Thema:

Implementierung eines Frameworks zur Simulation einer Agenten-basierten Minensuche

Ausarbeitung
im Rahmen des Seminars
„Unterstützung von Landminendetektion durch Bildanalyseverfahren und Robotereinsatz“

im Fachgebiet Informatik
am Institut für Informatik, Computer Vision and Pattern Recognition Group

Themensteller: Prof. Dr. Jiang
Betreuer: Dr. Dietmar Lammers

vorgelegt von:

Boris Beumers	Heiner Frühling	Jochen Olejnik
Steinfurter Straße 101a	Von-Kluck-Straße 5	Steinfurter Straße 61
48149 Münster	48151 Münster	48149 Münster
0179 / 7060987	0179 / 1248652	0251 / 2842732
bbeumers@uni-muenster.de	heiner.fruehling@web.de	olejnik@uni-muenster.de

Abgabetermin: 2004-01-05

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Während der kalte Krieg in den Industrieländern mit dem Fall der Sowjetunion 1991 sein Ende fand, leiden insbesondere die Länder der dritten Welt weiter unter den Folgen. In einer Reihe von Stellvertreterkriegen wurden seit dem zweiten Weltkrieg Unmengen von Landminen verlegt, von denen bis heute nur ein Bruchteil gesichert und geräumt wurde. Derzeitige Schätzungen gehen von über 100 Millionen Landminen weltweit als Altlast aus. Obwohl im Rahmen der Ottawa-Konferenz von 1997 mehr als 120 Staaten sich vertraglich verpflichtet haben, auf die Entwicklung, den Einsatz und die Verbreitung von Anti-Personen-Minen zu verzichten, werden sie dennoch ständig weiter entwickelt und aufgrund der geringen Kosten weiter eingesetzt, da wichtige Nationen wie die USA, Russland und China den Vertrag nicht ratifiziert haben. Die Entwicklung von leistungsfähigen Minenräumverfahren ist daher eine Aufgabe von globaler Bedeutung.

Die Räumung der Minen erfolgt derzeit meistens von Hand durch menschliche Minenräumexperten, die unter dem Einsatz ihres Lebens jedes verdächtige Objekt in einem zeitraubenden Prozess einzeln untersuchen, oder durch Spezialfahrzeuge, die die Minen mechanisch zur Explosion bringen. Bedeutende Fortschritte in den Forschungsgebieten der Robotik, künstlichen Intelligenz sowie der Bildgewinnung und -bearbeitung könnten in Zukunft die Grundlage für eine einfachere und ungefährlichere Form der Minendetektion und -räumung bilden. Mit Sensoren bestückte Roboter fahren die Minenfelder ab und übersenden ihre Wahrnehmungen an lernfähige Programme, die anhand erlernter Muster die Landminen erkennen. Im Rahmen der Entwicklung eines solchen Systems erscheint es sinnvoll, den Entwurf bereits vor Baubeginn kostengünstig in einem Simulator zu testen, der die Eigenschaften des Geländes und der Minen realistisch nachbildet. Die vorliegende Ausarbeitung beschreibt den Aufbau und die wesentlichen Eigenschaften eines solchen im Rahmen des Seminars entwickelten Simulationsprogramms für die Minensuche unter Einbeziehung von Agenten. Im zweiten Kapitel wird ein Überblick über die zu simulierende Aufgabenstellung sowie die Architektur der Anwendung gegeben. Das dritte Kapitel befasst sich eingehend mit den einzelnen Modulen und kann als Leitfaden für die Benutzung der Anwendung und Implementierung der Agenten dienen. Die Ausarbeitung schließt mit einer kurzen Zusammenfassung.

2 Aufbau der Anwendung

2.1 Abstraktion der Realität

Unsere Aufgabe war das Erstellen eines Frameworks zur Simulation multiagenten-basierter Minensuche. Um möglichst viele Umweltsituationen abbilden zu können, haben wir versucht das Framework so variabel wie möglich zu machen. Der Benutzer soll möglichst viel modifizieren können. Dabei ist für die Modifikationen, mit Ausnahme des Roboterhirns, kein Programmieraufwand nötig, da diese Änderungen über eine GUI geschehen. In der realen Welt gibt es eine Vielzahl von Objekten, die Einfluss auf einen Minensuchroboter haben können. Da diese nicht alle berücksichtigt werden können haben wir einige vereinfachende Annahmen getroffen. Unsere Welt besteht aus Kacheln, wobei ein Roboter genau auf einer Kachel steht und auch genau eine Kachel groß ist. Das Gelände besitzt diskrete Höhenstufen. Ein Roboter wird aus drei Komponentenarten zusammengesetzt. Er besitzt genau einen Antrieb, „beliebig“ viele Sensoren und genau eine Steuereinheit in Form eines Agenten. Der Agent wurde zuvor vom Benutzer programmiert. Der Antrieb und die Sensoren wurden vom Benutzer über die GUI angelegt. Welche Parameter im Einzelnen modifiziert werden können, wird weiter unten genau beschrieben.

2.2 Softwarearchitektur

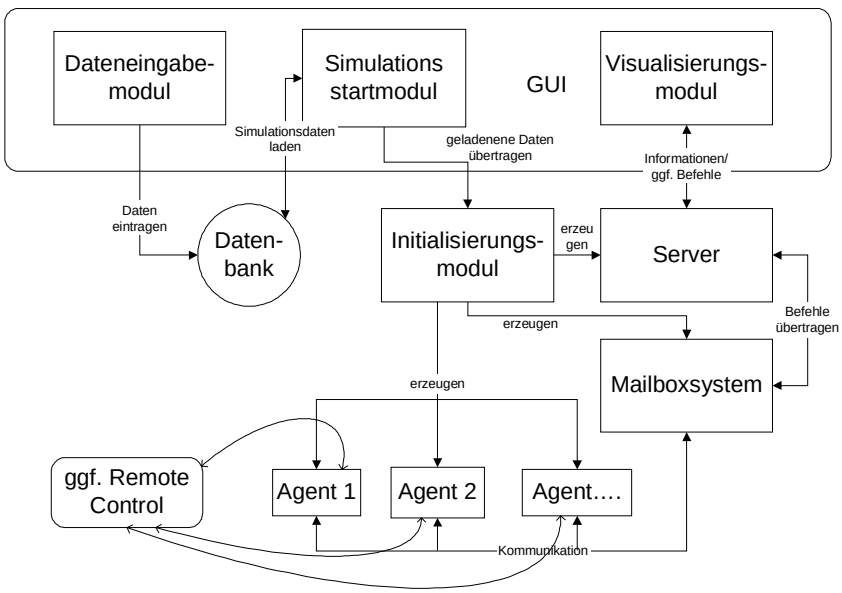


Abb. 2.1: Softwarearchitektur

Die obige Softwarearchitektur beschreibt die Funktionsweise unserer Software. Sie dient auch als roter Faden durch unsere Ausarbeitung.

Im Vorfeld der Simulation werden im Dateneingabemodul Sensoren, Antriebe und Roboter, bestehend aus einem Antrieb, ein oder mehreren Sensoren und einigen eigenen Attributen, erstellt. Ebenso werden die Spielfelder erstellt, auf denen dann später die Simulation abläuft. Über das Dateneingabemodul wird die Datenbank gefüllt. Im Simulationsmodul wird das Spielfeld aus der Datenbank geladen. Den Agenten werden Roboter zugewiesen. Diese erhalten dann auch ihre Startposition auf dem Spielfeld. Sind diese Daten fertig eingegeben, werden sie an das Initialisierungsmodul übergeben. Das Initialisierungsmodul erzeugt nun für den Server, das Mailboxsystem, sowie für jeden einzelnen Agenten (Roboter-Agent-Zuordnung) einen eigenen Thread. Dem Server wird das Spielfeld übergeben. Wenn der Benutzer im Roboterhirn programmiert hat, dass ein Agent eine Benutzerschnittstelle braucht, so wird auch diese erzeugt. Nun beginnt die eigentliche Simulation.

