



Universität zu Lübeck

Institut für Multimediale und Interaktive Systeme

Direktor: Prof. Dr. rer. nat. Michael Herczeg

# **CASi – Context Awareness Simulator**

Projektarbeit

vorgelegt von

Moritz Bürger, Marvin Frick und Tobias Mende

Prüfer

Prof. Dr. rer. nat. Michael Herczeg

wissenschaftliche Begleitung

Felix Schmitt, M.A.

# Kurzfassung

In unserer Projektarbeit im Rahmen des Praktikums im Bereich Interaktions- und Mediengestaltung haben wir uns mit der Konzeption und Entwicklung eines Context Awareness Simulators beschäftigt.

Ziel des Projektes war es, einen Simulator zu entwickeln, mit der das MACK Framework und die dazugehörigen Reasoner ohne spezielle Hardware getestet werden können. Auf Grund der Komplexität dieser Aufgabe haben wir uns dafür entschieden, ein abstraktes Simulationsframework zu entwickeln, da die Software ohnehin sehr flexibel sein muss, um verschiedene Anwendungsgebiete simulieren zu können.

Zunächst haben wir eine Analyse des Einsatzgebietes, der Anwender und notwendigen Funktionen vorgenommen, um mit diesen Ergebnissen ein Konzept für einen modularen Simulator zu entwerfen. Das von uns entwickelte Konzept sah vor, dass einzelne Module durch klar definierte Schnittstellen voneinander abgegrenzt sind und mit geringem Aufwand ausgetauscht werden können. Dies betrifft insbesondere Sensoren, Aktuatoren, Kommunikationshandler, Benutzungsschnittstellen und die Simulationserzeugung, da diese Komponenten stark vom Einsatzgebiet des Simulators und dem Simulationskontext abhängen.

Wir verwenden einen benutzerzentrierten Entwicklungsprozess, um die späteren Nutzer bei wichtigen Entscheidungen während der Planung und Implementierung miteinzubeziehen. Hiermit wollten wir die Qualität und Gebrauchstauglichkeit unserer Software sicherstellen.

## Schlüsselwörter

Simulator, Context Awareness, Software Entwicklung, Framework, Projektarbeit

# Abstract

This project report displays our approach of planning and developing a simulation software for context aware systems. It covers both the aspects of the initial planning as well as the development phases itself. This work originates from the course interaction- and media development at the University of Lübeck.

The primary goal was to develop a simulator that interacts with the established MACK framework and its reasoners, without the need to build concrete hardware modules. Due to the complexity of simulations, we decided to create a more generic framework which allows easy adoption to other context and environments.

Therefore we started with the analysis of the context, the users and the needed features. We developed the concept with the focus on modularity and simple and well defined interfaces to separate the components. Particularly sensors, actuators, communication handlers, user interfaces and simulation generation are subject to different context.

We have chosen a user centered design approach for the development of the software, so we involved the users in important decisions about the simulator. This was to ensure the quality and usability of our software.

## Keywords

context awareness, simulator, software development, framework, project work

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Ziele der Arbeit . . . . .	1
1.2	Stand der Technik . . . . .	2
1.3	Vorgehensweise . . . . .	2
<b>2</b>	<b>Analyse</b>	<b>4</b>
2.1	Problemanalyse . . . . .	4
2.2	Benutzeranalyse . . . . .	8
2.3	Kontextanalyse . . . . .	9
<b>3</b>	<b>Konzeption</b>	<b>10</b>
3.1	Systemarchitektur . . . . .	10
<b>4</b>	<b>Realisierung</b>	<b>16</b>
4.1	Realisierung der Einzelmodule . . . . .	16
<b>5</b>	<b>Evaluation</b>	<b>31</b>
5.1	Ziel . . . . .	31
5.2	Methoden und Vorgehen . . . . .	32
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	<b>33</b>
6.1	Zusammenfassung . . . . .	33
6.2	Offene Punkte . . . . .	34
6.3	Ausblick . . . . .	35
	<b>Abbildungen</b>	<b>36</b>
	<b>Quelltexte</b>	<b>37</b>
	<b>Quellen</b>	<b>38</b>
	Literatur . . . . .	38
	Weblinks . . . . .	38
	<b>Anhänge</b>	<b>39</b>

## *Inhaltsverzeichnis*

A	Projektstruktur . . . . .	39
B	Ausführungsanweisungen . . . . .	41
<b>Erklärung</b>		<b>42</b>

# 1 Einleitung

Die Arbeit beschreibt die Planung und Entwicklung eines Simulators zur realitätsnahen Context Awareness Simulation. Die Motivation für dieses Projekt ist vor allem die Entwicklung einer Testumgebung, die die Analyse des MACK-Frameworks ohne hohe Hardwarekosten ermöglicht.

*MACK* ist ein Framework, welches am IMIS entwickelt wurde und das zur Konstruktion von Systemen, die Context Awareness bieten sollen, benutzt werden kann. Es besteht aus verschiedenen, modularen Komponenten, die in unterschiedlichen Kombination implementiert werden können. Beispiele für solche Systeme sind unter anderem folgende Szenarien:

- ein Bürogebäude, in dem die Unterbrechbarkeit der Kollegen erkannt und sichtbar gemacht wird,
- ein interaktives Museum, in dem sich die (elektronische) Führung an den Besucher anpasst,
- ein ambientes System, in dem sich der Wohnraum an die Bewohner anpasst oder
- ein Krankenhaus, in dem das Personal über die Verfügbarkeit von anderen Mitarbeitern oder bestimmten Behandlungsräumen informiert wird.

*MACK* ist aus dem Projekt *MATe*, das ebenfalls am IMIS ins Leben gerufen wurde, entstanden. *MATe* ist ein System, das in Büros benutzt werden soll, um die Unterbrechbarkeit der Mitarbeiter zu erkennen und anzuzeigen. Es soll außerdem die Möglichkeit bieten, bestimmte Nachrichten zwischen den Kollegen zu verschicken. Das System sammelt Daten mit Hilfe von Sensoren im Büro und verschiedene Reasoner versuchen anhand dieser Information die Unterbrechbarkeit und weitere Attribute der Teilnehmer festzustellen. Diese werden mittels Aktuatoren in der Umgebung erkennbar gemacht.

## 1.1 Ziele der Arbeit

Das Ziel unserer Arbeit ist vor allem die Entwicklung eines modularen Simulators, der sowohl flexibel erweitert, als auch universell eingesetzt, werden kann. Insbesondere soll das Programm so modular sein, dass nicht nur Systeme, die auf *MACK* basieren, damit getestet werden können.

Da das Programm aber voraussichtlich zunächst nur zur Simulation von *MATe*-Komponenten benutzt wird und uns nur der *MATe*-Server zum Testen zur Verfügung steht, haben wir als Beispielsimulation

eine Bürosimulation entworfen und vorerst auch nur Sensoren und Aktuatoren der MATE Anwendung implementiert.

Ein weiteres Ziel ist der Entwurf einer Beschreibungssprache, mit der man sowohl eine Simulation also auch neue Sensoren und Aktuatoren definieren kann. Die Planung dieser XML-basierten Sprache haben wir zwecks besserer Übersicht über den Umfang des Projektes zu Beginn vorgenommen. Da der Fokus der Entwicklungsphase allerdings auf der Universalität und Stabilität des Simulators lag, haben wir die Implementierung der Sprache und eines entsprechenden Interpreters zurück gestellt, weshalb die Beschreibung der Simulationen vorerst nur in Java möglich ist.

### 1.2 Stand der Technik

Zum Zeitpunkt der Entwicklung gab es bereits einen Context Awareness Simulator mit dem Namen *Siafu* (Martin & Nurmi (2006)), der umfangreiche und konfigurierbare Simulationen ermöglicht. Dieser Simulator besitzt jedoch keinerlei Schnittstellen für die Integration von Sensoren und Aktuatoren.

Unsere Recherchen ergaben, dass es bereits Arbeiten gibt, die sich mit dem grundlegenden Gedanken von Context Awareness Simulation auseinander gesetzt haben (Nguyen et al. (2010)) oder die Nutzung dieser Simulatoren als Werkzeug in der Anwendungsentwicklung beschreiben (Pirainen et al. (2007)). Diese Arbeiten brachten jedoch keine öffentlich zugänglichen und erweiterbaren Implementierungen hervor.

Aus diesen Gründen haben wir uns für die Entwicklung eines eigenständigen Simulators entschieden, da die Weiterentwicklung und Anpassung des bestehenden Simulators einen sehr hohen, schwer kalkulierbare Aufwand bedeutet hätte. Ein weiterer Faktor für die Entscheidung zur Neuentwicklung war auch, dass der bestehende Simulator bereits seit 2007 nicht weiter entwickelt wurde und somit eine grundlegende Überarbeitung notwendig gewesen wäre.

### 1.3 Vorgehensweise

Bei der Planung des Projektes haben wir zunächst eine Anforderungsanalyse durchgeführt. Hierfür haben wir Gespräche mit Entwicklern am IMIS, Felix Schmitt und Jörg Cassens, geführt, um ein tiefgehendes Verständnis vom MACK-Framework, der MATE-Implementierung und den Anforderungen an einen Simulator zu erhalten. Die genannten Personen waren zugleich die Hauptinteressenten am Produkt und unsere direkten Ansprechpartner während der Entwicklungsphase.

Im Anschluss haben wir Gespräche mit Olof-Joachim Frahm und Daniel Wilken geführt, die ebenfalls im Rahmen der MATE-Entwicklung am IMIS tätig sind, um weitere technische Einblicke in das Framework und laufende Arbeiten zu erhalten. Olof-Joachim Frahm hat im letzten Jahr ein Whiteboard<sup>1</sup>

---

<sup>1</sup>Die Whiteboard-Anwendung sorgt serverseitig für die Koordination und Weiterleitung von Nachrichten an und zwischen

für Reasoner entwickelt, das es ermöglicht, Ergebnisse verschiedener Reasoner zu kombinieren und eine zentrale Kommunikationsschnittstelle zwischen diesen bereitstellt. Daniel Wilken ist im Rahmen seiner Arbeit unter anderem mit der Entwicklung von Sensoren und Aktuatoren vertraut und beschäftigt sich ebenfalls mit der Entwicklung der Reasoner. Während der Entwicklungsphase haben wir uns Rückmeldungen von diesen Personen eingeholt.

Nach den Gesprächen haben wir bestehende Simulatoren getestet und recherchiert, ob bereits Arbeiten zu diesem Thema durchgeführt wurden. Die Recherchen führten zu dem Entschluss, ein Simulationsframework zu entwickeln, dass flexibler als bestehende Simulatoren ist, da diese unter Berücksichtigung der Anforderungen ungeeignet waren.

Im Anschluss an die Recherchen haben wir die Architektur und eine eigene Beschreibungssprache auf XML-Basis geplant und uns unabhängig von der Architektur textuell drei Szenarien (Büro, Krankenhaus, Museum) überlegt, mit denen wir die Vollständigkeit und Flexibilität der Architektur verifizieren konnten. Diese Vorbereitungen waren die Grundlage, auf der wir die Programmstruktur aufgesetzt haben.

Die eigentliche Entwicklung ist feature-driven und user-centered. Aus diesem Grund haben wir zunächst den Simulationskern und dann die einzelnen Komponenten entwickelt und unserem Betreuer zu den wöchentlichen Treffen eine aktuelle Version als jar-Datei bereitgestellt. So konnten wir frühzeitig die Rückmeldungen der späteren Anwender zu unseren Ansätzen bekommen und auf Wünsche und Anregungen der Benutzer reagieren.

In Kapitel 2 beschreiben wir die Analysen, die der Planung und Entwicklung des Simulators vorausgegangen sind. Die Projektplanung sowie die Konzeption der Architektur werden im Kapitel 3 beschrieben. Über die Entwicklung des Simulationsframeworks schreiben wir in Kapitel 4. Eine detaillierte Beschreibung der Simulation, mit der die MATe-Anwendung getestet werden kann, findet sich im Kapitel 4.1.2. Abschließend beschreiben wir im Kapitel 5 mit welchen Methoden wir die Qualität der Software und das Erreichen unserer Ziele sichergestellt haben und bringen in Kapitel 6 eine Zusammenfassung unserer Arbeit sowie Ausblicke auf offene Aspekte und Erweiterungsmöglichkeiten des Simulators, die als Thema für weitere Projekte dienen können.

---

den Reasonern (Frahm (2011)).



## 2 Analyse

In diesem Kapitel werden die Analyseschritte und die Vorüberlegungen beschrieben, die vor der Konzeption und Implementierung der Anwendung durchgeführt wurden. Hierzu gehen wir in Abschnitt 2.1 zunächst auf die Probleme und Einsatzszenarien ein und beschreiben dann in Abschnitt 2.2 die Ergebnisse der Benutzeranalyse. Abschließend präsentieren wir in Abschnitt 2.3 unsere Kontextanalyse.

