CORE

Metadata, citation and similar papers at core.ac.uk

Provided by Hochschulschriftenserver der HTWG Konstanz

Entwurf einer auf XML basierenden Beschreibungsspache für Benutzerschnittstellen im Kontext von Mobile-Agenten-Systemen

Axel Baldauf

Konstanz, 8. April 2003

DIPLOMARBEIT

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Fachhochschule Konstanz

Hochschule für Technik, Wirtschaft und Gestaltung

Fachbereich Informatik/Technische Informatik

Thema : Entwurf einer auf XML basierenden Beschreibungsspache für Benutzerschnittstellen im Kontext von Mobile-Agenten-Systemen

Diplomand : Axel Baldauf
Handelisgartenweg 2
78462 Konstanz

Firma : Institut für graphische Datenverarbeitung (IGD)
der Fraunhofer Gesellschaft
Abteilung A8
Fraunhoferstrasse 5
D - 64283 Darmstadt

Betreuer : Prof. Dr. Oliver Bittel
Fachbereich Informatik, Fachhochschule Konstanz

Dipl.-Inform. Jan Peters
IGD Darmstadt

Eingereicht : Konstanz, 8. April 2003

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Axel Baldauf, geboren am 15. Januar 1967 in Singen/Htwl., ehrenwörtlich,

(1) dass ich meine Diplomarbeit mit dem Titel:

„Entwurf einer auf XML basierenden Beschreibungsspache für Benutzerschnittstellen im Kontext von Mobile-Agenten-Systemen

am Institut für graphische Datenverarbeitung (IGD)
der Fraunhofer Gesellschaft in Darmstadt unter
Anleitung von Professor Dr. Oliver Bittel selbständig
und ohne fremde Hilfe angefertigt habe und keine anderen als
in der Abhandlung angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate aus der Literatur
sowie die Verwendung der Gedanken anderer Autoren an den
entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 8. April 2003

Abstract

Mobile-Agenten erweitern die Möglichkeiten von verteilten Systemen dadurch, dass sie lauffähigen Code transportieren und diesen ausführen können. Sie können sich frei im Netzwerk bewegen, dort nach Informationen suchen und Aufgaben im Namen ihrer Auftraggeber ausführen.

Viele Mobile-Agenten-Plattformen verwenden Java als Laufzeitumgebung, dadurch sind sie unabhängig vor dem darunterliegenden Betriebssystem. Auf eine Unterstützung einer Anzeigegeräte-unabhängigen Benutzer-Agent-Interaktion wird aber meist nicht geachtet.

Betrachtet man sich diese Anzeigegeräte mit den unterschiedlichen Eigenschaften in Display-Größe, Farbfähigkeit und Möglichkeiten für die Ein-und Ausgabe, besonders im Zusammenhang mit mobilen Geräten, z.B. Smartphpones und PDAs, ist es verständlich warum der Benutzer oft noch auf herkömmliche Geräte zur Interaktion angewiesen ist.

Der Fokus dieser Diplomarbeit lag auf der Entwicklung eines Frameworks, das graphische Java-Benutzerschnittstellen in einem geräteunabhängigen XML-Format beschreibt und daraus ein geräteabhängiges Format erzeugt. Ein besonderes Augenmerk lag dabei in der einfachen Erweiterbarkeit des Frameworks, um zukünftige Ausgabeformate zu unterstützen, weswegen die Technik XSLT zum Einsatz kam. Nach dem Übermitteln der GUI-Daten werden über einen Rückkanal die Benutzerinteraktionen wieder dem Framewok zugeführt.

Da die meisten mobilen Geräte einen Webbrowser integriert haben, eignen sich die Markup-Sprachen HTML oder WML besonders gut als Ausgabeformate für diese Geräte.

Das Framework erlaubt darüberhinaus sowohl die lokale als auch die entfernte Interaktion mit den Software-Agenten. Sowohl für den Transport der GUI-Daten als auch für die Interaktion mit Agenten verwendet das Framework die Sicherheitsmechanismen der Agenten-Plattform SeMoA (Secure Mobile Agents).