Die Kommunikation zwischen Agenten und Server und den Agenten untereinander findet ausschließlich über das Mailboxsystem statt. Möchte ein Agent eine Aktion ausführen, wie sich bewegen, scannen oder eine Mine räumen, sendet er dies als Nachricht an der Server und dieser verarbeitet dann die Anfrage, abhängig vom Spielfeld und den vorher getätigten Einstellungen. Die Agenten fangen nach ihrer Initialisierung an, ihren Algorithmus abzuarbeiten. Das Visualisierungsmodul zeigt dem Benutzer den aktuellen Stand der Simulation. Die dazu benötigten Daten werden vom Server geliefert. Die Simulation läuft dann so lange weiter, bis der Benutzer sie abbricht. In den folgenden Abschnitten werden die einzelnen Module der Anwendung genauer beschrieben.

3 Module der Anwendung

3.1 Dateneingabemodul

In diesem Kapitel werden die Einstellungsmöglichkeiten der einzelnen Elemente der Simulation anhand der Eingabemasken im Dateneingabemodul erläutert. Ein Antrieb hat einen Namen und ein Gewicht. Abhängig vom Gelände werden Passierbarkeit, Energieverbrauch beim Überfahren und Geschwindigkeit festgelegt. Ein Sensor hat einen Namen und ein Gewicht. Für jedes Gelände ist die Scan-Dauer und der Energieverbrauch pro Scan einstellbar. Zusätzlich kann festgelegt werden, ob der Sensor durch dieses Gelände hindurch scannen kann. Auf einem Feld kann dann der Sichtbereich dieses Sensors zusammengeklickt werden.

Jede Kachel dieses Sichtbereiches hat eine bestimmte Sensorebene. Durch unterschiedliche Sensorebenen kann dargestellt werden, dass der Sensor für die gleiche Kachel bei unterschiedlicher Entfernung zwischen Sensor und Objekt unterschiedliche Werte liefert. Das rote R steht dabei für einen Roboter, der nach Norden schaut. Relativ zu diesem Roboter muss angegeben werden, auf welchen Feldern welche Sensorebene gesehen wird. Scannt der Roboter in eine andere Richtung wird dieser Sichtbereich später entsprechend rotiert. Durch diesen ziemlich frei einstellbaren Sensorbereich kann genau spezifiziert werden wie weit dieser Sensor scannt und in welche Richtung er dies tut (z.B. rundum oder scheinwerferförmig nach vorn oder nur in die Ferne oder nur das Feld auf dem der Roboter steht, etc.).

Ein Geländetyp hat wieder einen Namen. Das Bild, was man angeben muss, ist das Bild, mit dem diese Geländeart im Visualisierungsmodul und im Geländeeditor angezeigt wird. Jedes Gelände hat eine Länge und eine Breite in Kacheln. Auf diese Weise kann man beliebig große rechteckige Objekte (z. B. Häuser, Brücken, Mauern) erstellen. Ein Geländetyp besitzt eine von drei Höhenstufen. Höhenstufe null ist dabei flach, eins mittelhoch (ca. 1m) und zwei hoch. Nun muss noch für jede Sensorebene jedes Sensors eine Sensormeldung angegeben werden.

Eine Sensormeldung ist eine Adresse (URL) zu einer beliebigen Datei auf dem Rechner. Scannt ein Roboter mit einem Sensor ein Gelände so bekommt er als Rückgabewert diese Adresse. Dadurch können verschiedenste Rückgabewerte für verschiedenste Sensoren hinterlegt werden (z. B. Bilder, Histogramme, Sounds, Text, Java-Programme, welche dann dynamisch Werte generieren, etc). Da für jede Sensorebene eines Sensors ein eigener Rückgabewert eingetragen wird kann man hier genau festlegen, wie sich der

Rückgabewert mit den Ebenen ändert. Die Dateien, auf welche die Adressen verweisen, müssen vorher vom Benutzer erstellt werden. Außerdem muss das Roboterhirn wissen, um welchen Dateityp es sich handelt, um die Datei, auf welche diese Adresse verweist, richtig laden und interpretieren zu können.

Ein Roboter hat einen Namen, einen Energievorrat, ein Gewicht und ein Bild (die Sendeleistung wird zwar in die Datenbank geschrieben, aber vom Server nicht verarbeitet). Das Bild dient wieder zur Visualisierung während der Simulation. Der Roboter bekommt genau einen Antrieb und eine beliebige Anzahl an Sensoren, wobei er jeden Sensor höchstens einmal besitzt. Das Gewicht des Gesamtroboters ergibt sich dann aus der Summe des hier angegebenen Gewichtes und der Gewichte des Antriebes und der Sensoren.

Eine Mine hat einen Namen und ein Bild auf dem Spielfeld. Nun bekommt jede Mine für jede Sensorebene jedes Sensors und jedes Gelände einen Rückgabewert. Dieser Rückgabewert ist eine Adresse auf dem Rechner, wie auch der Rückgabewert bei den Geländearten und beschreibt das, was beim Scannen zurückgegeben wird.

Jedes Spielfeld hat eine bestimmte Länge und Breite, sowie ein Standardgelände, mit dem das Feld zu Anfang gefüllt ist. In den einzelnen Kacheln befinden sich Zahlen, welche für die Höhenstufe dieser Kachel stehen. Es kann für jede Kachel eine von 10 Höhenstufen gewählt werden, der Standardwert ist null. Im Spielfeldeditor kann man zwischen den Modi „Zeigen“ und „Schreiben“ wechseln. Klickt man im Zeigenmodus auf eine Kachel, so stellen sich die Drop-Down-Menüs für Höhe, Mine und Gelände auf den Wert der angeklickten Kachel. Im Schreibenmodus werden der angeklickten Kachel die in den Drop-Down-Menüs eingestellten Werte zugeordnet; jedoch nur die Werte, die neben „Schreiben:“ ausgewählt sind. So können z. B. ausschließlich die Höhen verändert werden. Mit den Checkboxen neben „Zeigen:“ können die drei Einstellungsmöglichkeiten zur besseren Übersicht ein- bzw. ausgeblendet werden. Das Spielfeld lässt sich in sechs Stufen zoomen. Nachdem das Spielfeld benannt wurde, kann es in der Datenbank gespeichert werden.

Es können Umgebungen angelegt werden, denen zusammengehörige Elemente, d. h. Gelände, Minen, Roboter, Sensoren und Antriebe zugeordnet werden. Um die Eingabemasken übersichtlicher zu gestalten werden alle Elemente, die nicht zur aktuellen Umgebung gehören, ausgeblendet. Es ist immer genau eine Umgebung im Eingabemodul aktiv.

3.2 Simulationsstartmodul

Um eine Simulation zu starten müssen die dazu erforderlichen Daten über das Simulationsstartmodul eingegeben werden. Unter „Datei-> Simulation“ starten befindet sich dessen Eingabemaske. Im Drop-Down-Menü für das Spielfeld wird das entsprechende Spielfeld ausgewählt, auf dem die Simulation stattfinden soll, woraufhin dieses aus der Datenbank gelesen und in Objekte geladen wird. Dieses Spielfeld ist ebenfalls zoombar und es können einzelne Komponenten ein- bzw. ausgeblendet werden. Im Agent-Drop-Down-Menü sind die Namen aller Agenten gelistet, die zur Robotersteuerung verwendet werden können. Jedem Roboter werden ein Roboterobjekt[1], ein Agent, eine Startblickrichtung und ein Name zugeordnet. Der Agent kann dann im Folgenden diesen Roboter steuern. Durch den Knopf „Roboter einfügen“ wird der Roboter unter seinem Namen in das Auswahlfeld eingetragen. Wählt man ihn dort aus, kann man ihn daraufhin auf dem Spielfeld platzieren. Auf diese Weise werden alle Roboter angelegt und auf dem Feld platziert. Die Roboter werden beim platzieren auf dem Spielfeld in Objekte geladen. Die so geladenen Objekte werden dann an das Initialisierungsmodul übergeben.

