Fachhochschule Köln
Cologne University of Applied Sciences

Evaluation von Multiagentensystemen zur Erstellung von virtuellen Wegesystemen einschliesslich Erstellung eines Prototypen

Evaluation of Multi Agent Systems for their Use in the Creation of Virtual Road Networks Inclusive of the Development of a Prototype

Bachelorarbeit

ausgearbeitet von

Robert Giacinto

zur Erlangung des akademischen Grades

Bachelor of Science

vorgelegt an der

Fachhochschule Köln
Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

im Studiengang

Medieninformatik

Erster Prüfer: Prof. Dr. Horst Stenzel
Fachhochschule Köln

Zweiter Prüfer: Prof. Dr. Wolfgang Konen
Fachhochschule Köln

Gummersbach, im Februar 2010

Adressen:

Robert Giacinto
Am Sandberg 28
51643 Gummersbach
robert.giacinto@gmail.com

Prof. Dr. Horst Stenzel
Fachhochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
stenzel@gm.fh-koeln.de

Prof. Dr. Wolfgang Konen
Fachhochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
wolfgang.konen@fh-koeln.de

Kurzfassung

Der immer weiter steigende Bedarf an qualitativ hochwertigen, grafischen Inhalten in Computerspielen sorgt dafür, dass neue Wege gefunden werden müssen, um diese mit verfügbaren Ressourcen produzieren zu können. Hier hat sich in einigen Bereichen die Verwendung von prozeduralen Techniken zur computergestützten Generierung von benötigten Modellen bereits bewährt. Ein Bereich, der aufgrund seiner Komplexität besonders interessant für den Einsatz von prozeduralen Methoden ist und im Bereich der Computerspieleindustrie noch am Anfang steht, ist die Simulation und Erzeugung von virtuellen Städten oder anderen urbanen Strukturen.

Ziel dieser Arbeit ist es, ein Systemkonzept zu präsentieren, das auf Grundlage eines erweiterbaren Bedürfnissystems von Personen ein virtuelles Wegenetz erzeugt, das als Folge der Interaktionen der Personen mit der simulierten Welt entsteht. Es soll gezeigt werden, dass einzelne Agenten ohne ein globales Wissen oder eine vorhandene Planungsinstanz nach dem Beispiel von sozialen, staatenbildenen Insekten nur mithilfe einer Kommunikation auf Pheromonbasis zu in Computerspielen einsetzbaren Simulationsergebnissen kommen.

Abstract

The growing need for high-quality graphic content in computer games makes it necessary to find new ways to produce it with already available resources. In some areas, the use of procedural techniques for the computer-assisted generation of required assets has proven its worth. The simulation and generation of virtual cities or other urban structures is especially interesting for the use of procedural methods and is still in the early stages with respect to the computer game industry.

This paper's goal is to present a system design that generates a virtual road system based on an extendable system of people's needs; it develops according to the interactions of the agents with the simulated world. It will be shown that individual agents without any global knowledge or available planning authority will achieve simulation results applicable in computer games, using nothing but pheromone-based communication, following the example of eusocial insects.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Computerspiele für den Massenmarkt haben, gefördert durch die steigende Leistungsfähigkeit von Computern im Consumer Segment, in den letzten Jahren eine große Entwicklung erlebt. Hierunter fällt nicht nur die sich immer weiter verbessernde Qualität Spielegrafik, die von den Konsumenten mit jeder neuen Generation von Grafikprozessoren auch erwartet wird, sondern auch der Anspruch der Spieler in modernen Spielen eine dynamische, interaktive und glaubhafte Welt präsentiert zu bekommen. Von diesen Spielen erwartet der Kunde, dass sie ihm so wenig Begrenzungen oder verdefinierte Wege vorgeben, wie eben möglich, damit er die Welt auf seine eigene Art und Weise erkunden kann. Dies hat zu einer Welle von sogenannten Sandbox-Games, wie die Grand Theft Auto (GTA) Reihe (11), Just Cause (12) oder Prototype (13). Allen Spielen dieser Kategorie ist gemein, dass der Hauptaugenmerk nicht auf die Geschichte des Spiels gelegt wird, sondern auf die Möglichkeit des Spielers, sich so frei wie möglich in dieser virtuellen Welt entfalten zu können und ihm keine Entscheidungen vom Spielverlauf vorgegeben werden. Dies stellt die Entwicklung von Spielen vor vollkommen neue Herausforderungen, bei denen die freie Entscheidung des Spieler, etwas zu tun (oder nicht zu tun) und die Welt nach eigenen Vorstellungen zu erkunden, mit in die Planung einbezogen werden muss.

Unabhängig von dem Genre des Spiels werden in der Proktuktion eines neuen Spiel unterschieliche Kulissen benötigt, um die präsentierte Welt interessant und lebendig wirken zu lassen. Hierunter fallen unter anderem die im Spiel vorhandene Vegetation, die unterschiedliche geologische Beschaffenheit der Landschaft, die ein Spieler erkunden kann, und urbane Strukturen, wie einzelne Häuser, Straßen zwischen Dörfern oder die Dörfer selbst. Diesen neuen Anforderungen an die Entwicklung von modernen Computerspielen kann man nur begebnen, indem die Produktionsprozesse entsprechend angepasst werden, so dass die vorhandenen Ressourcen in den Entwicklungsstudios effizienter genutzt werden können, um eine größere optische Vielfalt und Dynamik der virtuellen Welt zu erreichen.

Die neuen Ansprüche haben dazu geführt, dass in den vergangenen Jahren immer häufiger prozedurale Techniken zur Erzeugung von Inhalten in Computerspielen eingesetzt werden, um Künstler von Routinearbeiten zu entlasten und sie vom Computer, entweder zur Entwicklungszeit des Spiels oder zur Laufzeit, generieren zu lassen. Für einen Überblick über unterschiedliche, in Spielen Verwendung findende prozedurale Techniken, sei auf (14) verwiesen.

In dieser Arbeit soll es speziell um die Erzeugung von urbanen Strukturen ge-

hen, die in Computerspielen eingesetzt werden können. Gerade in Spielen, in denen der Spieler eine frei begehbare Welt erkunden kann, ist es wichtig diese mit Leben und urbanen Strukturen zu füllen, um sie glaubhaft und interessant wirken zu lassen. Hierfür wurden in den letzten Jahren einige Systeme oder Konzepte präsentiert, die Entwicklern helfen können, urbane Strukturen für Spiele, unterstützt vom Computer, zu erzeugen. Eine Übersicht solcher Methoden, basierend auf verschiedenen Techniken, wird in Kapitel2 gegeben.

Die dort vorgestellten Methoden richten sich nicht direkt an die Entwicklung von Spielen, sondern lassen sich zu einem großen Teil im Bereich der Geosimulation, also der Simulation urbaner Phänomene mithilfe des Computers aus der Sicht der Stadtplanung, oder der Computergrafik ansiedeln. Sie bieten aber einen guten Einblick über den derzeitigen Stand der Forschung und wie unterschiedlich der Ansatz sein kann, der zu einer virtuellen Stadt führt. Die in Kapitel2 vorgestellten Arbeiten lassen sich in zwei große Kategorien einteilen. Die Arbeiten von Parish und Müller (Kapitel 2.1) bzw. Greuter et al. (Kapitel 2.2) sind beides top-down Lösungen, die von dem globalen Bild einer Stadt Möglichkeiten untersuchen, diese Strukturen zu beschreiben und zu visualisieren. Kapitel 2.3 beschreibt einen bottom-up Ansatz, wie er in der Geosimulation üblicher ist. Hier werden die urbanen Strukturen als ein emergentes Phänomen verstanden. Es geht von der Idee aus, dass sich ein chaotisches System, das sich in keinem Equilibrium befindet, schwer durch alles umfassende Gleichungssysteme global beschreiben lässt. Aus diesem Grund gibt Kapitel 3 eine Einführung in die der Geosimulation zu Grunde liegenden Gedanken.

Auf Grundlage dieser Gedanken versucht der letzte Teil der Arbeit in den Kapiteln 4 und 5 einen Ansatz zu entwickeln, der die Entstehung eines Straßennetzes auf Basis von lokalen Interaktionen von Agenten mit ihrer Umwelt modelliert. Ein wesentlicher Punkt hierbei ist, dass die Agenten eine rein lokale Sicht auf ihre Welt haben und dennoch Wissen an andere Agenten weitergeben können, ohne einen lokalen Wissensspeicher besitzen zu müssen. Eine Technik, die sich in den letzten Jahren für diese Art von Agentensystemen bewährt hat und in der Entwicklung des Prototyps mit einbezogen wurde, ist die der pheromongesteuerten Agenten (15). Die Ergebnisse und eine ihre Einschätzung werden im letzten Kapitel der Arbeit präsentiert.

2 Marktübersicht

Dieses Kapitel stellt drei unterschiedliche Arbeiten vor, die sich mit der prozeduralen Erzeugung von Städten auseinandersetzen. Jede Arbeit benutzt als Grundlage für ihre Ergebnisse einen anderen Ansatz. Die ersten beiden sind Vertreter einer top-down Perspektive, bei der die Stadt als eine Entität angesehen wird, die prozedural erzeugt werden soll.

Die in Kapitel 2.1 beschriebene CityEngine verwendet eine erweiterte Form von Lindenmayer-Systemen zur Definition der zu erzeugenden Stadt und ist das einzige von den hier vorgestellten Systemen, das kommerziell vertrieben wird. Kapitel 2.2 beschreibt die Undiscovered City, einen von Parish und Müller entwickelten Prototypen, der als einziger Städte in Echtzeit generiert und visualisieren kann. Der CityBuilder in Kapitel2.3 implementiert eine bottom-up Lösung auf Basis eines Multiagenten-Systems, das Erkenntnisse aus der Geosimulation und der Stadtplanung verwendet, um eine Landnutzungs- und Straßennetzkarte zu erstellen.

2.1 CityEngine

Parish und Müller haben in (1) ein System zur prozeduralen Erzeugung von virtuellen Städten vorgestellt, das zum einen leicht erweiterbar, zum anderen mit wenigen Benutzereingaben auskommen soll. Die von ihnen getaufte CityEngine haben sie 2001 das erste Mal auf der SIGGraph präsentiert.

Die CityEngine besteht aus mehreren Submodulen, die für die Generierung einzelner Aspekte der Stadt (Straßennetz, Gebäudeplätze, die Erzeugung der Gebäudegeometrien und der entsprechenden Polygone) verantwortlich sind. Diese Module bilden eine Toolpipeline, an deren Ende die erzeugte, virtuelle Stadt steht.

Der erste Schritt im Prozess der Stadterzeugung besteht darin ein entsprechendes Straßennetz aufzubauen. Hierfür benötigt die CityEngine mehrere Eingaben des Benutzers in Form von Bilddaten. Die Daten, die so an das Programm übergeben werden, lassen sich in zwei Oberkategorien aufteilen. Einmal die geographischen Karten, in denen die Landerhebung und die Art (Wasser, Land, Vegetation) der jeweiligen Fläche definiert wird, und die soziostatistischen Karten, in denen sich Informationen zur Bevölkerungsdichte, der Landnutzung, den zu verwendenden Straßenformen und der maximalen Höhe von Gebäuden dem Programm übergeben lassen.

In diesem ersten Teil der Pipeline wird ein erweitertes L-System verwendet, das sie Self-Sensitive L-System nennen. Dieses trägt dem Anspruch der leichten Erweiterbarkeit Rechnung, indem es die Definition und Anpassung von benötigten Parametern

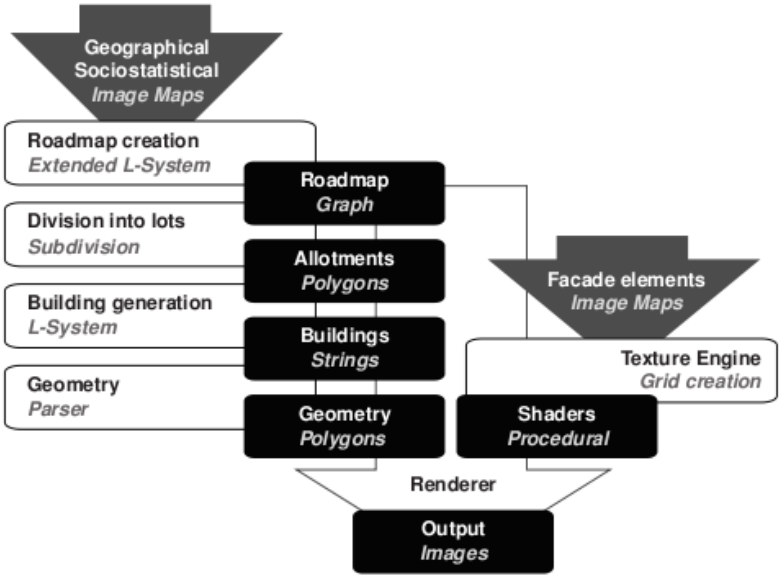

Abbildung 1: Pipeline der CityEngine (Bild aus (1))

nicht innerhalb der Grammatik des L-Systems definiert, sondern in externen Funktionen, deren nachträgliche Änderung wesentlich einfacher ist als das Anpassen vieler Ersetzungsregeln innerhalb des L-Systems. So erzeugt das L-System nur ein allgemeines Template, das sie *ideal successor* nennen. Dieses Template wird dann von den externen Funktionen (*globalGoals* und *localConstraints*) so abgeändert, dass es bestimmten globalen Zielen und lokalen Beschränkungen zur Straßenerzeugung folgt.

Als erstes werden die globalen Ziele durch die Funktion *globalGoals* überprüft. Die dabei überprüften Kriterien sind die Verbindung von Bevölkerungszentren durch Hauptstraßen oder Autobahnen. Außerdem achtet die *globalGoals*-Funktion darauf, dass golbale, übergeordnete Straßenpattern eingehalten werden. In der CityEngine sind radiale Pattern, wie sie in Städten wie Paris mit nur einem zentralen Kern zu finden sind, sowie rechteckigen Pattern, wie man es in Städten wie New York vorwiegend antrifft, und dem Folgen der größten oder kleinsten Steigung, dem sogenannten San Fancisco Stil, implementiert. Wird kein globales Stadtmuster vorgegeben, so gelten nur die lokalen Beschränkungen und die Eingaben über die soziostatistischen Karten.

Nachdem die globalGoals-Funktion ihre Parametervorschläge an die localConstraints-Funktion weitergereicht hat, überprüft diese, ob das neue Straßensegmet innerhalb eines ungültigen Gebiets endet oder dieses kreuzt und ob es Überschneidungen mit anderen Straßen oder Straßenkreuzungen gibt, die sich innerhalb eines bestimmten

Abstands zu diesem Straßensegment befinden. Handelt es sich bei dem Straßensegment um ein Teil einer Autobahn, das ein ungültiges Gebiet, zum Beispiel einen Bereich mit Wasser, kreuzt, so wird dieses Segment markiert und kann im späteren Verlauf, bei der Erzeugung der 3D-Geometrie, z.B. durch eine Brücke ersetzt werden.

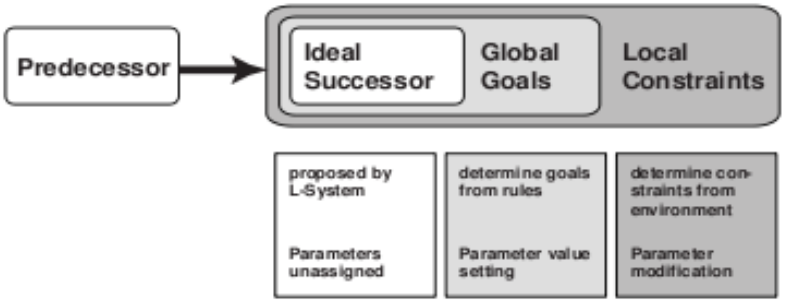

Abbildung 2: Anwendung der Anpassungsfunktion auf den Vorschlag der Engine (Bild aus (1))

Nachdem alle Straßenenden überprüft wurden, ob sie in einem erlaubtem Gebiet liegen, wird die Umgebung dieser Endsegmente auf Nachbarstraßen und Kreuzungen untersucht. Dabei werden folgende Regeln angewandt, um ein Straßennetz aufzubauen.

- Überkreuzen sich zwei Straßen, so wird an ihrem Schnittpunkt eine Kreuzung erzeugt.
- Liegt das Ende einer Straße in der Nähe einer bestehenden Kreuzung, so wird das Straßensegment so verlängert und seiner Ausrichtung verändert, so dass die Straße der Kreuzung hinzugefügt werden kann.
- Sind zwei Straßen so nah beieinander, so dass sie sich fast schneiden, so wird das Segment der Straße verlängert, bis es zu einem Schnitt kommt.

Nachdem die Erzeugung des Straßennetzes abgeschlossen ist wird es als Input an das Allotment Submodul übergeben. Dieses ist dafür verantwortlich die Bereiche zu definieren, in denen Gebäude stehen werden. Hierfür wird das Straßennetz als Begrenzung für die einzelnen Bauflächen verwendet. Die so entstandenen Flächen, auch Blocks genannt, werden zufällig unterteilt bis ein bestimmter, vom Benutzer bestimmbarer, Grenzwert erreicht ist. Die Parzellen, die zu klein geworden sind oder keinen direkten Zugang zu einer Straße bieten, werden verworfen und aus dem Modell entfernt.

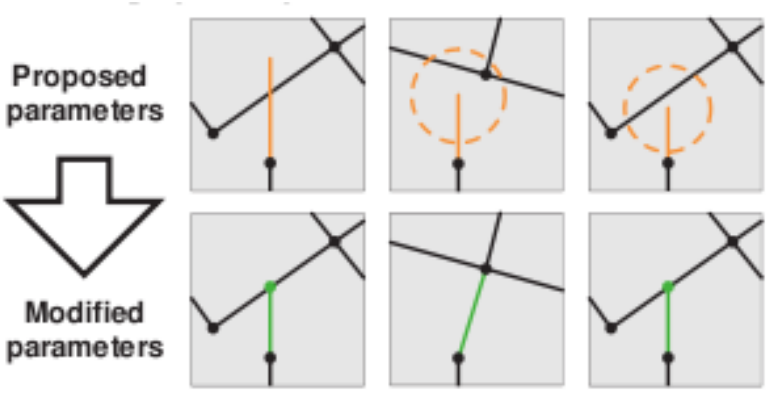

Abbildung 3: Anwendung der lokalen Constraints auf die Straßensegmente (Bild aus (1))

Im darauf folgenden Schritt wird für jede Parzelle durch ein stochastisches L-System eine Gebäudegeometrie erzeugt, deren Aussehen von der Flächennutzung in dem Bereich bestimmt wird. Unterschiedliche Produktionsregeln sind in der CityEngine für Hochhäuser, gewerbliche Gebäude und Wohnhäuser implementiert. Der Ausgangspunkt für die Geometrieerzeugung ist eine Boundingbox, die die Form der jeweiligen Parzelle besitzt. Abhängig vom Gebäudetyp werden an dieser Boundingbox die jeweiligen Produktionsregeln angewandt.

Die in diesem L-System definierten Transformationen sind Skalierung und Translation. Zusätzlich können geometrische Templates, wie Antennen oder bestimmte Dachverzierungen, dem Modell hinzugefügt werden. Diese Art der Gebäudegeneration ermöglicht zusätzlich ein leicht zu implementierendes LOD-Feature, indem die unterschiedlich aufgelösten Gebäudemodelle eines jeden Produktionsschritts behalten und als LOD-Modell verwendet werden können. Der letzte Schritt besteht in der Texturierung der entstandenen 3D-Modelle. Da die Verwendung von hochaufgelösten Gebäudetexturen aus Fotos von echten Gebäuden, die auf das Modell projeziert werden, zu speicheraufwendig und zu rechenzeitintensiv wäre, verwendet die CityEngine ein halbautomatisiertes Verfahren zur Erzeugung der benötigten Texturen.

Hierfür verwenden sie ein von Ebert in (16) beschriebenes Verfahren zur funktionalen Komposition von Texturen, indem auf mehreren Ebenen die erzeugten oder vorbereiteten Texturkomponenten in einem Grid prozedural angeordnet und auf das Gebäudemodell projeziert werden.

Die Vielfalt der Gebäude übersteigt die der Gebäude von Greuter et al. um ein Vielfaches, vor allem, da die erzeugten Gebäudelots nicht auf ein rechteckiges Ver-

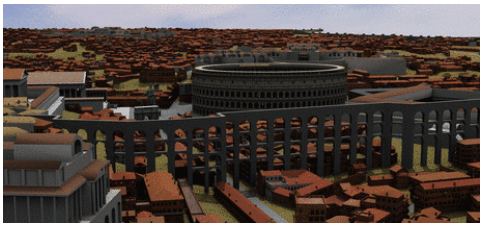

Abbildung 5: Ein Nachbau des alten Roms mithilfe der CityEngine (2)

hältnis festgesetzt sind, sondern durch die Form der Blocks. Zwar ist es möglich eine Stadt nur auf Grundlage einer Landkarte erzeugen zu lassen, die Beispiele aus (1) verwenden aber durchgängig mehr als diese vom Benutzer bereitgestellte Eingabe, um die gewünschten Resultate zu erhalten. In jüngerer Zeit wurde die CityEngine auch erfolgreich zur Rekonstruktion des alten Pompeii (3) und Roms (2) eingesetzt und konnte auch dort ein stimmiges Aussehen der Stadt und ihrer Architektur erzeugen. Wieviel Aufwand es aber wirklich ist, die entsprechenden Architekturtypen und Gebäudestile in die CityEngine zu integrieren, ist aus der vorhandenen Literatur nicht ersichtlich. Neben einer Echtzeitansicht in einem Fenster der CityEngine können die Ergebnisse auch über eine Exportfunktion an anderer Stelle wiederverwendet werden. Die, durch die Engine erzeugten, Modelle sind nach dem Export vollkommen statisch und müssen von Künstlern entsprechend belebt bzw. verändert werden.