Das vorgestellte Framework lässt sich in SeMoA einfach integrieren und gestattet die individuelle Konfiguration der Interaktionsmethode mit einem Agenten.

Aufgabenstellung für die Diplomarbeit des
Herrn cand.-Inform. Axel Baldauf
Matrikel-Nr. 265 225

Thema: "Entwurf einer auf XML basierenden Beschreibungssprache für Benutzerschnittstellen im Kontext von Mobile-Agenten-Systemen"

Am Fraunhofer IGD wird im Projekt Secure-Mobile-Agents (SeMoA) an einer sicherheitsorientierten Plattform für mobile Agenten geforscht. Ein grundlegender, jedoch oft unbeachteter Bereich in diesem Forschungszweig ist die Bereitstellung eines einheitlichen Rahmens für die Interaktion zwischen Benutzern und Agenten.

Eine wichtige Eigenschaft der Agenten stellt dabei deren Mobilität, also die Migrationsfähigkeit von Programm, Daten und aktuellem Zustand von einem Wirtssystem auf ein anderes, dar. Durch die Verwendung der Programmiersprache Java^{TM} wird Plattformunabhängigkeit bezüglich der Ausführung von mobilem Code bereits gewährleistet. Begründet durch unterschiedliche Display-Größen, Farb-/Graphikfähigkeit bzw. der Ein- und Ausgabemöglichkeiten der Wirtssysteme im Allgemeinen ist eine plattformunabhängige Interaktionsumgebung zwischen Benutzer und Agent durch herkömmliche graphische Benutzerschnittstellen allerdings nicht immer gegeben. Besonders bei der Betrachtung von Szenarien mit mobilen Endgeräten (modernen Funktelefonen und PDAs) wird dies deutlich.

Im Rahmen der Diplomarbeit soll aus diesem Grund ein Framework entwickelt und implementiert werden, welches die Transformation einer graphischen Benutzerschnittstelle in eine plattformunabhängige, XML-basierte Beschreibungssprache realisiert. Dazu gehören auch entsprechende Module, welche die generische Beschreibungssprache wieder in konkrete, gerätespezifische Ausgabeformate konvertieren und einen Rückkanal für die Benachrichtigung über Benutzeraktionen zur Verfügung stellen.

Dieses Framework soll darüber hinaus eine Kapselung der Interaktionsumgebungen aller auf dem Wirtssystem befindlichen Agenten in einem einheitlichen Rahmen ermöglichen. Die Nutzung der Interaktionsumgebung seitens der Agenten, analog dem Zugriff auf eine Ressource des Wirtssystems, soll durch das Framework kontrolliert werden können.

Im theoretischen Teil der Arbeit sollen folgende Aspekte berücksichtigt werden:

Literaturrecherche Es soll der State-of-the-Art in den Bereichen (graphische) Benutzerschnittstellen, Dialogbeschreibung sowie Generierung, Syntaxanalyse und Transformation von XML und XSLT zusammengetragen werden.

Generelle Probleme Es sind die generellen Schwierigkeiten aufzuzeigen, die sich durch herkömmliche Verfahren (z.B. Nutzung von Java^{TM}-Swing-GUIs) für die Interaktion mit

mobilem Code ergeben.

Grundlegende Interaktionskomponenten Eine Analyse der wesentlichen, zur Interaktion benötigten Basiselemente bei graphischen Benutzerschnittstellen und der Fragestellung, inwieweit Dialogentwurf und graphische Gestaltung entkoppelt werden kann bzw. soll.

Evaluierung Es ist zu prüfen, ob und inwiefern Performance-Einbußen durch den Zwischenschritt der Benutzerschnittstellen-Transformation, besonders im Szenario mit mobilen Endgeräten, durch die sich ergebenden Vorteile vertretbar sind und wo sich diese bemerkbar machen.