3.3 Initialisierungsmodul

Die Initialisierung der Simulation und die Kontrolle des Ablaufs werden vom SimulationController[2] übernommen. Zunächst wird der Gameserver instantiiert und mit dem Spielfeld initialisiert. Dann erzeugt ein AgentLoader anhand der Klassennamen der gewählten Agenten die konkreten Instanzen, versorgt diese mit Informationen über die von ihnen kontrollierten Robotertypen und erzeugt die entsprechende GUI, falls der Agent eine benötigt. Der Agent erhält die Informationen über den von ihm kontrollierten Roboter in Form eines Informationsobjektes RoboterInfo. Über die Ausgestaltung dieses Informationsobjektes kann bei einer konkreten Simulation entschieden werden, welche Informationen den Agenten von Beginn an zur Verfügung stehen[3]. Sinnvolle Daten sind die derzeitige Ausrichtung des Roboters, der Energievorrat und die zur Verfügung stehenden Sensoren. Zusätzliche Informationen

[1] Im Roboterobjekt sind die Daten des Robotertyps wie Antrieb, Sensoren und den zum Roboter gehörenden Attributen, welche in der Robotereingabemaske eingetragen wurden enthalten.
[2] Die Klasse, die das Initialisierungsmodul beinhaltet heißt SimulationController.
[3] Die Aktualisierungsmöglichkeiten der Informationen werden im Server und den entsprechenden Protokollen implementiert.

können Energieverbräuche der einzelnen Komponenten des Roboters oder die aktuelle Position auf dem Spielfeld sein. Dieses RoboterInfoObjekt kann auch zu einem späteren Zeitpunkt vom Server erfragt werden um z. B. den aktuellen Energiestand zu erfragen.

Abschließend wird das Mailboxsystem initialisiert. Der Vorgang schließt mit der Erzeugung und dem Start der einzelnen Threads ab. Eine grafische Darstellung findet sich im Anhang. Der SimulationController übernimmt auch das Entfernen eines Roboters aus dem Simulationsablauf, falls dieser auf eine Mine fährt.

3.4 Mailboxmodul

Der gesamte Simulationsablauf basiert auf dem asynchronen Austausch von Nachrichten. Bei der Initialisierung wird daher jedem Agenten eine Mailbox zugewiesen, die dem Agenten die Interaktion mit seiner Umwelt ermöglicht. Sie wird durch die Klasse AgentMailbox implementiert. Sie enthält die Default-Methoden, die für das Senden von Nachrichten im System benutzt werden müssen und sich deshalb auch in der Klasse *Agent* wieder finden. Für die Kommunikation unter den Agenten stehen eine Broadcast-Methode sowie eine Methode zur direkten Kommunikation mit einem einzelnen Agenten über seinen Namen zur Verfügung[4]. Für die Kommunikation mit dem zugeordneten Roboter und der Gewinnung von Sensormeldungen sendet der Agent Steuernachrichten an den GameServer. In jede ausgehende Nachricht wird automatisch der Absender durch die Mailbox eingefügt.

Das Grundelement der Kommunikation ist das in Abb. 3.1 dargestellte Message-Objekt. Neben Absender und Empfänger kann eine Nachricht einen beliebigen Inhalt vom allgemeinen Typ *Object* haben. Dies ermöglicht dem Entwickler das Versenden von beliebig strukturierten Inhalten, wie z. B. das in Abb. 3.1 dargestellte ScanResult-Objekt, dass ein standardisiertes Format für die Rückgabe beim Benutzen eines Sensors definiert.

[4] Der Name kann über einen Broadcast ermittelt oder bei einer Antwort aus der Nachricht entnommen werden.

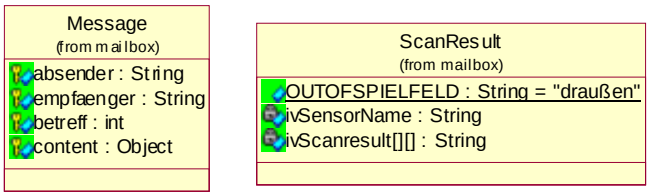


Abb. 3.1: Klassendiagramm für die Nachrichtenobjekte

Um die Kommunikation zwischen den beteiligten Agenten und dem Server sinnvoll zu regeln, müssen Protokolle definiert werden, die festlegen, welche Nachrichten in welcher Reihenfolge von welchen Beteiligten im Hinblick auf ein durch die Kommunikation zu erreichendes Ziel ausgetauscht werden. Ein Protokoll wird im dargestellten Framework an zwei Stellen implementiert. Zum einen spielt für die Protokollimplementierung der Betreff einer Nachricht eine wichtige Rolle. Er enthält in Analogie zum Internet-Protocol numerische Konstanten, die eine feste Bedeutung haben. Zusätzlich zum Betreff kann auch der Inhalt Bestandteil einer Protokollvereinbarung sein wie im Beispiel des Sensor-Scan-Ergebnisses. Zum anderen muss das Protokoll in den Verarbeitungsschleifen der beteiligten Kommunikationspartner implementiert werden[5]. Protokolle, die die Kommunikation zwischen den Agenten regeln, sind eine entscheidende Voraussetzung für die in der Einführung des Kapitels postulierte Kollaboration bei der Problemlösung. Da Protokolle jedoch anwendungsspezifisch sind, ist es Aufgabe des Anwenders diese zu erstellen. In unserem Framework sind lediglich die Protokolle für die Kommunikation zwischen Agent und Server festgeschrieben, die den Roboter steuern und Sensor-Scans durchführen.

In der Klasse Message ist ein verbindlicher Satz von sprechend benannten Konstanten definiert, die im Quellcode über den Qualifier Message.KONSTANTENNAME einfach aufgerufen werden[6]. Das Sequenzdiagramm in Abb. 3.2 beschreibt einen einfachen Nachrichtenaustausch zwischen Agent und Server, der das Ziel hat, den Roboter auf dem Spielfeld um ein Feld weiterzubewegen. Die einzelnen Beschriftungen bezeichnen den Betreff der Nachricht. Bei Erhalt der MOVEFORWARD-Nachricht leitet der Server z.B. die Bewegung ein und hat den Abschluss der Bewegung dem Agenten mit

[5] Siehe Absätze zur Serverschleife und zur Agentenschleife.
[6] Insbesondere wenn der Editor automatische Quellcode-Vervollständigung unterstützt.

MOVEFINISHED zu bestätigen. Falls sich der Roboter bereits bewegt, ist die Nachricht zu ignorieren und eine Fehlermeldung zu senden[7].