Abbildung 4: Die Rekonstruktion der Stadt Pompeii mithilfe der CityEngine (3)

2.2 Undiscovered City

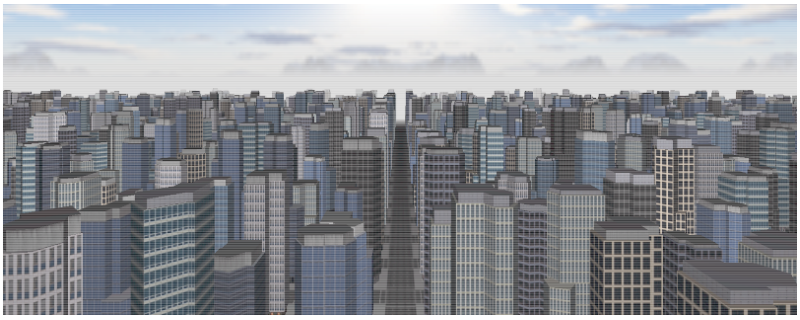

Abbildung 6: Resultat der gridbasierten Ansatzes von Greuter et al. (Bild aus (4))

Das von Stefan Greuter et al. im Jahr 2003 (10; 4) vorgestellte System zur prozeduralen Erzeugung von pseudo-unendlichen Städten wurde in mehreren Artikeln in ihren Grundlagen beschrieben. Das Ergebnis ihrer Arbeit ist ein Prototyp, der im selben Jahr unter dem Namen "Undiscovered City" präsentiert wurde.

Diese erzeugt virtuelle Städte beliebiger Größe in Echtzeit, die von dem Benutzer der Anwendung interaktiv erforscht werden können. Der Prototyp verwendet als Grundlage für das zu generierende Straßennetz ein gleichförmiges Netz, an dem die Straßen der virtuellen Stadt ausgerichtet werden. Dieser Ansatz ermöglicht das Erzeugen von Straßennetzen, wie man sie aus modernen Metropolen, wie New York City kennt. Der freie Platz zwischen den einzelnen Straßen wird mit Gebäuden gefüllt, deren Grundriss und 3D-Geometrie auf Basis von geometrischen Primitiven zusammengesetzt und extrudiert werden. Damit die Stadt in Echtzeit generiert und visualisiert werden kann, wird in dem Prototypen eine Technik, die sie in ihren Artikeln View Frustrum Filling nennen, verwendet. Hierbei wird vor dem Rendering ein Sichtbarkeitstest zwischen der Kamera und den einzelnen Zellen der Stadt durchgeführt. Befindet sich eine Zelle im Sichtbereich der Kamera und innerhalb der sichtbaren Distanz, so wird das entsprechende Element (dies kann ein Gebäude oder eine Straße sein) berechnet und angezeigt.

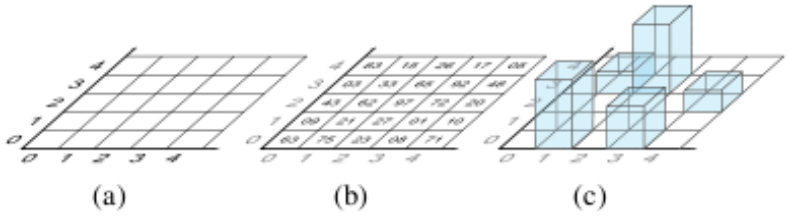

Abbildung 7: (a) das 2D-Grid, (b) die Seeds, die für die Gebäudeerzeugung benötigt werden und jeder Zelle durch die Hashfunktion zugeordnet wurde, (c) die Gebäude, die den einzelnen Zellen zugeordnet wurden (Bild aus (4)).

Nicht nur die Straßennetze der virtuellen Stadt werden zur Laufzeit berechnet, auch die Gebäude werden über prozedurale Methoden erzeugt. Das Aussehen der Häuser definiert sich über eine 32-Bit-Zufallszahl, deren Stellen einzelne visuelle Eigenschaften, wie Höhe und Breite des Gebäudes oder die Anzahl unterschiedlicher Etagen bestimmt.

Um eine ausreichende optische Varianz zu gewährleisten, arbeitet ihr Prototyp mit einer Hashfunktion, die den einzelnen Zellen der virtuellen Stadt eine Zufallszahl zuweist. So werden nah beieinanderliegende Zufallszahlen breit über das komplette Netz gestreut. Dies ist wichtig, da Zufallszahlen mit einer kleinen Differenz zu optisch ähnlichen Gebäuden führen. Bevor ihr Programm die Stadt rendern kann, müssen die einzelnen Gebäudemodelle erzeugt und in einem Cache für nachfolgende Frames gespeichert werden. Dafür werden die benötigten Grundrisse der Häuser zufällig bestimmt, indem für jeden Iterationsschritt dem Grundriss ein weiteres Polygon, an einer zufälligen Stelle, hinzugefügt wird.

Der Algorithmus, der für die Generierung verwendet wird, lässt sich in Pseuocode folgendermaßen beschreiben.

Algorithm 1 Pseudocode des Algorithmus zur Erzeugung der Gebäudegrundrisse. (Entnommen aus (10))

```
src <- random polygon
for every building iteration do
        tmp <- random polygon
        rotate tmp randomly about y axis
        translate tmp to random vertex in src
        src <- src union tmp
```

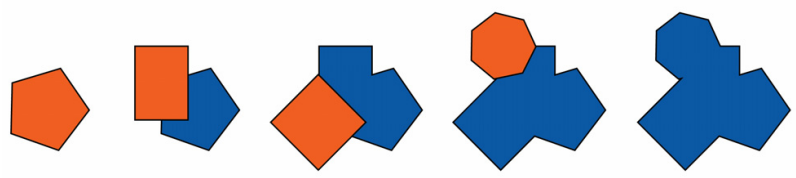

Abbildung 8: Die einzelnen Iterationsstufen der Grundrissgeneration (Bild aus (4))

In der Implementierung verwenden sie für ihren Prototypen einen Least-Recently-Used (LRU) Algorithmus, um zu bestimmen, welche Gebäude im Display-Cache gehalten werden sollen und so nicht in jedem Frame neu generiert werden müssen. So werden die Gebäudemodelle, die längere Zeit nicht mehr zur Anzeige benötigt wurden, durch Gebäudemodelle ersetzt, die erst vor kurzem sichtbar waren.

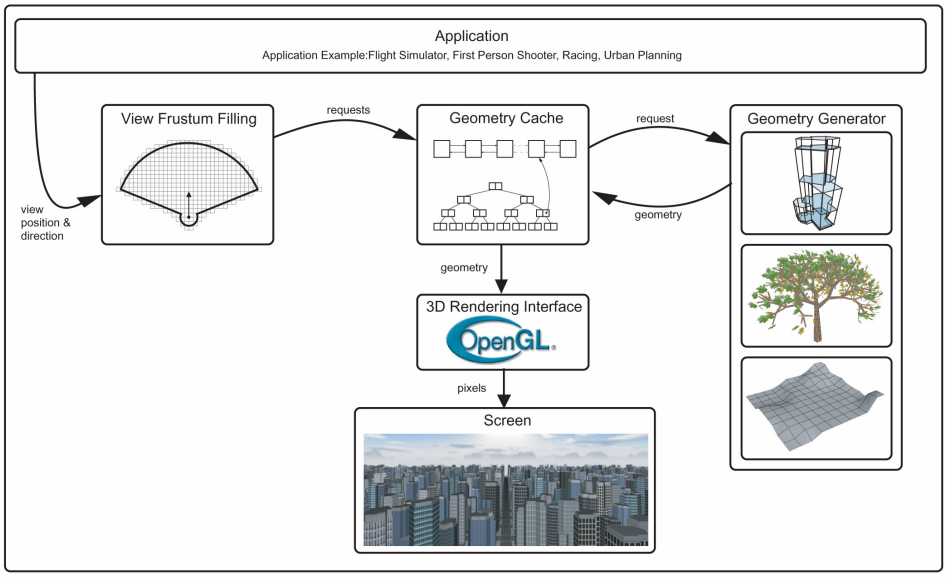

Abbildung 9: Schemadarstellung der von Greuter et al. entworfenen Architektur

Der hier vorgestellte Prototyp zur Generierung von prozeduralen Städten stellt einen interessanten Ansatz dar, auch wenn sich die optische Vielfalt durch die strenge Anordnung an einem Grid in Grenzen hält und es kaum Möglichkeiten gibt, das Aussehen der Stadt zur Laufzeit des Programms zu beeinflussen. So ist die Definition von bestimmten Vierteln (Wohn, Industrie- und Gewerbeviertel) mit unterschiedlichen optischen Parametern nicht möglich; auch Freizeitflächen wie Parks werden nicht berücksichtigt. Der Prototyp zeigt aber, dass es möglich ist komplette Städte

in einer Framerate zu generieren, die es ermöglicht das Ergebnis interaktiv erfahrbar zu machen.

2.3 CityBuilder

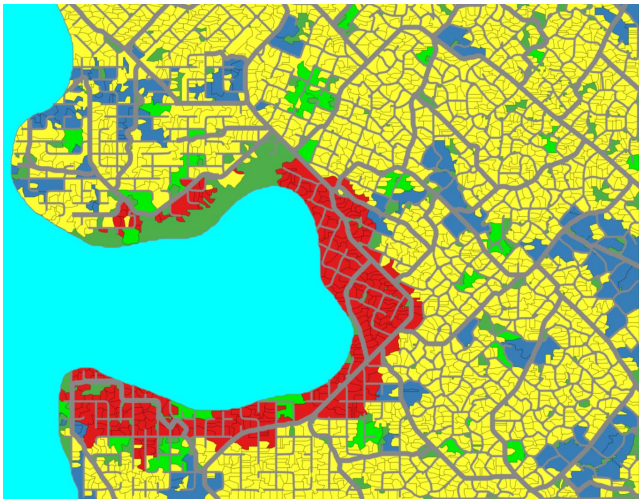

Abbildung 10: Das Ergebnis einer Landnutzungssimulation, das vom CityBuilder erstellt wurde. (aus (5))

Wilensky et al. beschreiben in (5) die Verwendung von Multi-Agenten-Systemen zur Erzeugung von urbanen Landnutzungskarten. Ihre Implementierung, der CityBuilder, verwendet NetLogo, eine Java-Implementierung der Sprache Logo, die eine Plattform zur Erforschung von MAS-bezogenen Fragestellungen bietet. Hierbei liegt das Augenmerk der Arbeit nicht auf der kompletten Erzeugung einer virtuellen Stadt, wie es zum Beispiel bei der CityEngine der Fall ist, sondern auf einer Simulation der unterschiedlichen Landnutzungen innerhalb eines urbanen Systems. Die Simulation verwendet unterschiedliche Typen von Agenten, um die Landnutzung und den Straßenbau zu simulieren. Durch diesen Ansatz ist es nicht nur möglich die Landnutzung einer virtuellen Stadt zu erzeugen, man kann auch auf die Entwicklungsgeschichte dieser Stadt zugreifen und diese, bei ihrer Modellierung, berücksichtigen. Der CityBuilder ist in der Lage fünf unterschiedliche Landnutzungen (gewerbliche, industrielle, private, sowie die Verwendung als Straße oder Parkfläche) zu simulieren. Die Straßenerzeugung basiert auf drei Typen von Agenten, die für die Erweiterung des Straßennetzes verantwortlich sind.

- *Tertiary Extenders* kümmern sich darum, dass unerschlossenes Land erreichbar ist.
- *Tertiary Connectors* sorgen dafür, dass die einzelnen *tertiary roads* ausreichend

miteinander verbunden sind.

- *Primary Developers* sind dafür verantwortlich, dass die Einwohner der Stadt sich innerhalb dieser schnell genug bewegen können.

Die Art und Weise, wie die Straßen in der Landschaft entwickelt werden, kann durch vom Künstler bestimmbare Variablen definiert werden. Hierfür kann er die entsprechenden Constraints in die Landschaft malen, an denen sich dann die Agenten orientieren, oder global für die Karte über Stellregler definieren.

So kann der Künstler direkten Einfluss auf das Aussehen der Stadt nehmen, indem er über die gemalten Constraints zum Beispiel die Dichte des Straßennetz, die Kurvigkeit der Straßen, das Ausmaß der Verbindungen zwischen den einzelnen Straßen oder die Art der Landnutzung in diesem Bereich bestimmen bzw. fördern kann. Zur Bestimmung der Landnutzung verwendet die Simulation jeweils einen separaten Agententypen für jeden existierenden Nutzungstyp (kommerzielle, industrielle und private Nutzung). Jeder dieser Developer-Agenten versucht entsprechende, noch nicht entwickelte, Zellen mit einem hohen Landwert für seine eigene Nutzung zu finden. Jeder Agententyp bewertet hierbei die Zellen der Karte auf unterschiedliche Art und Weise und fügt sie zu größeren Flächen, den Parzellen, zusammen. So versucht der Developer-Agent für private Nutzungsflächen zum Beispiel Gebiete zu finden, die abseits von belebten Straßen und Ballungen von industriellen Nutzungsflächen liegen. Die Simulation besitzt keine Abbruchkriterien, sondern läuft solange der Benutzer sie nicht manuell beendet. Zur Laufzeit ist es ihm möglich die Agenten indirekt durch die Verwendung der gemalten Constraints oder durch die Änderung von globalen Konstanten zu kontrollieren.

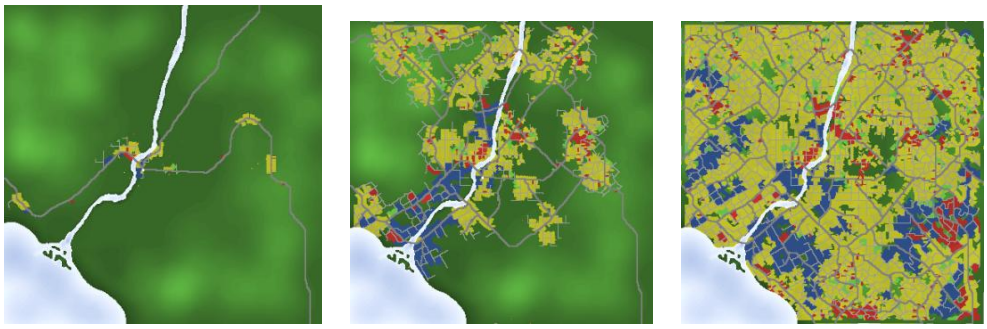

Abbildung 11: Entwicklung des Simulationsergebnisses des CityBuilders über die Zeit. Die blauen Flächen sind Gebiete mit industrieller Nutzung, die roten Flächen stehen für Gebiete mit kommerzieller Nutzung und gelbe Flächen stehen für Wohngebiete. (aus (5))

Da der CityBuilder in einer interpretierten Sprache geschrieben ist, ist die Geschwindigkeit, mit der die Simulation abläuft, gering. So dauert es auf einem handelsüblichen Computer ca. 15 Minuten, um eine Stadt mit den Ausmaßen einer Kleinstadt zu simulieren. Davon abgesehen bietet die Anwendung aber viele unterschiedliche Möglichkeiten, mit denen der Benutzer Einfluss auf die Entwicklung der Stadt nehmen kann. Gerade bei der Erzeugung des Straßennetzes bieten die Agenten eine gute Möglichkeit einen flüssigen und realistischen Übergang zwischen kleinen, gewundenen und gerade Hauptstraßen zu generieren.

3 Grundlagen der Geosimulation

Wie man im vorherigen Kapitel sehen kann gibt es zwei unterschiedliche Ansatzmöglichkeiten, um eine Stadt in ihrer Entwicklung bzw. ihrem Aussehen zu beschreiben. Die Arbeiten von Parish et al. und Greuter et al. sind gute Beispiele für eine Top-Down-Lösung des Problems. Sie beschreiben das gewünschte Verhalten und Aussehen in L-Systemen oder ordnen die generierten Gebäude an einem fixen Grid an. Die zweite Möglichkeit ist eine Bottom-Up-Lösung, wie die von Wilensky et. al. Ein solcher Ansatz erfordert eine andere Heransgehensweise an das Problem, um zu dem gewünschten Ergebnis zu kommen. Hierfür haben sie sich der Erkenntnissen und Methoden der Geosimulation, die anhand von Simulationen mit zellulären Automaten oder Multi-Agenten-Systemen die einzelnen Aspekte der Stadtentwicklung verstehen möchte, bedient.

Die Geosimulation entstand in der Urbanologie aus der Erkenntnis heraus, dass es sich bei Städten nicht um Systeme handelt, die sich, wie in den frühen 1950er und 1960er Jahren angenommen, in einem Equilibrium befinden, sondern die dynamischen Prozessen unterworfen sind, die sie verändern und in ihrer Entwicklung beeinflussen. Bis auf wenige Ausnahmen in der Anfangszeit dieses Umbruchs, wie zum Beispiel Wilsons Vorschlag statische Modelle in die dynamischen Prozesse einzubinden (17), gingen die meisten Bestrebungen dahin, die bis dahin bekannten Standardmodelle durch mehr dynamische Beziehungen zu erweitern (6). Erst durch die Verbreitung leistungsfähigerer Computer, die es ermöglichten die rechenintensiven Modelle der Urbanologie in einer realistischen Zeit zu berechnen, wurde es auch möglich, Modelle zu entwickeln und zu testen, die aus wesentlich mehr disaggregierten Einheiten bestanden. Vor allem ab Beginn der 1990er Jahren konzentrierten sich die Entwicklungen auf die Verwendung von zellulären Automaten und Multi-Agenten-Systemen zur Beschreibung und Simulation von urbanen Prozessen (17**?**). Ein Beispiel für die Verwendung von zellulären Automaten in der Geosimulation ist die Segregation von Bevölkerungsgruppen innerhalb einer Stadt. Ein einfaches Modell zur Beschreibung dieser Segregation ist das von Schelling 1968 vorgestellte Modell, in der die Segregation als ein Resultat der Selbstorganisation einzelner Zellen angesehen wird (18).

Die Zellen bekommen zu Beginn der Simulation eine zufällige Zugehörigkeit zu einer bestimmten Gruppe (zum Beispiel “Biertrinker” oder “Weintrinker”) zugeordnet.

Die Startkonfiguration t(0) ist also defininiert als $t(0) = a_i$, mit a = “Biertrinker” oder “Weintrinker” und i = dem Index der Zelle. Jede Zelle ändert in jedem dar-

auffolgenden Simulationsschritt seine Zugehörigkeit (seine Meinung), wenn in einer 3x3 Nachbarschaft, um diese Zelle, mehr als 4 Nachbarn der jeweils anderen Gruppe angehören. Diese sehr einfache Regel führt zu einem Verhalten, das schnell zu einem stabilen segregierten Zustand führt (Abb. 12(b)), der sich selbst dann nicht wesentlich ändern, wenn man diesen Zustand durch das Hinzufügen eines Zufallsrauschen aus dem Gleichgewicht (Abb. 12(d)) bringt.

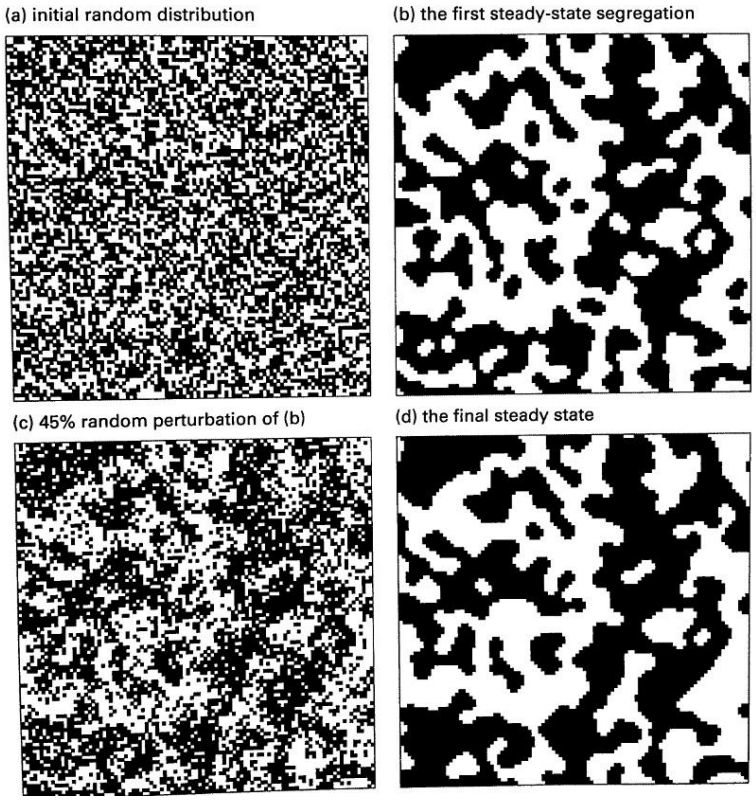

Abbildung 12: Die Entstehung von Segregationspattern durch Meinungsänderung aus (6)

So einfach dieses Modell in seiner Beschreibung auch erscheinen mag, es kann zu einem guten Grad die Muster von Gruppenbildungen (auch Segregation genannt) in Städten gut beschreiben. Vergleicht man die Muster aus Abbildung 12 mit dem Aussehen einer realen Segregationskarte in Abbildung 13, so erkennt man schnell die Ähnlichkeit.