Related Work Existierende Beschreibungssprachen für Benutzerschnittstellen und Realisierung von Benutzer/Agenten-Interaktion in anderen Plattformen für mobile Agenten.

Die zu entwerfende Architektur soll in Form eigenständiger Module realisiert werden, welche dann als sogenannte Dienste in die SeMoA-Architektur einzubinden sind. Dabei soll vor allem auf die flexible Erweiterbarkeit in Bezug auf die unterstützten Ausgabeformate Rücksicht genommen werden. Die implizite Nutzung des entwickelten Frameworks soll für den Agentenentwickler möglichst transparent geschehen.

Unter der Annahme, dass sich als Ausgabeformate der konvertierten Benutzerschnittstellen unter anderem die weit verbreiteten, sogenannten Markup Languages HTML, WML oder VoiceXML eignen, welche in entsprechenden Clients (z.B. einem Webbrowser) dargestellt werden, ergibt sich neben dem direkten Zugriff auf die Interaktionsumgebung eines Agenten auf dem Wirtssystem auch die Möglichkeit des Zugriffs von entfernten Rechnern aus. In diesem Fall ist es sinnvoll, sich auch Gedanken über die Sicherheitsimplikationen in Bezug auf die transportierten Daten zu machen.

Darmstadt, den 01.10.02

Betreuer:
Dipl.-Inform. Jan Peters

Danksagung

Danken möchte ich an dieser Stelle meinen Eltern, die immer für mich da waren wenn ich sie am dringendsten gebraucht habe, meiner Freundin Karin, für ihre starke Unterstützung und ihr Verständnis während der Diplomarbeit. Dank gebührt aber auch meinen Professoren an der Fachhochschule Konstanz, die mir das notwendige Wissen vermittelt haben, im besonderen meinem Prüfer Dr. Oliver Bittel, für die freundliche Unterstützung im Rahmen der Diplomarbeit. Dank auch an die Mitarbeiter des CRCG in Providence, USA und an die Mitarbeiter am IGD in Darmstadt, im besonderen Dipl.-Inform. Jan Peters, ohne dessen Einsatz der mit der Diplomarbeit verbundene USA Aufenthalt am CRCG nicht möglich geworden wäre, und der die Diplomarbeit durch seine fachliche Kompetenz entscheidend beeinflusst hat.

Axel Baldauf Konstanz, April 2003

Inhaltsverzeichnis

I Grundlagen **1**

1 Einleitung **2**

1.1 Motivation 2

1.2 Zielsetzung 2

1.3 Überblick 3

1.4 Konventionen 3

2 Agententechnologie **4**

2.1 Software-Agenten 4

2.2 Multiagentensysteme 5

2.3 Mobile Agenten 6

3 Graphische Benutzerschnittstellen **8**

3.1 Entstehung 8

3.2 JAVA GUI 9

3.3 HTML 11

4 Vergleichbare Arbeiten **13**

4.1 X-Window-System 13

4.2 XML-Talk 14

4.3 Rapid-GUI-Development für das Web 15

4.4 Plattformunabhängige Benutzerschnittstellen 15
4.5 Sonstige 15
4.6 Bewertung 16

5 XML und XSLT **17**
5.1 XML 17
5.2 XSLT 19
5.3 XSLT-Grundlagen 19
5.4 XSLT-Techniken 20

6 SeMoA **25**
6.1 Intention von SeMoA 25
6.2 Systemarchitektur 26
6.3 Sicherheitsmechanismen 27

II Systementwicklung **29**

7 Problemdefinition **30**
7.1 Anforderungen 30
7.2 Einsatzbereich 30

8 Lösungsansatz **32**
8.1 Darstellung 32
8.2 Transformation 33
8.3 Kontrolle und Verwaltung 33
8.4 Transport und Weiterleitung 34
8.5 Systemarchitektur 34
8.6 Vorabbewertung des Lösungsansatzes 35

