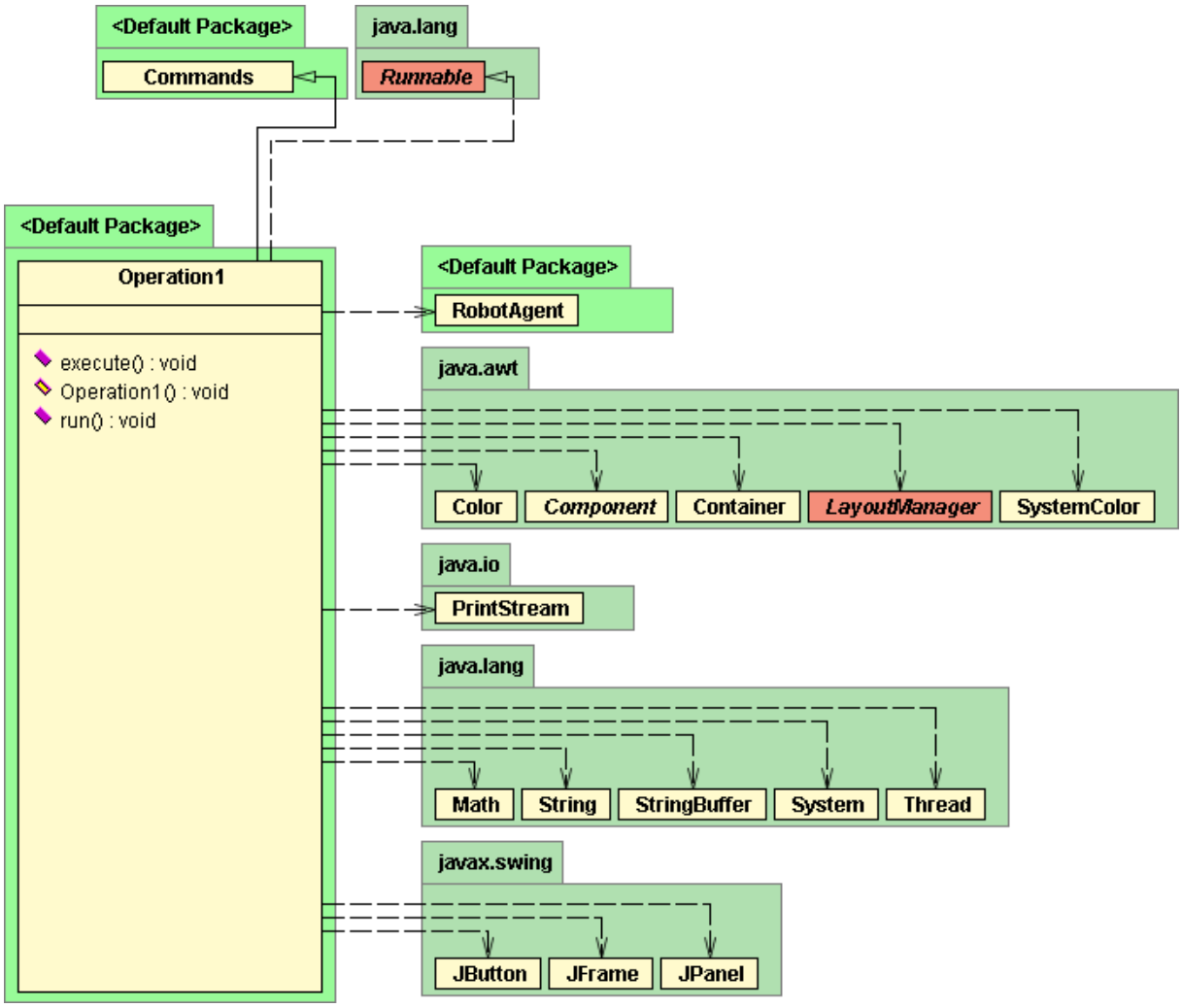

Abbildung 7: Ausschnitt des Klassendiagramms - Befehlsklasse

4.4.2 Serveragent

Der Serveragent (siehe Anhang: Abbildung 8) erbt ebenfalls von der Klasse **`Agent`**. Er bekommt – wie jeder Agent - von der JADE-Plattform einen **`Agent Identifier`** (**`AID`**) zugewiesen, der für eine eindeutige Identifizierung sorgt. Eine seiner Aufgaben ist die Registrierung weiterer Agenten (**`register()`**). Die Adressen aller Agenten werden in einer Kollektion gespeichert (**`robotNames`**), damit jeder Agent während der Simulation angesprochen werden kann. Zusätzlich verfügt der Serveragent über einen direkten Verweis auf den Simulations- und Feldagenten (**`fieldAgent`** bzw. **`simAgent`**), um eine aufgrund des hohen Nachrichtenaustausches unkomplizierte Kommunikation zu ermöglichen. Des Weiteren werden Referenzen auf verschiedene Behaviour-Objekte erzeugt, mit deren Hilfe der Serveragent Daten mit den übrigen Agenten austauscht.