### 2.1 Problemanalyse

Bei der Entwicklung des MACK-Frameworks geht es vor allem um die Entwicklung eines zentralen Systems, das aus verschiedenen Reasonern besteht, die auf Basis von Sensordaten Zustände von Personen ermitteln können und diese an Aktuatoren weiterleiten. Dieses System soll durch verschiedene externe Komponenten wie Sensoren und Aktuatoren erweitert werden, um auf beliebige Kontexte adaptiert werden zu können. Hierbei entstehen komplexe Abhängigkeiten, da die ermittelten Werte der Sensoren vom Verhalten der Personen abhängen, welches wiederum zu einem gewissen Anteil von den Meldungen der Aktuatoren abhängen kann.

Um die Effizienz und Vollständigkeit der Reasoner zu verifizieren gibt es bislang nur die Möglichkeit, diese mit Hardware-Implementierungen der Sensoren und Aktuatoren zu verknüpfen und denkbare Szenarien durchzuführen. Dies eröffnet zum einen das Problem, dass zunächst die Hardware entwickelt werden muss, was mit hohem Zeit- und Kostenaufwand verbunden ist. Zum anderen muss das System von vielen Benutzern eingesetzt werden, um repräsentative Ergebnisse zu erhalten.

Darüber hinaus gibt es bisher unerforschte Einsatzgebiete für das MACK-Framework, für die es weder die Reasoner noch die notwendige Hardware gibt. Bevor die teure Hardware entwickelt oder eingekauft wird, soll getestet werden, ob das System für den angedachten Anwendungszweck geeignet ist.

Selbst mit, in ausreichendem Umfang vorhandener, Hardware ist es nur schwer möglich, die Wechselwirkung zwischen den Personen und dem Framework zu testen, da dies die Beobachtung der Benutzer in ihrem täglichen Handeln und eine entsprechende Verhaltensanalyse voraussetzt. Diese Analyse ist nur schwer zu realisieren und insbesondere dann sehr aufwändig, wenn die Auswirkungen minimaler

Änderungen an Reasonern oder dem Framework an sich, verifiziert werden sollen.

Um eine Idee dafür zu bekommen, in welchen Einsatzgebiete das Framework eingesetzt werden könnte und welches demnach Gebiete sind, für die Simulationen realisierbar sein sollten, haben wir uns verschiedene Szenarien überlegt. Mit den nachfolgenden Szenarien haben wir im Verlauf der Planung und Implementierung verifizieren können, dass das Simulationsframework flexibel genug ist um auf verschiedene Einsatzgebiete adaptiert werden zu können.

Die Implementierung aller Szenarien im beschriebenen Umfang ist nicht Teil unseres Projektes. Wir beschränken uns hierbei auf eine grundlegende Implementierung des Büro-Szenarios in einer, ähnlich der im nachfolgenden beschriebenen, Form.

### Szenario: Büro

Das Büroszenario ist das Standardszenario des IMIS, für das bereits Teile einer Hardware-Implementierung und entsprechende Reasoner vorliegen.

Es gibt mehrere Büros, abgehend von einem Korridor. Jedes Büro hat einen Schreibtisch, der eine Dropzone<sup>1</sup> und einen Cubus<sup>2</sup> beherbergt. Außerdem steht in jedem Büro ein Telefon und es gibt eine akustische Überwachung<sup>3</sup>, mit der festgestellt werden kann, ob der dort arbeitende Anwender gerade spricht.

Es gibt an jeder Bürotür ein interaktives Türschild und eine Ampel, die Besuchern Informationen über die Unterbrechbarkeit des Bürobesitzers liefern. Vom Flur abgehend gibt es Toiletten, eine Teeküche und einen Konferenzraum. Die Toiletten melden, wenn sie belegt sind, aus Gründen der Privatsphäre jedoch nicht, wer sie gerade benutzt. Personen haben die Möglichkeit, bevor sie aufstehen, zu sehen, ob die Toilette gerade besetzt ist. In der Teeküche gibt es einen Kaffeeautomaten, der gerne von Mitarbeitern zum Pausengespräch genutzt wird. Im Konferenzraum werden täglich die Tagesziele besprochen. Zu diesen Meetings müssen alle Mitarbeiter kommen.

Das Standardverhalten, wenn Personen keine spezifische Aktion vorhaben, ist, im Büro am Schreibtisch zu sitzen, sich Kaffee zu holen oder eine der Toiletten aufzusuchen.

Mögliche Zufallskomponente, die einen Anwender in seiner aktuellen Aktion unterbrechen könnten, sind zum Beispiel das Klingeln des Telefons oder andere Kommunikationsmittel. Wenn der Adressat im Radius des Kommunikationsmittels ist, beschäftigt er sich für eine zufällige Dauer mit der Kommunikation und unterbricht dafür seine Arbeit.

---

<sup>1</sup>Bei der Dropzone handelt es sich um eine Station, in die verschiedene Benutzer Marken, bzw. ihre Schlüssel, legen können. Das Gerät erkennt dann, welche Marken vorhanden sind und führt so Rückschlüsse auf die im Raum befindlichen Personen durch.

<sup>2</sup>Der Cubus ist ein Würfel, dessen Oberseite den vom Framework ermittelten Zustand des Besitzers angibt. Durch Drehen des Würfels kann eine explizite Änderung des Zustands erwirkt werden. Weitere Informationen befinden sich in Abschnitt 3.1 von Kindereit et al. (2011).

<sup>3</sup>Gemeint ist das in Abschnitt 3.5 von Kindereit et al. (2011) beschriebene *Mike*.

### Szenario: Museum

In einem Museum gibt es verschiedene Bereiche mit unterschiedlichen Themengebieten, zum Beispiel könnte es in einer Ausstellung über die menschlichen Sinne die Bereiche „Sehen“, „Hören“ und „Fühlen“. Außerdem gibt es Toiletten und einen Souvenirladen.

Im Museum halten sich Besucher (20 – 100) und verschiedene Mitarbeiter (10 – 20) auf. Einige Mitarbeiter machen Führungen mit Besuchern durch bestimmte Bereiche, einige Mitarbeiter bewachen empfindliche oder wertvolle Ausstellungsstücke und können Besuchern Wege zu bestimmten Bereichen wie Toiletten und dem Ausgang weisen.

Am Ein- und Ausgang befindet sich ein Lageplan für das Museum, an verschiedenen Ausstellungsstücken gibt es Informationstafel und beim Eintritt in das Museum können die Besucher einen elektronischen Führer mitnehmen. Der elektronische Führer ist Sensor und Aktuator zugleich. Er ermöglicht dem Besucher, sich die Hinweise auf den Informationstafeln vorlesen zu lassen und merkt sich, welche Ausstellungsstücke und Bereiche des Museums den Besucher interessieren. Auf Grundlage dieser Daten macht er Vorschläge, welche anderen Bereiche und Ausstellungsstücke interessant für diesen Besucher sein könnten. Außerdem schlägt er Führungen durch das Museum vor, die ebenfalls auf die Interessen des Benutzers angepasst sind. Der elektronische Führer könnte zusätzlich beim Betreten des Souvenirladens anzeigen, welche Souvenirs dem Besucher gefallen könnten.

Es gibt noch weitere Sensoren innerhalb der Bereiche, die feststellen können, wie viele Personen sich dort aufhalten. Durch Aktuatoren an den, in diese Bereiche führenden, Türen könnte den Besuchern angezeigt werden, welche Bereiche gerade überfüllt oder wenig besucht sind.

### Szenario: Krankenhaus

In einem Krankenhaus auf einer Intensivstation arbeiten zehn Krankenschwestern und zwei Ärzte. Es gibt zehn Einzelzimmer und zehn Doppelzimmer. Insgesamt gibt es 30 Patienten. Die zehn Patienten in den Einzelzimmern sind in einem kritischen Zustand und deshalb an ein Health-Monitoring-System (HMS) angeschlossen, welches den Sauerstoffgehalt im Blut, den Blutdruck, den Puls und die Atemfrequenz erfasst. Außerdem stuft das System den Patienten anhand dieser Werte in Kategorien ein: „sehr gut“, „gut“, „stabil“, „kritisch“, „lebensgefährlich“ und „unbekannt“.

Alle Mitarbeiter verfügen über einen Tablett PC, über den sie innerhalb der Station lokalisiert werden können. Darüber hinaus werden die Geräte zur Protokollierung der Visiten verwendet. Wenn ein Patient in den Zustand „kritisch“ oder schlechter wechselt, werden beide Ärzte über das Tablett informiert. Sie erhalten die Zustandseinschätzung des HMS, die Analysewerte und die Information, welcher Patient in welchem Raum betroffen ist.

Die Station hat außerdem zwei Computerarbeitsplätze im Schwesternzimmer, sowie zwei, mit Computern ausgestattete, Behandlungszimmer, in denen sich gleichzeitig die Schreibtischarbeitsplätze der

Ärzte befinden. Alle Computer sind mit einem „Desktop Activity Analyzer“<sup>4</sup> ausgestattet.

Die Simulation beginnt am 1.11.2011 um 4:30. Eine der beiden Nachtschwestern macht einen Kontrollgang. Danach begibt sie sich zurück ins Schwesternzimmer und arbeitet für zwei Stunden am PC. Um 5:00 Uhr wechselt das HMS in Raum 1 von „stabil“ auf „kritisch“. Wenn das Panel im Schwesternzimmer dies anzeigt, soll eine Schwester sofort in das entsprechende Zimmer gehen und Gegenmaßnahmen vornehmen, so dass der Zustand im Anschluss mindestens „stabil“ ist. Danach setzt die Schwester ihr vorherige Aktivität fort.

Um 5:30 Uhr macht die andere Schwester einen Rundgang. Um 6:00 Uhr kommt der erste Arzt. Er holt sich zunächst einen Kaffee aus der Küche und geht danach in sein Zimmer um dort für 30 Minuten am Computer die Vorkommnisse der letzten Nacht durchzugehen (lesen). Der zweite Arzt kommt um 6:15 Uhr, holt sich ebenfalls einen Kaffee und geht danach in sein Zimmer um an seinem Computer die Vorkommnisse der letzten Nacht zu analysieren. Um 6:45 Uhr kommen die ersten fünf Tagesschwestern. Um 7:00 Uhr treffen sich alle anwesenden Mitarbeiter zur Besprechung im Schwesternzimmer (1 Stunde). Danach gehen die Nachtschwestern nach Hause. Um 9:00 wird der Patient aus Zimmer 3 in den OP gebracht. Arzt 1 operiert für 3 Stunden zusammen mit 5 Schwestern, die dafür zur Arbeit kommen, den Patienten. Während dieser Zeit ignoriert er alle Meldungen, d.h. er unterbricht diese Aktion nicht, selbst wenn das System einen Vorfall bei einem anderen Patienten meldet.

Um 10:00 Uhr melden die HMS in R4 und R5, dass der Zustand von „gut“ auf „lebensgefährlich“ wechselt. Der freie Arzt unterbricht seine Aktivität und geht in das erste Zimmer. Die Schwester, die am nächsten ist, geht in das zweite Zimmer. Dem operierenden Arzt wird die Situation via Tablett Computer gemeldet. Der freie Arzt erkennt eine Fehlfunktion des Systems und setzt den Zustand wieder auf „gut“. Anschließend geht er zum zweiten kritischen Patienten.

Um 11:00 Uhr beginnen der Arzt und zwei Schwestern mit der Visite. Währenddessen meldet sein Smartphone eine kritische Situation im OP, so dass er zunächst in den OP geht, um dem operierenden Arzt zu assistieren. Nach einer Stunde sind beide Ärzte mit der Operation fertig und gehen den Aktionen nach, die sie während der Operation ignoriert haben, sofern diese nicht bereits von anderen erledigt wurden.

Jeder Arzt hat eine Schwester, die stündlich zu ihm kommen muss, um Aktuelles mit ihm zu besprechen. Bevor sie dies tun, prüfen sie mit ihrem Tablett, ob der jeweilige Arzt verfügbar ist und wo er sich befindet. Das System liefert hierzu die Raumnummer und den Status des Arztes („Unterbrechbar“, „Nicht unterbrechbar“ oder „Unbekannt“).

---

<sup>4</sup>Der DAA überwachte die Aktivität und die Programmart, die sich im Vordergrund befindet. Siehe Abschnitt 3.4 von Kindereit et al. (2011).

## 2.2 Benutzeranalyse

Bei der Zielgruppe handelt es sich um erfahrene Entwickler, die mit dem Simulator ihre Anwendungen im Bereich Context Awareness testen möchten und die Anwendung als Werkzeug bei ihrer Arbeit einsetzen. In erster Linie sind das die Mitarbeiter des IMIS, die das MACK-Framework entwickeln und gleichzeitig die Auftraggeber dieses Projektes sind. Sie wollen bestimmte Teile des MACK-Systems wie zum Beispiel seine Reasoner testen und evaluieren.

Da wir die späteren Benutzer als Experten eingestuft haben, standen wir von Anfang an viel mit ihnen in Kontakt, um beispielsweise die spätere Spezifikationsprache der Simulationen festzulegen. Unser Ergebnis war hier, dass mindestens grundlegende Kenntnisse in Java und XML gegeben sind und deshalb die Simulationen mittels dieser Sprachen spezifiziert werden können. Die Erstellung der Simulation nur anhand von Java-Klassen ist zwar umständlich, aber so konnten wir den Fokus auf die Entwicklung des eigentlichen Simulators setzen.