Abbildung 13: Ausschnitt aus einer Segregationskarte aus real erhobenen Daten (7)

Natürlich gibt es in der Geosimulation wesentlich komplexere Modelle, die zum Beispiel zur Simulation von Wegesystemen von Fußgängern oder globalen, urbanen Strukturen verwendet werden können. Hier verschwimmen, mit Hinblick auf die Simulation von Wegesystemen, zum Teil auch die Grenzen zu Forschungen der verteilten künstlichen Intelligenz.

Die Beschreibung und Einführung dieser Modelle an dieser Stelle würde aber den Rahmen dieser Arbeit übersschreiten und so soll folgend noch ein weiteres Modell auf Basis von zellulären Automaten beschrieben werden, das Vicsek-Szalay Modell. Dies hat zur grundlegenden Erkenntnis in der Urbanologie beigetragen, dass Pattern, wie sie auch in urbanen Entwicklungen zu finden sind, aus der örtlichen Durchschnittsbildung von Entwicklungspotenzialen hervorgehen können (19). Im Vicsek-Szalay-Modell wird das Entwicklungspotenzial einer Zelle durch die gemittelte Summe der Nachbarzellen ihrer Von-Neumann-Nachbarschaft bestimmt. Diese Durchschnittsbildung wird in ihrem Modell noch durch das Hinzufügen eines weißen Rauschens erweitert, so dass sich das Modell zu keinem Zeitpunkt im Zustand eines Equilibriums befinden kann. Vicsek und Szalay erhalten nun eine sichtbare Struktur aus dieser Potenzialbildung, indem sie einen Grenzwert definieren, zu dem eine Zelle sich entwickeln kann und ihren Zustand ändert. Zwar kann es in in diesem Modell passieren, dass in späteren Zeitschritten das Potenzial für eine bestimmte Zelle wieder unter den benötigten Grenzwert fällt, aber eine Zelle, die einmal entwickelt ist, bleibt dies auch, auch wenn der Potenzialgrenzwert wieder unterschritten werden sollte. Für einen Entwicklungsgrenzwert von 4,5 und einer Startverteilung von -1, 1 im Potenzialfeld, kommt es für die Zeitschritte t = 0, 25, 250, 1000, 1500, 2000, 2500 und 3000

zu den Strukuren, die in Abbildung 12 zu sehen sind und sehr stark an Verteilungen und Muster in urbanen Entwicklungen erinnern.

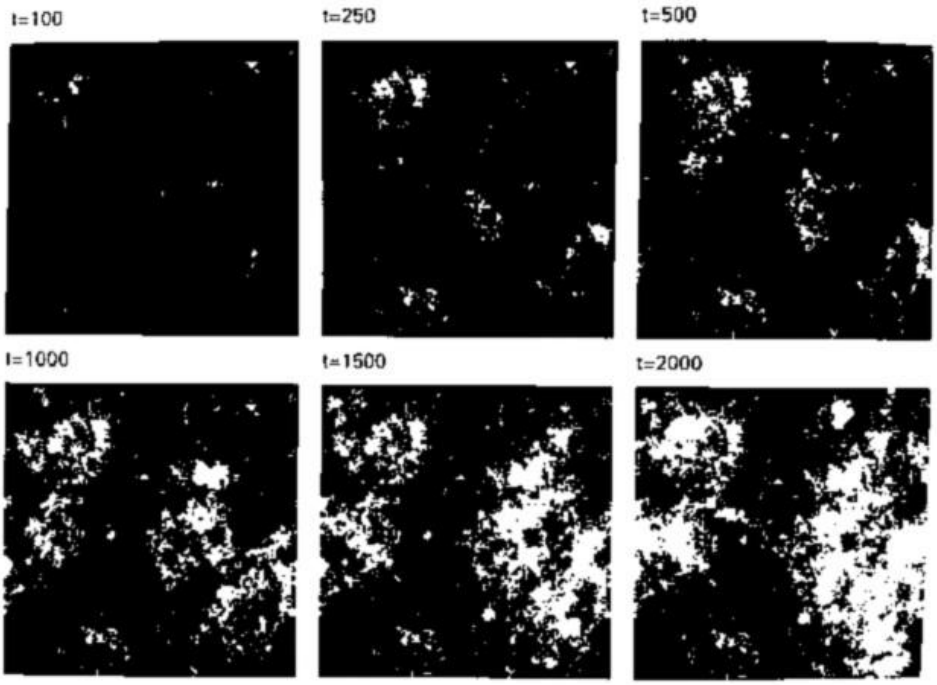

Abbildung 14: Ergebnis der Simulation des Vicsek-Szalay-Modells aus (6)

Eine Erklärung, wie solche Strukturen aus einem so zufälligen Prozess entstehen können, lässt sich auf einem phänomenologischen Level finden. Die zufällige Ausgangssituation produziert beim Mitteln klar abgegrenzte Muster, in denen sich bestehende Nachbarschaften und größere Anhäufungen in ihrem Wertebereich annähern. Das Hinzufügen des Zufallsrauschen wirkt wie das Einbringen von neuer Energie in das System und stellt sicher, dass es zu keinen großen, einförmigen Ballungen von gleichwertigen Potenzalbereichen kommt. So entstehen diese sichtbaren, fraktalen Strukturen. Die hier vorgestellten Modelle und weitere, die in (6) oder als Anwendung für NetLogo unter (20) zu finden sind, sind nicht dazu gedacht, die urbane Entwicklung als solche in einem großen Modell beschreiben und simulieren zu können. Es geht viel mehr darum zu erkennen, dass es möglich ist, mit oft nur wenigen Annahmen und Regeln, die Entwicklung von einzelnen urbanen Phänomenen, mithilfe von zellulären Automaten oder Multi-Agenten-Systemen, präsentieren zu können. Dies bietet der Forschung die Möglichkeit auf experimenteller Basis zu Ergebnissen zu kommen.

Vor allem in Bereichen, in denen emergente Phänomene betrachtet werden, die durch die Interaktion einzelner Entitäten entstehen, sind Simulationen durch Agenten oder zelluläre Automaten besonders geeignet. In vielen Fällen ist es möglich die Simulation durch bestehende Theorien und Modelle zu stützen und zu verifzieren. Im Allgemeinen ist es aber schwierig die Validität von Simulationsergebnissen in Bereichen zu garantieren, für die es keine existierenden Modelle gibt, mit denen die Ergebnisse abgeglichen werden können. Die Geosimulation bietet aber ein gutes Handwerkszeug zur experimentellen Erforschung von urbanen Phänomenen am Computer und kann so helfen, die Natur der emergierenden, komplexen Prozesse besser zu verstehen.

4 Anforderungen an den Prototypen und sein zu Grunde liegendes Modell

Die meisten der im zweiten Kapitel vorgestellten Arbeiten kommen aus dem Bereich der Geosimulation und haben ihren Hintergrund vor allem in der Urbanologie. Es handelt sich um Systeme, die Prinzipien zur Modellierung von Städten in Computern untersuchen sollen. Ihr Ziel ist es daher auch, große und komplexe, urbane Systeme besser verstehen oder dreidimensional visualisieren zu können.

Hauptziel dieser Arbeit ist nicht die vollständige Simulation einer virtuellen Stadt, sondern die Erzeugung eines künstlichen Straßennetzes, das durch Agenten auf Grundlage ihrer Interaktionen mit ihrer Umwelt und untereinander entsteht. Hierfür bedient sich diese Arbeit der Erkenntnisse zur Lebensweise sozialer, staatenbildender Insekten wie Ameisen, die auf der Basis von Pheromonen untereinander Informationen austauschen und die Entstehung einer geordneten globalen Organisationstruktur ein emergentes Phänomen ist, das ohne globales Wissen der einzelnen Ameisen oder eine lenkende Kontrollinstanz auskommt. Das Projekt geht einen ähnlichen Weg wie (5), indem die Entwicklung dieses Netzes als ein emergentes Phänomen von Interaktionen der beteiligten Agenten verstanden wird. Im Gegensatz zu besagter Arbiet ist die Grundlage aber nicht ein mathematisches Model der Prozesse eines urbanen Systems, das die Entscheidungen der Agenten mit beeinflusst, sondern ein grundlegendes Bedürfnismodell, das die Entscheidungen der Agenten bestimmt, indem das oberste Ziel der Agenten eine möglichst hohe Bedürfnisbefriedigung ist.

Daher konzentriert sich die Implementierung des zu erstellenden Prototypen auf die, für dieses System essentiellen Komponenten und einer erweiterbaren Architektur, die in Zukunft leicht zu erweitern sein soll. Ziel für den Prototypen muss es also sein, ein System bereitzustellen, das ohne viel Aufwand an neue Anforderungen, wie neue Gebäude oder Bedürfnisse, anzupassen ist.

Im Folgenden sollen die einzelnen Komponenten des Modells in kurzen Beschreibungen und losgelöst von ihrer eigentlichen Implementierung vorgestellt werden, damit der Leser sich einen Überblick verschaffen kann, ohne von den technischen Details abgelenkt zu werden.

Das hier entwickelte Modell besteht im wesentlichen aus zwei logischen Einheiten, auf die in den nächsten Abschnitten eingegangen werden soll; die Welt der Simulation und die Personen, die mit der Welt in konstanter Interaktion stehen.

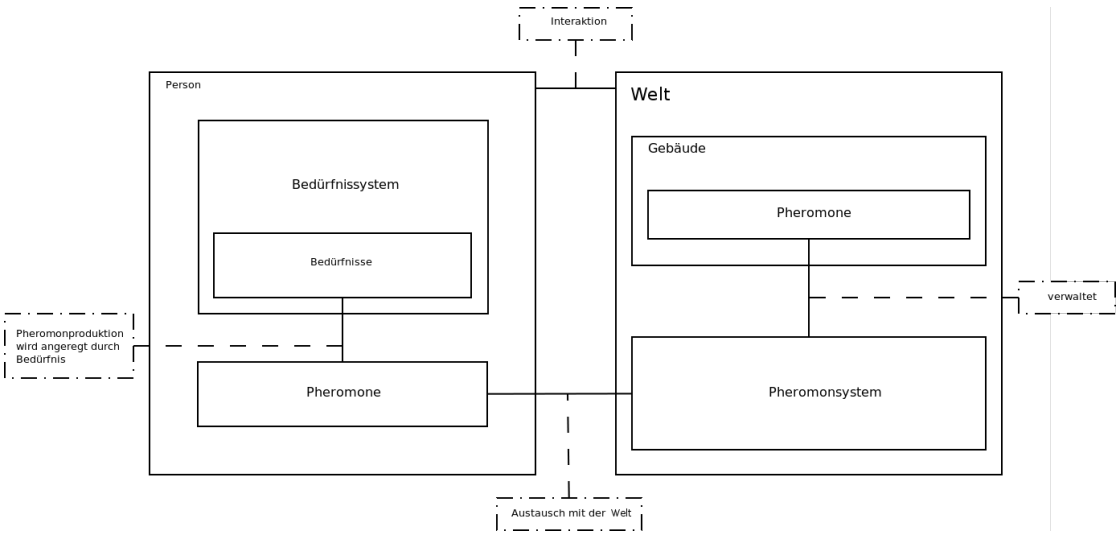

Abbildung 15: Übersicht über die im Modell vorhandenen Komponenten

4.1 Personen-Modell

Personen in der Simulation führen ein Leben, das von unterschiedlichen Bedürfnissen beeinflusst wird, die in Konkurrenz zueinander stehen. Der kurzen Zeit, in der diese Arbeit entsteht, ist es geschuldet, dass nur eine sehr vereinfachte Abstraktion eines Menschen in diesem Modell existiert.

Eine Person kommt als erwachsener Mensch in der Welt an und erfährt keine Alterung und die mit ihr verbundenen Veränderungen, wie den sich ändernden Lebensgewohnheiten oder neu auftretenden Bedürfnissen. Alle Bedürfnisse werden über ein Bedürfnissystem verwaltet, das als Hauptentscheidungsinstanz in jeder Person vorhanden ist und die entsprechenden Aktionen einer Person auslöst.

4.1.1 Bedürfnisse einer Person

Eine grundlegende Antriebskraft für das Handeln der Personen innerhalb der Simulation dieses Modells ist ein zu befriedigendes Bedürfnis. Ein Bedürfnis kann die Befriedigung eines körperlichen Zustands wie Müdigkeit oder Hunger sein, kann aber auch sozialer (Bedürfnis nach sozialer Interaktion) oder wirtschaftlicher (Bedürfnis nach Arbeit, Wohlstand) Natur sein. Ob ein Bedürfnis von einer Person zu einer bestimmten Zeit befriedigt wird, hängt davon ab, wie stark dieses zu dem entsprechenden Zeitpunkt ist und eine entsprechende Ressource vorhanden ist, die der Befriedigung dieses Bedürfnisses dient. Das Modell bezieht folgende Bedürfnisse, die

nach (21) jedem Menschen zu eigen sind, mit ein:

- vegetative Bedürfnisse
  - Hunger
  - Schlaf
  - Sicherheit (Bedarf nach Wohnraum)
- soziale Bedürfnisse
  - Bedürfnis nach Kommunikation mit anderen Personen
- Sicherheitsbedürfnisse
  - Bedürfnis nach finanzieller Sicherheit (Arbeitsplatz)

4.1.2 Das Bedürfnissystem

Personen treffen ihre Entscheidungen auf Basis von Bedürfnissen, die im Laufe der Zeit befriedigt werden müssen, um einen Grad von Zufriedenheit zu erreichen. Wobei der Grad an Zufriedenheit als das Maß der Abwesenheit von unbefriedigten Bedürfnissen verstanden wird. Dieser Zustand ist erreicht, wenn

- alle Bedürfnisse befriedigt wurden, oder
- kein Bedürfnis einen kritischen Grenzwert überschritten hat und erneut vom Bedürfnissystem zur Befriedigung ausgewählt wurde.

Das Bedürfnissystem verwaltet die vorhandenen Bedürfnisse und evaluiert die Wahrscheinlichkeit, mit der ein bestimmtes Bedürfnis befriedigt werden muss. Die Wahrscheinlichkeit hängt von unterschiedlichen Faktoren ab, die sich, abhängig vom jeweiligen Bedürfnis, unterscheiden können.

Hat ein Bedürfnis den Schwellenwert erreicht, bei dem es von dem Bedürfnissystem zur Befriedigung ausgewählt wird, so aktviert das Bedürfnissystem das entsprechende Verhalten der Person, das für diese Zeit die Kontrolle über die Person übernimmt, bis das Bedürfnis befriedigt wurde.

Das Bedürfnis nach sozialer Interaktion stellt einen gewissen Sonderfall in diesem Modell dar, da es das einzige Bedürfnis ist, bei dem der Agent als Emitter von Duftstoffen auftritt. Der Agent signalisiert damit, dass er sozialen Kontakt sucht. Agenten, die diesen Duftstoff in ihrer Umgebung wahrnehmen, können darauf bei Bedarf reagieren und der Spur bis zum suchenden Agenten folgen.

4.1.3 Pheromone

Da Personen nur auf Grundlage von lokalen Informationen Entscheidungen treffen können, sollte nicht nur ihr eigenes, aktuelles Wissen mit einbezogen werden, sondern auch das von anderen Entitäten. Eine Lernstrategie, die dies ermöglicht und in diesem Modell verwendet wird, ist eine Strategie auf Basis von Duftstoffen, wie man sie bei Staaten bildenden Insekten findet. Hierbei lenken unterschiedliche Duftstoffe und ihre jeweilige Stärke das kollektive Verhalten des Insektenstaates.

Die Duftstoffe, die verwendet werden, sollen folgende Eigenschaften (vgl. hierzu (15)) erfüllen:

- Duftstoffe vom gleichen Typ, an einer Position, addieren sich in ihrer Stärke
- die Stärke der Duftstoffe nimmt über die Zeit hinweg ab
- Duftstoffe erfahren eine Diffusion und verteilen sich im umliegenden Raum
- Duftstoffe können von anderen Personen wahrgenommen werden und können auf Grundlage dieser Wahrnehmung neue Entscheidungen treffen

Eine Person verwendet die Informationen der Duftstoffe, um existierende Wege in der Welt zu entdecken und ihnen zu folgen. Gleichzeitig hinterlässt jede Person eine neue Duftspur auf dem Weg zu ihrem Ziel. Um einen existiereden Weg zu finden, folgt die Person der Duftspur der vorhandenen Wege und dem gewünschten Ziel, in dem die Person dem Verlauf des Gradienten folgt. Nähere Details zu der Art und Weise wie die Person ihr Ziel mit Hilfe der Pheromone findet, finden sich im Kapitel zur Implementierung des Modells.

4.2 Welt-Modell

Die Welt ist die Entität, mit der die Person in konstanter Interaktion steht, um seine eigenen Bedürfnisse zu befriedigen. Die Hauptaufgabe der Welt ist die Verwaltung der Gebäude, die Personen als Ressourcenquellen dienen und der in der Welt befindlichen, Duftstoffe.

4.2.1 Gebäude

Gebäude dienen der Repräsentation von Ressourcenquellen innerhalb der Welt. Sie verfügen über einen unbegrenzten Vorrat und können von den Agenten besucht werden, um ein entsprechendes Bedürfnis zu befriedigen.

Damit die Gebäude in der Welt zu finden sind und die Personen nicht nur zufällig auf sie treffen, emitieren Gebäude einen Duftstoff, der den Agenten das Lokalisieren der Quelle vereinfachen kann. Gebäude können zwei unterschiedliche Arten von Duftstoffen produzieren. Die erste Art funktioniert analog eines Broadcastings. Der Duftstoff wird in die Welt entlassen und wirkt auf alle Personen in der Welt gleichermaßen, die zweite Art ist ein Duftstoff, der nur von einer Person verarbeitet werden kann. Diese Art wird von Wohngebäuden verwendet, damit Personen ohne ein internes Wissen über die Karte auskommen und zu ihrer Wohnung zurückfinden können.

Folgende Gebäudetypen sind in dem Modell des Prototypen vorgesehen:

- Wohngebäude dienen als Heim für eine Person und verwenden individualisierte Duftstuffe für den jeweiligen Bewohner.
- Bäckereien als Beispiel für einen Nahrungsprovider, der von den Personen verwendet werden kann, um bei Hunger das Bedürfnis nach Nahrung zu befriedigen. Bäckereien verwenden allgemeine Duftstoffe, die von jeder Person in der Welt verwendet werden können.
- Geschäftsgebäude dienen als Arbeitgeber für Personen und befriedigen das Bedürfnis nach Arbeit. Geschäftsgebäude propagieren ihre Dienste auch über Duftstoffe.

4.2.2 Pheromonsystem

Das Pheromonsystem der Welt verwaltet alle Duftstoffe, die sich zu einem Zeitpunkt t in der Welt befinden. Gebäude müssen hierfür ihren speziellen Duftstoff zu Beginn

bei dem System registrieren. Das Pheromonsystem übernimmt danach die Aufgabe der Diffusion und der graduellen Abnahme über die Zeit. Zusätzlich verwaltet es auch die Duftspuren, die die Personen bei ihren Wegen durch die Welt hinterlassen und wendet Diffusion und Reduktion der Duftstärke auch auf diese an.

Da es sich bei den Wegspuren der Personen um keinen spezifischen Duft handelt, sondern dieser bei allen Personen gleich ist, muss er zu Beginn nicht bei dem Pheromonsystem angemeldet werden. Anders sieht es bei den Duftstoffen aus, die die Personen abgeben, die auf der Suche nach sozialer Interaktion sind. Diese sind individuelle Duftstoffe, die dem System zu beginn bekanntgegeben werden müssen.

5 Technische Dokumentation des Prototypen

Dieser Teil der Arbeit soll sich mit den unterschiedlichen Erwägungen und der Implementierung beschäftigen und diese im Rahmen dieser Dokumentation präsentieren.

5.1 Evaluation verfügbarer Multiagenten-Systeme

Um einen Prototypen zu entwickeln, der die Entwicklung von Wegesystemen bottom-up ermöglicht, fiel die Entscheidung direkt zu Beginn auf eine Implementierung in einem Multiagenten-System. Diese bieten bei einem bottom-up Ansatz den Vorteil, dass die einzelnen, in der Simulation involvierten Entitäten, als einzelne Agenten im Programm modelliert werden können. Diese Agenten übernehmen in der Simulation die einzelnen Aufgaben, die bei der Analyse des Systems als notwendige Grundbausteine herausgefunden wurden.

Da Multiagenten-Systeme in den letzten Jahren verstärkt in unterschiedlichen wissenschaftlichen Disziplinen und der Wirtschaft zur Lösung von Simulations- oder Anwendungsproblemen verwendet werden, gibt es einige bestehende Frameworks, die vor Beginn der Implementierung auf ihre Nützlichkeit hin untersucht werden sollten. Hierfür wurde versucht, die einzelnen Systeme, auf Grundlage von kleineren Proof of Concepts, in unterschiedlichen Punkten zu bewerten.

Die für die Entscheidungsfindung wesentlichen Kriterien sind:

- sollte in Java programmiert worden sein
- sollte als Open Source Lösung verfügbar sein
- Dokumentation
- Aktivität der Community
- empfundene Lernkurve

Im weiteren wurden nur Frameworks untersucht, die kostenlos und mit offenem Quelltext verfügbar waren. Die Multiagenten-Systeme, die näher untersucht wurden, waren JADE, Repast und Swarm. Von der Entwicklung eines eigenen Multiagenten-Systems wurde direkt von Beginn an abgesehen, da in der kurzen Zeit, die für die Bachelorarbeit zur Verfügung steht, kein System entwickelt werden kann, dass ausreichende Funktionalität zur Implementierung eines Prototypen liefern würde. Ganz abgesehen von den vielen Problemen, wie Inkonsistenzen in der Datenhaltung oder

Race Conditions, die durch die hohe Parallelität eines Multiagenten-Systems auftreten können.