5 Was leistet die Software?

Die Software ist eine Simulationsumgebung für Landminen suchende Roboter. Sie bietet den Entwicklern solcher Roboter die Möglichkeit, Detektionsalgorithmen in einer virtuellen Umgebung zu testen und gegebenenfalls zu verbessern.
Da eine Nachbildung der Realität aufgrund der zu hohen Komplexität nicht möglich ist, basiert das Simulationsmodell auf einigen vereinfachenden Annahmen.
Eine der grundlegendsten Vereinfachungen ist die Aufteilung des Spielfeldes in gleichgroße Kacheln. Jede Kachel verkörpert eine 2m² große Fläche. Die Kachelgröße kann aber ohne weiteres im Nachhinein geändert werden. Ein Roboter kann sich auf eine benachbarte Kachel bewegen; innerhalb einer Kachel ist eine Bewegung nicht möglich. Auch eine Drehung kann nicht frei, sondern nur in 8 vorgeschriebene Richtungen (Nord, Nordost, Ost,..., Nordwest) geschehen. Die gesamte Spielfeldgröße ist auf 150 x 150 Kacheln beschränkt. Diese Größe ist für Testzwecke ausreichend und ermöglicht die Verwendung von bestimmten Programmiertechniken (Kacheln werden als JButton dargestellt).
Für jede Kachel sind Sensorinformationen hinterlegt. Benutzt ein Roboter einen Sensor, so werden die angeforderten Daten durch den Simulationsagenten möglichst realitätsnah zurückgegeben. Momentan verfügt das Programm über einen Stoßsensor, der einem Roboter Hindernisse anzeigt, sowie über die 3 gängigsten Detektionssensoren: Fotosensor, Infrarotsensor und Metalldetektor. Weitere Sensoren können im Nachhinein hinzugefügt werden (siehe Bewertung). Viele Umweltfaktoren - wie Temperatur und Wind - bleiben aber unberücksichtigt, weil für eine sinnvolle Einbringung dieser (Stör-)Faktoren die notwendigen Kenntnisse über die formalen Zusammenhänge zwischen den einzelnen Faktoren fehlen.
Ferner bietet die Simulationsumgebung die Wahl zwischen 2 Landschaftstypen: Wüste und Gras. Je nach Wahl ändert sich das Erscheinungsbild des GUI, die hinterlegten Feldinformationen (z.B. für den Fotosensor) sowie der Energieverlust der Roboter beim Ausführen von Aktionen. Jedes Spielfeld kann nur einen Landschaftstyp annehmen.
Neben dem Landschaftstyp kann der Feldtyp für jede Kachel eingestellt werden. Feldtypen sind „Boden“, „Fluss“, „Hindernis“, „Brücke“, „Asphalt“ und „Feldweg“. Je

nach Feldtyp ändert sich der Energieverbrauch beim Durchführen von Roboteraktionen. Auch die Tatsache, dass Minen mit unterschiedlicher Wahrscheinlichkeit bei bestimmten Feldtypen auftreten (z.B. Antipanzer-Mine werden vorzugsweise auf Straßen platziert) kann bei der Spielfelderstellung berücksichtigt werden.

Die Höhenunterschiede werden vereinfacht in diskrete Höhenebenen unterteilt. Je höher die Ebene, desto dunkler wird die Kachel in dem GUI farblich dargestellt. Höhenunterschiede zwischen Kacheln beeinflussen den Energieverlust bei Roboterbewegungen.

Zurzeit sind nur überirdische Minen in die Simulationsumgebung integriert. Durch Austauschen der Bilder im entsprechenden Verzeichnis (siehe Handbuch) können aber ohne weiteres andere Minenarten integriert werden.

Überdies können Roboter ihre Sensoren immer nur auf eine Kachel anwenden; diese ist durch die aktuelle Roboterposition und die –blickrichtung festgelegt. Diese Einschränkung ist durchaus vertretbar, da es bei Minensuchrobotern üblich ist, nur Flächen in unmittelbarer Nähe zu untersuchen. Wird ein optischer Sensor benutzt, so wird ein aus ca. 50cm Höhe aufgenommenes Bild (Foto bzw. Infrarotbild) von der betroffenen Kachel zurückgegeben. Bilder können beliebig der Simulationsumgebung hinzugefügt werden.