9 Analysemodell **37**
9.1 Use-Case-Diagramme 37
9.2 Klassendiagramme 40

10 Implementierung **45**

10.1 Prototyp 45

10.2 Aufbau des Prototyps 45

10.3 Integration in SeMoA 49

III Resümee **52**

11 Evaluation und Ausblick **53**

11.1 Funktionsweise des Frameworks 53

11.2 Kurze Evaluation 55

11.3 Ausblick 56

11.4 Schlussbemerkung 57

IV Anhang **58**

A Protokollspezifikation **59**

A.1 Protokollspezifikation 59

A.2 Übersicht der Agent-GUI-Klassen 62

B Initialisierungsdateien für SeMoA **63**

B.1 demo1.conf 63

B.2 rc.network 63

B.3 whatis.conf 67

C XSLT-Stylesheets **70**

C.1 global.xslt 70

C.2 allhosts_ms.xslt 72

C.3 allagents_ms.xslt 74

C.4 html_agent_gui_ms.xslt 76

D Beispiel-GUI mit XML-Datei **84**

D.1 Java^{TM}-GUI und HTML-GUI 84

D.2 Beispiel der dazugehörigen GUI-Beschreibung in XML 85

E Akronyme **87**

Tabellenverzeichnis

1.1 UML Zeichenerklärung 3

5.1 Übersicht der Achsen 21

5.2 Abkürzung der Achsen 22

9.1 Zuordnung der *package* Namen 41

A.1 Bisherige Agent-GUI-Klassen 62

Abbildungsverzeichnis

4.1 X-Server und Client auf unterschiedlichen Rechnern 14

5.1 SGML Stammbaum 18

5.2 Beispiel für ein XML-Dokument 18

5.3 Schematik einer XSLT-Transformation 19

5.4 XSLT-Stylesheet Beispieldatei 21

6.1 Sicherheitskonzept von SeMoA 28

8.1 Systemarchitektur 35

9.1 Use Case Diagramm vom Agenten-Dialog unter Verwendung von Java 37

9.2 Use Case Diagramm vom Übersetzten des Agenten-Dialogs 38

9.3 Use Case Diagramm vom Übersetzten und Weiterleiten des Agenten-Dialogs . 38

9.4 Detailliertes Use Case Diagramm vom Weiterleiten des Agenten-Dialogs . . . 39

9.5 Detailliertes Use Case Diagramm vom Übersetzten des Agenten-Dialogs 39

9.6 Detailliertes Use-Case-Diagramm vom Weiterleiten der Benutzereingaben . . . 40

9.7 Klassendiagramm der `agentgui` Interfaces 41

9.8 Klassendiagramm der `control` Interfaces 42

9.9 Klassendiagramm der `guiExchange` Interfaces 44

9.10 Klassendiagramm aller Interfaces 44

10.1 Klassen aus dem Packet `agentgui` 46

10.2 Klassen aus dem Packet `control` 47
10.3 Klassen aus dem Packet `guiExchange` 48
10.4 Klassen aus dem Packet `servlet` 49
10.5 Integration als Dienst in SeMoA 50
10.6 Alle Klassen des Frameworks 51

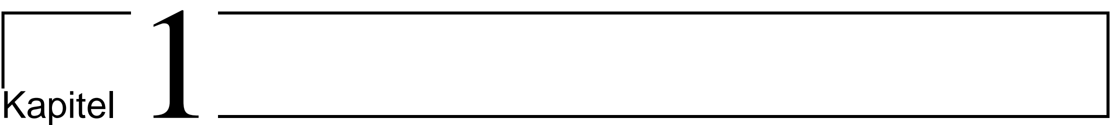
11.1 Browserdarstellung aller SeMoA-Hosts 53
11.2 Browserdarstellung aller SeMoA-Agenten eines Hosts 54
11.3 Browserdarstellung einer Agent-GUI mit Checkboxen 54
11.4 Java^{TM}-Darstellung einer Agent-GUI mit Checkboxen 55
11.5 Browserdarstellung der `JumpingAgent`-GUI 55
11.6 Java^{TM}-Darstellung der `JumpingAgent`-GUI 55