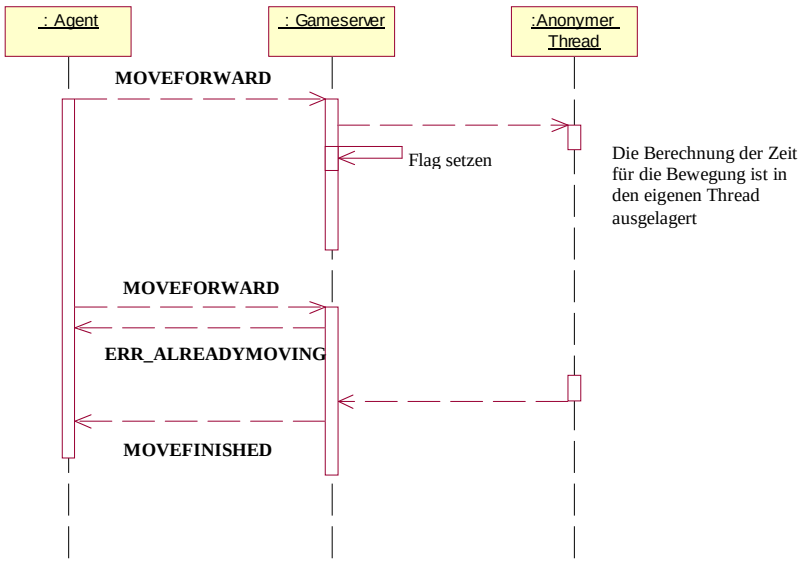


Abb. 3.2: Sequenzdiagramm für die Durchführung einer Bewegung des Roboters

3.5 Agenten

Die Steuerung der in der Simulation platzierten Roboter wird in der Simulation durch einen Software-Agenten durchgeführt. Derzeit existiert in der Literatur keine vereinheitlichte Definition für den Begriff Agent[8], es haben sich jedoch einige wesentliche Eigenschaften herauskristallisiert, die einen Agenten charakterisieren. Ein Agent dient der Lösung einer ihm übertragenen Aufgabe, handelt bei der Lösung (teil-) autonom und kann mit anderen Agenten zusammenarbeiten. Dabei kann er mit anderen Agenten, Systemen, seiner Umwelt und dem Auftraggeber kommunizieren. Agenten werden besonders in verteilten Systemen eingesetzt und haben Zugriff auf verteilte Ressourcen. Das Verhalten eines Agenten kann sich gegebenenfalls im Lauf der Zeit durch Maschinenlernen an geänderte Bedingungen anpassen, so dass ein intelligent wirkendes Verhalten entsteht.

[7] Genaue Details im Kapitel zum GameServer.
[8] Vgl. Kühnel, R.: Agentenbasierte Softwareentwicklung. München 2001, S. 203

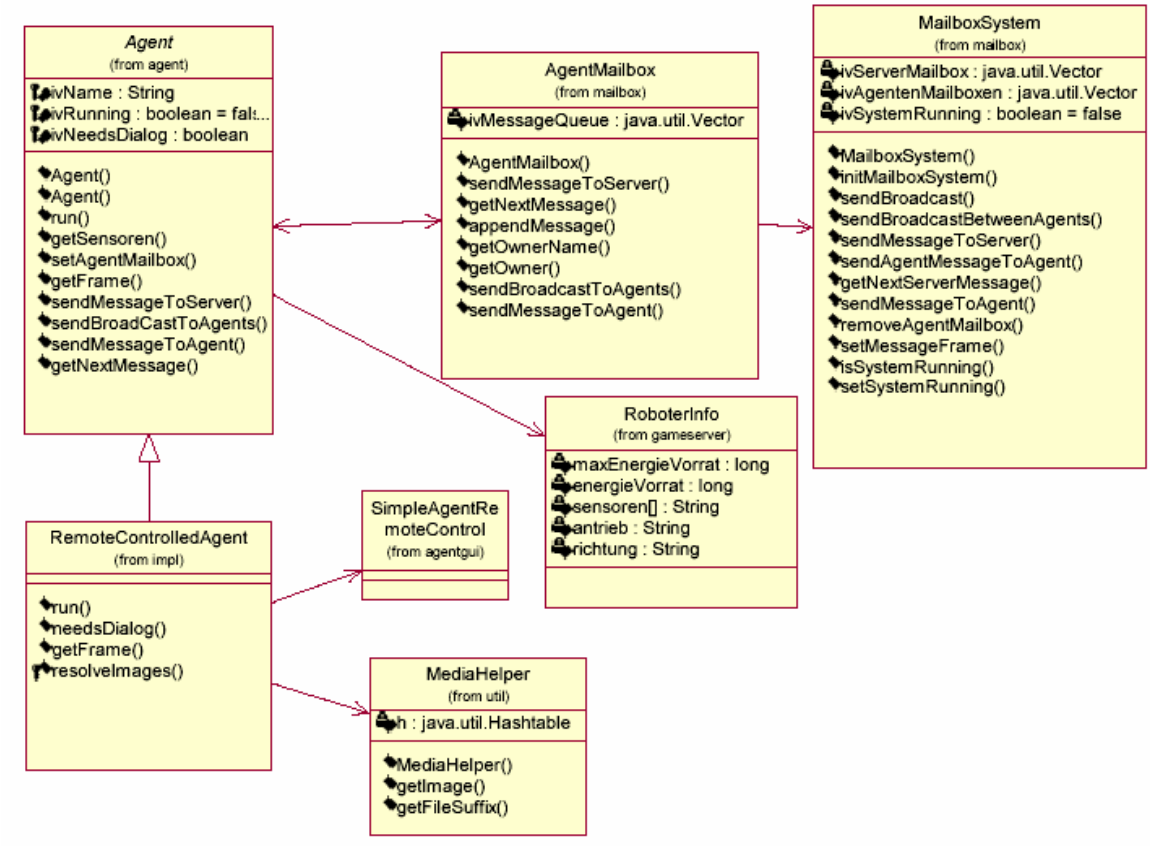


Abb. 3.3: Klassendiagramm für den Bereich des Agenten

Die vorliegende Implementierung stellt keine echte verteilte Umgebung in Form einer Interprozesskommunikation zwischen mehreren unabhängigen Prozessen dar, sondern benutzt aus Vereinfachungsgründen das Java Thread API. Innerhalb des Gesamtprozesses der Simulation wird jeder Agent mit einem eigenen Thread initialisiert, so dass die Zuteilung der Rechenzeit des ausführenden Rechners von der verwendeten Java-Implementierung durchgeführt wird. Das in Abb. 3.3 gezeigte Klassendiagramm zeigt die einem Agentenprogrammierer zur Verfügung gestellte Umgebung des Frameworks. Die Kommunikation zwischen den Agenten untereinander und zwischen Agent und Server geschieht wie weiter oben beschreiben. Jede Implementierung eines Agenten muss von der Basisklasse *Agent* abgeleitet werden. Falls die Implementierung in der Simulationsstart-GUI zur Verfügung stehen soll, besteht die Verpflichtung, diese im Package *agent.impl* abzulegen, da die Anwendung nur dort nach Implementierungen sucht. Die Basisklasse implementiert das Interface *Runnable* aus dem Java Threading API und überschreibt die Methode *run()*. In der abgeleiteten konkreten Agentenimplementierung enthält die *run()*-Methode die komplette Steuerungslogik. Die Variable ivRunning ist ein Flag, das den Agenten von

außerhalb starten und stoppen kann und muss in der *run()*-Methode abgeprüft werden. Über eine Benutzeroberfläche kann eine Möglichkeit zur Kommunikation zwischen Agent und Benutzer geschaffen werden, die bidirektional genutzt werden kann, z. B. falls der Agent eine Entscheidung an den Benutzer delegieren oder der Benutzer den Agenten steuern möchte. Dazu sind die Methoden *needsDialog()* und *getFrame()*, die bei der Initialisierung des Systems per Callback aufgerufen werden, zu überschreiben, sowie eine Implementierung des entsprechenden Dialogfensters zur Verfügung zu stellen. Die im Klassendiagramm zu Demonstrationszwecken aufgeführte Beispielimplementierung stellt einen nicht-autonomen Agenten dar, der die über die Fernsteuerung gegebenen Kommandos des Benutzers in Bewegungen des Roboters umsetzt. Die Klasse *MediaHelper* stellt einen Datencache bereit, um das Laden der Bilder für die Fernsteuerung zu beschleunigen.