Weiterhin ist die Berechnung des Energieverlustes stark vereinfacht. Die Formel bezieht zwar die unterschiedlichen Landschafts- und Feldtypen sowie den überwundenen Höhenunterschied mit ein, jedoch werden Antrieb, Motor und Geschwindigkeit der Roboter vernachlässigt. Die Formel kann aber im Simulationsagenten geändert werden.

Der Benutzer der Simulationssoftware hat die Option, Spielfelder manuell oder automatisch zu erstellen. Die automatische Kartenerstellung generiert in kurzer Zeit zufällige Karten, damit Roboteralgorithmen ausreichend getestet werden können. Extreme Spielfeldsituationen können durch manuelle Kartenerstellung oder durch die Nachbearbeitungsfunktion gestellt werden. Da Karten abgespeichert und geladen werden können, ist es den Programmierern möglich, bestimmte Spielfeldsituationen nach Ausbesserung oder Verbesserung der Roboteralgorithmen nachzustellen.

5.1 Erweiterbarkeit

Die Simulationsumgebung kann in der Version 1.0 aus Komplexitätsgründen und zeitlichen Beschränkungen zu dessen Realisierung nicht alle denkbaren Aspekte einer Landminenerkennung berücksichtigen. Jedoch ist die technische Möglichkeit der Erweiterung durchaus gegeben.

Vorraussetzung ist dabei ein genaues Verständnis des JADE – Frameworks. Zudem ist auch eine fürsorgliche Einarbeitung in die Klassen des mit dieser Arbeit erstellten Simulationsframeworks erforderlich, um die Gefahr von unerwünschten und nicht vorhersehbaren Effekten bei der Ausführung der Simulationsumgebung hinreichend gering zu halten.

Die vorhandene Infrastruktur der Simulationsumgebung erlaubt die prinzipiell „einfache“ Erweiterung um Komponenten wie zusätzliche Sensoren, Integration eines Rundensystems oder die Berechnung von Energiewerten im Simulationsagenten. Allerdings ist zu berücksichtigen, dass z.B. schon im Falle einer Sensorerweiterung zusätzlich auch komplexe Änderungen an den GUIs (Anzeigen von Sensorinformationen) erforderlich sind! Überdies sind bei einer etwaigen Erweiterung der Funktionalität alle Agenten der Administration betroffen, was eine sorgfältige Analyse der bestehenden Interdependenzen einzelner Agenten erforderlich macht. Der Zeitaufwand ist dabei auf Grund der großen Anzahl an beteiligten Klassen unter Umständen recht hoch.

Um die Einarbeitung in das Simulationsumgebungs – Framework zu erleichtern, wird im Anhang 9.2 eine Fallstudie präsentiert. Die dargestellten Diagramme und textuellen Erläuterungen sollen in grundlegende Konzepte der Implementierung einführen.

6 Bewertung von JADE

Das JADE – Framework hat die Entwicklung der Simulationsumgebung sehr beschleunigt. Die in Kapitel 1.4 geschilderten Anforderungen an ein MAS werden von JADE kompromisslos unterstützt.

Auffällig war die moderate Einarbeitungszeit in das JADE – Framework (2 Wochen). Kleinere Funktionsfähige MAS können bereits nach 1 bis 2 Tagen intensiver Einarbeitung implementiert werden.

Die Verwaltung der Roboter- sowie der Administrationsagenten wird komplett vom MAS übernommen. Der Nachrichtenaustausch zwischen Agenten ist hervorragend organisiert. Neben der Unterstützung des FIPA – (Nachrichten-) Standards sind auch ohne größeren Aufwand eigene Standards realisierbar.

Zudem ist es mit Hilfe der JADE – Plattform problemlos möglich, die Simulationsumgebung als verteilte Simulation laufen zu lassen. JADE verwaltet verteilte Agentenplattformen erstaunlich stabil und effizient. Der Aufwand zum initialisieren einer verteilten Agentenplattform ist identisch mit dem Aufwand zum Erstellen einer lokalen Agentenplattform. Vielleicht ist es nicht sofort ersichtlich, warum gerade eine Simulationsumgebung von der Möglichkeit einer verteilten Lauffähigkeit profitiert. Da aber eine Entwicklung von Roboterprogrammen typischerweise von mehreren Personen erfolgt und diese oftmals über ein Netzwerk verbunden sind, ergibt sich hier schon sofort eine Rechtfertigung. Ein Host kann in einem Netzwerk z.B. die Administration verwalten und einzelne Roboter können dann extern von einem anderen Rechner über den Remote – Management – Agent gestartet werden. Dies ermöglicht einen verteilten Test von verschieden Roboter – Algorithmen in ein und derselben Umgebung, bei welchem die „Entwickler“ nicht vor Ort anwesend sein müssen.