Begibt man sich auf die Suche nach passenden Multiagenten-Systemen, so stößt man im wissenschaftlichen Kontext vor allem auf die zwei letztgenannten Vertreter, Repast und Swarm. Diese wurden in der Evaluationsphase möglicher Frameworks auch als erstes untersucht. Swarm schied nach einigen kleineren Versuchen wieder aus, da Swarm zwar eine Java-Binding besitzt, das Framework selbst aber in Objective-C geschrieben wurde. Zusätzlich war gerade die Einsteigerdokumentation veraltet, wodurch sich die Evaluation auf Repast und JADE konzentrierte.

5.1.1 Repast

Repast wird in seiner aktuellen Version Repast Simphony als SourceForge-Projekt (8) gehostet. Es bietet viele Features, die man sich von einem ausgereiften Multiagenten-System wünscht, das in unterschiedlichen akademischen Disziplinen eingesetzt werden können soll. Hierunter fallen die Einbindung externer Tools, wie das Statistikprogramm R, das grafische Editieren von Agenten (Abb. 16), die Verwendung von High-Level Sprachen wie Groovy und die automatische Visualisierung der Simulation. Die aktuelle Version baut auf dem Eclipse RCP auf und lässt sich auch nur in Verbindung mit Eclipse voll nutzen. Eine Verwendung der Bibliothek aus anderen IDEs heraus ist zwar theoretisch möglich, ist aber praktisch kaum umsetzbar, da die Konfiguration der Simulation nur mithilfe der in Eclipse integrierten Wizards wirklich sinnvoll ist, wie sich in der näheren Beschreibung von Repast im Folgenden zeigen wird.

Die Projektseite liefert Beispiele und eine Einführung in die wichtigsten Funktionen, um bei Repast einen Einstieg zu finden. Leider konzentriert sich ein Großteil der Dokumentation auf die Verwendung des grafischen Editors für Agenten, der es ermöglicht über Flowcharts die Logik des Agenten zu definieren, ohne eine Zeile Code schreiben zu müssen. Man ist zum Teil auf externe Dokumentation angewiesen, die vor allem die Verwendung der API, präziser beschreibt, als die offizielle Dokumentation.

Als erstes soll anhand eines kurzen HelloWorld Beispiels die grundlegende Arbeitsweise mit Repast demonstriert werden. Das Framework kann entweder einer existierenden Eclipse-Installation über eine Update-Site hinzugefügt werden, oder man benutzt die Repast-Installtion von der Projektseite, die allerdings nur für Windows und Mac verfügbar ist. Bei der Windows/Mac-Installation sind alle benötigten

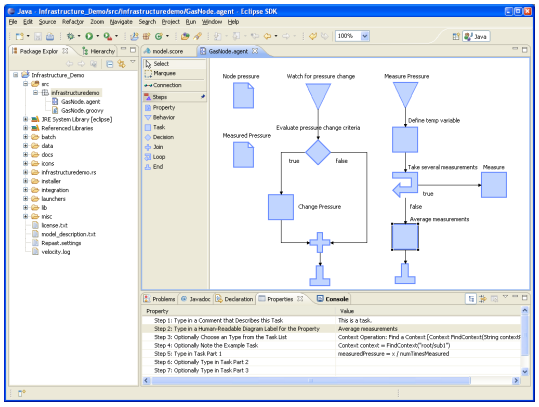

Abbildung 16: Grafischer Editor zur Modellierung der Logik eines Agenten in Repast. ((8))

Zusatzbibliotheken direkt vorhanden, für Linux müssen die fehlenden Komponenten per Hand dem Classpath hinzugefügt werden. Dies sind Java3D, Java Advanced Imaging (JAI) und Jogl.

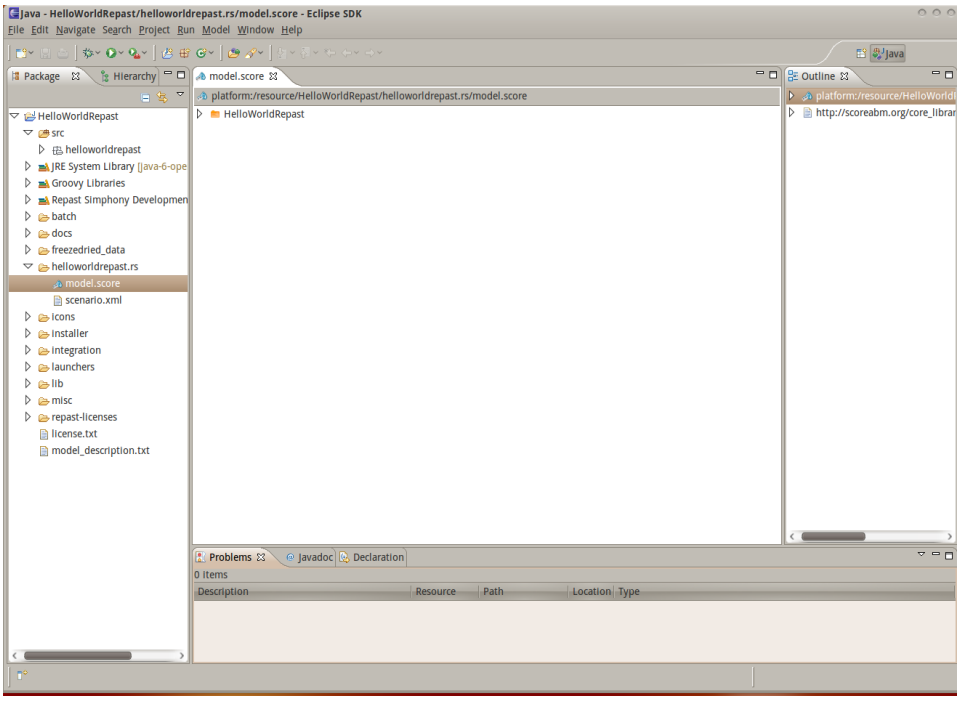

Abbildung 17: Neues Repast-Projekt in Eclipse

Nachdem diese Bibliotheken dem Classpath hinzugefügt wurden, kann man ein

neues Repast-Projekt in Eclipse wie gewohnt erstellen. Der Wizard erzeugt hierfür eine Vielzahl von verschiedenen Verzeichnissen, die man für das kleine Bespiel aber nicht weiter benötigt. Der relevante Ordner ist der src-Ordner für die Javaklassen und die model.score-Datei, die sich im <Projektname>.rs Verzeichnis des Projekts befindet. Hierbei handelt es sich um eine Metadatei, in der alle wichtigen Informationen über die Simulation, ihre Projektionen, unter denen man bei Repast jede mögliche Visualisierung der Simulation versteht, und die vorhandenen Agenten gespeichert werden. Wichtig ist, dass jede Klasse vorher in der Metadatei bekanntgegeben wird, da sie sonst in der Simulation selbst nicht verfügbar wäre.

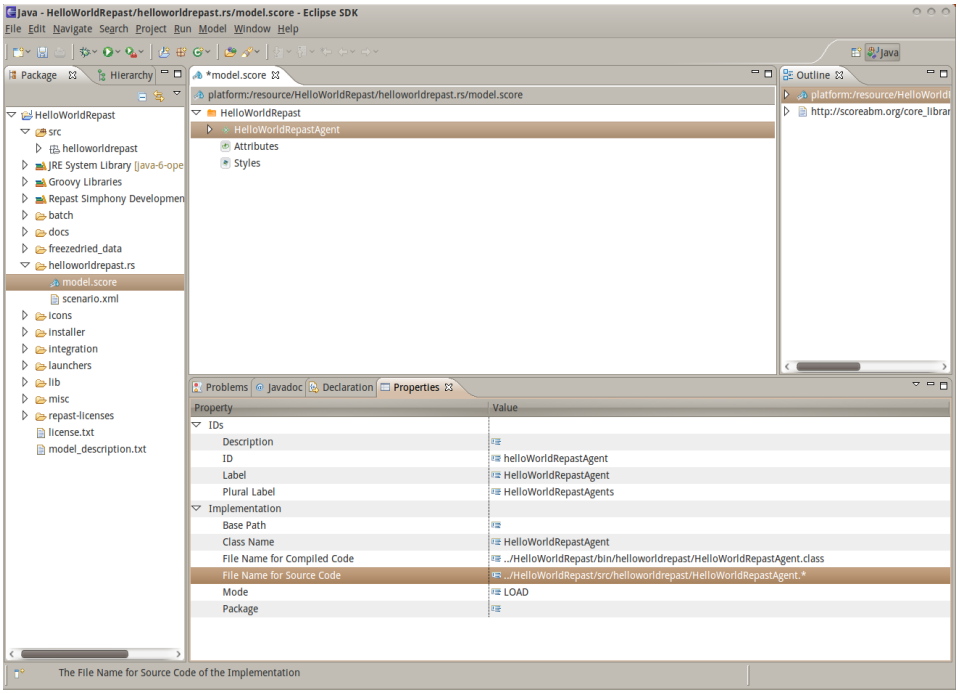

Abbildung 18: Die Deklaration eines neuen Agenten in Repast und die Property-Informationen zu dieser Entität.

Im ersten Schritt wird ein Kontext für die Agenten angelegt. Zwar kann man direkt den Rootkontext mit dem Namen des Projekts verwenden, es bietet sich aber an, die unterschiedlichen Agenten, von Beginn an, in separaten Kontexten zu organisieren. Um einen neuen Kontext der model.score Datei hinzuzufügen geht man wie folgt vor:

1. Man öffnet das Kontextmenü durch einen Rechtsklick auf den Rootkontext “HelloWorldRepast”.

2. In dem Menü findet man unter “Create Member” → “Context” die Option einen

neuen Kontext der Simulation hinzuzufügen.

Eclipse legt in den Metadaten des Projekts einen neuen Kontext mit dem Namen "HelloWorldRepastContext" an. Die dazugehörige Javaklasse muss mit genau diesem Namen dem src Ordner des Projekts hinzugefügt werden. Die *HelloWorldRepastContext*-Klasse muss die Klasse DefaultContext oder einer ihrer Superklassen erweitern. Innerhalb des Kontexts werden alle Individuen eines Agententyps verwaltet. Leider wird hier eine der Schwächen von Repast sichtbar. Es existiert zwar eine Javadoc, sie ist aber größtenteils leer. Die Entwickler haben die wenigstens Klassen vollständig dokumentiert und man muss sich in vielen Fällen den Nutzen einer bestimmten Klasse aus Beispielen der recht knappen Dokumentation oder anderen Onlinequellen, die sich selten auf die aktuelle Version von Repast beziehen, erschließen.

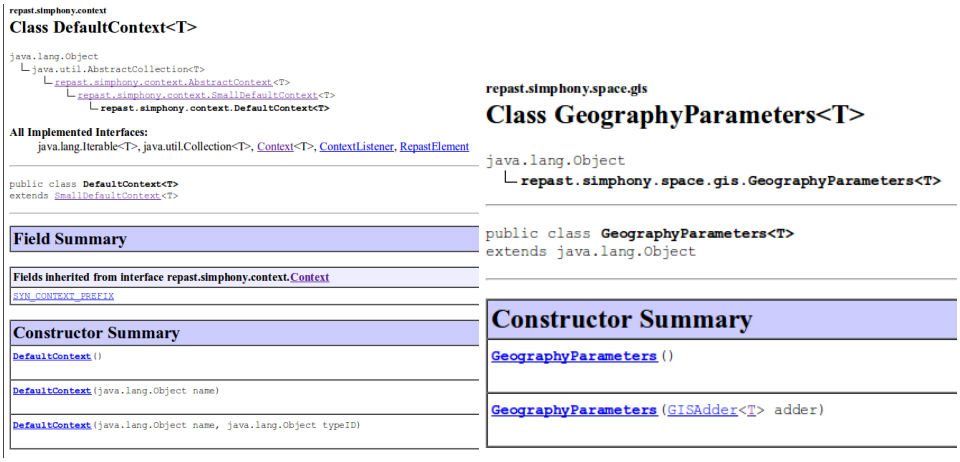

Abbildung 19: Zwei Beispiele für die vielen Klassen, die in der API von Repast nicht dokumentiert sind, aber regelmäßig verwendet werden

Der Kontext wird in diesem Beispiel zusätzlich dafür verwendet, neue Agenten, die die Nachricht ausgeben sollen, der laufenden Simulation hinzuzufügen. Agenten werden der Simulation auf eine ähnliche Weise bekanntgegeben wie ein neuer Kontext. Als erstes muss auch für den Agenten, der später die Hello World Nachricht ausgeben soll, ein Eintrag in den Metadaten der Simulation angelegt werden.

1. Man öffnet das Kontextmenü durch einen Rechtsklick auf den vorher erstellten Kontext "HelloWorldRepastContext".

2. Über "Create Member" → "Agent" legt man einen neuen Agenten innerhalb des Kontexts in der model.score an.

Es erscheint ein neuer Eintrag, den man nach Belieben umbenennen kann, falls der automatisch generierte Name des Agenten nicht passen sollte. In den Properties des Eintrags kann man nachlesen, wie der Agent in der Simulation bezeichnet wird und wo Repast die Klasse des Agenten erwartet. Bei Bedarf können auch diese Informationen angepasst werden.

Nachdem der Agent in den Metadaten deklariert wurde, erstellt man die entsprechende Klasse im src-Verzeichnis des Projekts. Hier befinden sich schon zwei Dateien, die vom Wizard automatisch erzeugt wurden und nur von dem grafischen Agenteneditor benötigt werden. Falls man diesen nicht verwenden möchte, kann man die zwei Dateien ohne Bedenken löschen.

Die Klasse wird über den New Class-Wizard von Eclipse erzeugt und entsprechend der Deklaration in den Metadaten der model.score benannt. Ein Agent in Eclipse kann jedes POJO (Plain Old Java Object) sein. Soll ein Agent eine bestimmte Methode automatisch in regelmäßigen Abständen oder zu einem abgestimmten Zeitpunkt ausführen, so annotiert man die Methode mit der @ScheduledMethod Annotation. Die ScheduledMethod Annotation nimmt unterschiedliche Parameter entgegen, wobei folgende drei für dieses Beispiel wichtig sind:

- start: die Anzahl der Iterationen, bevor diese Methode ausgeführt wird
- interval: die Anzahl der Iterationen, die diese Methode pausiert wird, bevor sie das nächste Mal wieder aufgerufen wird
- priority: gibt die Priorität dieser Methode an, was von Bedeutung sein kann, wenn es mehr als eine Methode gibt, die in einem Zeitschritt ausgeführt werden soll

Eine solche Methode soll in diesem Beispiel die HelloWorld-Nachricht ausgeben. Der Quelltext des Agenten sieht dann folgendermaßen aus:

Algorithm 2 HelloWorld-Agent in Repast. Die Methode sayHelloWorld() wird von Repast genau einmal im ersten Iterationsschritt der Simulation aufgerufen.

```
1 package helloworldrepast;
2 import repast.simphony.engine.schedule.ScheduledMethod;
3
4 public class HelloWorldRepastAgent {
5         @ScheduledMethod(start = 1, interval = 0, priority = 1)
6         public void sayHelloWorld() {
7                 System.out.println("Hello World from me!");
8         }
9 }
```

Nachdem der Agent als Eintrag in der model.score und als Javaklasse existiert, kann nun die Klasse für den HelloWorldRepastContext geschrieben werden. In dem Konstruktor des Kontexts wird auch der Agent angelegt, der die Hello World Nachricht über seine *sayHelloWorld* Methode ausgeben wird.

Algorithm 3 Der Kontext in Repast, in dem der Agent ausgeführt wird.

```
1  package helloworldrepast;
2  import repast.simphony.context.DefaultContext;
3
4  public class HelloWorldRepastContext extends
5          DefaultContext<HelloWorldRepastContextAgent>{
6          public HelloWorldRepastContext() {
7                  super("HelloWorldRepastContext");
8                  add(new HelloWorldRepastContextAgent());
9          }
10 }
```

Um die Simulation nun ausführen zu können, klickt man auf den Run Project Button und wählt als Run Configuration “Run HelloWorldRepast Model” aus. Es öffnet sich das Repast Hauptfenster, in dem man zum Starten der Simulation auf den Play Button in der oberen Toolbar klicken muss. Im Outputfenster von Eclipse sollte sich nun die Ausgabe des HelloWorldRepastAgents finden.

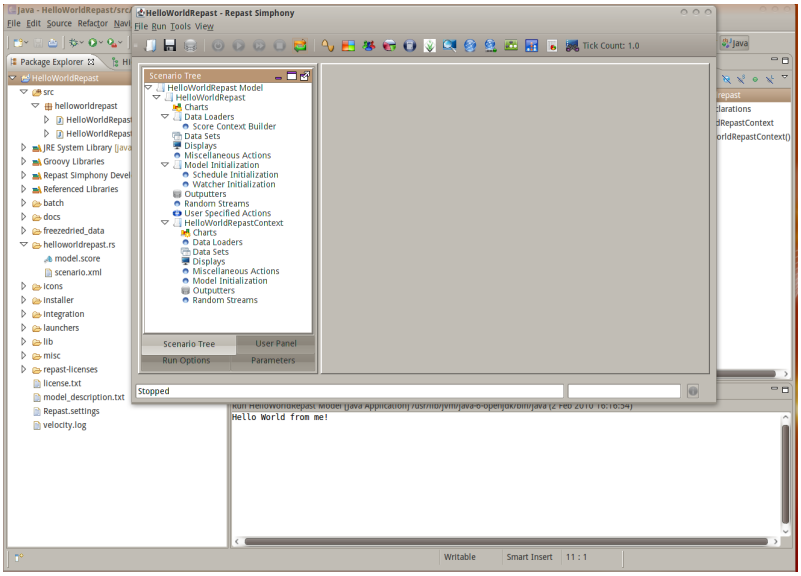

Abbildung 20: Das Repast Hauptfenster und die Ausgabe des Agenten in der Konsolenausgabe von Eclipse.

5.1.1.1 Fazit zu Repast Simphony

Zwar bietet Repast in seiner aktuellen Version eine Vielzahl von Möglichkeiten zur Erstellung von Multiagenten-Systemen, aber gerade die fehlende Dokumentatoin macht es schwer viele der Arbeitsschritte nachvollziehen zu können. Zusätzlich ist Repast Simphony die erste Version, die auf das Eclipse RCP aufbaut und dieses als Grundlage für weitere Entwicklungen macht.

Während der Evaluationsphase ist es häufiger passiert, dass das Repast Simphony Hauptfenster, das beim Starten eines Modells erscheint, nicht vollständig initialisiert werden konnte und man dadurch keine Kontrolle über die Anwendung hat. Leider konnte der Grund für dieses Fehlverhalten nicht herausgefunden werden. Diese Schwierigkeiten und die fehlende bzw. unzureichende Dokumentation haben dafür gesorgt, dass dieses Framework für die Entwicklung des Prototypen in dieser Arbeit nicht weiter in Betracht gezogen werden konnte.

5.1.2 JADE

JADE (22) ist ein Toolkit zur Entwicklung von agentenbasierten Anwendungen. Es wird von der Telecom Italia entwickelt und unter der GPL zur Verfügung gestellt. Das Framework ist FIPA[1]-konform und bietet dadurch die Möglichkeit mit anderen, heterogenen Systemen Informationen über Systemgrenzen hinaus austauschen zu können. Im Gegensatz zu Repast bietet JADE keine integrierte Möglichkeit zur Visualisierung der Agenten in einem geographischem Raum an. Dies liegt vor allem an den unterschiedlichen Zielsetzungen der beiden Projekte. Repast richtet sich direkt an eine wissenschaftliche Community, die mit Multiagenten-Systemen arbeitet und diese visualisieren möchte, JADE dagegen ist ein Framework zur Erstellungen von agentenbasierten Softwaresystemen, in denen die einzelnen Komponenten der Software autonome Agenten sind. Da JADE an kein RCP gebunden ist und bei den Kernfunktionalitäten auch keine anderen Abhängigkeiten an externen Bibliotheken besitzt, die nicht im J2SE zu finden sind, ist das Framework wesentlich leichtgewichtiger.

Die Architektur von JADE baut auf der Referenzarchitektur des FIPA-Standards auf, die beide im Programmer's Guide (9) von JADE vorgestellt werden. Dieser Überblicksartikel wird regelmäßig von den Entwicklern aktualisiert und bietet einen guten, technischen Überblick über die wichtigsten Funktionen des Framework, beinhaltet aber noch nicht die aktuellen Änderungen aus dem letzten Jahr, die mit der Version 3.7 eingeführt wurden.