Um eine Simulation zu beschreiben, sollen folgende Teile definiert werden können:

- eine Umgebung, beispielsweise ein Gebäude mit Räumen und Türen,
- Personen, die an der Simulation beteiligt sind. Diese werden im Weiteren *Agents* genannt.
- verschiedene Sensoren und Aktuatoren, die beteiligt sind und getestet werden sollen,
- Aktionen, die von Agents oder Sensoren/Aktuatoren angestoßen werden, um reale Abläufe zu simulieren.

Eine weitere wichtige Frage an die erfahrenen Benutzer war, welche Art von Benutzungsschnittstellen für sie implementiert werden sollten. Wir haben erfahren, dass sowohl besonderer Wert auf detaillierte Logausgaben gelegt wird als auch eine einfache grafische Oberfläche erwünscht ist, da diese für das Erlangen eines ersten Eindrucks von der Arbeit des Systems besonders gut geeignet ist. Die grafische Oberfläche soll hierbei ausschließlich zur Visualisierung und nicht zum Erstellen der Simulation dienen. Andernfalls wäre der Aufwand der Implementierung zu groß gewesen. Auf den Ablauf der Simulation soll in unserer Implementierung nur sehr begrenzt Einfluss genommen werden, zum Beispiel durch Verändern der Simulationsgeschwindigkeit oder Pausieren der Simulation.

Im gesamten Entwicklungsprozess haben wir eng mit den späteren Nutzern zusammengearbeitet und uns möglichst früh Rückmeldung zu offenen Fragen und Entwicklungsentscheidungen geholt. So wollten wir sicherstellen, dass die Benutzer direkt an der Entwicklung beteiligt sind und somit mögliche Fehlentwicklungen vermieden werden.

## 2.3 Kontextanalyse

Der Simulator soll im IMIS auf den Arbeitsplätzen der Entwickler laufen. Die Arbeitsplätze sind mindestens Quad-Core-Computer. Der Simulator soll während der Entwicklung zur Verifikation einzelner Änderungen eingesetzt werden. Außerdem sollen längere Simulationen durchgeführt werden, deren Ergebnisse später mit Hilfe der Logfiles ausgewertet werden können. Das System muss also sowohl im Vorder- als auch im Hintergrund laufen können.

Wegen des hohen Netzwerkaufkommens ist es optimal, Server und Simulator auf einer Maschine laufen lassen zu können und den Netzwerkverkehr über lokale Loopback-Devices abzuwickeln.

## 3 Konzeption

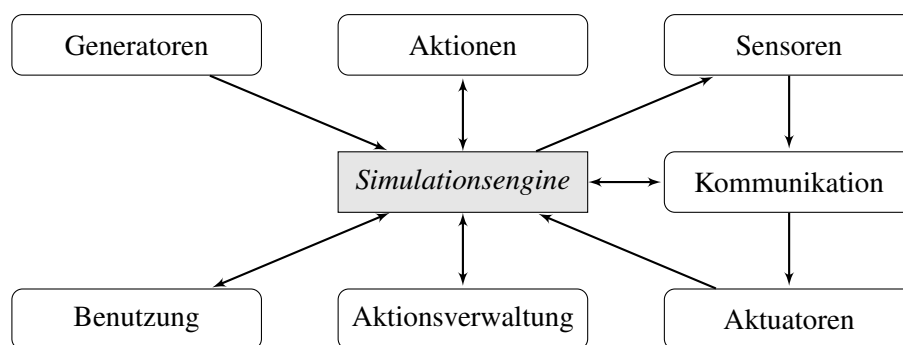
In diesem Kapitel beschreiben wir die Konzepte, die wir im Rahmen unserer Analysen und der Planungsphase erarbeitet haben. Zu diesem Zweck beschreiben wir zunächst in Abschnitt 3.1.1 den Aufbau und die logische Struktur der *SimulationEngine* und der *SimulationClock*.

Die weiteren Abschnitte gliedern sich nach dem in Abbildung 1 dargestellten Schema.

### 3.1 Systemarchitektur

In Abbildung 1 wird die Modularität der Architektur des Simulators deutlich. Die Komponenten in den weißen Kästen stellen Module dar, die durch den Austausch weniger Codezeilen ausgewechselt werden können. Diese Komponenten sind zum Teil simulationsspezifisch. Zum Beispiel sind Sensoren, Aktuatoren und Aktionen vom Simulationsumfeld abhängig. Darüber hinaus bestehen auch die Abhängigkeiten Sensoren  $\leftrightarrow$  Aktionen und Aktuatoren  $\leftrightarrow$  Aktionen, da Sensoren auf bestimmte Aktionen reagieren und Aktuatoren bestimmte Aktionen auslösen können.

Die Pfeile in der Abbildung geben den Hauptinformationsfluss zwischen den Komponenten an.



**Abbildung 1:** Unterteilung der Simulatorarchitektur in Module

### 3.1.1 Simulationsengine

Der Kern des Simulationsframeworks ist die Simulationsengine, die im wesentlichen aus den Klasse `SimulationEngine` und `SimulationClock` besteht. Sowohl die Uhr als auch die Maschine sind als Singleton realisiert und können deshalb aus jeder Klasse angesprochen werden.

Die `SimulationClock` repräsentiert die Zeit in der Simulation. Die Geschwindigkeit der Uhr, und somit die Geschwindigkeit der Simulation, kann skaliert werden. Der Skalierungsfaktor in der Uhr gibt dabei an, wie viele Millisekunden in Echtzeit einer simulierten Sekunde entsprechen. Somit resultiert ein niedriger Wert in einer höheren Geschwindigkeit.

Klassen können das `ISimulationClockListener`-Interface implementieren und sich bei der Uhr als Zuhörer registrieren, um über Events wie das Ticken der Uhr, das Pausieren, Starten und Stoppen der Simulation informiert zu werden.

Die Engine hält die simulierte Welt (`World`) bereit, welche die Konfiguration und das Verhalten der Simulation beschreibt. Darüber hinaus hält die Engine Referenzen auf den Kommunikationshandler, der in unserer Implementierung für die Kommunikation mit dem Awareness-Hub zuständig ist.

### 3.1.2 Generatoren

Generatoren erzeugen `World`-Objekte, die von der Simulationsengine simuliert werden können.

Ein Generator muss sich darum kümmern, Räume, Sensoren, Aktuatoren, Agenten und Aktionen zu erzeugen und geeignet zu verknüpfen. In unserer Implementierung gibt es einen Generator in dem fest einprogrammiert Java-Objekte erzeugt werden.

Eine weitere Möglichkeit für einen Generator stellt zum Beispiel ein XML-Handler dar, der Java-Objekte aus einer XML-Beschreibung generiert.

Generatoren können in der Main-Klasse (`CASi`) des Simulators ausgetauscht werden. Die einzigen Voraussetzungen, die ein Generator erfüllen muss, sind, dass er die `IWorldGenerator`-Schnittstelle implementiert und eine vollständige `World`-Instanz erzeugt.

### 3.1.3 Benutzungsschnittstellen

Benutzungsschnittstellen können bereit gestellt werden, indem eine Implementierung des Interfaces `IMainView` in der Main-Klasse des Simulators an den `MainController` übergeben wird. Die Benutzungsschnittstellen können sowohl passiv als auch interaktiv gestaltet werden. Hierfür können sich Implementierungen der Schnittstelle als Zuhörer (*Listener*) bei den Objekten des Modells registrieren und auf die `SimulationClock` und die Engine zugreifen, da beide Komponenten als Singleton realisiert sind.



### 3.1.4 Aktionen

Aktionen können allgemein durch folgende Parameter spezifiziert werden:

**priority** Die Priorität einer Aktion als Ganzzahl zwischen 0 und 10.

**duration** Die Dauer in Sekunden oder `-1`, falls keine Dauer angegeben wird (z.B. bei `Move`-Aktionen, die dann fertig sind, wenn das Ziel erreicht wurde).

**earliestStartTime** ein Zeitpunkt (`SimulationTime`), ab dem die Aktion gestartet werden darf oder `null`, falls die Aktion zu einem beliebigen Zeitpunkt gestartet werden kann.

**deadline** ein Zeitpunkt, zu dem die Aktion erledigt sein muss oder `null`, falls keine Deadline angegeben wurde.

**state** der aktuelle Status der Aktion, z.B. `SCHEDULED`, wenn die Aktion einem Agenten hinzugefügt wurde, dieser sie aber noch nicht ausgeführt hat, `ONGOING`, wenn die Aktion gerade durchgeführt wird oder `COMPLETED`, wenn die Aktion erfolgreich durchgeführt wurde.

Es gibt `AtomicActions` und `ComplexActions`, wobei letztere aus einer Liste von atomaren Aktionen bestehen können. Mit diesem Konstrukt lassen sich unter anderem Aktionen wie das Aufsuchen und Reden mit einer anderen Person (`Agent`) beschreiben.

Um eine neue Aktion zur Verfügung zu stellen, muss eine Klasse implementiert werden, die von einer der abstrakten Klassen `AtomicAction` oder `ComplexAction` erbt. Die eigentliche Aktion muss in der `internalPerform(AbstractComponent performer)`-Methode beschrieben werden. Sollten vor der ersten Ausführung der Aktion Konfigurationsschritte notwendig sein, können diese in der Methode `preActionTask(AbstractComponent performer)` beschrieben werden. Analog dazu gibt es auch eine Methode `postActionTask(AbstractComponent performer)`, die vom Framework aufgerufen wird, nachdem der Job erledigt wurde.

Wenn eine Dauer angegeben wurde, kümmert sich das Simulationsframework um das Dekrementieren der Zeit, in diesem Fall muss die `internalPerform(AbstractComponent performer)`-Methode immer `false` zurückgeben. Sobald diese Methode `true` zurückgibt, gilt die Aktion als erledigt und es wird im Falle einer komplexen Aktion mit der nächsten atomaren Aktion fortgefahren.

Aktionen müssen vom Generator erzeugt werden und einem oder mehreren Agenten hinzugefügt werden.

### 3.1.5 Aktionsverwaltung

Zur Aktionsverwaltung hat jeder Agent einen `IActionScheduler`, der im Konstruktor ausgetauscht werden kann. Die Aufgabe des Schedulers ist es, die Aktions-Sammlungen bereit zu halten. Das Konzept sieht vor, dass es für jeden Agenten drei Listen von Aktionen gibt:

**todoList** Diese Liste beinhaltet Aufgaben, die in jedem Fall während der Simulation entsprechend ihrer oben beschriebenen Parameter ausgeführt werden sollen.

**actionPool** Diese Menge enthält Aufgaben, die optional ausgeführt werden können, wenn der Agent gerade keine anderen Aufgaben zu erledigen hat. Wenn eine Aufgabe aus dem Pool abgeschlossen wird, wird diese nicht aus dem Pool entfernt. So können sich wiederholenden Zufallsaktionen simuliert werden.

**interruptActionList** In diese Liste werden während der Simulation Aktionen eingeordnet, die in jedem Fall unmittelbar als nächste Aktionen ausgeführt werden müssen. Sinnvoll ist dies insbesondere dann, wenn ein Agent darauf angewiesen ist, dass ein anderer Agent mit ihm interagiert. Dem zweiten Agenten kann dann eine Aktion auf die Interrupt-Liste gesetzt werden, damit er unmittelbar bei der nächsten Gelegenheit mit dem ersten Agenten interagiert und keine anderen Aktionen vorzieht.

Die Kernfunktionalität eines Action-Schedulers ist es, mit geeigneten Algorithmen zu jedem Zeitpunkt eine Aktion auszuwählen. Hierzu ruft der Agent die `getNextAction()`-Methode auf, woraufhin der Scheduler eine Aktion auswählt und diese von der Liste löscht.

### 3.1.6 Kommunikationshandler

Ein Kommunikationshandler muss das `ICommunicationHandler`-Interface implementieren. Kommunikationshandler sind für die Verwaltung der Kommunikation mit einem beliebigen Gegenpart außerhalb des Simulators zuständig. Ein konkretes Beispiel ist ein Netzwerkhandler, der die Kommunikation über eine Netzwerkschnittstelle verwalten kann. In der aktuellen Implementierung ist der `MACKNetworkHandler` ein Kommunikationshandler, der den Informationsaustausch mit dem MACK Server realisiert.

Innerhalb der Simulation interagiert der Kommunikationshandler mit Sensoren und Aktuatoren, die beide das Interface `ICommunicationComponent` implementieren. Die Komponenten registrieren sich in der Initialisierungsphase des Simulators beim Handler, indem die Methode `register(ICommunicationComponent comp)` aufgerufen wird.

Während der Simulation können die Komponenten die `send(ICommunicationComponent sender, Object message)`-Methode aufrufen, um eine Nachricht an den Handler zu senden. Umgekehrt kann der Handler bei den Komponenten die Methode `receive(Object message)` aufrufen, um eine neu eingetroffene Nachricht zu übermitteln.

Zu beachten ist, dass die Nachrichten zunächst vom Typ `Object` sind, um einen hohen Abstraktionsgrad zu gewährleisten. Von welchem Typ die Nachrichten wirklich sind, muss in Sensoren, Aktuatoren und dem Handler während der Implementierung entschieden werden.