A.1 Sequenz-Diagramm der `guiExchange.protocol` Client-Klassen 59
A.2 Sequenz-Diagramm der `guiExchange.protocol` Server-Klassen 60
A.3 Sequenz-Diagramm der Server-Klassen: Sammeln der Methoden Argumente . 61

D.1 Java^{TM}-Darstellung einer Agent-GUI mit Radiobuttons 84
D.2 Browserdarstellung einer Agent-GUI mit Radiobuttons 84

Teil I

Grundlagen



Einleitung

1.1 Motivation

Das Internet ist mittlerweile zur festen Einrichtung geworden, das WWW zum Informationslieferanten. War es anfangs nur der Computer, so sind es heute eine Vielzahl diverser Geräte, die sich des Internetzugangs bedienen, hierzu zählen unter anderem die folgenden:

PC, Laptop, Palm PC, Mobiltelefon, Spielekonsole, Unterhaltungselektronik und neuerdings auch diverse Haushaltsgeräte.

Diese Entwicklung beruht auf dem Bedürfnis des Anwenders, möglichst uneingeschränkt von Ort und Zeit auf Informationen aus dem Internet zugreifen zu können. Dieser Trend ist längst Programm in der Industrie und wird auch in Zukunft zu immer neueren Produkten führen.

Das Forschungsprojekt SeMoA am Institut für graphische Datenverarbeitung in Darmstadt (IGD), folgt diesem Trend. Es stellt eine einheitliche Schnittstelle für die Interaktion oben genannter Geräte mit seiner Agentenplattform zur Verfügung und ermöglicht dadurch die entfernte Kommunikation mit seinen Agenten.

1.2 Zielsetzung

Jede graphische Benutzerinteraktion mit einem SeMoA-Agenten setzt das Vorhandensein einer SeMoA- und damit verbunden einer Java^{TM}- Installation voraus. Ziel der Diplomarbeit ist es, SeMoA um die Funktionalität zu erweitern, eine Interaktion mit Agenten von möglichst vielen internettauglichen Geräten aus zu ermöglichen, auf denen selbst nicht zwingend SeMoA , Java^{TM}oder eine sonstige Erweiterung installiert sein muss. Es ist eine Lösung angestrebt, bei der der Agentenprogrammierer Java^{TM}-GUI[1] Komponenten wie gewohnt verwendet und Be-

[1]GUI: Graphic User Interface

stehendes ohne Änderung integrieren kann. Im Idealfall sollen diese GUI-Komponenten identisch auf den Ausgabegeräten dargestellt werden, unter Berücksichtigung der Display-Eigenschaften.

1.3 Überblick

Der erste Teil der Arbeit vermittelt die theoretischen Grundlagen, die zur Lösungssuche und Lösungsfindung Verwendung fanden. Der zweite Teil beschreibt den Prozess der Systementwicklung, beginnend mit der Problemdefinition bis hin zur Implementierung. Der dritte Teil, das Resümee, bildet den Abschluss der Arbeit.

1.4 Konventionen

Klassennamen, Datentypen und Quelltextauszüge sind in `Schreibmaschinen Schrift`, Schlüsselwörter und Begriffsdefinitionen in *kursiver Schrift* dargestellt.

Abkürzungen und ihre Langfassung finden sich im Anhang E.

Gebräuchliche englische Begriffe aus der Informatik werden ohne weitere Erklärung verwendet, alle anderen werden, sofern eine Übersetzung möglich ist, ins Deutsche übersetzt.

Es gelten die Regeln der neuen deutschen Rechtschreibung.