[1] FIPA = Foundation for Intelligent Physical Agents ist eine Standardisierungsorganisation der IEEE, die Softwarestandards für die Entwicklung von Multiagenten-Systemen in heterogenen Systemen bietet. http://www.fipa.org/

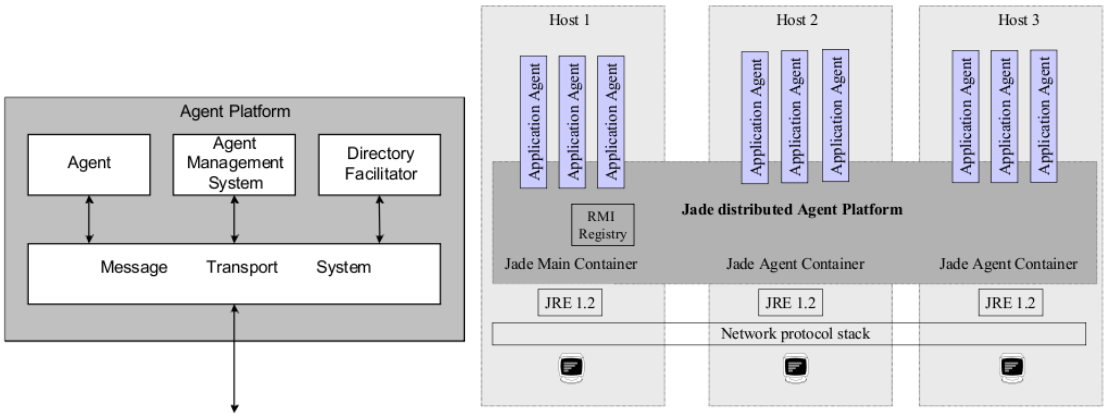

Abbildung 21: Die FIPA-Standard Referenzarchitektur (links) und ihre Implementierung in der JADE Architektur (rechts) (9)

Die Referenzarchitektur des FIPA-Standards besteht aus vier Komponenten:

Agent der Agent ist die Komponente, die die Programmlogik der Anwendung enthält

Agent Management System (AMS) das AMS verwaltet den Lebenszyklus des Agenten, sowie die einmaligen IDs der Agenten und bietet ein White Pages Service[2] für die Agenten zur Laufzeit an

Directory Facilitator (DF) der Directory Facilitator bietet den Agenten zur Laufzeit einen Yellow Pages Service[3] an, den die Agenten durchsuchen oder modifzieren können

Message Transport System (MTS) das MTS ist eine Softwareschnittstelle, die sich um die Kommunikation zwischen den einzelnen Komponenten innerhalb einer Plattform und zwischen unterschiedlichen Plattformen kümmert

Die Architektur von JADE bildet diese Referenzarchitektur genau ab. Als Agent Plattform fungiert bei JADE die JVM, in der die Agenten, das AMS und der DF über RMI mit Remotesystemen verbunden werden. Das AMS und der DF sind Agenten, die bei Systemstart automatisch erzeugt werden und nur im Hauptcontainer existieren. Alle Agenten, auch die auf Remotesystemen existieren, verwenden das AMS und den DF des Hauptcontainers über die Remoteverbindung.

[2] *White Pages Service* bezeichnet in diesem Zusammenhang einen Dienst, der jedem Agenten eine eindeutige Nummer zuweist, über die dieser gefunden werden kann.

[3] *Yellow Pages Service* bezeichnet einen Dienst, bei dem sich Agenten mit einem eigenen Dienst, den sie anbieten, registrieren können, so dass sie von anderen Agenten gefunden werden können, um die entsprechende Dienstleistung beanspruchen zu können.

Ein in JADE erstellter Agent registriert sich als erstes automatisch mit dem AMS und bekommt eine eindeutige ID (die *AID*) zugewiesen. Mit dieser kann ein Agent direkt angesprochen werden. Optional nach der Registrierung beim AMS kann der Agent sich selbst beim DF mit den jeweiligen, eigenen Services anmelden. Dies ist zwar kein Muss, wird aber als Good Practice von den JADE-Entwicklern empfohlen.

Ein Agent in JADE sollte so aufgebaut sein, dass seine auszuführende Logik in Verhaltensklassen gekapselt wird, die bei Bedarf von dem Agenten instantiiert werden können. Da der Prototyp starken Gebrauch von unterschiedlichen Verhaltensklassen macht, soll in zwei Beispielen ihre Arbeitsweise verdeutlicht werden. Das erste Beispiel ist ein einfacher Hello World Agent, der "Hello World" auf der Konsole ausgeben soll. Das zweite Beispiel demonstriert die Interaktion zwischen zwei aktiven Verhaltensweisen, die aufeinander reagieren, indem das erste Verhalten "Hello" und das zweite "World" auf der Konsole in regelmäßigen Intervallen ausgibt.

JADE bietet einige grundlegende Verhaltensklassen (Unterklassen der Klasse *jade.core.behaviours.Behaviour*), die für die meisten Fälle ausreichen. Hierunter fallen

OneShotBehaviour ein Behaviour, das den enthaltenen Code genau einmal ausführt

CyclicBehaviour ein Verhalten, das periodisch ausgeführt wird

TickerBehaviour ein Verhalten, das, in im Konstruktor definierten Intervallen, den enthaltenen Code ausführt

SimpleBehaviour ein Verhalten, das als Grundlage für eigene Implementierungen dienen kann. Es führt seinen Code aus, solange die Methode done() *false* zurückgibt

Der Agent für das erste Beispiel lässt sich in wenigen Zeilen realisieren. Jeder Agent in Jade erweitert die Klasse *jade.core.Agent*. Notwendige Initialisierungen, wie das Instantiieren von Verhaltensklassen, können in der Methode *setup* geschehen, die für diesen Zweck überschrieben wird. Für die einmale Ausgabe auf der Konsole bietet sich das Benutzen des *OneShotBehaviour*s an. Mit *addBehaviour(Behaviour newBehaviour)* kann das gewünschte Verhalten dem Agenten in der *setup* Methode hinzugefügt werden. Da es sich um ein übersichtliches Behaviour handelt, wird es als anynome Klasse übergeben. Der Aufruf der Methode *doDelete* (Alg. 4, Z. 12) entfernt den Agenten aus dem System und beendet die JADE-Plattform, da dies der einzige Agent in der Plattform war.

Algorithm 4 Einfacher "Hello World"-Agent in JADE

```
1  package helloworld;
2  import jade.core.Agent;
3  import jade.core.behaviours.OneShotBehaviour;
4
5  public class HelloWorldJADEAgent extends Agent{
6      @Override
7      protected void setup() {
8          addBehaviour(new OneShotBehaviour() {
9              @Override
10             public void action() {
11                 System.out.println("Hello World!");
12                 doDelete();
13             }
14         });
15     }
16 }
```

Das zweite Beispiel demonstriert die Verwendung des *CyclicBehaviour*, des *TickerBehaviour* und wie man die Klasse *jade.lang.acl.ACLMessage* einsetzt, um eine Kommunikation zwischen Agenten zu realisieren.

Algorithm 5 Der *HelloAgent* in JADE

```
1  package helloworld;
2  import jade.core.AID;
3  import jade.core.Agent;
4  import jade.core.behaviours.TickerBehaviour;
5  import jade.lang.acl.ACLMessage;
6
7  public class HelloAgent extends Agent {
8      @Override
9      protected void setup() {
10         addBehaviour(new TickerBehaviour(this, 5000) {
11             private int counter;
12             @Override
13             protected void onTick() {
14                 if (counter > 4) {
15                     doDelete();
16                 } else {
17                     counter++;
18                     System.out.print("Hello ");
19                     ACLMessage worldRequest = new ACLMessage(ACLMessage.REQUEST);
20                     worldRequest.setConversationId("SAY WORLD");
21                     worldRequest.addReceiver(new AID("worldAgent", AID.ISLOCALNAME));
22                     send(worldRequest);
23                 }
24             }
25         });
26     }
27 }
```

5.1.2.1 HelloAgent in JADE

Ein *TickerBehaviour* oder eine Unterklasse davon nimmt im Konstruktor die Instanz des Agenten (Alg. 5, Z.10), dem dieses Behaviour gehört, und ein Zeitintervall in Millisekunden entgegen. In diesem Beispiel soll das *TickerBehaviour* alle fünf Sekunden aktiv werden. Bei jeder Ausführung wird der Code in der *onTick* Methode ausgeführt. Damit das Programm nur fünfmal auf der Konsole Hello World ausgibt, wird bei jedem Durchlauf ein Counter um eins hochgezählt. Beim fünften Durchlauf wird *onDelete* aufgerufen und das Programm dadurch beendet. Das *TickerBehaviour* erstellt, nach dem Ausgeben der "Hello"-Nachricht auf der Konsole, eine *ACLMessage*.

Eine *ACLMessage* ist eine Nachricht, die zwischen den Agenten verschickt werden kann, um Informationen auszutauschen. Sie ist FIPA-konform und kann daher auch von anderen Agenten, die nicht in Java programmiert wurden, verstanden werden.

Die Integerkonstante, die der Konstruktor einer *ACLMessage* entgegennimmt definiert die Art der Nachricht, in diesem Fall ein Request an einen anderen Agenten. Die nächste Zeile ist in diesem kleinen Beispiel eher optional, da das Verhalten des *WorldAgent* nur in einer Form auf die empfangenen Nachrichten reagieren kann. Da in dem Prototypen *ConversationIDs* häufig Verwendung finden, soll der Gebrauch dieser in diesem Beispiel mit verdeutlicht werden.

Die *ConversationIDs* sind die Betreffzeilen der verschickten *ACLMessage* und können von Agenten zur Filterung der eigenen Messagequeue verwendet werden, damit sie deren Bearbeitung an ein separates Behaviour deligieren können.

Der letzte Schritt vor dem Senden der Nachricht ist das Hinzufügen des Empfängers zu der *ACLMessage*. Das dies nicht auf nur einen Empfänger beschränkt ist, erkennt man auch an dem Namen der Methode, die der Nachricht eine zusätzliche *AID* eines Empfängers hinzufügt. Da es in dem Beispiel nur einen weiteren Agenten gibt, dessen Namen bekannt ist, kann die *AID* des Empfängers über das Erzeugen eines neuen *AID* Objekts erzeugt werden. Der erste Parameter ist der Name des Agenten, wie man ihn bei seiner Instantiierung festgelegt hat. Dieser Name wird in JADE als *localname* bezeichnet und muss innerhalb des Konstruktors als solcher über den zweiten Parameter gekennzeichnet sein. In größeren Systemen, in denen man keine Kenntnisse aller lokalen Namen voraussetzen kann, ist der übliche Weg, den DF zu befragen. Dieser liefert eine Liste von *AID*s zurück, die als Empfänger der Nachricht dienen können.

Die *send* Methode verschickt die Nachricht. Diese wird von der Plattform an den Empfänger übermittelt und in seine Messagequeue platziert.

Algorithm 6 Der WorldAgent in JADE

```
1  package helloworld;
2  import jade.core.Agent;
3  import jade.core.behaviours.CyclicBehaviour;
4  import jade.lang.acl.ACLMessage;
5  import jade.lang.acl.MessageTemplate;
6
7  public class WorldAgent extends Agent{
8      @Override
9      protected void setup() {
10         addBehaviour(new CyclicBehaviour() {
11             @Override
12             public void action() {
13                 ACLMessage m = receive(MessageTemplate.MatchConversationId("SAY WORLD"));
14                 if(m != null){
15                     System.out.println("world!");
16                 }else{
17                     block();
18                 }
19             }
20         });
21     }
22 }
```

5.1.2.2 WorldAgent in JADE

Das CyclicBehaviour des WorldAgents erwartet eine Nachricht mit der ConversationID “SAY WORLD” und reagiert auf diese mit einer Ausgabe des Worts “hello” auf der Konsole. Kommt eine Nachricht mit dieser ID an, so schreibt der Agent das Wort und wartet auf die nächste Nachricht, die mit dieser ConversationID in der Nachrichtenschlange des Agenten erscheint. Damit das CyclicBehaviour nicht konstant aufgerufen und die Messagequeue nach einer passenden Nachricht durchsucht wird, wird am Ende, wenn keine abzuarbeitende Nachricht mehr in der Schlange vorhanden ist, im else-*Teil die block* Methode des Behaviours aufgerufen. Dieser Aufruf bewirkt, dass das CyclicBehaviour solange nicht mehr ausgelöst wird, bis eine neue Nachricht in der Messagequeue des Agenten vorhanden ist. *Damit* es bei einer leeren Messagequeue nicht zu einer versehentlichen *NullPointerException* kommt, empfehlen die JADE-Entwickler den hier verwendeten Standardaufbau der Nachrichtenverarbeitung.

1. Nachricht aus der Messagequeue abholen
2. Überprüfen, ob das *ACLMessage-Objekt* nicht *null* ist.
   a) Ist Referenz nicht *null*, wird die Nachricht verarbeitet.
   b) Ist Referenz *null,* so wird das Verhalten mit *block()* suspendiert, bis eine neue Nachricht in der Nachrichtenschlange ankommt.

5.1.2.3 Fazit zu JADE

JADE bietet ein umfassendes Framework, das allgemein zur Entwicklung von Software auf Basis von autonomen Agenten verwendet werden kann. Dies ist ein grundlegender Unterschied zu Repast; hier ist das Haupteinsatzgebiet die Simulation von Multiagenten-Systemen im hautpsächlich akademischen Bereich. Aus diesem Grund bietet JADE auch keine integrierten Möglichkeiten zur Visualisierung der Agenten. Diese muss, falls sie benötigt wird, von Hand selbst erstellt werden, was unter Umständen einen Mehraufwand bedeuten kann. In diesen Fällen liegt die Wahl des passenden Frameworks an der Ausrichtung des Projekts und ob man die fehlende Dokumentation durch die Mailinglist kompensieren kann.

Ein weiterer Punkt, der für die Wahl von JADE als Basis des Prototypen sprach, ist die sehr aktive Mailinglist des Projekts, in der jede ankommende Frage von den hauptverantwortlichen Entwicklern bei der Telecom Italia selbst binnen kurzer Zeit beantwortet wird. Gerade Anfängerfragen werden dort genauso schnell und zuvorkommend behandelt, wie weiterführende Fragen oder Probleme.

Die gute Konzeption des Frameworks, die vielen Tools, die mit der Plattform mitgeliefert werden, und der schnelle, im Ganzen problemlose, Einstieg durch die Guides und mitgelieferten Beispielanwendungen ließen die Wahl auf JADE fallen.

5.2 Implementierte Architektur

Der in dieser Arbeit implementierte Prototyp versucht das in Kapitel 4 beschriebene Modell so genau wie möglich umzusetzen. Wie auch das Modell, besteht der Prototyp aus zwei wesentlichen Komponenten. Die eine ist die Repräsentation der Welt, in der die Personen existieren, die andere ist die Person selbst, die in Interaktion mit der Welt und anderen Agenten steht.

Da der Prototyp viele unterschiedliche Behaviours implementiert, um das Verhalten des Modells in JADE abzubilden, sind die Behaviours auf Seite der Simulation strikt nach ihrem Aufgabenbereich benannt. Folgende Nomenklatur wird dafür verwendet:

- Der erste Teil des Namen steht für die Oberkategorie (Person, Scent (Verhalten die duftstoffrelevante Behaviours implementieren), Building).
- Der zweite Teil des Namens beschreibt die Funktion des jeweiligen Verhaltens.

Die Abbildung des Domänenklassendiagramms verdeutlicht die Struktur der für den Prototypen wichtigsten Klassen.

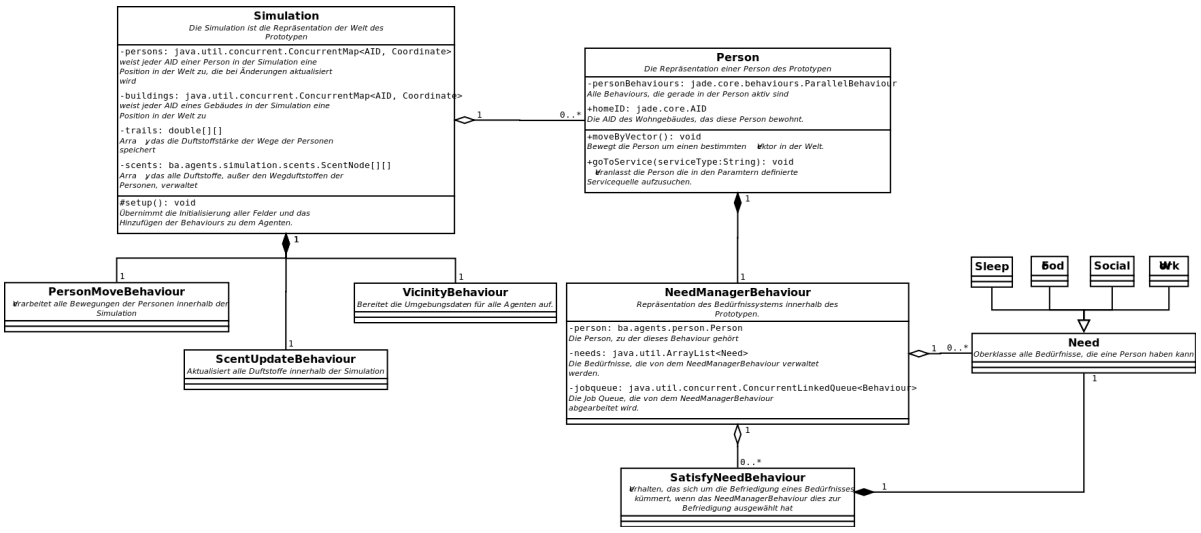

Abbildung 22: Das Domänenklassendiagramm des Prototypen

Die Simulation verwaltet den globalen Zustand der Welt, zu dem das Pheromonsystem, das die einzelnen Duftstoffe, die in der Welt zu finden sind, und die Position der Gebäude sowie der Personen gehört. Es wird hierbei, wie schon in der Besprechung des Modells erwähnt, zwischen einfachen und komplexen Duftstoffen unterschieden.

Die einfachen Duftstoffe kommen in dem Prototypen nur in Form der von der Person gegangenen Wege vor. Die komplexen Duftstoffe sind entweder direkt an eine bestimmte Person in der Simulation gerichtet oder fungieren als ein Broadcasting-Kommunikationsmittel.

5.2.1 Simulation der Welt

Die Simulation ist die Repräsentation der Welt und verwaltet den Zustand aller Agenten. Darunter fallen die Personen, die sich in der Welt befinden und die Gebäude, die in der Welt gebaut wurden. Damit die Simulation die einzelnen Anfragen der Personen und Gebäude bearbeiten kann, gibt es eine Vielzahl unterschiedlicher Behaviours, die nach der Instanziierung der Simulation in der *setup* Methode geladen werden.

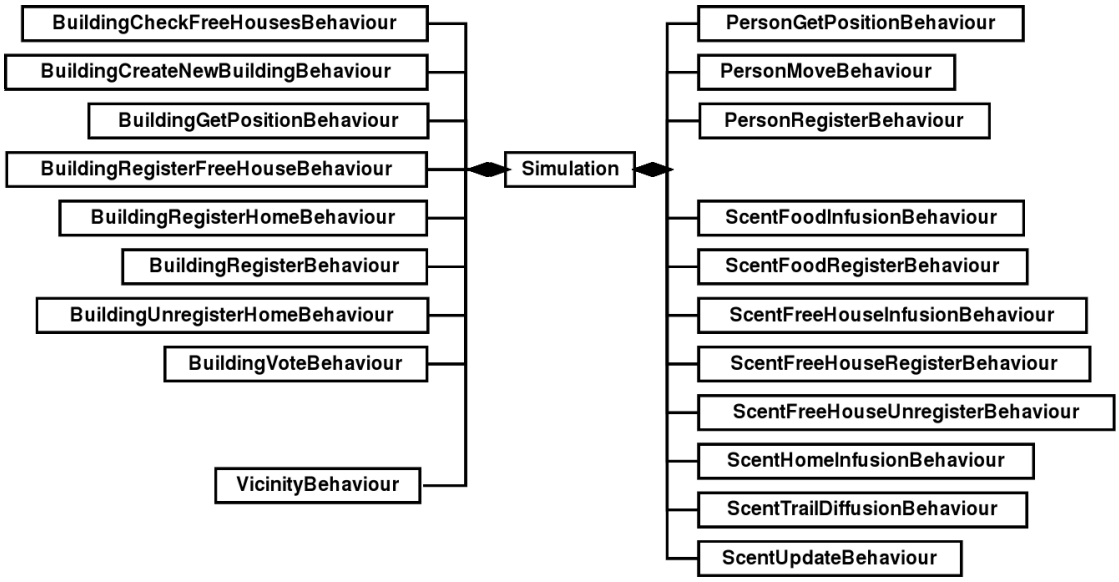

Abbildung 23: Klassendiagramm der Simulation und der implementierten Behaviours

Die Behaviours repräsentieren die einzelnen Aufgaben, die die Simulation in diesem Prototypen übernimmt. Die in den Behaviours gekapselte Logik wird entweder über eine *ACLMessage* mit einer der Aktion entsprechenden *ConversationID*, die man als Konstante dem Interface *ba.util.CID* entnehmen kann, ausgelöst oder es sind Unterklassen des TickerBehaviours, die in Intervallen ausgeführt werden, die in der Simulationklasse als Konstante definiert sind.

Die Hauptaufgabe der Simulationklasse besteht darin, die in der Messagequeue ankommenden Nachrichten entgegenzunehmen. Das getriggerte Behaviour übernimmt für die jeweilige Aufgabe die Arbeit. Dieses Konzept hat den Vorteil, dass Verhaltensmuster durch ihre Kapselung gut zu finden und auch auszutauschen sind. Zusätzlich bietet es auch die Möglichkeit, Behaviourklassen über Reflection dynamisch nachzuladen, ohne das die Simulationsklasse von der neuen Funktionalität etwas wissen muss. Diese Idee wird aber in dieser Arbeit nicht weiter verfolgt.

Bevor nun die Arbeitsweise der wichtigsten Behaviours und der Simulation als Ganzes näher betrachtet wird, sollen die einzelnen, implementierten Behaviourklassen der Übersicht halber in ihrer Bedeutung für die Simulation angesprochen werden.