Allerdings ergaben sich zu einem recht späten Zeitpunkt der Entwicklung der Simulationsumgebung grundlegende Schwierigkeiten, welche die Grenzen der Anwendungsfelder des JADE - MAS dokumentierten. JADE stellt zwar ein Framework zur Verfügung, um autonome Agenten modellieren und implementieren zu können. Autonome Agenten erscheinen auch im Hinblick auf die Beschreibung von Robotern, welche sich selbstständig in einem Minenfeld bewegen können, zunächst als eine geeignete Abstraktion zum Zwecke der Modellierung einer Simulationsumgebung für Minensuchroboter. Jedoch zeigte

sich, dass JADE für dieses Vorhaben zu sehr abstrahierte und einzelne spezielle Aspekte von Simulationsumgebungen zu wenig unterstützen konnte.

Durch das überaus *Thread - orientierte* Design der JADE - Plattform war es von vornherein sehr schwer, ein geeignetes eigenes Simulations - Framework zu entwickeln, auf dessen Basis Minensuchroboter programmiert werden konnten. Größte Schwierigkeit hierbei war, von dem *Verhalten* eines JADE - Agenten zu einem Befehl eines Roboters zu abstrahieren. Rückgabewerte z.B. von Sensordaten sind nicht direkt über die so genannten *Behaviours* realisierbar. Zudem ergibt sich aus der Nebenläufigkeit der Agenten - Agenten ein Zuordnungsproblem: „Wie können Roboterbefehle in der vorgesehenen Reihenfolge bearbeitet werden, wenn JADE derartige Befehle quasi gleichzeitig versucht, abzuarbeiten?“. Der *Widerspruch* von thread-bedingter *Nebenläufigkeit* von Aktionen/ Verhalten und dem allgemein anzutreffenden *sequenziellen Abarbeiten* von Befehlen bei Robotern ist nur „schwer aus der Welt zu programmieren“! Abhilfe konnte nur durch die zusätzliche Entwicklung eines *Scheduling-mechanismus* geschafft werden. Allerdings gestaltete sich dies als die zentrale Schwierigkeit bei der Simulationsumgebungs – Entwicklung.

JADE ist im Nachhinein als Unterbau für eine Simulationsumgebung nur nach erheblichen Klimmzügen zu rechtfertigen. Jedoch ist JADE ein vorbildliches MAS, sofern es nur seinem „bestimmungsgemäßen“ Anwendungsgebiet (Supply-Chain-Management, Internet-Bots…) zugeführt wird. Dann erweist sich JADE nämlich als höchst programmierfreundlich und sehr effizient. Zudem ist JADE offen bezüglich zukünftiger Standards und kann sogar recht problemlos in geeignete (JAVA-) Applikationen eingebettet werden.

7 Abbildungsverzeichnis

8 Literaturverzeichnis

David M. Geary: *Graphic JAVA 2.0 – Die JFC beherrschen (Swing)*, Band 2, 3.Auflage, Markt+Technik, 2000

http://www.landmine.de
http://sun.java.com
http://jade.tilab.com
http://www.fipa.org

Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB, formerly CSELT)
Giovanni Rimassa (University of Parma): JADE PROGRAMMER'S GUIDE

Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT)
Giovanni Rimassa (University of Parma): JADE ADMINISTRATOR'S GUIDE