In der spezifischen Simulation und dem `MACKCommunicationHandler` sind Nachrichten im-

mer vom Typ `String`, wobei die Nachrichten im XML-Format sind, welches auch das MACK-Framework verwendet. Ausgehende Nachrichten werden von den Sensoren und Aktuatoren unter Verwendung der `MACKProtocolFactory` erzeugt.

### 3.1.7 Sensoren und Aktuatoren

Da Sensoren und Aktuatoren viele Gemeinsamkeiten aufweisen, erben alle konkreten Sensoren und Aktuatoren von der abstrakten Klasse `AbstractInteractionComponent`, die auch das `CommunicationComponent`-Interface implementiert.

Beide Komponenten haben einen Bereich, in dem sie auf Aktivitäten reagieren oder bei den Agenten Aktivitäten auslösen können. Der Bereich kann als Ausrichtung (Blickrichtung), Öffnungswinkel und Radius definiert werden, indem der entsprechende Konstruktor verwendet wird. Werden diese Parameter nicht angegeben, wird der komplette Raum, in dem sich eine Komponente befindet überwacht. Mit der Methode `setShapeRepresentation(Shape shape)` kann darüber hinaus eine beliebige Shape als Überwachungsbereich festgelegt werden.

Die Komponenten können die Methoden `boolean checkInterest(AbstractAction action, Agent agent)` und `boolean checkInterest(Agent agent)` implementieren, um anzugeben ob die jeweilige Komponente an einem Agenten und/oder einer Aktion eines Agentens interessiert ist. Standardmäßig geben beide Methoden `false` zurück. In der Initialisierungsphase werden alle interessierten Komponenten den Agenten als `IExtendedAgentListener` hinzugefügt und werden damit automatisch informiert, wenn ein Agent, an dem die Komponente interessiert ist, eine Aktion durchführt.

Bereits in der Elternklasse `AbstractInteractionComponent` aller Sensoren und Aktuatoren wird darauf geachtet, dass die eigentliche Methode zum Reagieren auf Agenten und Aktionen (`boolean handleInternal(AbstractAction action, Agent agent)`) nur dann aufgerufen wird, wenn sich der Agent im Überwachungsbereich der Komponente befindet.

Der einzige Unterschied zwischen Sensoren und Aktuatoren ist, dass der Rückgabewert der `handleInternal`-Methode von Sensoren ignoriert wird. Bei Aktuatoren oder Mischformen von Sensoren und Aktuatoren ist der Rückgabewert als Erlaubnis für eine Aktion zu verstehen. Wenn also ein Agent eine Aktion durchführt und die Methode `false` zurückgibt, bricht der Agent die Aktion ab.

Der Typ der Komponenten wird angegeben, indem die `type`-Variable gesetzt wird. Sie kann die Werte `SENSOR`, `ACTUATOR` und `MIXED` annehmen. Der Typ `MIXED` kann unter anderem für Wearables wie z.B. Smartphones sinnvoll sein.

Wearables können mit dem Konzept realisiert werden, indem mit der Methode `setAgent(Agent agent)` ein Agent gesetzt wird, der der Hauptinhaber der Komponente ist. Darüber hinaus muss mit der Methode `setWearable(boolean wearable)` festgelegt werden, dass die Komponente ein Wearable ist. Dann wird jedes mal, wenn der Agent seine Position ändert, die Position der

Komponente ebenfalls neu gesetzt.

## 4 Realisierung

In diesem Kapitel beschreiben wir die Umsetzung der in Kapitel 3 beschriebenen Konzepte. Die Unterteilung dieses Kapitels basiert auf der in Abbildung 1 dargestellten Struktur des Simulators. Nachfolgend beschreiben wir die Realisierung der Module, die ein fester Bestandteil der Simulatorarchitektur sind und geben Beispiele für den Arbeit mit ihnen.

### 4.1 Realisierung der Einzelmodule

#### 4.1.1 SimulationEngine und SimulationClock

##### *SimulationEngine*

Die `SimulationEngine` ist als Singleton realisiert. Dies hat den Vorteil, dass es zur Laufzeit nur eine Instanz von ihr geben kann, die in allen Klassen verfügbar ist. Diese Instanz hält die `World`, den `CommunicationHandler` bereit. Beide Objekte können vor dem Start der Simulation, d.h. vor dem Start der `SimulationClock` mit den jeweiligen Setter-Methoden gesetzt werden. Der Aufruf dieser Methoden nach dem Start der Simulation erzeugt einen `IllegalAccessException`.

Während der Simulation können die Welt und der Kommunikationshandler über die Getter-Methoden abgerufen werden. Auf diesem Weg können unter anderem Sensoren und Aktuatoren den Kommunikationshandler erreichen. Auch die graphische Oberfläche holt sich die zu zeichnenden Objekte über die Simulationsengine.

##### *SimulationClock*

Die `SimulationClock` ist das Herzstück der Simulation. Sie stößt sämtliche Aktionen der Komponenten des Modells an. Die Clock ist wie die Engine ebenfalls mit dem Singleton-Designpattern realisiert. So ist sicher gestellt, dass es nur eine Uhr geben kann, die aus allen Klassen erreichbar ist.

Die `SimulationClock` verwendet das Listener-Konzept, um Komponenten, die an der Simulationszeit interessiert sind regelmäßig über Änderungen zu informieren. Komponenten die sich als Listener bei der Uhr registrieren möchten müssen dazu die `ISimulationClockListener`-Schnitt-

stelle implementieren und können dann mit dem in Quelltext 1 beschriebenen Code hinzugefügt und analog wie in Quelltext 2 beschrieben wieder entfernt werden.

```
SimulationClock.getInstance()
    .addListener(ISimulationClockListener);
```

**Quelltext 1:** Beispiel für das Hinzufügen eines Listeners an die `SimulationClock`

```
SimulationClock.getInstance()
    .removeListener(ISimulationClockListener);
```

**Quelltext 2:** Beispiel für das Entfernen eines Listeners von der `SimulationClock`

Über das Listener-Konzept werden in jeder simulierten Sekunde Aktionen bei den Listeners angestoßen. Standardmäßig ist es nicht möglich, eine Java-Collection während des Durchlaufens dieser zu verändern. Somit wäre es unmöglich, dass sich eine Komponente als Listener abmelden kann. Die `SimulationClock` speichert jedoch alle Hinzufüge- und Entfernanfragen an die Listener-Collection zwischen und führt diese vor dem nächsten Durchlauf aus.

Die Uhr basiert auf einem Timer-Thread, der alle  $x$  Millisekunden auslöst. Wenn die Uhrzeit skaliert wird, wird mit dem nächsten Tick ein neuer Timer eingerichtet. Das Pausieren der Simulation stoppt nicht den Timer-Thread sondern verhindert lediglich, dass dieser die `SimulationTime` inkrementiert.

#### 4.1.2 Generatoren

Für das Erstellen einer neuen Simulation reicht es aus, eine Simulationswelt (der Klasse `World`) zu erzeugen. Dieser Vorgang wird als Generierung bezeichnet und von einem Generator realisiert. Ein Beispiel, wie eine solche erzeugt werden kann, ist in diesem Kapitel an Hand der Demonstrationssimulation dargestellt.

Aus Modularitätsgründen haben wir uns für das Konzept eines leicht austauschbaren Generators entschieden. Dafür wurde die Schnittstelle `IWorldGenerator` entworfen, welches nur eine Methode (`generateWorld()`) definiert, die von allen konkreten Generatoren als Schnittstelle genutzt werden sollte. Darüber hinaus gibt es im Generatorpaket (`de.uniluebeck.imis.casi.generator`) einige Helferklassen, deren Name mit `Collector` gekennzeichnet ist und einen `Linker`.

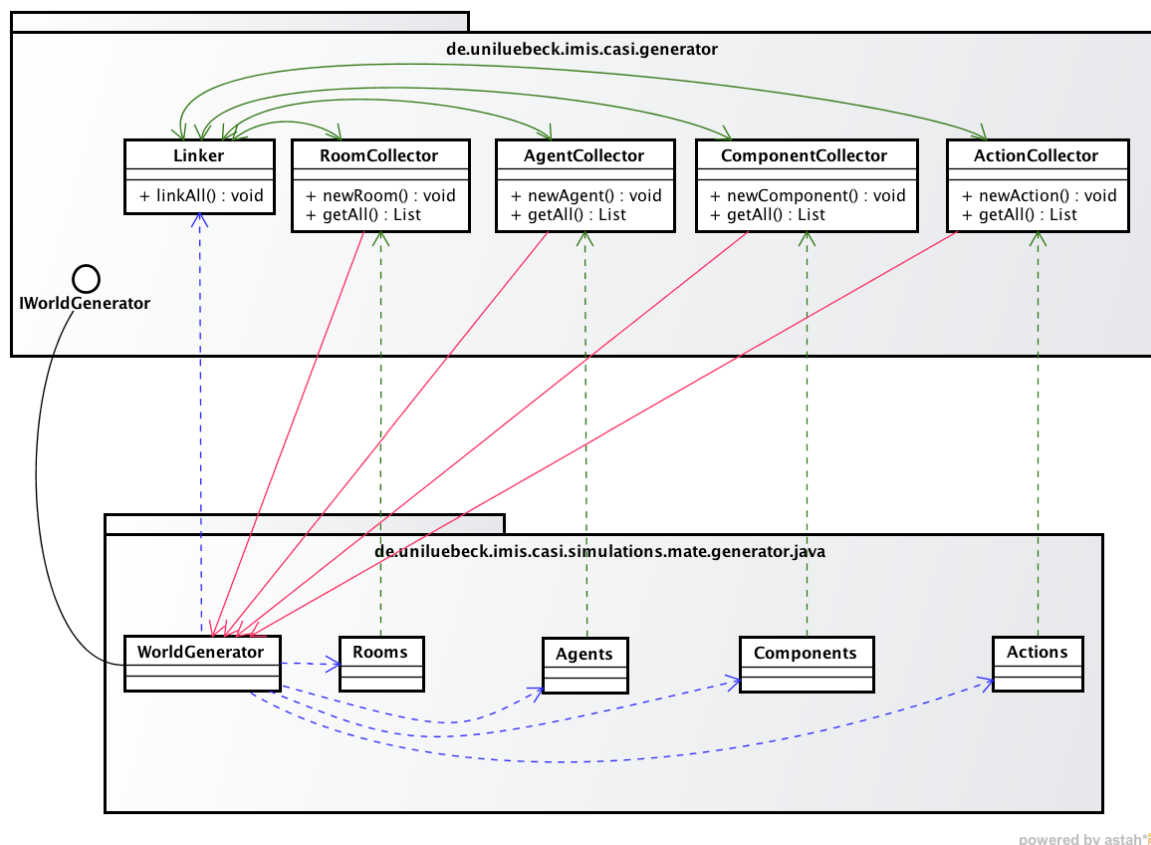
Durch den gewählten Ansatz der starken Abstraktion ist es möglich Simulationen auf die unterschiedlichsten Arten zu beschreiben ohne Änderungen am Simulator vornehmen zu müssen. Das von uns gewählte, am kurzfristigsten umzusetzende Vorgehen ist es, direkt aus Java-Klassen heraus die Objekte der Simulationswelt zu erzeugen und zu verknüpfen. Weitere Möglichkeiten wären:

- Beschreibungssprachen wie XML, YML oder JSON
- Erzeugung durch eine eingebettete Skriptsprache wie LUA oder JRuby

In einem frühen Konzept unserer Projektarbeit war insbesondere eine Beschreibung durch eine spezifizierte XML (Arbeitstitel: CASiX) vorgesehen. Diese wurde jedoch im Laufe des Semesters wegen nicht unerheblichen Aufwand zurückgestellt. Da eine saubere und klare Schnittstelle definiert wurde, ist es ohne Weiteres möglich, dass diese später noch hinzugefügt werden.

#### Generierung durch Java-Code

Für das Erstellen einer neuen Simulation hat es sich als sinnvoll erwiesen, ein neues Paket zu erstellen und darin die `IWorldGenerator`-Schnittstelle zu implementieren. Innerhalb dieses Paketes ist eine Aufteilung in weitere Klasse zu empfehlen.



**Abbildung 2:** abgewandeltes UML-Klassendiagramm zur Darstellung der Aufteilung eines Generators. *Blau gestrichelt:* Aufruf statischer Methoden, *grün gestrichelt:* Einfügen von Objekten, *grün:* Verknüpfung von Objekten untereinander, *rot:* Entgegennehmen von gesammelten Objekten

In Abbildung 2 ist, exemplarisch an unserer Demosimulation, eine solche Aufteilung gezeigt. Dort ist das untere Paket für die MATE-Simulation zu sehen. Darin: der eigentliche `WorldGenerator`, welcher die zu generierenden Teile durch Aufrufe von statischen Methoden an die andere Klassen delegiert. Hier gezeigt durch gestrichelten blauen Linien. Die jeweiligen Klassen (alle als `static` deklariert), arbeiten dann die, in ihnen beschriebenen, Java-Ausdrücke ab. Daraus resultierende Objekte werden in den, vom CASi-Kern bereitgestellten, Kollektoren gesammelt (gestrichelte grüne Pfeile). Die Reihenfolge der Erzeugung ist besonders zu beachten, da für einige Objekte andere bereits vorhanden sein müssen. Es hat sich die Reihenfolge: erst Räume, dann Agenten, dann Sensoren und Aktuatoren und schlussendlich Aktionen etabliert. Wenn alle Erzeugerklassen abgearbeitet sind, ruft der `WorldGenerator` den `Linker` auf. Dieser ist auch vom CASi-Kern bereitgestellt. Er übernimmt das Verknüpfen der Objekte untereinander, so zum Beispiel das Hinzufügen der für die Agenten erzeugten Aktionen in deren Todo-Liste. Als letzten Schritt nimmt der Generator alle Komponenten (rote Pfeile) und erzeugt daraus ein `World`-Objekt.