BuildingCheckFreeHousesBehaviour Dieses Verhalten überprüft auf Anfrage eines Agenten, ob es noch freistehende Wohnhäuser in der Simulation gibt.

BuildingCreateNewBuildingBehaviour Dieses Verhalten wird in regelmäßigen Intervallen aufgerufen und baut ein neues Gebäude des Gebäudetyps, das per Abstimmung von den meisten Agenten verlangt wird.

BuildingGetPositionBehaviour Dieses Hilfsverhalten, das benutzt werden kann, um die genaue Position eines Gebäudes mitgeteilt zu bekommen.

BuildingRegisterFreeHouseBehaviour Dieses Verhalten wird von neu gebauten Wohnhäusern benutzt, um sich als unbewohnt zu melden oder von Wohnungen, aus denen Personen wieder ausziehen. Nach der Aktivierung dieses Verhaltens befindet sich das Wohnhaus wieder im Pool der bezugsfähigen Gebäude.

BuildingRegisterHomeBehaviour Dieses Verhalten registeriert eine neue Wohnung - Person Beziehung, nachdem eine Person in ein freistehendes Haus eingezogen ist.

BuildingUnregisterHomeBehaviour Dieses Verhalten entfernt eine existierende Wohnung - Person Beziehung, wenn eine Person aus einem Haus wieder auszieht.

BuildingVoteBehaviour Dieses Verhalten nimmt die Abstimmung der Personen entgegen, die vorschlagen ein neues Gebäude bauen zu lassen. Die in diesem Verhalten gesammelten Stimmen werden vom *BuildingCreateNewBuildingBehaviour* ausgewertet.

PersonGetPositionBehaviour Dieses Verhalten teilt dem Sender dieser Anfrage seine aktuelle Position mit.

PersonMoveBehaviour Dieses Verhalten bewegt eine Person um einen, in der *ACL-Message* angegebenen, Vektor. Gleichzeitig sorgt dieses Verhalten dafür, dass ein entsprechender Duftstoff an der Position des Agenten hinterlassen wird.

PersonRegisterBehaviour Dieses Verhalten registriert eine neue Person bei der Simulation.

Da jegliche Entscheidungen von den Personen auf Grundlage von Duftstoffen gefällt werden, soll in dem nächsten Abschnitt über die Implementierung des Pheromonsystems gesprochen werden. Gerade dieser Teil war wesentlich arbeitsaufwendiger als zu Beginn des Projekts gedacht war. Die Gründe hierfür sollen nachfolgend erläutert werden.

5.2.1.1 Das Pheromonsystem

Die Art und Weise wie die Simulation die Duftstoffe verwaltet wurde im Laufe dieser Arbeit mehrfach verändert. Der erste Ansatz war, alle Duftstoffe nach dem gleichen Prinzip, wie auch die Duftspuren der Wege verwaltet werden, zu implementieren. Diese werden global in einem Zahlenarray kummulativ gespeichert und bei den Bewegungen der Personen entsprechend verändert.

In dieser Implementierung konnte der Agent die einzelnen Gebäude und den Rückweg zu seiner Wohnung nur finden, weil er die Simulation nach den Koordinaten seins Ziels fragen konnte. Er verwendete die Duftstoffe nur, um sich bezüglich der Wege zu orientieren. Eine Person war aber in dieser Variante noch nicht in der Lage, nur mit lokalem Wissen ein bestimmtes Gebäude zu finden.

Dass Personen nur mit lokalem Wissen auskommen, war aber eine der Anforderungen an den Prototypen. Die erste Verbesserung der Implementierung war, dass die fraglichen Stellen, in denen auf globales Wissen zugegriffen wurde, verändert wurden, indem für jedes vorhandene Gebäude ein neues Array mit dem Duftstoff des Gebäudes eingeführt wurde. Dies führte aber zu dem Problem, dass eine beliebig große Anzahl von Arrays verwaltet werden können musste, da jedes Gebäude und jede Person sein eigenes Array für jeden möglichen Duftstoff benötigte. Mit den neu gewonnen Erfahrungen konnte dann das Pheromonsystem entworfen und implementiert werden, wie es in der aktuellen Fassung des Prototypen eingesetzt wird.

Das Pheromonsystem baut auf Scent-Objekten auf, die in einem Array von Scent-Nodes verwaltet werden. Dies bietet den Vorteil, dass nur noch ein Array für eine beliebige Anzahl von unterschiedlichen Duftstoffen benötigt wird und diese durch ihre Repräsentation als Klasse an mögliche, geänderte Anforderungen angepasst werden

können.

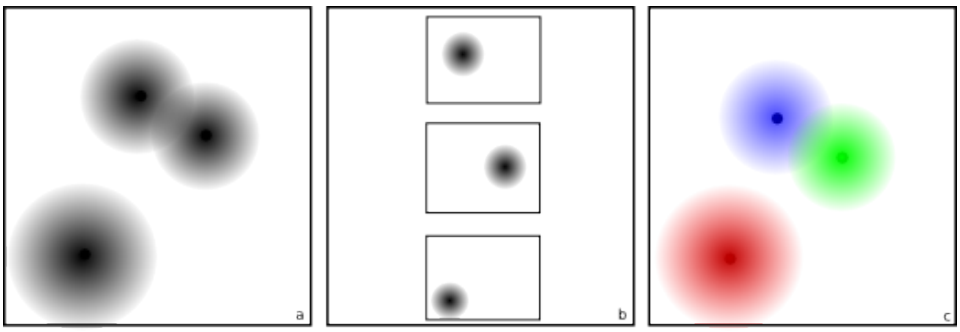

Abbildung 24: Unterschiedliche Ansätze zur Verwaltung der unterschiedlichen Pheromone. Alle Duftstoffe eines Gebäudetyps (z.B. Wohnhäuser) werden in einem Array gehalten (a). Für jedes Gebäude wird ein separates Array verwendet (b). Es wird ein Array für alle Gebäude verwendet, aber jedes Gebäude hat seinen eigenen Duft (c).

5.2.1.1.1 ScentNodes

Die Klasse *ScentNode* speichert Informationen der verschiedenen Duftstoffe, die über Unterklassen der Klasse *Scent* repräsentiert werden. Sollte der Simulation eine weitere Klasse von Duftstoffen hinzugefügt werden, die durch die vorhandenen Kategorien nicht abgedeckt wird, so muss die Klasse *ScentNode*, die für jede Klasse von Duftstoffen eine Implementierung des Interfaces *java.util.List* bereitstellt und zur Verwaltung der in der Simulation vorhandenen Duftstoffe verwendet wird, entsprechend angepasst werden.

Wird ein neuer Duftstoff bei der Simulation registriert, so wird der einfacheren Handhabung wegen dieser Duftstoff mit einem Wert von 0 jeder *ScentNode* hinzugefügt, um sich unnötige Überprüfungen, ob ein Duftstoff in einer Zelle vorhanden ist oder nicht, beim Update der Node sparen zu können. Die entsprechende Registrierungsmethode muss in der ScentNode vorhanden sein oder gegebenfall hinzugefügt werden, wenn es sich um einen neuen Duftstoff handelt, der dem Prototypen hinzugefügt wurde.

Die *ScentNode* bietet die Methoden *updateNode* und *decay(double rate)* an, die von der Simulation über das Behaviour *ScentUpdateBehaviour* zyklisch aufgerufen werden. Beide Methoden sind dafür zuständig, dass den in dem Modell geforderten Eigenschaften der Duftstoffe (örtliche Diffusion und zeitlicher Verfall) Rechnung getragen wird. *UpdateNode* führt eine Diffusion mit den Nachbarzellen aus, *decay(double rate)* reduziert die Stärke eines Duftstoffes um einen Wert, der in der Klasse der

Simulation definiert werden kann.

Algorithm 7 Die updateNode Methode der Klasse ScentNode, in der die lokale Diffusion durchgeführt wird. Die Variablen indexX und indexY stehen für den Index der aktuellen Node in dem Array. Neighbours ist eine Referenz auf das gesamte ScentNode-Array, um den Zugriff auf die Nachbarzellen gewährleisten zu können.

```
1  public void updateNode() {
2     for (int i = 0; i < foodScents.size(); i++) {
3        FoodScent s = foodScents.get(i);
4        double newVal = (neighbours[indexX + 1][indexY].getScent(s).getValue()
5                    + neighbours[indexX - 1][indexY].getScent(s).getValue()
6                    + neighbours[indexX][indexY + 1].getScent(s).getValue()
7                    + neighbours[indexX][indexY - 1].getScent(s).getValue()
8                    + neighbours[indexX - 1][indexY - 1].getScent(s).getValue()
9                    + neighbours[indexX + 1][indexY - 1].getScent(s).getValue()
10                   + neighbours[indexX - 1][indexY + 1].getScent(s).getValue()
11                   + neighbours[indexX + 1][indexY + 1].getScent(s).getValue()) / 8;
12       s.setValue(newVal);
13    }
14    for (int i = 0; i < houseScents.size(); i++) {
15       FreeHouseScent s = houseScents.get(i);
16       double newVal = (neighbours[indexX + 1][indexY].getScent(s).getValue()
17                   + neighbours[indexX - 1][indexY].getScent(s).getValue()
18                   + neighbours[indexX][indexY + 1].getScent(s).getValue()
19                   + neighbours[indexX][indexY - 1].getScent(s).getValue()
20                   + neighbours[indexX - 1][indexY - 1].getScent(s).getValue()
21                   + neighbours[indexX + 1][indexY - 1].getScent(s).getValue()
22                   + neighbours[indexX - 1][indexY + 1].getScent(s).getValue()
23                   + neighbours[indexX + 1][indexY + 1].getScent(s).getValue()) / 8;
24       s.setValue(newVal);
25    }
26    for (int i = 0; i < homeScents.size(); i++) {
27       HomeScent s = homeScents.get(i);
28       double newVal = (neighbours[indexX + 1][indexY].getScent(s).getValue()
29                   + neighbours[indexX - 1][indexY].getScent(s).getValue()
30                   + neighbours[indexX][indexY + 1].getScent(s).getValue()
31                   + neighbours[indexX][indexY - 1].getScent(s).getValue()
32                   + neighbours[indexX - 1][indexY - 1].getScent(s).getValue()
33                   + neighbours[indexX + 1][indexY - 1].getScent(s).getValue()
34                   + neighbours[indexX - 1][indexY + 1].getScent(s).getValue()
35                   + neighbours[indexX + 1][indexY + 1].getScent(s).getValue()) / 8;
36       s.setValue(newVal);
37    }
38 }
```

Wollte man eine andere Art von Diffusion oder Abklang der Simulation hinzufügen, so wäre dies die einzige Stelle, in der diese Änderung vorzunehmen wär. Denkbar wäre natürlich auch eine Implementierung als Klasse, die zur Diffusion verwendet werden würde. Aufgrund der begrenzten Zeit dieser Arbeit wurde aber darauf verzichtet, die Klassenstruktur komplexer zu gestalten.

5.2.1.2 Gebäude

Gebäude sind die Repräsentation aller Ressourcenquellen innerhalb der Simulation. Ihre Grundklasse ist, wie in der folgenden Abbildung zu sehen, die Klasse *ba.agents.buildings.Building* und wird von den entsprechenden Gebäudetypen, die in der Simulation vorkommen können, erweitert. Sie bietet die grundlegende Funktionalität, die benötigt wird, um ein Gebäude bei der Simulation zu registrieren, so dass die Unterklassen nur für die spezifischen Teil der Logik verantwortlich sind.

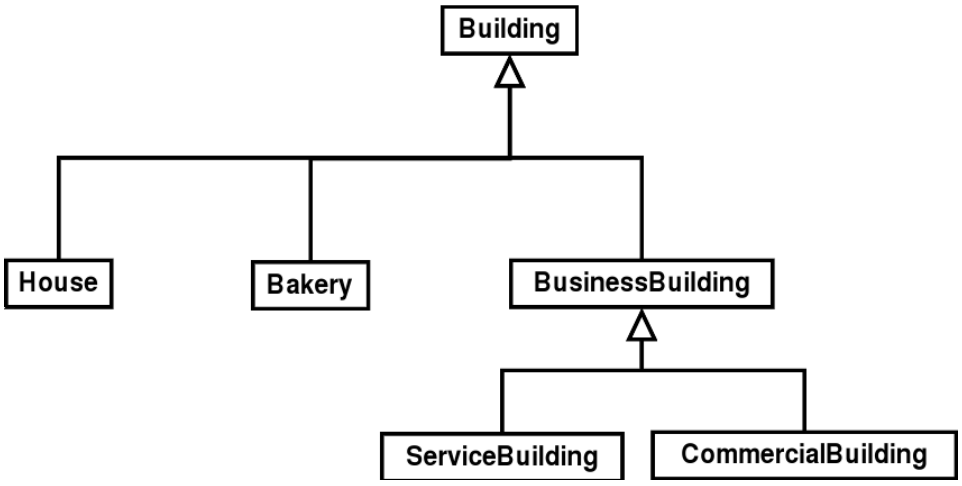

Abbildung 25: Klassendiagramm mit den im Prototypen implementierten Gebäudeklassen.

Damit Gebäude in der Simulation angelegt werden können, muss ihnen ihre Position als Parameter mitgeteilt werden. Da Gebäude nur über die Simulation selbst erstellt werden, geschieht dies über einen Aufruf in dem *BuildingCreateNewBuildingBehaviour* der Simulation.

Das Bauen eines neuen Gebäudes innerhalb des Prototypen ist von einem Voting der Personen abhängig, die bei Bedarf für den Bau eines neuen Gebäudes stimmen können.

Algorithm 8 Die createNewAgent Methode des *BuildingCreateNewBuildingBehaviour*s

```
1  private void createNewAgent(String agentName, Class<? extends Building> clazz) {
2      double pos[] = getBuildingPosition(clazz);
3      AgentContainer container = myAgent.getContainerController();
4      Double params[] = {pos[0], pos[1]};
5      AgentController controller = container.createNewAgent(agentName, clazz.getName(), params);
6      controller.start();
7  }
```

Der Ablauf des Erstellens eines neuen Gebäudes in der Simulation ist

1. Zählen der Stimmen der Personen, die ein neues Gebäude in der Welt benötigen in dem *BuildingVoteBehaviour.*
2. Auswerten der Stimmen in dem *BuildingCreateNewBuildingBehaviour.*
3. Erzeugen einer neuen Position für das Gebäude des benötigten Gebäudetyps in der Methode *getBuildingPosition* der Klasse *BuildingCreateNewBuildingBehaviour.*
4. Den Container der JADE-Plattform, der für das Gebäude zuständig sein soll, anweisen, einen neuen Agenten eines bestimmten Typs zu erzeugen.
5. Über den Controller des neu erstellten Agenten, das Gebäude starten.
6. Das Gebäude benutzt die Koordinaten, um sich selbst bei der Simulation über ein *RegisterBehaviour* zu registrieren.
7. Nach der Initialisierung registriert das Gebäude seinen Duftstoff bei der Simulation, um seinen Service den Personen anbieten zu können.

Abhängig von der Art des Gebäudes, das erstellt werden soll, wird eine gebäudetypabhängige neue Position für das Gebäude erzeugt und als ein Object-Array an die Methode *createNewAgent* des *AgentContainers* übergeben. Dadurch wird eine neue Instanz der übergebenen Klasse erzeugt. Die über das Object-Array an den Agenten weitergereichten Koordinaten werden von dem Gebäude verwendet, um sich über das *RegisterBehaviour* im Package *ba.agents.buildings.behaviours* bei der Simulation anzumelden.

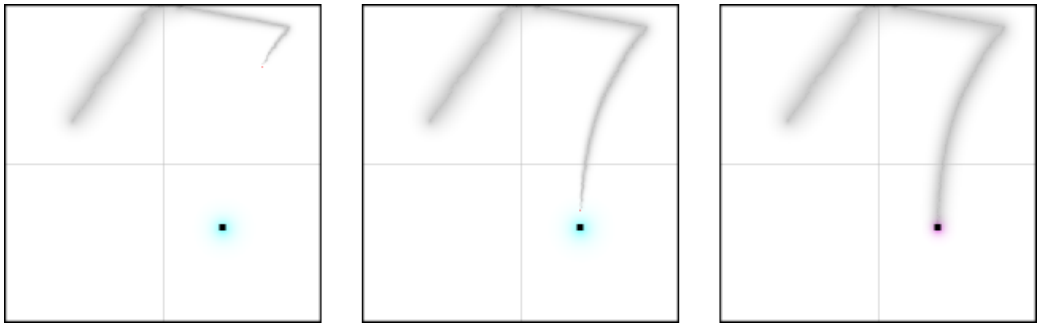

Abbildung 26: Eine Person auf der Suche nach einem Gebäude, das es beziehen kann. Der blaue Duftstoff repräsentiert freie Gebäde (linkes Biild). Der Duftstoff des freien Wohnhauses breitet sich in der Simulation aus und hilft der Person bei der Orientierung (mittleres Bild). Wenn die Person das Gebäude bezogen hat, wird der Duftstoff geändert und an die Person angepasst (rechtes Bild, roter Duftstoff).

Nachdem die Position des Gebäudes der Simulation bekannt ist, wird dem Pheromonsystem des Prototypen ein neuer Duftstoff hinzugefügt, der die Position und die Art des Services, die das Gebäude anbietet, in der Simulation propagiert. Ab diesem Zeitpunkt kann das Gebäude von Personen gefunden werden und zur Bedürfnisbefriedigung eingesetzt werden. Im Fall von Wohnhäusern wird nach dem Einzug einer Person der Duftstoff im System ausgetauscht. Dies ist notwendig, damit eine Person ihr eigenes Haus ohne globales Wissen finden. Hierfür teilt das Gebäude über den Duftstoff den Personen-Agenten mit, wem es gehört. Die Person kann dann über das Folgen der vorhandenen Wege und der Duftstoffe des Hauses zurückfinden.

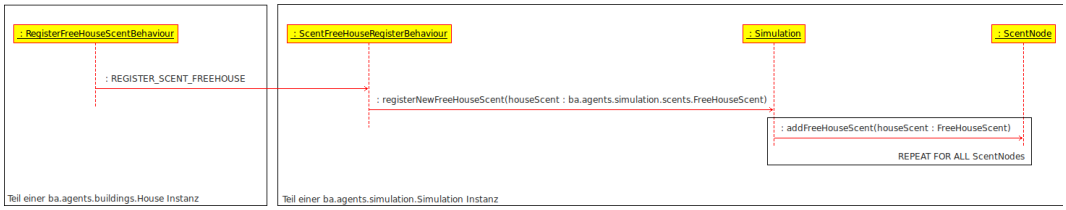

Abbildung 27: Sequenzdiagramm zur Anmeldung eines neuen FreeHouseScents bei der Simulation.

Wie man an dem Sequenzdiagramm sieht, arbeitet der Prototyp mit Nachrichten, die zwischen den einzelnen Agenten zur Kommunikation verschickt werden. Das Diagramm zeigt die Interaktion zwischen einem neu erstellten Wohnhaus, das einen neuen *FreeHouseScent* bei der Simulation registriert. Die Inter-Agentenkommunikation findet hierbei über das Verschicken von *ACLMessages* statt. Die Intra-Agentenkommunikation

verläuft über die API der einzelnen Agenten. In der laufenden Simulation sind derzeit nur die Wohnhäuser und die Bäckereien voll funktionsfähig. Dies liegt an den mehrmaligen Änderungen der Art und Weise, wie die Duftstoffe in der Simulation verarbeitet werden. Es sind im Prototypen die Grundlagen für die Verwendung der übrigen, im Klassendiagramm angebenen, Gebäudetypen vorhanden, aber das neue Pheromonsystem wurde bis zum Schreiben der Arbeit noch nicht vollends fertiggestellt.

5.2.2 Person

Personen sind die Grundlage der Simulation und ihrer Ergebnisse. Das Verhalten einer Person wird durch ihre Bedürfnisse gesteuert, die von einem NeedManagerBehaviour zentral verwaltet wird.

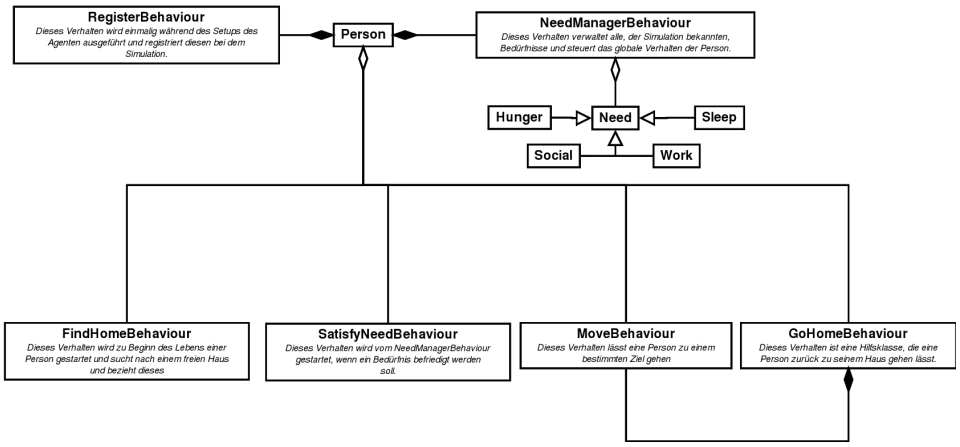

Abbildung 28: Personen - Klassendiagramm

Die Klasse Person instanziiert während ihres Setups zwei Behaviours. Das Erste kümmert sich um die Registrierung bei der Simulation, in dem es als OneShotBehaviour eine entsprechende Registrierungsnachricht an die laufende Simulation schickt, das zweite Verhalten ist der schon erwähnte NeedManager, der im nächsten Abschnitt genauer vorgestellt werden soll.