9 Anhang

9.1 Klassendiagramm - Serveragent

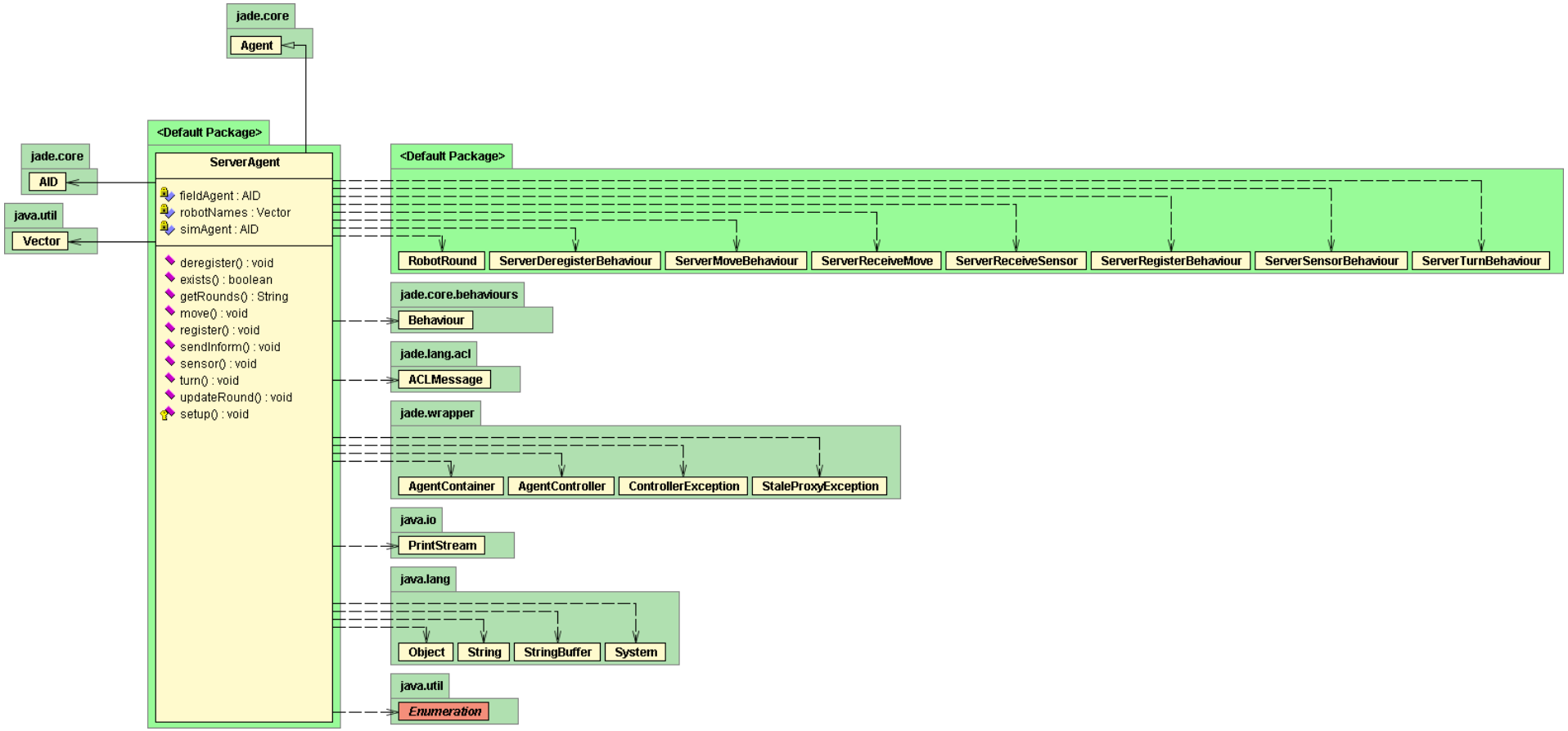

Abbildung 8: Ausschnitt des Klassendiagramms – Severagent

9.2 Fallstudie „Bewegen“

Generelle Überlegungen:

BEWEGEN ist ein Verhalten, das entweder ganz oder gar nicht ausgeführt werden kann (Transaktionseigenschaft). Dies hat folgende Gründe:

- entscheidet ein Roboter im Kontext seines autonomen Charakters, sich zu bewegen, dann ist es nur folgerichtig, ihn auch bewegen zu lassen ohne, dass dieses Verhalten zwischendurch unterbrochen wird (Prämisse unseres Robotermodells)
- nach dem Bewegen ist der Roboter dann wieder autonom in seinen Entscheidungen und kann Nachrichten von anderen Agenten entgegennehmen, Messinstrumente einsetzen oder sich wieder bewegen (dies rechtfertigt die modellinhärente Prämisse und belegt damit, dass ein transaktionales Bewegen nicht den Autonomiecharakter der beteiligten Roboteragenten einschränkt) => ein Roboter entscheidet sich autonom für BEWEGEN und ist nach dem Bewegen ebenfalls wieder fähig autonom zu handeln
- der Transaktionscharakter von BEWEGEN wird durch unser Kommunikationsmodell mit *Quittungen* hervorragend unterstützt

Im Folgenden sei der idealtypische Ablauf einer BEWEGEN – Aktion wiedergeben:

➔ Roboter entscheidet sich für das Verhalten BEWEGEN

➔ er ruft die Methode move() auf; innerhalb dieser Methode wird der Befehl in *RunRobotBehaviour* in eine „Warteliste“ geschrieben (Scheduling)

➔ wählt der Scheduler nun den BEWEGEN – Befehl, wird dem Roboter – Agent das Verhalten *RobotMoveBehaviour* hinzugefügt und zusätzlich sein Zustand *actionState* auf *true* gesetzt

➔ innerhalb von *RobotMoveBehaviour* wird eine Anforderungsnachricht an den Server gesendet; danach wechselt das verhalten in den Zustand zwei, in welchem es mit Hilfe von blockingReceive() auf eine Quittung vom Server wartet, in der Bestätigt wird, das die Anforderungsnachricht eingegangen ist; sollte diese nach einer vorgegebenen Zeitspanne nicht

eingegangen sein, wird der Vorgang wiederholt; sonst geht es weiter mit dem nächsten Schritt