Die Logik des Agenten sowie die verwendeten Protokolle werden, wie eingangs erwähnt, durch das Überschreiben der *run()-Methode* implementiert. Die Programmierung entspricht dabei exakt der normalen Programmierung mit Java Threads, wobei das in Abb. 3.4 dargestellte Ablaufschema als Orientierung gelten kann. Der Start der Threads erfolgt durch den SimulationController. Zunächst können einmalig auszuführende Tätigkeiten, wie z. B. die Erzeugung von Hilfsobjekten erfolgen, sofern dies nicht schon in der Initialisierungsphase der Simulation geschehen ist. Anschließend wird eine vom Agentenentwickler zu implementierende Endlosschleife gestartet, die die eigentliche Logik des Agenten ausführt. Zu den Basisaktionen gehört das Abfragen der Mailbox, was zu Beginn eines Schleifendurchlaufs erfolgen kann. Nach Auswertung des Betreffs in einer switch-case Anweisung werden Aktionen ausgeführt wie die Übersendung von Anweisungen an den Roboter, die Kommunikation mit anderen Agenten oder die Auswertung eines Scan-Ergebnisses. An dieser Stelle sind insbesondere auch die Protokolle zu implementieren, so dass jedem Betreff direkt die im Protokoll festgelegte Aktion zugeordnet wird. Falls die durchzuführende Aktion aus komplexen Verarbeitungsschritten besteht, empfiehlt es sich, sie in eine eigene Methode auszulagern, um die *run()-Methode* übersichtlich zu halten. Für Aktionen, die aus einer Vielzahl von elementaren Schritten und Schleifen bestehen oder deren Verarbeitung lange dauert, besteht die Möglichkeit, diese in einen eigenen, anonymen Thread auszulagern. Ein Beispiel für diese Vorgehensweise findet sich in der Implementierung agent.impl.RemoteControlledAgent, wo die Auflösung der

im Scan-Ergebnis übermittelten URLs in die Methode *resolveImages()* verlagert ist, da das Laden der Bilddateien von der Festplatte und die Instantiierung lange dauert. Dieses Vorgehen ist auch für die Anwendung von Bildbearbeitungsalgorithmen zu empfehlen.

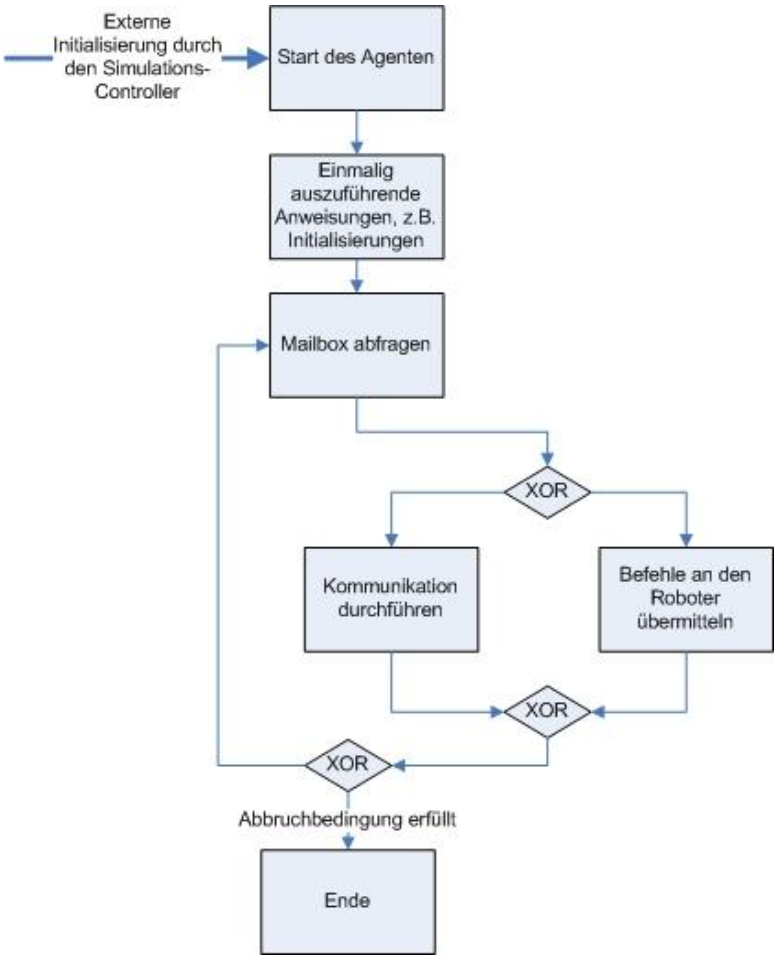


Abb. 3.4: Darstellung der Endlosschleife bei der Programmierung des Agenten

3.6 Server

3.6.1 Aufgabe

Der Server simuliert die Umweltbedingungen für die Simulation und stellt sicher, dass Agenten nur zulässige Aktionen durchführen. Ein Agent kann drei Aktionsarten durchführen. Er kann sich in eine der 8 Richtungen bewegen, in eine Richtung scannen oder eine vor ihm liegende Mine räumen. All diese Aktionen verbrauchen Zeit und Energie. Die Agenten können Nachrichten an die Mailbox es Servers schicken, um eine

bestimmte Aktion zu initiieren. Falls aus der Aktion eine Antwort resultiert, wird diese nach deren Ausführung an die Mailbox des entsprechenden Agenten gesandt.

3.6.2 Interne Objekte

Zum Server gehören verschiedene Objekte, an die Nachrichten verschickt werden, wenn entweder der Zustand der Simulationswelt geändert werden soll, oder eine Information über die Umwelt angefordert wird. Im Einzelnen sind es das Spielfeld, die Roboter, die Minen und die Sensoren. Das Spielfeld bietet dem Gameserver-Objekt die Möglichkeit, Referenzen auf Roboterobjekte durch Angabe des Namens zu erhalten. Das Roboterobjekt enthält die Methoden t*urn(int, int), move(int), removeMine(), getInformation()* und *doScan(String).* Minen und Sensoren werden nicht direkt vom Server angesprochen. Sie sind nur dem jeweiligen Roboter bzw. dem Spielfeld bekannt.

3.6.3 Hauptschleife

Während der Simulation durchläuft der Server eine Schleife, in der er überprüft, ob sich Nachrichten in der Mailbox befinden. Liegt dort eine Nachricht, so wird sie in einer „switch...case“-Anweisung dekodiert. Anhand des Absenders wird das entsprechende Roboterobjekt auf dem Spielfeld ermittelt und auf ihm eine entsprechende Methode aufgerufen. Die Verarbeitung wird so an den Roboter delegiert (z. B. wird die Bewegung vom Roboter durchgeführt und nicht vom Gameserver). Ist die so angestoßene Aktion beendet, veranlasst das ausführende Objekt den Server dem Agenten das Ende der Aktion mitzuteilen. Nachrichten von Agenten müssen als Betreff eines der folgenden Kommandos enthalten:

Anweisung (Message.ANWEISUNG)	*Aktion*
TURNNORTH, TURNWEST usw.	Rotation in die betreffende Himmelsrichtung
MOVEFORWARD, MOVEBACKWARD	Bewegen des Roboters in de entsprechende Richtung
SCAN	Ausführen eines Scans
PING	Erzeugt eine Serverantwort (überprüfen ob Server noch reagiert
GETINFO	Fordert ein Roboterinfobjekt mit den Daten: Energievorratsgröße, Energiestand, Sensornamen, Antriebsname, Blickrichtung.
REMOVEMINE	Entfernen einer Mine in Blickrichtung des

	Roboters

3.6.4 Serverfunktionen

Richtungsbestimmung: Die turn(int newDirX, int newDirY)-Methode

Diese Methode wird vom Gameserver-Objekt aufgerufen, wenn der Roboter seine Richtung ändern soll. Die Richtung wird roboterintern in eine X- und eine Y-Komponente aufgeteilt und mit den Werten -1, 0 und 1 belegt und daher auch so übergeben. Dies erleichtert die Bewegung, da so nur noch die Richtungskomponenten zu den aktuellen Koordinaten addiert werden müssen, um die neue Position zu erhalten. Wird ein falscher Wert übergeben, so wird dieser ignoriert. Wenn der Roboter gerade bereits eine andere Aktion ausführt oder keine Energie mehr besitzt, wird keine Rotation ausgeführt und eine Warnmeldung verschickt. Ansonsten werden die Felder *directionX* und *directionY* überschrieben.