Bevor diese Behaviours ausgeführt werden, registriert sich die Person unabhängig von der Simulation bei dem DF der JADE-Plattform (Algorithmus 9), wie es von den Entwicklern von JADE empfohlen wird, auch wenn von dem DF im weiteren Verlauf, in Bezug auf Personen, keinen Gebrauch gemacht wird.

Algorithm 9 Das Anmelden einer Person beim DF in der setup Methode des Agenten. Der DF wird über die DFService Utilityklasse der JADE-Plattform angesprochen.

```
1 ServiceDescription sd = new ServiceDescription();
2 sd.setName("PersonInWorld");
3 sd.setType("Person");
4
5 DFAgentDescription dfd = new DFAgentDescription();
6 dfd.setName(getAID());
7 dfd.addServices(sd);
8
9 DFService.register(this, dfd);
```

Nach dem Registrieren bei der Simulation und dem Starten des NeedManagers, übernimmt dieser die volle Kontrolle über die Entscheidungen, die eine Person im Laufe seines Lebens treffen wird. Die Methoden, die die Personenklasse zur Verfügung stellt, sind reine Utilityklassen, die von den unterschiedlichen Behaviours, die auf die Person Zugriff haben, verwendet werden können. Die Methoden umfassen hauptsächlich Hilfsmethoden, die von der Klasse *MoveBehaviour* zur Fortbewegung der Person verwendet werden.

5.2.2.1 Bedürfnisse

Personen können unterschiedlichste Bedürfnisse haben, die als Unterklassen der Klasse *Need* aus dem Package ba.agents.person.needs implementiert werden. Jedes Bedürfnis muss daher mehrere Methoden implementieren, die von dem Bedürfnismanager (siehe 5.2.2.2) bzw. dem Befriedigungsverhalten (siehe 5.2.2.2.1) der Person verwendet werden.

Die Klasse *Need* gibt folgende Methoden vor, die von ihren Unterklassen implementiert werden müssen:

- *evaluateProbability*
- *satisfyNeed*
- *updateNeed*
- *isSatisfied*

EvaluteProbability und *updateNeed* werden im *NeedManagerBehaviour* zur Verwaltung der Bedürfnisse verwendet. *UpdateNeed* aktualisiert den Zustand des Bedürfnisses und *evaluateProbability* bestimmt die Wahrscheinlichkeit, mit der dieses Bedürfnis befriedigt wird. Die Wahrscheinlichkeit bewegt sich für jedes Bedürfnis zwischen 0 und 1. Ob ein Bedürfnis aber befriedigt wird, entscheidet der Bedürfnismanager abhängig von der Priorität des untersuchten Bedürfnisses. Je höher die Grundwahrscheinlichkeit und die Priorität des Bedürfnisses, die in den Klassen als Konstante definiert werden kann, desto höher ist auch die Gesamtwahrscheinlichkeit, dass ein Bedürfnis ausgewählt wird.

Die Methoden *satisfyNeed* und *isSatisfied* werden innerhalb des *SatisfyNeedBehaviour* (Abschnitt 5.2.2.2.1) verwendet. Erstere wird aufgerufen, wenn die Person einen Ort erreicht hat, an dem dieses Bedürfnis befriedigt werden kann. Sie bewirkt eine entsprechende Reduktion des Bedürfnisses. Die zweite Methode informiert das

SatisfyNeedBehaviour der Person, ob das Bedürfnis ausreichend befriedigt wurde, um das Verhalten erfolgreich abzuschließen.

5.2.2.2 NeedManagerBehaviour

Der NeedManager ist die zentrale Kontrollinstanz der Implementierung der Personen in diesem Prototyp. Er übernimmt alle Aufgaben bezüglich der Auswahl der Bedürfnisse, die von der Person befriedigt werden sollen und ist der Zeitgeber für die Aktualisierung der Bedürfniswerte.

Das Flowchart in Abb. 29 verdeutlicht, welche Schritte der NeedManager bei jedem Update-Intervall durchführt. Das *NeedManagerBehaviour* ist als *TickerBehaviour* implementiert und wird während des Setup einer neuen Person instanziiert. Das Zeitintervall, in dem das Verhalten aktiviert wird, wird innerhalb der Klasse *NeedManagerBehaviour* in einer Konstante deklariert und kann bei Bedarf angepasst werden. Die Default-Zeit ist auf fünf Sekunden festgelegt.

Bei jedem Aufruf des Bedürfnismanagers einer Person schaut dieser nach, ob die Person noch mit der Befriedigung eines anderen Bedürfnisses aktuell beschäftigt ist. Dies kann das *NeedManagerBehaviour* über ein Feld der ihr assoziierten Person erfahren, das während der Befriedigung eines Bedürfnisses auf true gesetzt wird. Ist dies der Fall, so werden nur die im Manager registrierten Bedürfnisse aktualisiert, indem für jedes Bedürfnis die update-Methode aufgerufen wird. Diese ist mit dem individuellem Updateverhalten in den jeweiligen Need-Unterklassen implementiert.

Nach der Aktualisierung der Werte untersucht der NeedManager, ob es Bedürfnisse gibt, die von der Person befriedigt werden sollten. Hierfür untersucht er die einzelnen Bedürfnisse durch einen Aufruf der *evaluateProbability*-Methode jedes bekannten Bedürfnisses, das noch nicht in der Warteschlange der zu befriedigenden Bedürfnisse vorhanden ist. Ist es bereits in der Warteschlange vorhanden, so geht das Bedürfnismanagement davon aus, dass das Bedürfnis in absehbarer Zeit befriedigt werden wird und daher nicht noch einmal untersucht werden muss. Alle anderen Bedürfnisse, die eine Befriedigungswahrscheinlichkeit von mehr als 50% haben, werden in einer Liste möglicher Kandidaten vorgemerkt. Nachdem die Wahrscheinlichkeiten aller Bedürfnisse in dieser Liste normalisiert wurden, wählt der Needmanager das Bedürfnis mit der höhsten Wahrscheinlichkeit aus und fügt ein *SatisfyNeedBehaviour* mit dem ausgewählten Bedürfnis der Jobqueue der Person hinzu.

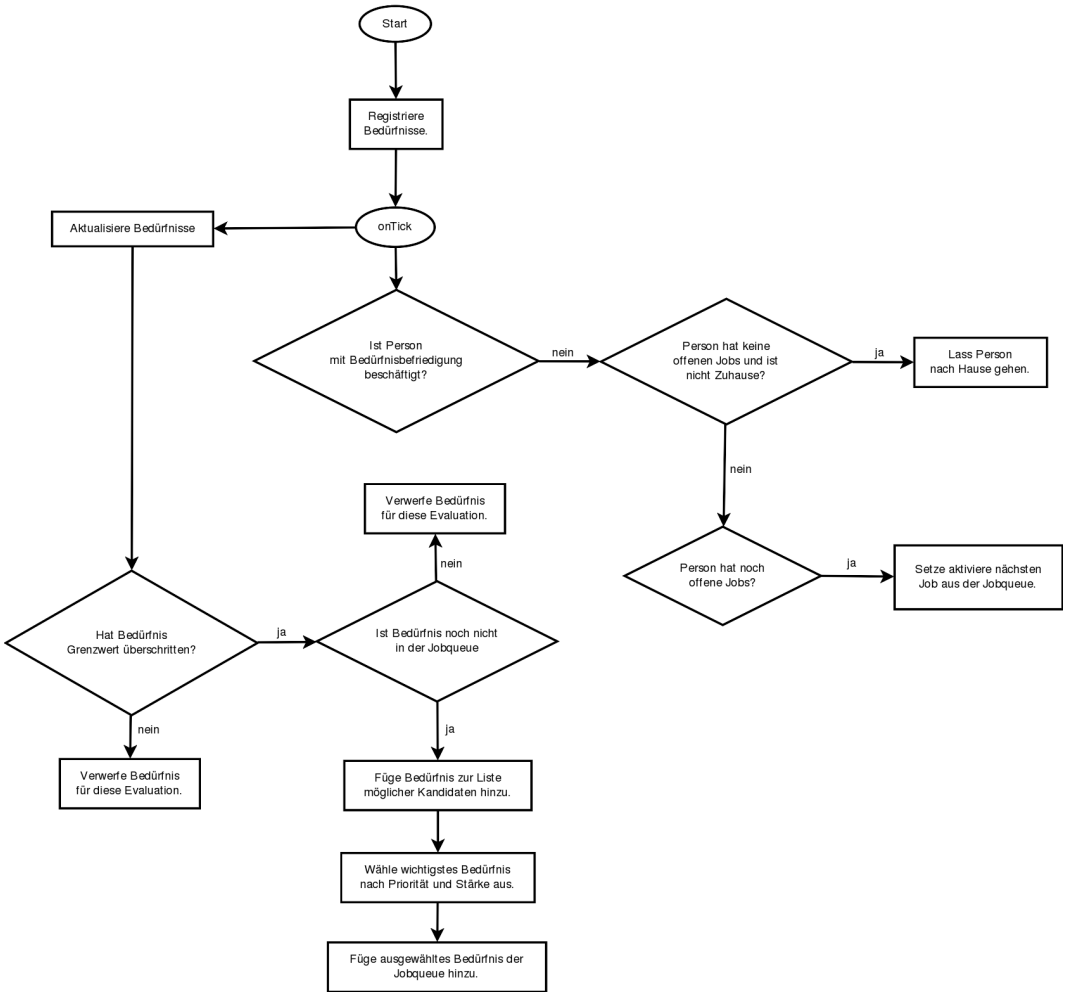

Abbildung 29: Flowchart der im NeedManagerBehaviour implementierten Logik.

5.2.2.2.1 SatisfyNeedBehaviour Ein *SatisfyNeedBehaviour* wird einer Person über sein *NeedManagerBehaviour* hinzugefügt und ist für den kompletten Ablauf der Bedürfnisbefriedigung zuständig.

Die Schritte, die das Bedürfnis hierbei sequenziell abarbeitet sind:

1. Überprüfen, ob ein Gebäude in der Simulation vorhanden ist, das der Person die Befriedigung ermöglicht. Ist kein entsprechendes Gebäude vorhanden, so gibt die Person seine Stimme zum Bau eines neuen Gebäudes vom benötigten Typ.

2. Ist ein Gebäude vorhanden, so wird ein neues *MoveBehaviour* in der Person

gestartet, das die Person zu dem Ort bringt, an dem die Person das gewählte Bedürfnis befriedigen kann.

3. Sobald die Person am Ziel angekommen ist, ruft das *SatisfyNeedBehaviour* die *satisfyNeed*-Methode des Bedürfnisses auf und befriedigt zu einem bestimmten Betrag das Bedürfnis bis die *isSatisfied*-Methode des Bedürfnisses *true* zurückgibt.

Über den DF kann das *SatisfyNeedBehaviour* herausfinden, ob ein neues Gebäude gebaut werden muss oder nicht. Der dafür benötigte Code ist in Alg. 10 dargestellt. Die statische Methode *search* gibt ein Array aller Agenten zurück, die den in der DFAgentDescription definierten Service anbieten. Ist die Länge des Arrays 0, so schickt das SatisfyNeedBehaviour eine VOTE-Nachricht an die Simulation, um für den Bau eines neuen Gebäudes vom benötigten Typ zu stimmen (Alg. 11) und gibt das Bedürfnis durch das Setzen des *inQueue* Feldes auf *false* wieder zur Evaluation durch den Bedürfnismanager frei.

Algorithm 10 Durchsuchen des DFs nach einem verfügbaren Gebäudetyp.

```
1 ServiceDescription sd = new ServiceDescription();
2 sd.setType(need.getServiceType());
3 DFAgentDescription dfd = new DFAgentDescription();
4 dfd.addServices(sd);
5 possibleServices = DFService.search(myAgent, dfd);
```

Algorithm 11 Das SatisfyNeedBehaviour überprüft, ob ein Gebäudetyp vorhanden ist, wenn nicht, wird für den Bau eines neuen Gebäudes gestimmt und das Behaviour beendet.

```
1 if (possibleServices.length == 0) {
2     ACLMessage m = new ACLMessage(ACLMessage.REQUEST);
3     m.setConversationId(CID.VOTE);
4     m.setContent(need.getServiceType());
5     m.addReceiver(Simulation.AID_SIMULATION);
6     myAgent.send(m);
7     need.setInQueue(false);
8     done = true;
9 }
```

Ist ein Ort, an dem die Person ihr Bedürfnis befriedigen kann, vorhanden, so fügt das *SatisfyNeedBehaviour* der Person ein *MoveBehaviour* mit dem S*ervice Type* des Bedürfnisses als Ziel hinzu und wartet mit dem eigenen Fortschreiten solange, bis die Person an dem Ziel des *MoveBehaviours* angekommen ist. Dies kann das Behaviour

über das Feld *atDestination* der Person überprüfen, das beim Erreichen des Ziels durch das *MoveBehaviour* auf *true* gesetzt wird.

Am Ziel angekommen, wird das *SatisfyNeedBehaviour* alle 500 Millisekunden aufgerufen und simuliert so, auf eine simple Art und Weise, dass die Befriedigung eines Bedürfnisses eine bestimmte Zeit benötigt. Wie lange diese Zeit dauert kann sich zwischen den Bedürfnissen unterscheiden, da das Intervall, in dem die Befriedigungsmethode aufgerufen wird, zwar fix ist, die Höhe der sukzessiven Befriedigung pro Aufruf der Methode aber durch den Gebrauch von Zufallswerten variieren kann. Ist das Bedürfnis befriedigt, so beendet sich das *SatisfyNeedBehaviour* und gibt das Bedürfnis zur erneuten Evaluation durch den Bedürfnismanager wieder frei.

5.2.2.3 Bewegungsverhalten von Personen

Das Bewegungsverhalten von Personen spielt die zentrale Rolle in der Implementierung des Prototypen. In der Klasse *MoveBehaviour* der Person finden alle Entscheidungen statt, die *bestimmen* wie eine Person das gewünschte Ziel findet. Die Klasse *VicinityBehaviour* der Simulation liefert auf Anfrage, gekapselt in der Klasse *VicinityResponse*, eine lokale Sicht der Welt, abhängig von der nachfragenden Person.

Die Implementierung des *MoveBehaviours* war während der Implementierungsphase des Prototypen am arbeitsintensivsten. Zum einen, weil die Bewegung das grundlegende Verhalten ist, das zu den Ergebnissen dieser Simulation führt, zum anderen, weil sich viele Designentscheidungen in dieser Klasse auf die Implementierung anderer Klassen direkt oder indirekt auswirken. Da die ersten Konzepte des Bewegungsverhaltens noch globales Wissen über die Position der gesuchten Gebäude beinhaltete, waren an das *MoveBehaviour* der Person und das *VicinityBehaviour* der Simulation ganz andere Anforderungen zu stellen. Da die Personen prinzipiell die richtige Richtung kannten, mussten sie in den ersten Implementierungen nur die Duftspuren der Wege und den globalen Richtungsvektor zum Ziel mit einbeziehen. Die Duftspur sollte in einem bestimmten Umkreis um die Person herum untersucht werden und die lokalen Maxima, die über einem bestimmten Grenzwert lagen, der Person übertragen werden. Die Person konnte dann mit den jeweiligen, mit einer Gewichtung versehenen Koordinate, seine Entscheidungen begründen.

Dies beachtete aber, zum einen, nicht die Anforderungen an den Prototypen, zum anderen war die Umkreissuche verhältnismäßig rechenintensiv, da ein KD-Tree Verwendung fand, der für jede Anfrage von einer anderen Person erneut erstellt werden musste. Daher wurde im Zuge der Erweiterung und Flexibilisierung des Pheromon-

systems und der Anpassung an die Anforderungen an den Prototypen auch das *MoveBehaviour* bzw. das *VicinityBehaviour* angepasst. Die an dem Verhalten beteiligten Klassen sind in Abb. 30 dargestellt. Die Architektur, mit der *VicinityRespose* als serialisierbares Objekt, hat den Vorteil, dass nach der Deserialisierung direkt auf das Objekt zugegriffen werden kann und komplexere Daten über eine ACLMessage nur als Strings verschickt werden können. Auf der anderen Seite schränkt diese Entscheidung aber die Verwendbarkeit ein, da serialisierte Objekte nur von JVM-basierten Sprachen, die mit Java-Objekten direkt arbeiten können, verwendet werden können.

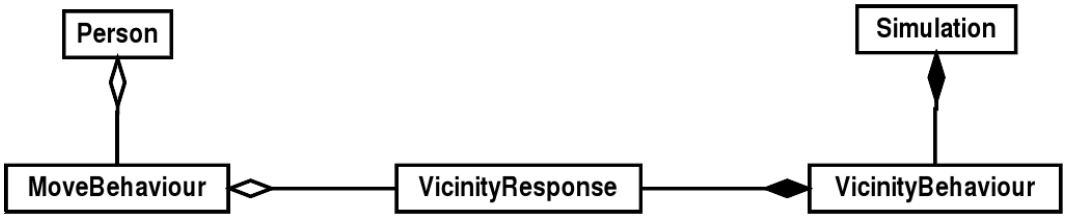

Abbildung 30: Domänenklassendiagramm der am Bewegungsverhalten einer Person beteiligten Klassen.

Um das Abrufen einer neuen *VicinityResponse bei* der Simulation zu vereinfachen, bietet die Klasse *Person* eine Hilfsmethode (Alg. 12), die das Befragen der Simulation übernimmt. Diese wird von dem *MoveBehaviour* einer Person verwendet, um die Duftstoffe der Moore-Nachbarschaft[4] einer Person für die weitere Verarbeitung zu erhalten.

Algorithm 12 Die Methode *getVicinity* der Klasse Person.

```
1  public VicinityResponse getVicinity() {
2      ACLMessage m = new ACLMessage(ACLMessage.REQUEST);
3      m.addReceiver(Simulation.AID_SIMULATION);
4      m.setConversationId(CID.GET_VICINITY);
5      send(m);
6      ACLMessage reply = null;
7      reply = blockingReceive(MessageTemplate.MatchConversationId(CID.GET_VICINITY));
8      VicinityResponse vr = null;
9      if (reply != null) {
10         vr = (VicinityResponse) reply.getContentObject();
11     }
12     return vr;
13 }
```

Eine *VicinityResponse* beinhaltet, wenn sie bei einer Person ankommt, die Informa-

[4]Die Moore-Nachbarschaft bezeichnet die 8er-Nachbarschaft(links, rechts, oben, unten und die Diagonalen) um einen Punkt herum.

tionen über alle verfügbaren Duftstoffe in ihrer Moore-Umgebung und ihre Stärke. Für jedes Nachbarfeld werden diese Informationen in einer *VicinityScentNode*, die vergleichbar mit der *ScentNode* des Pheromonsystems aus 5.2.1.1.1 ist, gespeichert. Der Unterschied zwischen einer *VicinityScentNode* und einer *ScentNode* sind zusätzliche Hilfsmethoden, auf die eine Person zugreifen kann, um die Informationen in der VicinityResponse zu verarbeiten.

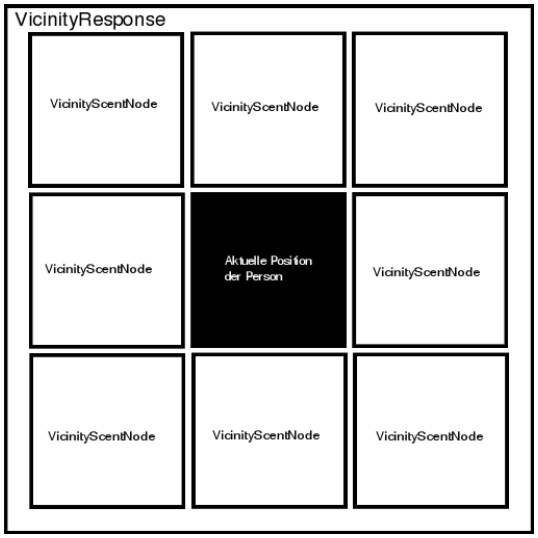

Abbildung 31: Der Aufbau einer VicinityResponse.

Ein *MoveBehaviour* bekommt als Ziel nur einen Service Typ übergeben. Welcher Ort von der Person genau angesteuert wird, hängt von der aktuellen Position ab und davon welcher Duftstoff für die Person am dominantesten ist. Der allgemeine Ablauf eines *MoveBehaviours* ist:

1. Abrufen der aktuellen Position. Diese wird dazu verwendet, um zu überprüfen, ob die Person nicht an einer Stelle "festhängt".

2. Abrufen der aktuellen Umgebungsinformationen von der Simulation.

3. In festen Intervallen wird die zurückgelegte Strecke der Person bestimmt.

    a) Ist die zurückgelegt Strecke zu kurz, muss davon ausgegangen werden, dass die Person in Schleifen läuft. In dem Fall wird das *MoveBehaviour* pausiert, um die Auswirkung der Wegduftstoffe zu reduzieren und die Person zu befreien. (Mehr dazu weiter unten in diesem Abschnitt)

4. Hat die Person neue Umgebungsinformationen von der Simulation erhalten, so bewegt sich eine Person nach folgenden Regeln fort.

   a) Ist kein Duftstoff des gesuchten Services verfügbar und bewegt sich die Person in keine zufällige Richtung, so wählt die Person eine neue, zufällige Richtung aus und bestimmt eine zufällige Anzahl von Schritten, die die Person in diese Richtung gehen wird.

   b) Ist kein Duftstoff des gesuchten Services verfügbar und es sind noch nicht alle Schritte in die zufällig bestimmte Richtung gegangen, so geht die Person einen weiteren Schritt in diese Richtung, nachdem diesem ein zufälliges Rauschen hinzugefügt wurde.

   c) Ist kein Duftstoff des gesuchten Services verfügbar, aber eine Duftspur eines vorhandenes Weges, so geht die Person in die vorher zufällig bestimmte Richtung, wird aber von der Duftspur des Weges angezogen.

   d) Ist ein Duftstoff des gesuchten Services verfügbar, so folgt die Person diesem Duft. Sie lässt sich aber von anwesenden Wegen anziehen, um ihnen so lange wie möglich zu folgen. Zusätzlich kommt eine leichte Ablenkung durch ein zufälliges Rauschen hinzu

5. Liegt die Stärke des Service-Duftstoffes über einem bestimmten Grenzwert (zur Zeit bei über 50%), so ist die Person seinem Ziel nah genug.