Die Verwendung folgender Symbole ist in *Unified Modeling Language* Diagrammen (UML-Diagrammen) nicht immer einheitlich. Für diese Arbeit gelten deshalb im Kontext von Java^{TM}-Klassendiagrammen die nachfolgenden Erklärungen. Die Symbole sind hierbei immer als zwischen zwei Klassen stehend zu betrachten (links vom Symbol steht Klasse A, rechts davon B):

Symbol	Name	Erklärung
——>	association	Klasse A verwendet Klasse B
- - - -▷	realization	Klasse A implementiert das Interface B
——▷	generalization	Klasse A erbt von B; B ist Oberklasse von A
◇——	aggregation	Klasse B ist ein Teil von A, A verwendet B
◆——	composition	Klasse A erzeugt eine oder mehrere Instanzen von B

Tabelle 1.1: UML Zeichenerklärung

Agententechnologie

Dieses Kapitel gibt einen Überblick über die Begriffe *Software-Agenten, Multiagentensysteme* und *Mobile Agenten*.

2.1 Software-Agenten

Der Begriff Agent stammt vom lateinischen Wort *agere* (handeln, wirken) ab. Er bezeichnet im alltäglichen Gebrauch jemanden, der im Auftrag eines anderen dessen Interessen vertritt und in dessen Namen agiert. Der Agent verfügt dabei meist über besondere Fähigkeiten oder Kontakte, über die der Auftraggeber nicht verfügt. Menschliche Agenten finden sich in den unterschiedlichsten Bereichen wie Sport, Spionage, Finanzwesen oder Unterhaltung. Die Informatik hat den Begriff des Agenten übernommen und setzt ihn in vielfältiger Weise ein.

Es gibt jedoch keine allgemein gültige Definition für Software-Agenten. Die Entwickler erster Software-Agenten formulierten deshalb unterschiedliche Eigenschaften, die Software-Agenten von anderem Programmcode unterscheiden.

Agenten mit der Eigenschaft ...

- *Autonomie:* entscheiden selbst wann sie handeln.
- *Reaktivität:* reagieren auf ihre Umwelt.
- *Anpassungsfähigkeit:* passen sich ihrer Umgebung an.
- *Kommunikationsfähigkeit:* können sich mit anderen Agenten oder dem Auftraggeber austauschen.
- *Mobilität:* bewegen sich frei in ihrer Umgebung.

- *Transparenz:* geben ihrem Auftraggeber Auskunft über ihr Handeln.
- *Robustheit:* haben eine hohe Fehlertoleranz und lösen Probleme weitestgehend selbst.
- *Kontinuität:* beharren darauf ihren Auftrag zu erledigen.

Die Autoren Richard Murch und Tony Johnson haben in ihrem Buch *Agententechnologie: Die Einführung* [13] diese Sammlung an Eigenschaften aus Ermangelung einer eindeutigen Definition in der Informatik aufgestellt. Obige Eigenschaften gelten für die Summe aller Agenten. Je nach Verteilung der Eigenschaften lassen sich Agenten folgenden Kategorien zuordnen.

- *Intelligente Agenten:* zeichnen sich vor allem durch *Reaktivität*, *Anpassungsfähigkeit* und *Robustheit* aus.
- *Mobile Agenten:* verfügen unter anderem über die Eigenschaft *Mobilität*, siehe auch Abschnitt 2.3.
- *Lernende Agenten:* oder auch *adaptive Agenten* sind Agenten, die das Verhalten und die Vorlieben von Benutzern erfassen und ihrer Aufgabe entsprechend verwenden.

2.2 Multiagentensysteme

Ein Multiagentensystem besteht aus folgenden Elementen:

- *Umwelt:* Raum für die folgenden Elemente.
- *Menge von Objekten:* Agenten können auf andere Objekte Operationen anwenden.
- *Menge von Agenten:* Die aktiven Objekte der Umwelt.
- *Menge von Beziehungen:* Eine Beziehung verbindet Objekte miteinander.
- *Menge von Operationen:* z.B: Wahrnehmen, Konsumieren, Empfangen, Erzeugen, Modifizieren, Löschen.
- *Operatoren:* Regelwerk für die Anwendung der Operationen innerhalb der Umwelt.

Gilt für das System, dass die Anzahl der Agenten gleich der Anzahl von Objekten ist, handelt es sich um ein rein kommunizierendes Multiagentensystem, wie es in der Verteilten Künstlichen Intelligenz[1] häufig Verwendung findet.