Bewegung: Die move(int direction)-Methode des Roboterobjekts

Die Bewegung wird im folgenden Teil exemplarisch für die drei Zeit beanspruchenden Roboterfunktionen (Bewegung, Scannen, Minen entfernen) genau erklärt. Die *move*-Methode wird vom Gameserver-Objekt aufgerufen, wenn der Roboter bewegt werden soll. Als Wert für *direction* sind nur 1 für „vorwärts“ und -1 für „rückwärts“ möglich, bei falschen Werten wird keine Bewegung ausgeführt. Die Bewegung wird nur begonnen, wenn der Roboter gerade keine andere Aktion ausführt und noch Energie besitzt, ansonsten erhält der zugehörige Agent eine Warnmeldung. Wenn die Bewegung begonnen wird, wird zunächst das Roboterobjekt für weitere Aktionen gesperrt, indem das boolean-Feld *moving* auf „true“ gesetzt wird. Außerdem wird ein anonymer Thread erzeugt, der den weiteren Ablauf der Bewegung steuert. Die *move*-Methode gibt die Kontrolle nach dem Start des Threads sofort zurück an die *run*-Methode des Gameservers; der Thread läuft parallel dazu. Innerhalb des Threads wird zunächst gewartet, bis die zur Bewegung benötigte Zeit verstrichen ist. Diese wird berechnet durch:

$$\mathbf{t_{move} = 1m * (60000ms/min) / v_{antrieb}(Gelände)}$$

t_{move} Zur Bewegung benötigte Zeit in ms

$v_{antrieb}$(Gelände): Geschwindigkeit des verwendeten Antriebs in m/min abhängig vom Gelände

Vom Energievorrat des Roboters wird nun die für die Bewegung benötigte Energie abgezogen. Diese wird berechnet durch:

$$\mathbf{E_{move} = m * g * \Delta h(Gelände_{alt}, Gelaende_{neu}) + E_{Antrieb}(Gelände)}$$

E_{move}	Zur Bewegung benötigte Energie in Millyjoule
m	Gesamtmasse des Roboters in g
$g = 9{,}81\ m/s^2$	Gravitationskonstante
Δh	Höhenunterschied zwischen altem und neuem Gelände
$E_{Antrieb}$(Gelände)	Energie, die der Antrieb zum Befahren der Geländeart benötigt in Millyjoule

Als nächstes wird die Passierbarkeit des Zielfeldes untersucht[9]: Falls das Feld nicht passierbar oder bereits von einem anderen Roboter belegt ist, wird nichts weiter unternommen. Ist das nicht der Fall, wird überprüft, ob sich auf dem Zielfeld eine Mine befindet. Ist dort eine Mine, wird der Roboter aus dem System gelöscht, die *run*-Methode abgebrochen und der Agent angehalten, ansonsten werden dem Roboter die neuen Koordinaten zugewiesen. Am Ende der *run*-Methode wird das Feld moving auf „false“ gesetzt, um den Roboter wieder für neue Aktionen freizugeben. Der Agenten bekommt schließlich noch *Message.MOVEFINISHED*-Nachricht geschickt.

Geradeausfahren: Die Grenzen des Kachelmodells

Da nur Bewegungen von einer Kachel in eine der acht vorgegebenen Richtungen möglich sind, ergeben sich mehrere in Bezug auf die Strecke „optimale“ Wege, um von einem Feld auf ein weiter entferntes zu gelangen (s. Abb. 3.5). Die durch die Grenzen des Parallelogramms beschriebenen Wege und mehrere Wege innerhalb dieses Parallelogramms haben dieselbe Länge. Haben alle Felder dieselbe Geländeart und Höhe, so wird für eine Bewegung auf diesen Pfaden jeweils dieselbe Zeit und Energie benötigt. Es bestände theoretisch die Möglichkeit, innerhalb des Servers eine Methode zum Geradeausfahren zu implementieren, die entweder ein Zielfeld oder einen Winkel als Parameter erwartet, und Vergünstigungen bezüglich Bewegungsdauer und Bewegungsenergie gewährt, sodass die Kosten proportional zur tatsächlichen (euklidischen) Entfernung berechnet werden. Dieser Ansatz führt jedoch zu Problemen, wenn die Bewegung vor Erreichen des Zielfeldes abgebrochen wird oder eine

[9] Dieser Teil der Methode ist in einem synchronized-Block (Monitor für das Klassenobjekt) eingeschlossen, um zu vermeiden, dass zwei Roboter gleichzeitig auf ein freies Feld fahren, oder auf derselben Mine explodieren.

Bewegung auf dem direkten Weg (z. B. durch Hindernisse) nicht möglich ist und wurde daher nicht implementiert.

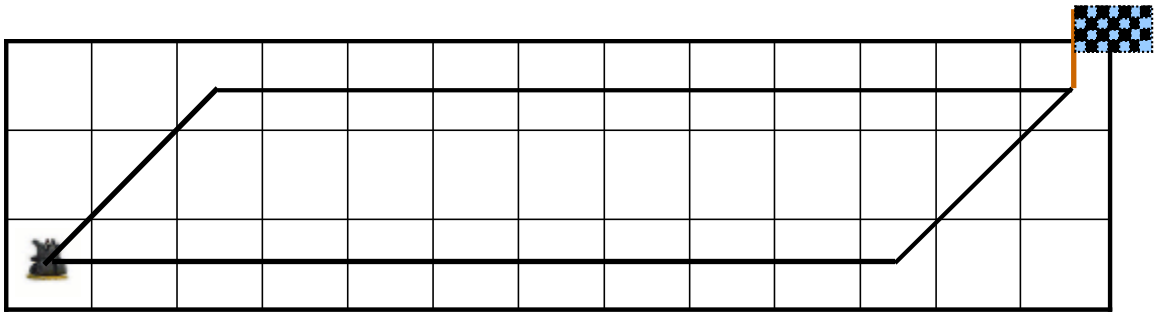


Abb. 3.5: Kürzeste Wege zwischen zwei Feldern

Scannen: Die doScan() Methode des Sensors

Bei einem Scan wird ein zweidimensionales Array mit URLs erstellt, die auf Sensormeldungen für die Felder im Sensorbereich verweisen. Dieses wird dem zugehörigen Sensor nach Abschluss des Scans über den Gameserver geschickt. Die *doScan()-Methode* des Sensors wird von der *doScan(String name)* Methode des Roboters aufgerufen, an die der Gameserver den Namen des gewünschten Sensors als Parameter übergibt. Zunächst wird wie bei der *Move-Methode* sichergestellt, dass noch keine andere Aktion ausgeführt wird. Dann wird der Sensor gesperrt und ein anonymer Thread gestartet, um die Zeit, die für den Scan festgelegt wurde zu messen. Die zum Scannen benötigte Energie ist in der Datenbank vorgegeben und wird daher nicht berechnet.