- ➔ Roboter schickt MOVE - Daten und wartet jetzt auf endgültige Antwort vom Server
- ➔ ist die Antwort für eine Bewegen-Aktion positiv, wird entsprechend den Werten in der Antwortnachricht der Energiewert vom Roboter angepasst sowie dessen Landkarte (GUI) aktualisiert und *actionState* auf *false* gesetzt
- ➔ ist die Antwort negativ (weil Roboter schon am Rand der Karte) wird lediglich die Karteninformation aktualisiert und *actionState* auf *false* gesetzt

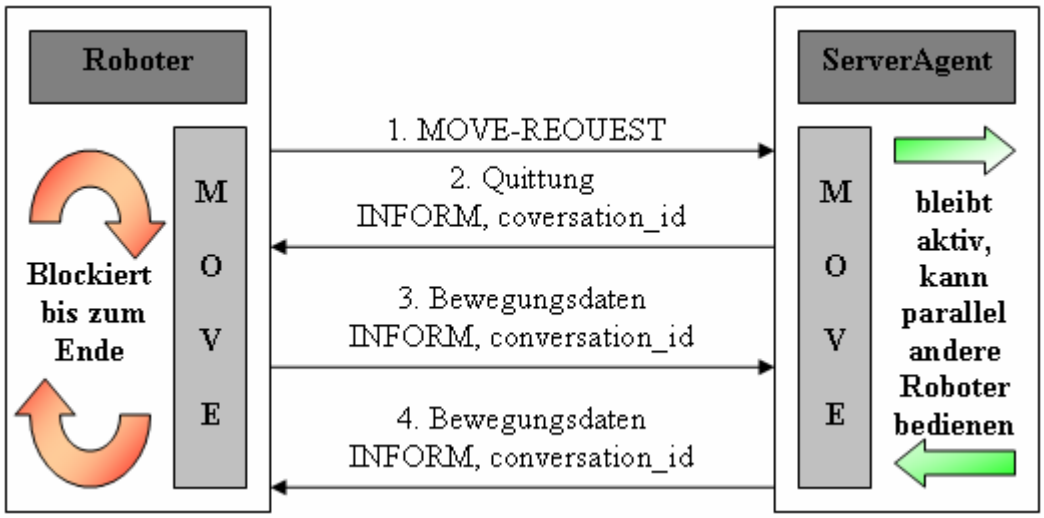

Abbildung 9: Nachrichtenaustausch beim Use-Case BEWEGEN

Neben dem Roboter – Agenten ist während der BEWEGEN – Aktion die gesamte Administration beteiligt. Der Server dient lediglich als Vermittler und Koordinator der BEWEGEN – Aktion.

Eine *Session* wird hierbei via *conversation_id* in den Nachrichten ermöglicht. Während der gesamten Aktion verwaltet der Server die conversation_id's mehrerer Roboter – Agenten für den Use-Case BEWEGEN.