#### 4.1.3 Benutzungsschnittstellen

Als Benutzungsschnittstelle für den Simulator haben wir die *Simple GUI* entwickelt. Diese soll die simulierte Welt grafisch darstellen und es ermöglichen, gewünschte Informationen abzurufen. Wir wollten, dass diese Benutzungsschnittstelle sowohl für die späteren Hauptbenutzer, die Simulationen testen, viele Funktionen bietet, aber auch für Benutzer verständlich ist, die von dem System noch keine fortgeschrittenen Kenntnisse haben. Zum Beispiel könnte sie auch benutzt werden, um anderen das MACK-Framework zu präsentieren und zu veranschaulichen.

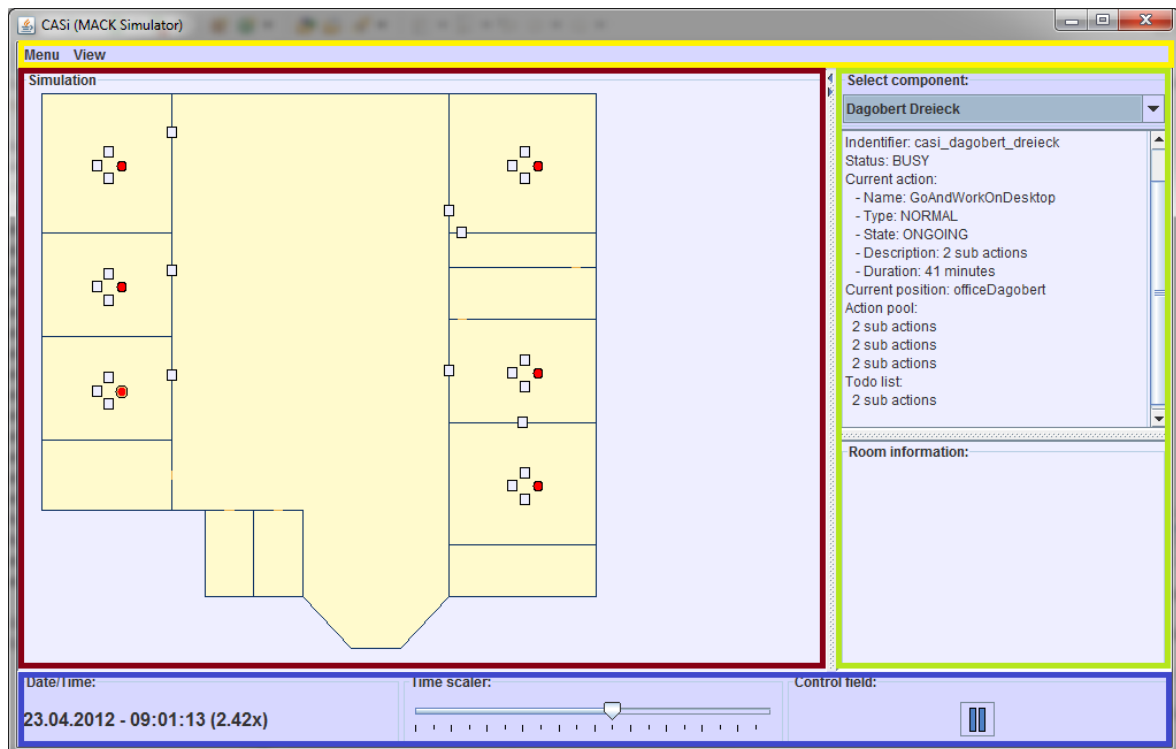
Die *Simple GUI* ist eines der Module des Simulators, die sich leicht durch ein passendes anderes ersetzen lassen. Wenn also beispielsweise die angezeigten Informationen nicht ausreichend oder für bestimmte Simulationen überflüssig und deshalb störend sind, kann man die Benutzungsschnittstelle austauschen. Realisiert wird dies durch das Interface `IMainView`, die neue Schnittstelle muss bei der Implementierung dieses Interface benutzen. Ein ganz einfaches Beispiel für eine Implementierung ist die Klasse `GuiStub`, die eingesetzt wird, wenn keine grafische Benutzungsschnittstelle erwünscht ist.

Für die graphische Oberfläche benutzen wir die Swing-Bibliothek, da wir aus dem Praktikum *Softwaretechnik* einige Erfahrung mit der Swing-Entwicklung haben.

Die *Simple GUI* ist aufgebaut, wie in Abbildung 3 zu sehen ist. Im Simulationsbereich (rot) wird die simulierte Welt gezeichnet. Der Informationsbereich (grün) an der rechten Seite bietet dem Benutzer die Möglichkeit detaillierte Informationen über Simulationskomponenten abzufragen. Zwischen dem Simulationsbereich und dem Informationsbereich befindet sich eine verschiebbare Trennung. Der Steuerungsbereich (blau) im unteren Teil lässt den Benutzer eingeschränkt Einfluss auf die Simulation nehmen. Die Menüleiste (gelb) im oberen Bildbereich bietet einige grundlegende Programmfunktionen. Letztere sollte in einem im Fenster ausgeführten Programm nicht fehlen, da sie den meisten



Benutzern vertraut ist und ihr Fehlen für eine gewisse Hilflosigkeit sorgen könnte.



**Abbildung 3:** Screenshot der Anwendung

Der Simulationsbereich besteht aus einer `JLayeredPane`. Auf dieser wiederum gibt es drei Ebenen:

- Ebene 1: Auf der tiefsten Ebene wird der Hintergrund gezeichnet. Dazu gehören die Räume mit ihren Wänden und Türen und die Einflussbereiche der Sensoren. Außerdem werden einige weitere, für das Debugging wichtige Elemente dargestellt, wie die zentralen Punkte der Räume und der Türen. Ebenfalls auf dieser Ebene liegen die Labels für Sensoren, Räume und Türen. In ungünstigen Fällen können sich die Bezeichner im Simulationsbereich überschneiden. Da aber die Identifizierung der Komponenten trotzdem möglich ist und die Berechnung für Bezeichnerposition und -größe für beliebige Simulationen recht schwierig ist, haben wir uns entschieden, das Problem zu vernachlässigen.
- Ebene 2: Auf dieser Ebene liegen die Sensoren und Aktuatoren. Diese sind ein wesentlicher Bestandteil der Simulation und sollen deshalb nicht durch Teile des Hintergrundes verdeckt werden. Jedes dieser Komponenten wird dargestellt mit Hilfe der Klasse `InteractionComponentView`, die von `ComponentView` erbt, welche wiederum eine `JComponent` ist. Ein Sensor oder Aktuator wird gezeichnet als ein Quadrat mit der Hintergrundfarbe der Simulation. Da die Sensoren und Aktuatoren beliebig viele Zustände annehmen können, haben wir darauf verzichtet zu implementieren, dass ihr Zustand an der Färbung erkennbar ist.

- Ebene 3: Auf der obersten Ebene befinden sich die Agenten. Diese haben wir als wichtigsten Bestandteil der Simulation eingestuft, deshalb sollen sie durch nichts anderes überdeckt werden. Jeder Agent wird dargestellt mittels der von uns implementierten Klasse `AgentView`, die ebenfalls von `ComponentView` erbt. Die Klasse `AgentView` implementiert das Interface `IAgentListener`. Mit den Methoden `stateChanged(STATE newState, - Agent agent)` und `positionChanged(Point2D oldPosition, Point2D newPosition, Agent agent)` erkennt diese Komponente, ob sich der Status oder die Position des Agenten verändert hat und zeichnet sich entsprechend neu. Agenten werden als Kreise gezeichnet, ihre Farbe entspricht ihrem Zustand, so bedeutet *rot* zum Beispiel, dass der Agent gerade beschäftigt ist.

Um den Simulationsbereich gut auszunutzen, skaliert sich die gezeichnete, simulierte Welt automatisch so, dass sie möglichst groß angezeigt wird.

Im Simulationsbereich gibt es einige Interaktionsmöglichkeiten. Verschiedene Teile der Simulation können per Mausklick ausgewählt werden, über diese Komponenten werden dann weitere Informationen angezeigt. Zu diesen Interaktionselementen gehören:

**Agenten** Wenn diese Komponente angeklickt wurde, wird sie markiert. Äußerlich erkennt man das daran, dass sie größer gezeichnet wird als die Anderen. Zur Zeit kann nur eine Komponente markiert sein, das bedeutet, wenn eine neue Komponente markiert wird, wird bei der alten die Markierung gelöscht.

**Sensoren/Aktuatoren** Auch diese Komponente wird markiert, wenn sie angeklickt wurde und wird dann größer gezeichnet.

**Räume** Für die Darstellung der Räume gibt es keine eigene Klasse, weil die Darstellung sich während der Simulation nicht verändert. Um Extrainformationen zu Räumen zu bekommen, muss auf die entsprechende Stelle auf dem Hintergrund geklickt werden.

Ein Problem war, dass sich die Sensoren, Aktuatoren und Agenten, die in den meisten Fälle auf zentralen Punkte der Räume gezeichnet werden, sich überlappen und dann nicht mehr anklickbar waren. Um die Anzeige der Simulation zu optimieren, werden diese Komponenten nun in einem Kreis um die Mittelpunkte der Räume angeordnet. Auf diese Weise können sie ohne Probleme selektiert werden. Diese automatische Anordnung wird aus Performancegründen in der *Simple GUI* nur bei den zentralen Punkten der Räume vorgenommen.

Der Informationsbereich dient zur Darstellung detaillierter Informationen zu bestimmten Komponenten der Simulation. Im oberen Bereich befindet sich eine Auswahlmöglichkeit (`JComboBox`). Hier sind alle Agenten, Sensoren und Aktuatoren aufgeführt. Die Gruppen sind mit Trennzeichen separiert und vom Benutzer auswählbar. Die Auswahl einer Komponente ist gleichbedeutend mit einem Mausklick auf die Komponente, sie wird also ebenfalls im Simulationspanel markiert. Im oberen Informationsbereich werden die Informationen zum aktuell markierten Agenten, Sensor oder Aktua-

tor angezeigt. Im unteren Informationsbereich werden Informationen zum zuletzt angeklickten Raum angegeben (beim Starten des Simulators ist dieser Bereich leer).

Der Steuerungsbereich gibt dem Benutzer die Möglichkeit Einfluss auf die laufende Simulation zu nehmen. Auf der linken Seite befindet sich die Anzeige der Zeit und des Datums der Simulation, anhand derer der Benutzer erkennt wie weit die Simulation fortgeschritten ist. Um die Simulationsgeschwindigkeit zu verändern, befindet sich in der Mitte ein Schieberegler ([JSlider](#)). Wir haben uns hier für einen Schieberegler entschieden, um einfach die minimale und maximale Simulationsgeschwindigkeit einhalten zu können. Der Schieberegler hat eine quadratische Skala, um sinnvolle Verhältnisse zwischen den Einteilungen des Reglers zu schaffen. Im rechten Bereich befindet sich noch ein Togglebutton zum Pausieren der Simulation. Wir benutzten hier die Metapher des Audiospielers, mit den bekannten Symbolen für *Pause* und *Wiedergabe*.

Die Menüleiste beinhaltet zwei Menüs. Unter dem Bezeichner *Menu* werden grundlegende Programmfunktionen angeboten:

**Pause/ Resume** Dieser Menüpunkt hat die gleiche Funktion wie der Pausebutton im Steuerungsbereich. Er pausiert die Simulation oder setzt sie fort.

**Exit** Dieser Menüpunkt beendet das Programm.

In dem Menü *View* lassen sich Einstellungen zur Anzeige im Simulationsbereich machen. Es können folgende Einstellungen gemacht werden:

**Show sensors/actuators** blendet die vorhandenen Sensoren und Aktuatoren ein oder aus.

**Show sensor/actuator area** blendet die Einflussbereiche der Sensoren und Aktuatoren ein oder aus.

**Show sensor/actuator labels** blendet die Namen sowie einige Statusinformationen zu den Sensoren und Aktuatoren ein oder aus.