Um das Sensorbild zu erstellen wird die Schablone um ihren Mittelpunkt in Blickrichtung des Roboters gedreht. Dann wird für jedes in der Schablone als sichtbar markierte Feld berechnet, ob es von anderen Feldern bzw. Objekten verdeckt ist. Dazu wird zunächst der direkte Weg vom Sensor zum zu scannenden Feld bestimmt (s. Abb. 3.6). Dieser entspricht dem durch waagerechte, senkrechte oder diagonale Schritte verfolgbaren Pfad, welcher sich am nächsten an der Strecke zwischen den Mittelpunkten der beiden Felder befindet (schwarze Linie). Da zwei Felder gleich weit von der direkten Verbindung entfernt sein können (senkrecht verbundene Felder), muss in diesem Fall festgelegt werden, welcher Pfad zur Berechnung zu benutzen ist. Es wird in dieser Situation immer dasjenige Feld gewählt, welches sich näher am Roboter befindet.

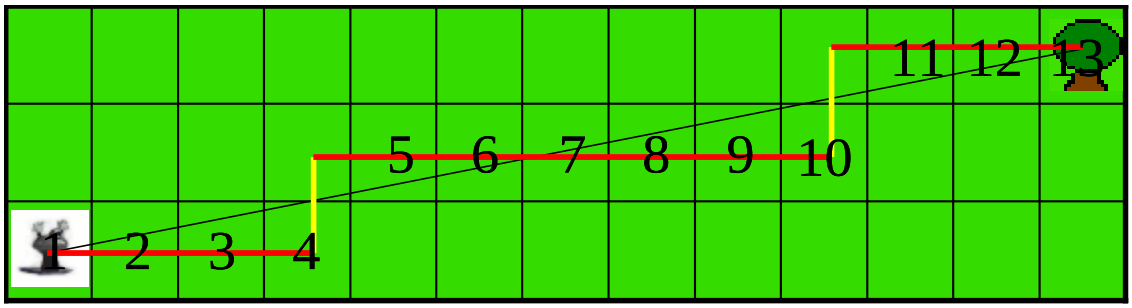


Abb. 3.6: Felder zwischen Roboter und Zielfeld

Entlang des bestimmten Pfades wird untersucht, ob das Zielfeld durch ein Objekt verdeckt wird. Dazu wird für jedes Feld seine Höhe berechnet. Diese setzt sich aus der Höhe des Geländes und, falls das auf dem Feld befindliche Objekt für den Sensor undurchsichtig ist, der Höhe des Objektes zusammen. Es wird nun das Maximum der Steigungen vom Sensor zu den Feldern auf dem Sichtpfad bestimmt. Übersteigt diese Maximalsteigung die Steigung vom Sensor zum Objekt, so ist das Objekt verdeckt und es wird kein Sensorwert für das Zielfeld zurückgegeben. Ist das Feld nicht sichtbar, so wird in dem Array, welches die gedrehte Schablone enthält, das entsprechende Feld als unsichtbar markiert. Nachdem alle Felder auf Sichtbarkeit überprüft wurden, werden anhand der veränderten, gedrehten Schablone Feldern Sensorebenen zugewiesen. Anhand von Sensorebene, Geländetyp und Mine auf dem Feld wird die entsprechende Sensormeldung bestimmt, welche an der entsprechenden Stelle im Rückgabearray eingetragen wird. Nach der Erstellung des Sensorbildes wird der Thread für die der Dauer des Scans entsprechenden Anzahl Millisekunden schlafen gelegt. Da die Berechnung verhältnismäßig viel Zeit in Anspruch nehmen kann, wird zu Beginn der doScan Methode die Systemzeit gespeichert. Die Differenz aus der Systemzeit nach der Berechnung und derjenigen vor der Berechnung wird von der Wartezeit abgezogen.

3.7 Ausgabemodul

Hier sieht der Benutzer das zoombare Spielfeld. Mit den Checkboxen können einzelne Elemente (Gelände, Höhen, Minen, Fog of War[10], Roboter) ausgeblendet werden. Bei eingeschaltetem Fog of War sind nur die Abschnitte der Karte zu sehen, die bereits von einem Roboter gescannt wurden. Über den Messageboxbutton kann man sich alle

[10] Der Fog of War ist eine Schablone, durch die Teile der Karte aus der Anzeige ausgeblendet werden können.

Nachrichten anzeigen lassen, mit denen die Roboter untereinander und mit den Server kommunizieren. Über „Simulation starten“ wird die Simulation in Gang gesetzt, der Simulation stoppen Knopf beendet sie. Mit dem Beendenbutton wird die Simulation beendet.

Während der Simulation kann sich der Benutzer nun ansehen, wie sich seine Agenten auf der vorgefertigten Karte verhalten und welche Nachrichten sie untereinander austauschen. Wenn es Agenten gibt, die eine GUI besitzen, so wird diese auch durch diese Modul aufgerufen.