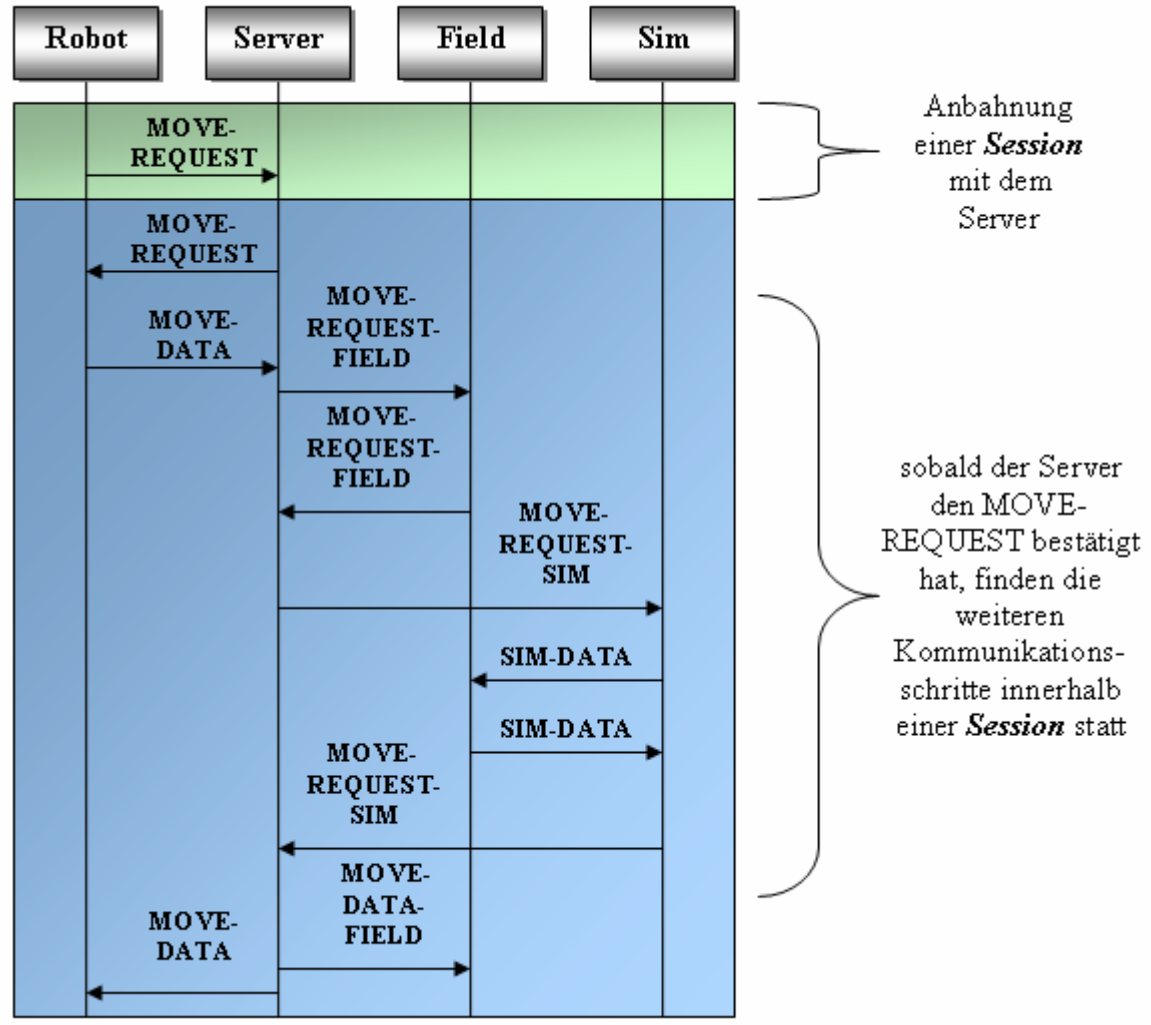

Abbildung 10: Interaktionsdiagramm BEWEGEN

Diagramm „Drehung“

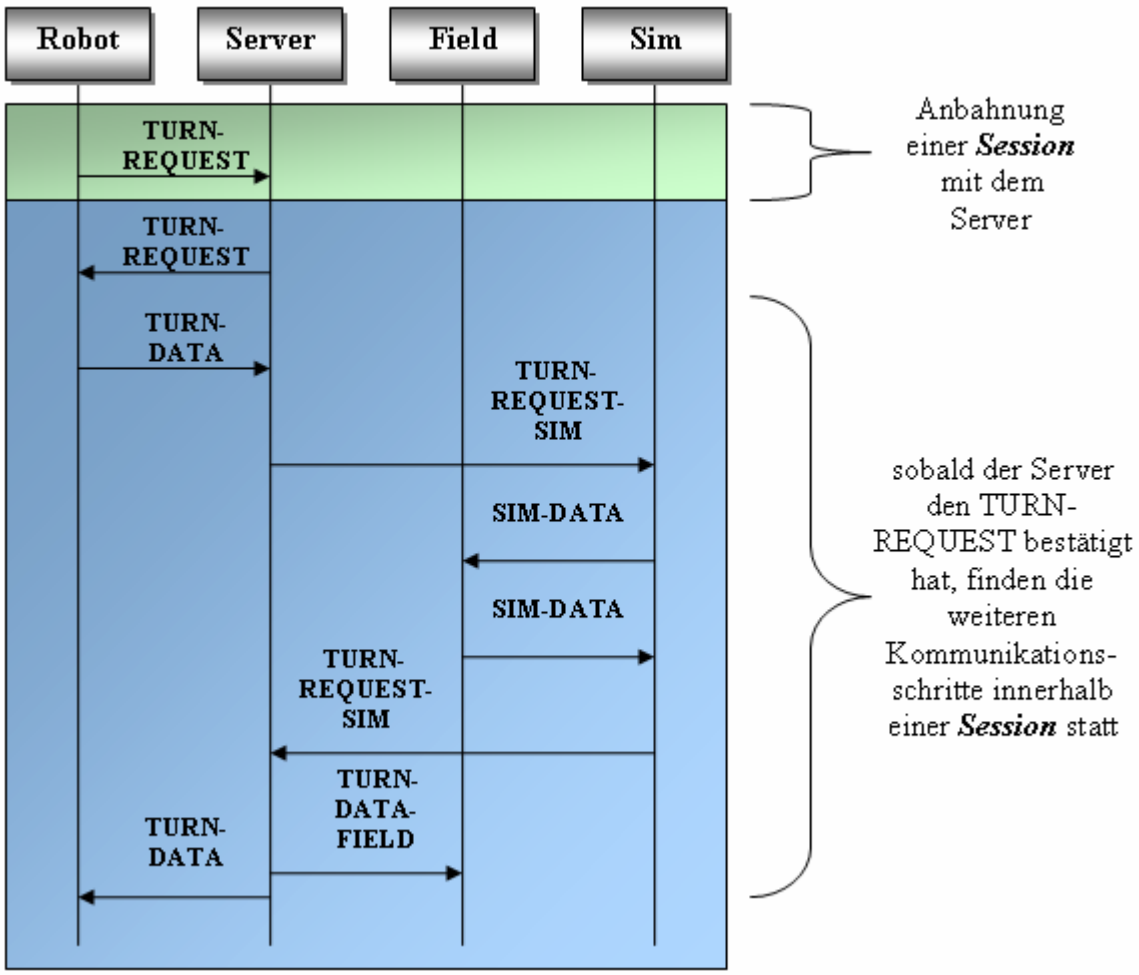

Abbildung 11: Interaktionsdiagramm DREHUNG

„Sensor Benuzen“

Verantwortliches Verhalten hierbei ist das *RobotSensorBehaviour*