6. Das *MoveBehaviour* wird beendet. Dabei wird das *atDestination* Feld der Person auf *true* gesetzt.

Sind Wegduftstoff und Duftstoffe des gesuchten Services verfügbar, so werden die einzelnen Duftstoffe in der Reihenfolge verarbeitet, wie sie in Alg. 13 zu sehen ist. In der Methode *addWeightedXScentDirection*, wobei X für den Namen des jeweiligen Duftstoffes steht, wird als *currentDirection* der absolute Winkel in Richtung des Pheromonmaximums gesetzt (Alg. 14). Als nächstes wird die Ausrichtung der Person, abhängig von den vorhandenen Wegspuren, verändert (Alg. 15). Der Agent dreht sich von seiner Ausgangsrichtung in Richtung des nächsten Weges. Dieses Verhalten garantiert, dass eine Person einem existierenden Weg so lange wie möglich folgt, bis diese ausbricht und gegebenenfalls einen unbekannten Weg zu der Servicequelle geht.

Algorithm 13 Die Reihenfolge der Duftstoffverarbeitung im MoveBehaviour einer Person.

```
1 currentDirection = addWeightedFoodScentDirection(currentDirection);
2 currentDirection = addWeightedTrailScent(currentDirection);
3 currentDirection = addNoise(currentDirection);
```

Algorithm 14 Beispiel einer *addWeightedXScent* Methode am Beispiel der *addWeightedFoodScent* Methode

```
1  private double addWeightedFoodScentDirection(double currentDirection) {
2     if (currentVicinity.isFoodScentAvailable()) {
3        int heading = currentVicinity.getFoodMaxDirection();
4        if (heading == VicinityResponse.UP) {
5           currentDirection = toRadians(90);
6        } else if (heading == VicinityResponse.UPPER_LEFT) {
7           currentDirection = toRadians(140);
8        } else if (heading == VicinityResponse.LEFT) {
9           currentDirection = toRadians(180);
10       } else if (heading == VicinityResponse.LOWER_LEFT) {
11          currentDirection = toRadians(225);
12       } else if (heading == VicinityResponse.DOWN) {
13          currentDirection = toRadians(270);
14       } else if (heading == VicinityResponse.LOWER_RIGHT) {
15          currentDirection = toRadians(325);
16       } else if (heading == VicinityResponse.RIGHT) {
17          currentDirection = toRadians(0);
18       } else if (heading == VicinityResponse.UPPER_RIGHT) {
19          currentDirection = toRadians(45);
20       }
21    }
22    return currentDirection;
23 }
```

Algorithm 15 Das Ändern der Grundausrichtung der Person auf Grundlage der existierenden Wegduftstoffe in der Nähe des Agenten.

```
1  private double addWeightedTrailScent(double currentDirection) {
2     if (currentVicinity.isTrailScentAvailable()) {
3        int trailMaxheading = -1;
4        if (mainDirection == VicinityResponse.UP) {
5           trailMaxheading = currentVicinity.getTrailsMaxDirectionUP();
6           if (trailMaxheading > -1) {
7              if (trailMaxheading == VicinityResponse.LEFT) {
8                 currentDirection = turnLeft(toRadians(25), currentDirection);
9              } else if (trailMaxheading == VicinityResponse.UPPER_LEFT) {
10                currentDirection = turnLeft(toRadians(40), currentDirection);
11             } else if (trailMaxheading == VicinityResponse.UP) {
12          } else if (trailMaxheading == VicinityResponse.UPPER_RIGHT) {
13             currentDirection = turnRight(toRadians(40), currentDirection);
14          } else if (trailMaxheading == VicinityResponse.RIGHT) {
15             currentDirection = turnRight(toRadians(25), currentDirection);
16          }
17       } else if (mainDirection == VicinityResponse.UPPER_LEFT) {
18          ...
19 }
```

In der derzeitigen Implementierung kann es passieren, dass eine Person von den umliegenden Duftstoffen der Wege, die sie selbst durch ihre Bewegung zusätzlich noch verstärkt, umschlossen wird und sich in einer Bewegungsschleife einfängt, die die Person bei fortschreitendem Fortbewegungsverhalten nicht durchbrechen kann. Da eine Person in diesem Zustand von fast gleich stark ausgeprägten Wegduftstoffen umgeben ist, bewegt sich dieser Agent, einmal in einer entsprechenden Konfiguration gefangen, in spiralförmigen (Abb. 32) Mustern um sich selbst. Aus diesem Grund wurde mit unterschiedlichen Methoden experimentiert, dieses Verhalten zu durchbrechen. Die erste Ausbruchsmöglichkeit ist das ignorieren aller Duftstoffe und das Durchbrechen der Kreisbewegung durch eine Fortbewegung in eine zufällige, neue Richtung. Die zweite Möglichkeit ist das kurzfristige Anhalten des Agenten und Abwarten, dass sich die Duftstoffe in der Nachbarschaft zum Teil wieder abgebaut haben und das normale Bewegungsverhalten dann wieder aufzunehmen.

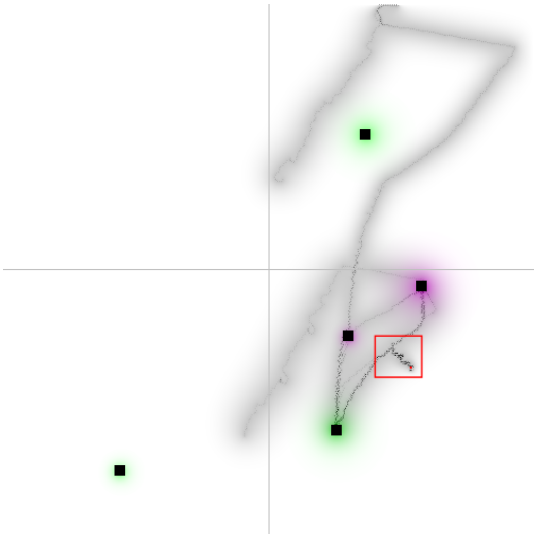

Abbildung 32: Eine Person, die in einer Schleifenbewegung gefangen ist.

Das erste Verhalten hat nach einige Versuchen nicht den gewünschten Effekt erzielt, da, vor allem wenn mehrere Agenten in der Simulation aktiv waren, die Wahrscheinlichkeit hoch war, dass sie an einer anderen Stelle wieder gefangen werden. Außerdem erschwert ein solches Ausbruchsverhalten die Entstehung robuster Hauptwege, da sie von den zufälligen Fährten der ausbrechenden Personen überlagert wurden.

Das alternative Pausieren des Agenten ist die Methode, die in der aktuellen Implementierung zu finden ist. Dieses deterministischere Verhalten der Agenten führt dazu, dass die Wege weiterhin in ihrer Form erhalten bleiben, sich durch den zeitlichen Verfall aber geringer auf die Richtungswahl der Person auswirken und es ihr so ermöglichen aus dem Schleifenverhalten ausbrechen zu können.

Die Entscheidung, ob das Bewegungsverhalten des Agenten pausiert werden soll, hängt von der Entfernung ab, die die Person in 20 Zeitschritten zurückgelegt hat. Liegt diese unterhalb des Grenzwerts der über die Formel *Anzahl der Zeitschritte zwischen den Messungen * Standardgeschwindigkeit einer Person*, so wird das *MoveBehaviour* für eine zufällige Zeit zwischen 100 und 250 Zeitschritten angehalten.

Ist die Person am Ziel, also einer Duftstoffquelle, angekommen, wird ihr das über das Setzen von *atDestination* auf *true* mitgeteilt und andere Behaviours können ihrerseits entsprechend darauf reagieren.

Algorithm 16 Die Methode prepareNextFrame und drawFrame der Visualisierung der Simulation.

```
1  public void prepareNextFrame() {
2      updateTrailColors();
3      updateLivingScentColors();
4      updateHomeScentColors();
5      updateFoodScentColors();
6  }
7
8  public void drawFrame(Graphics2D g) {
9      g.clearRect(0, 0, Simulation.WIDTH, Simulation.HEIGHT);
10     drawTrailsAndScents(g);
11     drawPersons(g);
12     drawBuildings(g);
13     g.setColor(Color.LIGHT_GRAY);
14     g.drawLine(0, Simulation.YOFFSET, Simulation.WIDTH, Simulation.YOFFSET);
15     g.drawLine(Simulation.XOFFSET, Simulation.HEIGHT, Simulation.XOFFSET, 0);
16 }
```

5.2.3 Visualisierung der Ergebnisse

Die Klassen, die zur Visualisierung der Simulation verwendet werden, finden sich im Package ba.util.view. Zwei dieser Klassen (*AnimatedWindow* und *AnimationListener*) wurden von Daniel Klein für die FH Köln als Java2D Animationsframework geschrieben und werden zu diesem Zweck in Lehrveranstaltungen eingesetzt. Die Verwendung dieser Klassen bot sich an, da sie genau die Funktionalität bieten, die in dieser Arbeit zur Visualisierung benötigt werden und durch die FH Köln schon zur Verfügung standen. Das Fenster der Visualisierung wird innerhalb der Klasse *Simulation* instanziiert und der AWT-EventQueue zum Start übergeben. Die Simulationsansicht verwendet eine integrierte Ausgabe, um die Gebäude, die Personen und die unterschiedlichen Duftstoffe in einem Fenster anzuzeigen. Hierfür greift die Instanz der Visualisierung auf unterschiedliche Felder der Simulation zu.

Die wesentlichen Schritte, die die Ansicht der Simulation für jeden Frame durchführt, sind in Alg. 16 zu sehen.

6 Zusammenfassung, Fazit und Future Work

Der Prototyp hat mehrere größere Codeänderungen in einer recht kurzen Zeit erlebt. Dies lässt sich damit erklären, dass unabhängig von den Anforderungen, die zu Beginn der Arbeit an den Prototypen gestellt wurden, die jeweiligen Implementierungen an auftretende Probleme angepasst werden mussten. Das beste Beispiel hierfür ist sicherlich das mehrfach überarbeitete Pheromonsystem, das mit jeder Änderung zwar komplexer wurde, aber auch flexibel genug, die unterschiedlichen Anforderungen erfüllen zu können. Daher muss der Prototyp als work in progress angesehen werden, der durch die letzten Änderungen in der Architektur genügend Flexibilität bietet, um auf seiner Grundlage weitere Funktionalitäten, gemäß den Anforderungen, hinzuzufügen.

Da diese Änderungen zum Teil mehr Zeit beansprucht haben als zu Beginn abzusehen war, konnten nicht alle vorher implementierten bzw. vorgesehenen Bedürfnisse vollständig in der zur Verfügung stehenden Zeit wieder eingebaut werden. Hierunter fallen die Bedürfnisse *Social* und *Work*. Um das Work Bedürfnis zu integrieren müssten die Behaviours hinzugefügt werden, die die jeweiligen WorkScents der Simulation hinzufügen und verwalten.

Es benötigt ein wenig mehr Arbeit das Social Bedürfnis dem Prototypen hinzuzufügen. Personen, die dieses Bedürfnis verspüren, müssen es in der Umwelt über das Aussenden von Duftstoffen bekanntmachen und Personen, die auf der Suche nach diesem Duftstoff sind, müssten dieser Duftspur ähnlich folgen können, wie sie es mit den Duftspuren von Wegen machen. Diese Funktionalität konnte innerhalb der Zeit der Implementierungsphase nach den neusten Änderungen nicht mehr vollkommen auf das neue System übertragen werden.

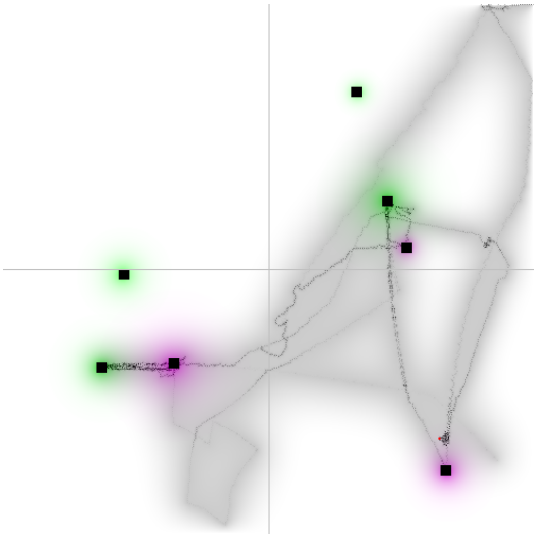

Abbildung 33: Simulationsergebnis mit 4 Agenten nach ca. 70 Minuten. Drei Wohnungen sind bezogen (violetter Duftstoff), eine vierte Wohnung ist noch unbesetzt (hellblauer Duft rechts oben), da sich die Person noch auf dem Weg zu diesem Haus befindet.

Trotz des Fehlens dieser zwei Bedürfnisse ist es möglich die Entwicklung eines Straßennetzes auf Grundlage der Bedürfnisbefriedigung der Personen zu verfolgen. Es entwickeln sich über die Zeit hinweg breitere Hauptstraßen mit angrenzenden Nebenstraßen, die seltener benutzt werden und wieder verschwinden, wenn sie von den Personen gar nicht mehr gebraucht werden. Es zeigt, dass auf Grundlage relativ weniger, einfacher Regeln, ohne eine globale Kontrollinstanz organisiert, dorfähnliche Strukturen entstehen können. Ein solches System bietet einem die Möglichkeit, über einen verhältnismäßig einfach zugänglichen Weg, durch die Definition von grundlegenden Verhaltensmustern und Bedürfnissen, ein dynamisches System zu erzeugen, das zum Beispiel im Bereich von Computerspielen eingesetzt werden könnte, um ansonsten statische Welten mit einer Art von simuliertem Dorfleben zu ergänzen, das dazu beitragen kann, dass zukünftige Spielerlebnisse mit neuen Entdeckungen verbunden sein können.

Der Prototyp befindet sich in einem Zustand, indem er alle Voraussetzungen bietet, um die Simulation an zusätzliche Anforderungen anpassen zu können. Die nächsten Schritte, die vom aktuellen Stand aus gegangen werden könnten, wäre die Einführung einer Tageszeit, die sich auf die Stärke und Priorität eines Bedürfnisses auswirken kann, oder die Einführung von Geld, mit dem die Person in Anspruch genommene Serviceleistung bezahlen muss.

Zusätzlich zu neuen Funktionen, ist ein weiterer Punkt für zukünftige Arbeiten, die Steigerung der Performance der Simulation. Zur Zeit ist die Simulation, aufgrund der vielen parallel laufenden Verhaltensmuster, nur in der Lage eine recht kleine Anzahl von Personen und Gebäuden zu verwalten, ohne dass die Simulationsgeschwindigkeit um ein Vielfaches abnimmt.

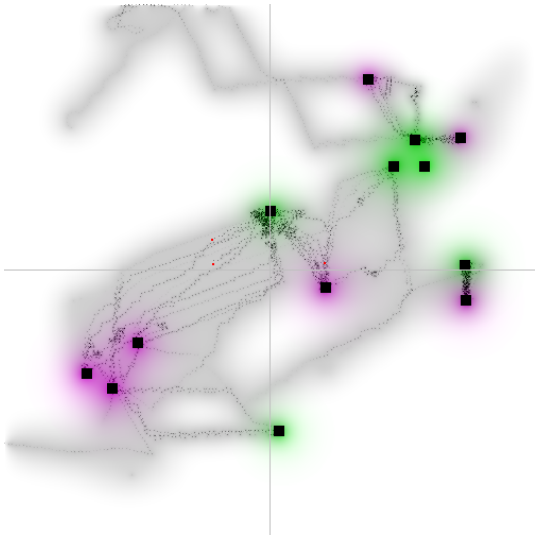

Abbildung 34: Ergebnis einer Simulation von 8 Personen, die über Nacht für einige Stunden lief.

Literatur

[1] Y. I. H. Parish and P. Müller, "Procedural modeling of cities." in *SIGGRAPH*, 2001, pp. 301–308. [Online]. Available: http://dblp.uni-trier.de/db/conf/siggraph/siggraph2001.html#ParishM01

[2] Procedural, "Rekonstruktion der stadt rom mithilfe der cityengine, http://www.procedural.com/cityengine/showcases/rome-reborn.html," 2008, [Online; Stand 19. Februar 2010]. [Online]. Available: http://www.procedural.com/cityengine/showcases/procedural-pompeii.html

[3] ——. (2008) Rekonstruktion der stadt pompeii mithilfe der cityengine, http://www.procedural.com/cityengine/showcases/procedural-pompeii.html. [Online; Stand 19. Februar 2010]. [Online]. Available: http://www.procedural.com/cityengine/showcases/procedural-pompeii.html

[4] N. S. G. L. Stefan Greuter, Jeremy Parker, "Real-time procedural generation of 'pseudo infinite' cities." ACM Press, 2003.

[5] U. Wilensky and B. Watson, "Procedural modeling of urban land use," 2005.

[6] M. Batty, *Cities and Complexity*. The MIT Press, 2005.

[7] L. R. Center. (2004, March) The segregation wall plan in hebron governorate. http://www.poica.org/. Applied Research Insitute Jerusalem. Letzter Besuch: 01.02.2010.

[8] R. S. P. Page, "Repast simphony," http://repast.sf.net, 2010, [Online; Stand 19. Februar 2010].

[9] T. T. G. R. Fabio Bellifemine, Giovanni Caire, "Jade programmer's guide," TILAB & University of Parma, Tech. Rep., 2008.

[10] S. Greuter, "Undiscovered worlds - towards a framework for. real-time procedural world generation." Melbourne DAC, 2003.

[11] Wikipedia, "Grand theft auto," [Online; Stand 19. Februar 2010]. [Online]. Available: http://de.wikipedia.org/w/index.php?title=Grand_Theft_Auto

[12] ——, "Just cause," [Online; Stand 19. Februar 2010]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Just_Cause_(video_game)

[13] ——, "Prototype," [Online; Stand 19. Februar 2010]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Prototype_(video_game)

[14] R. Giacinto, "Procedural content creation - eine methodenübersicht," 2009.

[15] H. V. D. Parunak, S. Brueckner, R. S. Matthews, and J. A. Sauter, "Pheromone learning for self-organizing agents." *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 35, no. 3, pp. 316–326, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/tsmc/tsmca35.html#ParunakBMS05

[16] D. S. Ebert, *Texturing & Modeling: A Procedural Approach.* Morgan Kaufmann, 2003.

[17] M. Batty, "Cellular automata and urban form: A primer," in *Journal of the American Planning Association.* American Planning Association, 1997, vol. 63, pp. 266–274.

[18] T. Schelling, "Models of segregation," in *Papers and Proceedings.* American Economy Review, 1969, vol. 58, no. 2, pp. 488–493.

[19] A. Vicsek C. Szalay, "Fractal distribution of galaxies modeled by a celular automaton-type stochastic process," in *Physical Review Letters.* Physical Review, 1987, vol. 38, pp. 2818–2821.

[20] Netlogo. Community models. http://ccl.northwestern.edu/netlogo/models/community/. [Online; Stand 19. Februar 2010].

[21] A. Maslow, *Motivation and personality*, 2nd ed., W. Holtzman and G. Murphy, Eds. New York [u.a.]: Harper & Row, 1970. [Online]. Available: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+022096191&sourceid=fbw_bibsonomy

[22] TIlab, "Jade - java agent development framework," http://jade.tilab.com/, [Online; Stand 19. Februar 2010].

A Glossar

Geosimulation die Geosimulation bezeichnet einen Forschungsbereich der Urbanologie, in der man versucht urbane Phänomene über Computersimulationen besser verstehen zu können

JADE Java Agent Development Environment, ein von der Telecom Italia entwickeltes Framework zur Entwicklung von agentenbasierter Software

RCP siehe Rich Client Plattform

Rich Client Plattform bei Rich Client Plattformen handelt es sich im Regelfall um Frameworks, die ein Grundgerüst für eigene Anwendungen bieten und die durch Module oder Plugins erweitert werden können. Bekannte Beispiele für RCPs sind das Eclipse RCP oder das Netbeans RCP, auf deren Basis viele kommerzielle und freie Produkte entstanden sind

Urban bezeichnet alles, das charakteristisch für eine Stadt ist oder zu ihr gehört. Hierunter fallen Lebensgewohnheiten, Architektur oder auch die Straßennutzung

Urbanologie oder Stadtplanung ist die Wissenschaft, die sich mit der sozialen und räumlichen Entwicklung von Städten auseinandersetzt

B Inhalt des externen Anhangs auf CD

- Bachelorarbeit
  - PDF der Bachelorarbeit
  - Source Code des Prototypen
- Quellen
  - Kopien verwendeter Quellen
  - Kopien verwendeter Websites

C Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 20. Februar 2010

Robert Giacinto