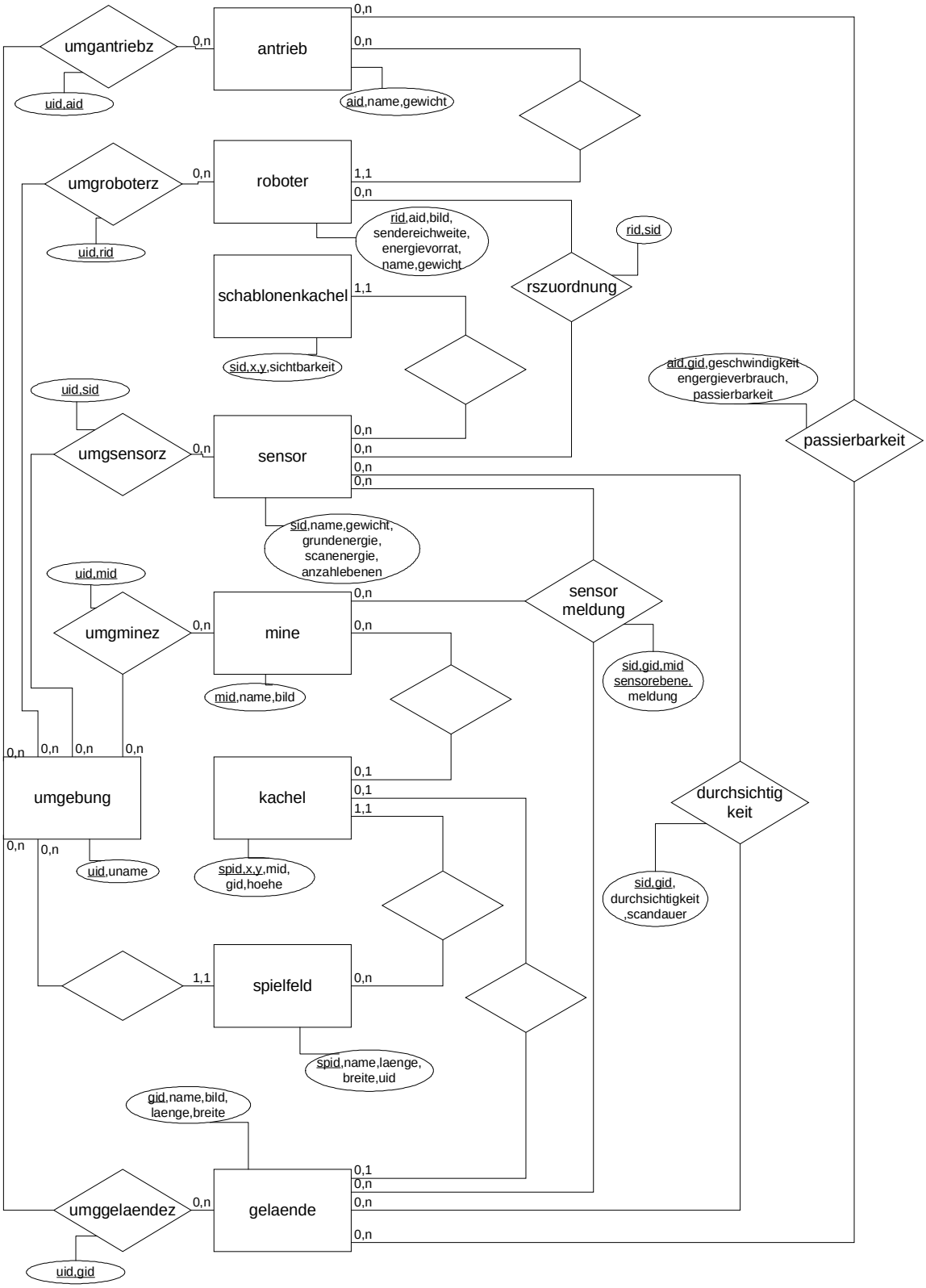
4 Zusammenfassung

Die bei der Minensuche gegebenen Bedingungen sind äußerst komplex und stellen die praktische Umsetzung der Minensuche mit Robotern und Bilderkennungsverfahren vor große Probleme. Das vorliegende Framework versucht, die Komplexität eines solchen Szenarios abzubilden, ohne den Anwender in seinen Möglichkeiten einzuschränken. Durch die Fähigkeit, diverse Antriebe und Robotergrundeinheiten sowie Sensoren mit dem exakten Wahrnehmungsfeld und wesentlichen Eigenschaften zu Robotern zusammenzustellen, sind viele Annäherungen an die Praxis geschaffen. Eine besondere Stärke des Frameworks liegt darin begründet, bei der Abbildung der Sensorrückgabewerte nicht allein auf Bilder beschränkt zu sein, sondern grundsätzlich jeden Datentyp zurückgeben zu können, z. B. Audiodaten oder Wahrheitswerte für die Umsetzung eines Passierbarkeitssensors.

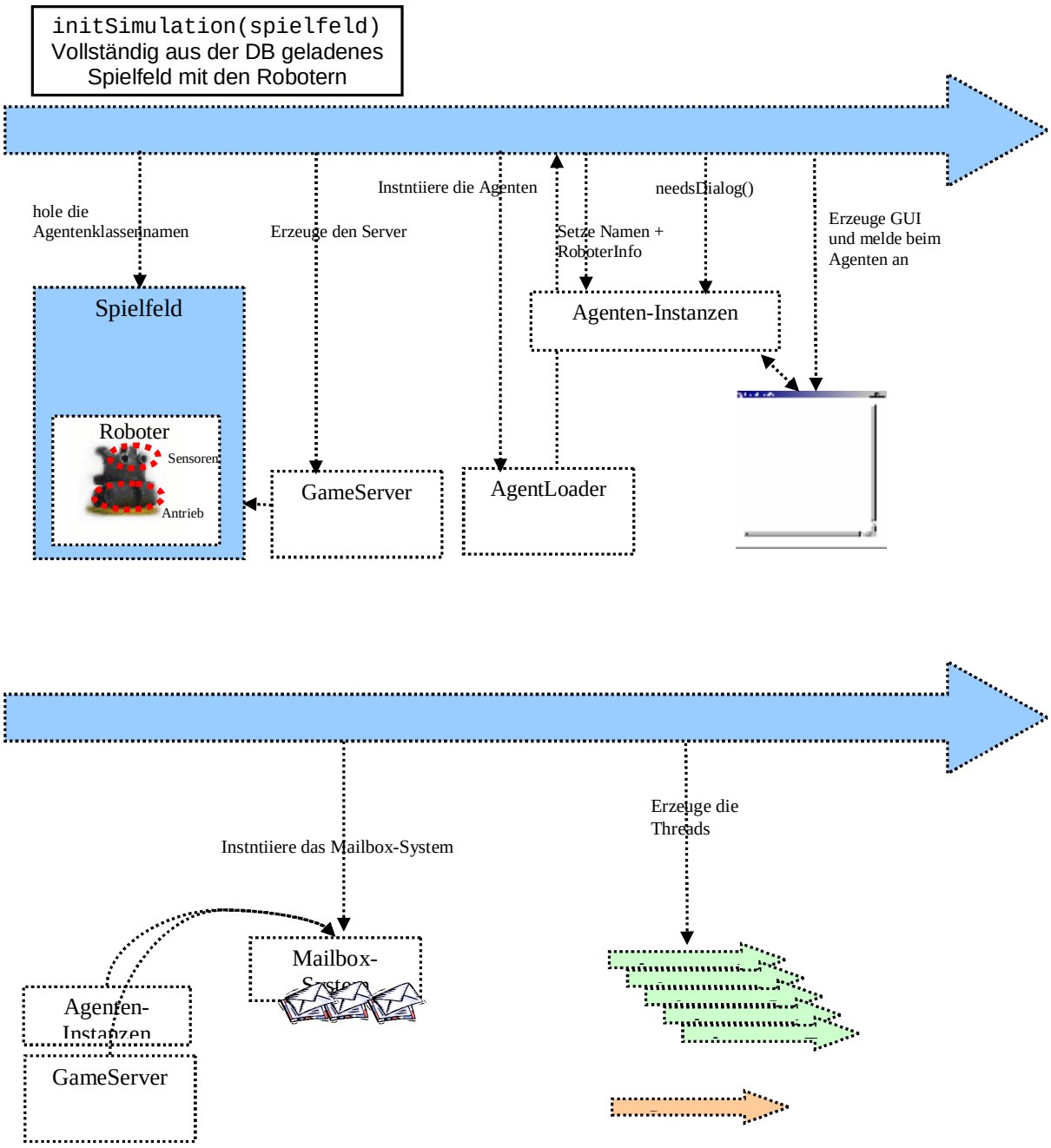
Die weitere Herausforderung bestünde nun darin das Framework weiter auszubauen und Agenten und Umgebungen für dieses zu erstellen.

Anhang

A Entity-Relationship-Model der Anwendung



B Übersicht über den Initialisierungsvorgang



C Installationsanleitung

C.1 Entwicklungsumgebung

Die Simulationsumgebung wurde entwickelt mit der **JDK-Version 1.4.1_02** und der Datenbank **MySQL** in der Version **4.012.** Diese müssen auf dem Rechner vorhanden sein.

C.2 Externe Bibliotheken

1. **JDBC-Connector**
   Als JDBC-Treiber wurde die Open-Source-Implementierung **MySQL-Connector 3.08** benutzt. Die mitgelieferte Zip-Datei ist zu entpacken, das Java-Archiv mit der Bezeichnung ***mysql-connector-java-3.0.8-stable-bin.jar*** ist in das Verzeichnis ***<java-installationsverzeichnis>/jre/lib/ext*** zu kopieren.

2. **Borland-spezifisches Layout**
   Für die Gestaltung des Layouts der Masken wird ein Borland-spezifisches Layout benutzt, das sich im mitgelieferten Java-Archiv ***jbcl.jar*** befindet. Dies ist ebenfalls in das Verzeichnis ***<java-installationsverzeichnis>/jre/lib/ext*** zu kopieren.

C.3 Erstellen der Tabellen

Zum Erstellen der Tabellen kann entweder die mitgelieferte Datei **tables.sql** ausgeführt werden, die die notwendigen SQL-Statements enthält oder die Methode **tabellenanlegen()** in der Klasse **editor.Sql.java** der Anwendung. Bei der zweiten Vorgehensweise müssen die in der Klasse definierten Verbindungsparameter entsprechend angepasst werden.

C.4 Installation und Start

Der mitgelieferte Quellcode ist in ein beliebiges Verzeichnis zu entpacken und zu kompilieren. Die main()-Methode zum Ausführen der Anwendung befindet sich in der Klasse **editor.Editor.java**. Die Simulationsumgebung steht nun mit einer leeren Datenbank bereit.