**Show door labels** blendet die Bezeichner der Türen ein oder aus.

**Show door points** blendet die zentralen Punkte von Türen ein oder aus.

**Show room labels** blendet die Namen der Räume ein oder aus.

**Show room points** blendet die zentralen Punkte der Räume ein oder aus.

Die Anzeigeeinstellungen werden durch Checkboxes ([JCheckBoxMenuItem](#)) vorgenommen, an denen leicht erkennbar ist, welche Elemente zur Zeit angezeigt oder ausgeblendet werden. Welche Komponenten zu Beginn eingeblendet sind, hängt davon ab, in welchem Modus das Programm gestartet wird. Ist der Entwicklermodus aktiviert, werden alle Elemente angezeigt, um Debugging zu ermöglichen. Anderenfalls werden nur die Sensoren und Aktuatoren eingeblendet und zusätzliche Informationen können bei Bedarf angezeigt werden. Das soll die Übersicht nach dem Programmstart bewahren.

#### 4.1.4 Aktionen

In unserer Implementierung sind verschiedenen Aktionen enthalten. In der Package-Struktur haben wir diese wie folgt aufgeteilt:

In `.simulation.model.actions` befinden sich Aktionen, die allgemein im Simulator durchgeführt werden können. Diese sind unabhängig vom MACK-Framework, Sensoren und Aktuatoren.

Im Paket `.simulation.model.mackActions` befinden sich Aktionen, die bestimmte Sensoren und Aktuatoren benötigen und somit abhängig von der jeweiligen Simulation sind.

##### *Allgemeine Aktionen*

Zu den wichtigsten allgemeinen Aktionen in unserer Implementierung zählen unter anderem:

**Move** Die `Move`-Aktion dient dazu, die Position eines Agentens zu Ändern. Hierbei wird in der Initialisierung der Aktion nur das Ziel angegeben. Zum Zeitpunkt der Ausführung der Aktion wird automatisch der Weg zum Ziel berechnet und pro Iteration die Position um eine bestimmte Anzahl von Wegpunkten verändert. Dieser Aktion kann keine Dauer übergeben werden, da sie automatisch terminiert, wenn das Ziel erreicht wurde. Gibt es keinen Weg zum Ziel. Bricht die Aktion automatisch ab.

**Follow** Mit der `Follow`-Aktion kann ein Agent einer `AbstractComponent` folgen. Somit ist es möglich, dass ein Agent einem anderen während eines Gespräches folgt. Die Aktion hat keine spezifische Dauer, sondern wird erst dann beendet, wenn `Follow.complete()` aufgerufen wird.

**SpeakTo** Mit dieser Aktion kann ein Agent mit einem anderen sprechen. Hierbei wird zunächst überprüft, ob beide Agenten im selben Raum sind. Außerdem wird überprüft, ob die Aktion der anderen Person eine Unterbrechung zulässt.

Wenn der Agent unterbrechbar ist, wird ihm eine Aktion vom Typ `Follow` in die `interruptList` gelegt, so dass der Agent für die Dauer des Gespräches dem Sprecher folgt und ebenfalls mit dem Gespräch beschäftigt ist. Dies verhindert, dass der andere Agent im Gespräch wegläuft. In der Methode `SpeakTo.postActionTask(AbstractComponent)` wird `Follow.complete()` aufgerufen und somit der Gesprächspartner wieder freigegeben.

**GoAndSpeakTo** Diese `ComplexAction` sorgt dafür, dass der ausführende Agent zunächst die Standard-Position des Gesprächspartners aufsucht, welche in der Regel sein Büro ist, und dort versucht, mit ihm zu reden.

In der aktuellen Implementierung bricht der Agent diese Aktion ab, wenn der Partner sich nicht in seinem Büro befindet. Diese Aktion kann aber erweitert werden, wenn es z.B. Aktuatoren wie das `DoorPlate` gibt, welches Auskunft über die wirkliche Position des Anderen gibt.

### Simulationsspezifische Aktionen

Die simulationspezifischen Aktionen hängen von Sensoren und Aktuatoren einer Simulationsimplementierung ab. In der aktuellen Implementierung gibt es unter anderem folgende Aktionen:

**WorkOnDesktop** Mit dieser Aktion kann man einen Agenten für eine bestimmte Zeit arbeiten lassen. Hierfür erzeugt man eine neue Instanz dieser Klasse und gibt im Konstruktor den Arbeitsplatz, die Programmklasse, die Frequenz und die Dauer an.

Die Programmklasse ist dabei äquivalent zu den Klassen, mit denen auch das MACK-Framework umgehen kann. Die Frequenz ändert sich während der Ausführung um einen normalen Arbeitsfluss zu simulieren.

**TurnCube** Mit der `TurnCube`-Aktion kann ein `Cube` (Cubus) auf einen bestimmten Status gedreht werden.

**PutKeyInDropZone** Diese Aktion fügt eine Referenz auf den Agenten in eine Liste einer Dropzone. Vorher wird überprüft, ob der Agent bereits einer anderen `DropZone` hinzugefügt wurde, um zu vermeiden, dass der Agent seinen „Schlüssel“ in mehrere Dropzones legen kann.

**RemoveKeyFromDropZone** Analog zur `PutKeyInDropZone`-Aktion kann mit dieser Aktion eine Referenz auf einen Agenten aus einer `DropZone` entfernt werden.

#### 4.1.5 Aktionsverwaltung

In unserer Implementierung nutzt die Verwaltungskomponente für Aufgaben im Wesentlichen den Scheduling-Algorithmus *Earliest Deadline First*. Darüber hinaus nutzt der Scheduler weitere Möglichkeiten um auch Parameter außer der Deadline zu berücksichtigen, da nicht jede Aufgabe eine Deadline haben muss.

#### 4.1.6 Kommunikationshandler

Die abstrahierte Beschreibung der Schnittstellen eines Kommunikationshandlers befindet sich im Paket `de.uniluebeck.imis.casi.communication`.

Eine Implementierung für die Zusammenarbeit mit dem MACK-Framework befindet sich in `de.uniluebeck.imis.casi.communication.mack`. *Bei Änderungen am MACK-Protokoll sind im Wesentlichen die Methoden in diesem Paket anzupassen.* Kommunikationshandler können die externe Kommunikation des Simulators ermöglichen. Dies bedeutet, dass sie sehr flexibel sein müssen, um jede erdenkliche Art von Kommunikation zu ermöglichen. Aus diesem Grund sind Kommunikationshandler als Implementierung der `ICommunicationHandler`-Schnittstelle realisiert. Sie müssen in der Main-Klasse `CASi` an den `MainController` übergeben werden. Es ist nur ein Kommunikationshandler zur Zeit vorgesehen. Benötigt man mehrere Handler, sind Subhandler denkbar, denen

die Nachrichten weiter gegeben werden.

In der Umsetzung des Simulators zum Testen des MACK-Frameworks wurde der Kommunikationshandler `MACKNetworkHandler` wie folgt realisiert:

Der Handler bekommt zu Beginn eine Konfigurationsdatei in XML-Format übergeben, die in diesem Fall Java-Beans-XML enthält, mit der die allgemeine Konfiguration der Netzwerkschnittstelle und die zur Verfügung stehenden `XmppIdentifier` aufgelistet sind. Jeder `XmppIdentifier` setzt sich aus drei `String`-Werten zusammen und repräsentiert eine Zeile aus der Tabelle `mate_userdevices` des Servers:

**componentOwner** Der eindeutige Name des Agents, dem die Komponente zugeordnet ist. Dieser `String` muss identisch mit dem Feld `username` in der Tabelle `mate_userdevices` sein.

**componentType** Die Art der Komponente. Dieser `String` muss identisch mit dem Feld `channel` in der Tabelle sein.

**id** Der XMPP Benutzername der Komponente ohne den Host (`@server.de`).

Die Netzwerkkonfigurationsdatei enthält für jeden `XmppIdentifier` einen Eintrag der in Quellcode 3 beschriebenen Form im Bereich der `ArrayList useableJabberIdentifiers`. In der Initialisierungsphase des Simulators wird jeder Komponente, die sich mit dem in Quellcode 4 gezeigten Befehl am `MACKNetworkHandler` anmeldet, eine Jabber-ID aus dieser Liste zugewiesen, sofern noch eine zum jeweiligen Typ passende verfügbar ist. Andernfalls wird eine Warnung ausgegeben und die Komponente nicht angemeldet. In der Registrierungsphase überprüft der Handler, ob der jeweilige Jabber-Identifizier bereits am XMPP-Server registriert ist. Ist dies nicht der Fall, versucht der Simulator automatisch, den Identifizier zu registrieren. Wenn mehrere Adressen noch nicht registriert wurden, kann es abhängig vom, in der Konfigurationsdatei angegebenen, `REGISTRATION_DELAY` zu längeren Wartezeiten kommen, bevor der Simulator mit der Simulation beginnt. Wenn jedoch alle Komponenten bereits registriert sind, kann die Simulation ohne Verzögerung starten.

Der `MACKNetworkHandler` erzeugt nach erfolgreichem Zuordnen der Identifier und dem Login am XMPP-Server für jede Komponente einen Chat mit dem MACK-Server. Über diesen Chat werden Pull- und Push-Anfragen gesendet und Rückmeldungen empfangen und an die betroffene Komponente weitergeleitet.

Sowohl die `send`-Methode im `ICommunicationHandler`-Interface, als auch die `receive`-Methode im `ICommunicationComponent`-Interface sehen vor, dass die Nachrichten vom Typ `Object` sind. Hierdurch ist ein hoher Abstraktionsgrad und somit eine hohe Flexibilität gegeben. Für die richtige Interpretation der Nachrichten ist somit der Entwickler des Handlers und der Sensoren und Aktuatoren verantwortlich.

In der MATE-Implementierung sind die Nachrichten immer vom Typ `String`, da sie die den Body der Chat-Nachrichten, also die eigentliche Nachricht im Format des MACK-Protokolls enthalten.

Zum Erzeugen der Nachrichten in den Aktuatoren und Sensoren kann die `MACKProtocolFactory` verwendet werden. Hierfür steht für Push- und Pull-Nachrichten je eine Methode zur Verfügung. Nachrichten, die vom MACK-Server empfangen werden, können mit der `parseMessage`-Methode in ein `MACKInformation`-Objekt zerlegt werden, das die Antwort in einer leicht zugänglichen Form bereit hält.

#### 4.1.7 Sensoren und Aktuatoren

In diesem Unterabschnitt beschreiben wir die verschiedenen, bereits durch uns realisierten, Sensoren und Aktuatoren um verständliche Beispiele für die Entwicklung weiterer Komponenten bereit zu stellen. Die modellierten Komponenten des MACK-Frameworks befinden sich im Paket `de.uniluebeck.imis.casi.simulation.model.mackComponents`.

##### *Gemeinsamkeiten von Sensoren und Aktuatoren*

Komponenten, die in regelmäßigen Abständen eine Aktion ausführen müssen, können sich als Zuhörer bei der `SimulationClock` hinzufügen. Sie können dann zum einen die Standardmethoden der `ISimulationClockListener`-Schnittstelle verwenden und darüber hinaus die `sendRecurringMessage(SimulationTime newTime)`-Methode überschreiben, die automatisch, in dem durch `AbstractInteractionComponent.PULL_INTERVALL` beschriebenen Intervall, aufgerufen wird. Diese Methode eignet sich besonders, um regelmäßige Pull-Nachrichten an den Server zu senden.

Alle Komponenten müssen die `getType()`-Methode überschreiben, um einen textuellen Wert für den Typ der Komponente anzugeben. Im Falle des MACK-Frameworks entspricht der Typ genau der vom Framework verwendeten Bezeichnung.

Darüber hinaus können Komponenten diverse `checkInterest(...)`-Methoden überschreiben, um ihr Interesse an bestimmten Agenten, Aktionen oder der Kombination aus beidem anzugeben. Von besonderer Bedeutung ist in diesem Fall die `checkInterest(Agent agent)`-Methode, da diese in der Initialisierungsphase des Simulators verwendet wird, um das Interesse der Komponenten an den Agenten festzustellen. Dieses Vorgehen dient dazu, die Größe der Liste der Komponenten, die informiert werden müssen, klein zu halten. Das Standardverhalten dieser Methode ist, `false` zurückzugeben. Wenn eine Komponente über Aktionen informiert werden möchte, muss sie dieses Verhalten überschreiben.

Es hat sich als sinnvoll herausgestellt, eine zentrale Funktionalität zu implementieren, mit der das Interesse von Komponenten an bestimmten Aktionen vor der eigentlichen Verarbeitung der Aktion festgestellt werden kann. Zu diesem Zweck stellt die Klasse `AbstractInteractionComponent` zwei Methoden bereit.



Mit der Methode `getInterestingPart (AbstractAction action)` wird versucht, aus einer übergebenen Aktion den für die jeweilige Komponente interessanten Teil herauszusuchen und zurückzugeben. Die Methode geht dabei wie folgt vor:

- Wenn die Aktion bereits interessant ist, wird sie unverändert zurückgegeben.
- Wenn es sich um eine komplexe Aktion handelt, wird überprüft, ob die aktuelle Unteraktion interessant ist. Wenn dies der Fall ist, wird nur dieser Teil zurückgegeben.
- In allen anderen Fällen wird einfach `null` zurückgegeben.

Diese Methode ist besonders nützlich, um den für eine Bearbeitung der Aktion relevanten Teil zu erhalten.


Mit der Methode `checkInterest (AbstractAction action)` wird nach dem oben beschriebenen Vorgehen überprüft, ob es sich bei dem Parameter um eine interessante Aktion handelt.