- ➔ zuerst vom Verhalten *RobotSensorBehaviour* eine Nachricht vom Typ REQUEST mit der Sprache SENSOR-REQUEST mit Inhalt „*Sensortyp*“ an den Server geschickt
- ➔ dessen Verhalten *ServerReceiveSensor* initialisiert das Verhalten *ServerSensorBehaviour*, welches dem RobotAgent antwortet

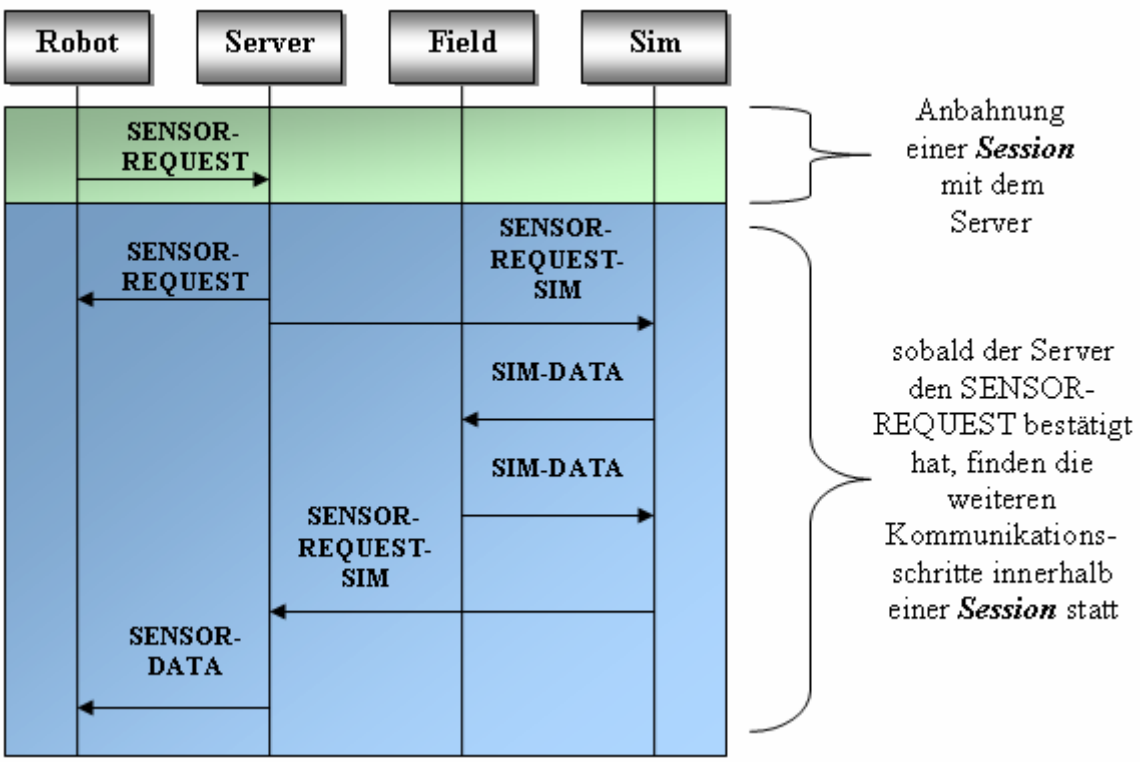

Abbildung 12: Interaktionsdiagramm Sensor