Verweise auf interessante Aktionstypen werden in der Sammlung `# interestingActions : Collection<Class>` in der Klasse `AbstractInteractionComponent` gespeichert. Implementierungen von Sensoren und Aktuatoren können z.B. im Konstruktor Werte zu dieser Liste hinzufügen:

```
interestingActions.add(SpeakTo.class);
interestingActions.add(HaveAMeeting.class);
```

**Quelltext 5:** Beispiel für das Hinzufügen interessanter Aktionstypen aus dem Konstruktor der Klasse `Mike`.

#### *Cube (MATE: Cubus)*

Der `Cube` besitzt eine Status-Variable, die Werte aus `Code.State` annehmen kann. Die Werte entsprechen den Textwerten, die das MACK-Framework für die Kommunikation mit dem Cubus verwendet. In der aktuellen Implementierung reagiert der `Cube` auf die `WorkOnDesktop`-Aktion, indem er anhand der Arbeitsgeschwindigkeit und der verwendeten Programmklasse sinnvolle Annahmen über die mögliche Tätigkeit der arbeitenden Person macht. Entspricht der aktuelle Status des `Cubes` nicht dem ermittelten, löst er mit einer bestimmten Wahrscheinlichkeit und einer Verzögerung von `Cube.TURN_CUBE_SCHEDULE_DELAY` Sekunden eine `TurnCube`-Aktion aus. Die Wahrscheinlichkeit kann über die `ConfigMap` definiert werden. Wir keine explizite Wahrscheinlichkeit in der `ConfigMap` des Agents gesetzt, wir die `Cube.DEFAULT_TURN_CUBE_PROBABILITY` verwendet. 

Der `Cube` fügt sich als Zuhörer zu der `SimulationClock` hinzu. Dies führt dazu, dass durch die Elternklasse `AbstractInteractionComponent` in regelmäßigen Abständen die `sendRecurringMessage (SimulationTime newTime)`-Methode aufgerufen wird. In dieser Me-



thode wird eine Pull-Nachricht gesendet, mit der die Werte für `activity` und `interruptibility` abgefragt werden.

### *Desktop (MATE: DesktopActivityAnalyzer)*

Der `Desktop` ist ein Beispiel für eine Komponente, die zunächst kein Interesse an einem Agenten hat. Die Interaktion mit diesem Sensor erfolgt durch die `WorkOnDesktop`-Aktion, die die `work(Program program, Frequency frequency)`-Methode aufruft, wenn ein Agent mit diesem Computer arbeitet.

Wenn sich die Werte für die Programmklasse und die Intensität der Arbeit ändern, sendet der `Desktop` diese Werte als Push-Nachricht an den MACK-Server.

### *DoorLight*

Das `DoorLight` ist ein Aktuator, der Personen davon abhalten kann, einen Raum zu betreten. Hierfür kann es einen Status aus `DoorLight.State` annehmen. Die möglichen Werte entsprechen den `interruptibility`-Werten, die der Server zur Verfügung stellt.

Der Aktuator sendet periodische Pull-Nachrichten an den Server, um den Werte für `interruptibility` abzufragen. Die Komponente ist so definiert, dass sie Interesse an allen Personen außer der zugehörigen hat. Es reagiert auf `Move`-Aktionen, um zu überprüfen, ob eine Bewegung in den Raum oder nur zufällig durch den beobachteten Bereich führt.

### *DoorSensor*

Der Türsensor ist ein `IDoorListener` und wird somit über die Veränderung des Türstatus informiert. Bei Statusänderungen sendet der Sensor den aktuellen Wert als Push-Nachricht an den MACK-Server.

### *DropZone*

Die `DropZone` ist ebenfalls eine passive Komponente, mit der über die Aktionen `PutKeyInDropZone` und `RemoveKeyFromDropZone` interagiert werden kann. Mit diesen Aktionen können Personen eine Referenz auf sich in der `DropZone` speichern. Hiermit wird das Ablegen und Entfernen eines Schlüssels simuliert.

Der Sensor sendet eine Liste von Namen an den Server, wenn sich diese ändert.

##### *Mike (MATE: MikeRophone)*

Dieser Sensor beobachtet standardmäßig den gesamten Raum, in dem er sich befindet. Er dient dazu, sprechende Personen zu erkennen. Um dieses Verhalten zu **Simulieren**, reagiert das *Mike* auf die Aktionen *SpeakTo* und *HaveAMeeting*. Bei der ersten Aktion nimmt es die Namen von den beiden Personen, die miteinander sprechen und sendet diese Liste an den Server. Bei der zweiten Aktion wählt der Sensor zufällig aus allen beteiligten Personen zwei aus, deren Namen er an den MACK-Server sendet.

Das Senden und erneute Ermitteln der Sprecher erfolgt automatisch. Die Zeit zwischen den Aktualisierungen variiert zufällig zwischen 10 und 120 Sekunden. Auf diese Weise wird sichergestellt, dass sich die Sprecher nicht zu schnell abwechseln und eine möglichst realitätsnahe Unterhaltung simuliert werden kann.

```

<void method="add">
  <object class="de.uniluebeck.imis.casi.communication
.mack.XmppIdentifier">
    <void property="componentOwner">
      <string>casi_hermann_matsumbishi</string>
    </void>
    <void property="componentType">
      <string>doorlight</string>
    </void>
    <void property="id">
      <string>CASI_MATe_doorlight_0</string>
    </void>
  </object>
</void>

```

**Quelltext 3:** Beispiel für eine XML Netzwerkkonfigurationsdatei

```

public void register(ICommunicationComponent comp);

```

**Quelltext 4:** Beispiel zur Registrierung einer kommunizierenden Komponente

## 5 Evaluation

In diesem Kapitel beschreiben wir, welche Komponenten wir evaluieren wollen und mit welchen Methoden wir dies gemacht haben. Wir beschreiben zunächst die Ziele, die wir mit der Evaluation verfolgt haben und erklären dann in Abschnitt 5.2 welche Methoden wir verwendet haben und wie wir vorgegangen sind.

### 5.1 Ziel

Es gibt verschiedene Bereiche in diesem Projekt, die mit unterschiedlichen Methoden überprüft werden konnten.

Wir verwenden einige `JUnit`-Testfälle um die Funktionalität einzelner Komponenten und Klassen des Simulators zu verifizieren. Ein Beispiel hierfür sind die Wegfindungsalgorithmen und weitere Komponenten des Modells, da das frühzeitige Erkennen von Fehlern in diesem Teil des Simulators besonders wichtig ist, um die Stabilität und Zuverlässigkeit des Simulators sicher zu stellen.

Des Weiteren haben wir eine Simulation entwickelt, die auch zum Testen der MATe Anwendung verwendet werden kann. Auf diese Art ist es möglich, den Simulator als Gesamtsystem zu testen und zu gewährleisten, dass das Simulationsframework mit frei definierten Simulationen umgehen kann und das Konzept vollständig in dem Sinne ist, dass verschiedene Simulationen realisiert und simuliert werden können.

Zu diesem Zweck dienen ebenfalls die vor Beginn der Implementierungsphase definierten Szenarien. Mit ihnen haben wir während der Entwicklungsphase permanent die Vollständigkeit des Systems verifiziert und gegebenenfalls die Komponenten erweitert.

Ein weiterer Bereich, der evaluiert werden muss, sind die Log-Ausgaben, die insbesondere über den Simulation-Logger geschrieben werden, da diese im späteren Produktivbetrieb für den Test des MACK-Frameworks von großer Bedeutung sind. Dies bedeutet, dass wir losgelöst von der eigentlichen Simulationsengine überprüfen müssen, ob die Logfiles ausreichende Informationen enthalten, um den Ablauf der Simulation nachvollziehen und Fehler oder Schwächen der Reasoner ablesen zu können.

Darüber hinaus muss evaluiert werden, ob der Simulator seinen Zweck erfüllt und verwertbare Infor-

mationen über das Verhalten des MACK-Frameworks generiert.

## 5.2 Methoden und Vorgehen

Insbesondere zu Beginn der Entwicklungsphase war es nicht möglich, das Gesamtsystem zu testen, da viele Klassen noch nicht oder nicht vollständig implementiert waren. Aus diesem Grund haben wir für Teilbereiche, deren Funktionalität besonders wichtig oder aufwändig ist, Testfälle entwickelt, mit denen das gekapselte Testen dieser Klassen möglich ist.

Die einzige effektive Methode zum Testen des Gesamtsystems stellt die Entwicklung einer umfangreichen Simulation dar. Da es bereits eine Implementierung der MATE-Anwendung gibt und ein Server für diese Anwendung zur Verfügung steht, haben wir ein Szenario entwickelt, das mit diesem Server zusammenarbeiten kann und somit die Interaktion zwischen dem MATE-Server und dem Simulator verifizieren kann.

Durch dieses Vorgehen kann gleichzeitig die Funktionalität mehrere Teilsysteme evaluiert werden. Hierzu zählen unter anderem folgende Bereiche:

- die Netzwerkschnittstelle (`MACKNetworkHandler`)
- die Benutzungsschnittstelle (`SimpleGui`)
- die Simulationsengine inklusive des Models
- die Flexibilität der Schnittstellen zur Beschreibung von Simulationen
- die Vollständigkeit der Log-Ausgaben
- die Interaktion zwischen Sensoren, Aktuatoren und der Netzwerkschnittstelle
- die Interaktion von Sensoren, Aktuatoren und der Simulation (z.B. mit den `Agents`)

Um zu verifizieren, dass der Simulator seinen Zweck erfüllt und die Entwicklung nicht in die falsche Richtung ausartet, haben wir wöchentlich eine ausführbare Version des Simulators unserem Betreuer zur Verfügung gestellt, um rechtzeitig ein Feedback zu erhalten. Darüber hinaus haben wir während der Implementierungsphase Rücksprache mit zukünftigen Anwendern und involvierten Personen in der MACK-Entwicklung gesprochen, um sicher zu stellen, dass der Simulator zur Verifikation des Verhaltens des MACK-Frameworks geeignet ist.

## 6 Zusammenfassung und Ausblick

In diesem Kapitel wird kurz aufgezeigt, was in dieser Arbeit erreicht wurde, welche Punkte dabei offen geblieben sind und in welcher Weise man diese in Zukunft angehen könnte.

### 6.1 Zusammenfassung

Wir haben es geschafft, ein Simulationsframework zu entwickeln, das so modular aufgebaut ist, dass sich zum einen verschiedene komplexe Abläufe simulieren lassen und zum anderen die Adaption auf andere Anwendungsgebiete möglich ist. Hiervon versprechen wir uns eine hohe Relevanz, sowohl für das MACK-Projekt, als auch für andere Projekte im Bereich der Context Awareness. Die Veröffentlichung unserer Software unter der LGPL soll dazu beitragen, anderen Projektgruppen die Weiterentwicklung zu ermöglichen. Aus diesem Grund haben wir auch auf die ausführliche Dokumentation in allen Teilen des Projekts geachtet.

Darüber hinaus haben wir eine Simulation entwickelt, mit der das MACK-Framework getestet werden kann. Diese Simulation erfüllt zum einen das ursprüngliche Projektziel, das Testen des MACK-Frameworks zu ermöglichen. Außerdem eignet sich diese Simulation als Beispiel für die Möglichkeiten des Simulators, da sie insbesondere die Funktionalität und Modularität der Software aufzeigt.

Die Methode des benutzerzentrierten Designs war sehr erfolgreich. Durch die vielen, frühzeitigen Rückmeldung der späteren Benutzer konnten wir uns immer schnell an Wünsche und Anregungen anpassen. So haben wir vermieden, dass sich nach der Arbeitsphase am Projekt noch sehr aufwendige Änderungen ergeben. In diversen Treffen untereinander und mit unserem wissenschaftlichen Begleiter haben wir eine strukturierte Arbeitsweise entwickelt. Diese half uns, das Projektziel in einer Art und Weise zu erreichen, die die Entwicklung für uns, als auch für zukünftige Benutzer und Entwickler der Software, besonders angenehm machte und machen wird.

Unsere Evaluation hat ergeben, dass unser System im Wesentlichen funktionsfähig ist und einen guten Ansatzpunkt für weitere Arbeiten an diesem Projekt bietet. Der Ablauf der Simulation wirkt auf uns ausreichend realistisch und sollte sich gut zum Testen des MACK-Frameworks eignen. Die Sensoren und Aktuatoren ändern ihre Werte in Abhängigkeit vom Geschehen und nehmen ebenfalls Einfluss darauf.

## 6.2 Offene Punkte

Im Rahmen unseres Projektes konnten fast alle der ursprünglich geplanten Punkte umgesetzt werden. Einzig die Beschreibungssprache CASiX wurde komplett gestrichen.

### 6.2.1 CASiX

Eine in der Analyse festgestellte Anforderung an unsere Software war die Möglichkeit, Simulationen durch eine Beschreibungssprache zu erzeugen. Im frühen Stadium unserer Arbeit haben wir uns, mit Hilfe der befragten Benutzer, für XML entschieden. Es sollte eine XML-Spezifikation in Form einer XML-Schema-Datei festgelegt werden. Unter dem Namen CASiX wurde ein erster Version spezifiziert.

Im weiteren Entwicklungsprozess wurde jedoch festgestellt, dass die Spezifikation in ihrer vorliegenden Form nicht ansatzweise flexibel genug war, um unseren Anspruch an die gebrauchstaugliche Gestaltung einer Simulation zu genügen. Daraufhin wurde der Schritt wieder zurück zur Spezifikation gegangen. Nach langer Diskussion haben wir uns schlussendlich dafür entschieden, unsere Arbeit ohne die Beschreibungssprache fortzusetzen. Da wir nicht nur in deren Planung, sondern, vor allem, in deren späteren Implementierung einen sehr hohen Zeitaufwand sahen, so dass die Realisierung zu Lasten der Entwicklung des eigentlichen Simulators gefallen wäre.

Ungefähr zur Projektmitte haben wir die Simulationsbeschreibungssprache in den letzten Projekt-Meilenstein verlegt, da die Entwicklung eines XML-Handlers zwar interessant und sinnvoll, aber keinesfalls maßgeblich das Projekt beeinflussend ist.

Wir sind der Meinung, dass die durch diese Entscheidung eingesparte Arbeit in unserem Projekt an anderen Stellen sinnvoller zum Einsatz kam.

Durch den stark modularen Aufbau der Software ist es jedoch für spätere Entwickler besonders einfach, diesen Aspekt noch einmal aufzugreifen und in CASi zu integrieren.

### 6.2.2 Antwortzeiten des MACK-Servers

Ein aktuelles Problem ist, dass bei schneller Simulationsgeschwindigkeit die Antworten des MACK-Servers zu stark verzögert sind. In diesem Fall passen die Sensor- und Aktuatorwerte nicht zum aktuellen Verhalten der Simulation. Diese können sich auf den gesamten Simulationslauf auswirken. Dagegen können wir uns zwei unterschiedlich einflussreiche Verbesserungen vorstellen.

Die Kommunikationskanäle könnten hinsichtlich ihrer Geschwindigkeit optimiert werden. Dies könnte zum Beispiel durch das Verlegen des XMPP-Servers (über den sämtliche Kommunikation läuft) auf die lokale Maschine des Awareness-Hubs geschehen. Somit könnte der Netzwerkverkehr lokal ab-

gewickelt werden, was die Packetlaufzeiten deutlich reduzieren dürfte.

Weit sinnvoller halten wir es jedoch, das MACK-Protokoll zu überarbeiten. Wenn dieses dahingehend erweitert wird, dass Zeitstempel und/oder andere Identifikationsmerkmale übermittelt werden, könnten sich der Awareness-Hub und der Simulator unabhängig von ihrer Geschwindigkeit austauschen. Reasoner für die zeitliche Aspekte wichtig sind, wären in der Lage, die erhöht Geschwindigkeit der Simulation zu verstehen und sich darauf anzupassen. Als Vorarbeit zu diesem Ansatz senden unsere Komponenten bereits in den Push-Nachrichten Zeitstempel mit.

### 6.3 Ausblick

Aspekte unserer Arbeit an die gut in der Zukunft angeknüpft werden kann, sind prinzipiell in zwei Teilbereiche aufzuteilen. Eine Weiterentwicklungsmöglichkeit ist es eigene, neue Simulationen zu entwerfen. Die dazu benötigten anderen Sensoren, Aktuatoren oder Aktionen sind unkompliziert zu erstellen. Man könnte sich an das, in dieser Dokumentation beschriebene, Vorgehen halten, oder sich an unserer Demonstrationssimulation orientieren.

Zum anderen sind auch für den CASi-Kern Erweiterungen vorstellbar. Dazu zählen insbesondere die Entwicklung einer umfangreicheren graphischen Schnittstelle, die erweiterte Interaktionsmöglichkeiten bereitstellt. Weitergehen könnten wir uns vorstellen, Simulationen interaktiv zu gestalten. Denkbar wäre eine Simulation in einer virtuellen Realität, in der die Benutzer die Agenten direkt steuern können. Dafür könnten Netzwerk-Mehrspieler-Spiele genutzt werden. Beispielsweise Minecraft web (2012) bietet dafür eine in Java programmierte quelloffene Serverkomponente an, welche sich durch geeignete Schnittstellen erweitern lassen könnte.



# Abbildungen

Abbildung 1: Unterteilung der Simulatorarchitektur in Module . . . . .	10
Abbildung 2: abgewandeltes UML-Klassendiagramm zur Darstellung der Aufteilung eines Generators. <i>Blau gestrichelt</i> : Aufruf statischer Methoden, <i>grün gestrichelt</i> : Einfügen von Objekten, <i>grün</i> : Verknüpfung von Objekten untereinander, <i>rot</i> : Entgegennehmen von gesammelten Objekten . . . . .	18
Abbildung 3: Screenshot der Anwendung . . . . .	20

## Quelltexte

Quelltext 1:	Beispiel für das Hinzufügen eines Listeners an die <code>SimulationClock</code> . . .	17
Quelltext 2:	Beispiel für das Entfernen eines Listeners von der <code>SimulationClock</code> . . . .	17
Quelltext 5:	Beispiel für das Hinzufügen interessanter Aktionstypen aus dem Konstruktor der Klasse <code>Mike</code> . . . . .	27
Quelltext 3:	Beispiel für eine XML Netzwerkkonfigurationsdatei . . . . .	30
Quelltext 4:	Beispiel zur Registrierung einer kommunizierenden Komponente . . . . .	30

# Quellen

## Literatur

(2012). Minecraft. URL <http://www.minecraft.net/>.

Frahm, O.-J. (2011). *MATe-Kommunikationsframework zur Einbindung multipler Reasoner*. Diplomarbeit, Universität zu Lübeck.

Kindereit, M., Momsen, S., Röhr, K. & Weiss, T. (2011). *MATe: Sensoren und Aktuatoren*. Technischer Bericht, Institut für Multimediale und Interaktive Systeme, Universität zu Lübeck.

Martin, M. & Nurmi, P. (2006). A Generic Large Scale Simulator for Ubiquitous Computing. In *Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, 2006 (MobiQuitous 2006)*. San Jose, California, USA: IEEE Computer Society.

Nguyen, T. V., Nguyen, H. A. & Choi, D. (2010). *Development of a Context Aware Virtual Smart Home Simulator*. Technischer Bericht, Department of Computer Engineering, Chonnam National University, Korea.

Piirainen, T., Niskanen, I., Kantorovitch, J. & Kalaoja, J. (2007). *Context Simulation and Support for Context-Aware Application Development*. Technischer Bericht, VTT Technical Research Centre of Finland.

# Anhänge

Folgend sind die Projektstruktur (Abschnitt A) inklusive der konzeptionellen Aufteilung einzelner Arbeitsteilgebiete in verschiedene Ordner und Pakete und eine Hilfestellung zum Ausführen des Simulationsprogramms (Abschnitt B) gegeben.

*Umfangreiche zusätzliche Informationen, die im Textverlauf stören würden, aber für die Arbeit wichtig sind, wie z.B. Programmcode, Fragebögen, Evaluationstabellen*

## A Projektstruktur

Um die Weiterentwicklung des Projektes zu erleichtern, befindet sich in diesem Abschnitt eine ausführliche Erklärung zur Struktur des Projektes. Dies beinhaltet zum einen die Paket- und Ordnerstruktur, aber auch Hinweise auf Funktionalitäten, die die Entwicklung erleichtern.

### A.1 Ordnerstruktur

Das Projekt ist in verschiedene Ordner unterteilt, deren Sinn nachfolgend erläutert wird:

**src** In diesem Ordner befinden sich die Beschreibungen und Implementierungen des Simulators und der Simulationen.

**test** Dieser Ordner enthält die JUnit-Testfälle für einige Klassen im `src`-Ordner. Hierbei entspricht die interne Paket-Struktur dieses Ordners der Struktur des `src`-Verzeichnisses.

**log** Logfiles, die während der Ausführung des Programms generiert werden, werden mit einem Zeitstempel in diesem Ordner abgelegt.

**doc** Die JavaDoc-Dokumentation des Simulators befindet sich in diesem Verzeichnis.

**dist** Im Distributionsverzeichnis finden sich ausführbare Versionen der Anwendungen als jar-Datei.

**sims** Beschreibungen für Simulationen, wie z.B. textuelle Erklärungen zu einzelnen Simulationen und die Netzwerk-Konfigurationsdateien befinden sich in diesem Ordner.

**bin** Dieser Ordner enthält die Binaries des Simulators.

**lib** In diesem Verzeichnis befinden sich externe Bibliotheken, die für die Arbeit am Projekt oder für Komponenten des Simulators notwendig sind.

**reports** Dieses Verzeichnis enthält Reports über die Durchführung der Testfälle.

## A.2 Paketstruktur

Alle Klassen, die zum Simulator gehören, befinden sich in Unterpaketen des Paketes `de.uniluebeck.imis.casi` im Ordner `src`. Nachfolgend wird der Sinn der verschiedenen Pakete erläutert:

**engine** Dieses Paket enthält die Basis des Simulators, die für die Koordination der Ausführung und die Verknüpfung der Komponenten sorgt. Hierzu zählen insbesondere die `SimulationEngine` und die `SimulationClock`.

**communication** In diesem Paket befinden sich die Schnittstellenbeschreibungen für Kommunikationshandler. In Unterpaketen befinden sich konkrete Implementierungen. So enthält `communication.mack` die Implementierung des `MACKNetworkHandlers`.

**generator** Das `generator`-Paket beinhaltet die Beschreibung der Schnittstellen, die von `World-Generatoren` implementiert werden müssen.

**ui** In diesem Paket befinden sich die Beschreibungen für Benutzungsschnittstellen. Unterpakete enthalten die Implementierungen verschiedener Schnittstellen. Das Paket `ui.simplegui` enthält die einfache graphische Benutzungsoberfläche.

**utils** Im `utils`-Paket befinden sich Klassen, die einen hohen Abstraktionsgrad aufweisen und somit leicht in andere Projekte portiert werden können, da sie größtenteils vom Simulator unabhängig sind.

**simulations** Dieses Paket kapselt in weiteren Unterpaketen die Beschreibungen einzelner Simulationen.

**simulation.model** Die Anwendungslogik befindet sich in diesem Paket.

**controller** Die Controller zur Verknüpfung der Komponenten und zur Steuerung des Programmflusses befinden sich in diesem Paket.

**logging** Dieses Paket beinhaltet Konfigurationsklassen für die Logger.

## A.3 Buildfile

Das Projektverzeichnis enthält das Ant-Buildfile `build.xml`, welches eine Reihe von Buildtargets definiert. Diese Ziele können entweder über die Konsole mit `ant <target-name>` oder aus Eclipse heraus ausgeführt werden. Nachfolgend befindet sich eine Beschreibung der wichtigsten Targets, eine vollständige Beschreibung aller Targets erhält man durch Ausführen des Befehls `ant -p` im

Projektverzeichnis:

**clean** räumt das Projektverzeichnis auf und löscht die Binäre-, Dokumentations- und Reportverzeichnisse.

**clean-all** führt das Target `clean` aus und löscht außerdem das Log- und Distributionsverzeichnis.

**test** führt die Testfälle aus.

**test** führt die Testfälle aus und erzeugt HTML-Reports im `reports`-Verzeichnis.

**doc** erzeugt die JavaDoc-Dokumentation

**jar** erzeugt ein ausführbares jar-File des Simulators (`dist/CASi.jar`)

**xmpp-registrator-jar** erstellt eine ausführbare Version des `XmppRegistrators`.

**main** Erstellt alle Distributionen und die JavaDoc-Dokumentation.

## B Ausführungsanweisungen

Da es sich bei dem Projekt um ein Java-Projekt handelt, gibt es keine Installation im herkömmlichen Sinne. In diesem Abschnitt wird beschrieben, wie die Programme ausgeführt werden können. Dieser Abschnitt beinhaltet Informationen zum Umgang mit dem Simulator und dem zuvor bereits beschriebenen `XmppRegistrator`.

### B.1 Simulator

Die jar-Version des Simulators kann, nachdem sie mit `ant jar` erzeugt wurde, mit `java -jar dist/CASi.jar <parameter>` ausgeführt werden. Eine ausführliche Dokumentation der möglichen Parameter erhält man mit `java -jar dist/CASi.jar --help`.

### B.2 XmppRegistrator

Der `XmppRegistrator` kann automatisch alle für die Simulation benötigten `XmppIdentifier` am XMPP-Server registrieren. Dies ist besonders hilfreich, wenn der Server ein Delay zwischen zwei Registrierungsoperationen benötigt, da somit vermieden werden kann, dass die Identifier erst beim Start der Simulation registriert werden und somit die Ausführung der Simulation verzögert wird.

Der `XmppRegistrator` verlangt als einzigen Parameter den Pfad zu einer Netzwerkkonfigurationsdatei und kann, nachdem er mit `ant xmpp-registrator-jar` erzeugt wurde, mit `java -jar dist/XmppRegistrator.jar <path-to-config-file>` ausgeführt werden.

# Erklärung

Wir versichern, die vorliegende Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Lübeck, den 9. Februar 2012

---

Moritz Bürger

---

Marvin Frick

---

Tobias Mende