Zu den Anwendungsgebieten der Multiagentensysteme hat Jacques Ferber in seinem Buch *Multiagenten-Systeme, Eine Einführung in die Verteilte Künstliche Intelligenz* [6] folgende Unterteilung vorgeschlagen:

[1] Teilgebiet der Künstlichen Intelligenz [5]

- *Das Lösen von Problemen:* Hierunter fallen drei Situationen für Software-Agenten: 1. Das Wissen kann entweder unter den Agenten verteilt sein. 2. Das Problem selbst ist verteilt. 3. Wenn weder das Wissen noch das Problem verteilt ist nutzen Agenten zur Problemlösung ihre Interaktionsfähigkeiten.

- *Multiagentensimulation:* Theoretische Modelle werden simuliert, um Vorhersagen oder Erklärungen für reale Ereignisse zu treffen.

- *Erschaffung künstlicher Welten:* Simulation spezieller Verhaltensweisen von Agenten in oft vereinfachten Abbildungen der realen Welt, zum Beispiel Agenten jagen gemeinsam die Beute.

- *Roboterkollektive:* Robotergruppen, die in der realen Welt zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen. Beispielsweise können die Gelenke eines Schweißroboters eigenständige Agenten sein, die sich gegenseitig zur Ausführung einer notwendigen Bewegung einbeziehen.

- *Programm Design:* Die Umsetzung jeder Komponente eines Programms in die Form eines Agenten, um so Daten zu sammeln, Dienste anzubieten, Dienste anzufordern, etc.

Zur letzten Gruppe zählt Jacques Ferber auch solche Agenten, die sich in einem Netz bewegen. Ihnen ist der folgende Abschnitt gewidmet.

2.3 Mobile Agenten

Ein mobiler Agent ist ein Agent, der im Auftrag eines Benutzers sich selbständig im Netzverbund zwischen Hostrechnern zur Erledigung seiner Aufgaben bewegt. Dazu kann er mit anderen Agenten sowie mit seinem Auftraggeber kommunizieren. Der Agent nutzt auf dem jeweiligen Wirtssystem, die von ihm benötigten freigestellten Dienste und kann mit anderen Agenten kooperieren. Zur Migration[2] von einem Rechner zum anderen wird der dynamische Prozesszustand des Agenten eingefroren und samt persistenter Daten an den Zielhost versendet. Dort wird der alte Zustand wieder hergestellt und der Agent kann seinen Dienst fortsetzen.

Mobile Agentensysteme haben gegenüber klassischen Client-Server Modellen, die *Remote Method Invocation* (RMI) Aufrufe verwenden, folgende Vorteile:

- *Eingesparte Bandbreite:* Üblicherweise werden Daten zur Prozedur gesendet. Im Falle von großen Datenmengen kann Bandbreite eingespart werden, wenn der Agent mitsamt Prozedur zu den Daten gesendet wird, um diese dort zu verarbeiten.

[2] Wechseln des Standorts

- *Verbesserte Reaktion:* Ein Agent vor Ort, beispielsweise in einer Echtzeitanwendung, kann schneller auf Veränderungen reagieren als dies durch entfernte Kontrollinstanzen möglich wäre.
- *Offline Verbindung:* Außer zur Migration auf den Zielhost braucht der Agent keine dauerhafte Verbindung. Dies ist besonders bei mobilen Geräten vorteilhaft.
- *Lokale Nutzbarkeit:* Sind gewisse Dienste oder Ressourcen aus verschiedenen Gründen nur lokal nutzbar, kann der Agent dazu dienen, diese nach außen verfügbar zu machen.

Die Idee des *mobilen Codes* ist nicht neu und wurde bereits zur Ansteuerung von Druckern, bekannt unter dem Namen *PostScript* verwendet. Neuer ist die Idee der *Applets*: hier wird Java^{TM}-Code versendet, der in sogenannten Applet-Viewern, zumeist integriert in Webbrowsern, ausgeführt wird.

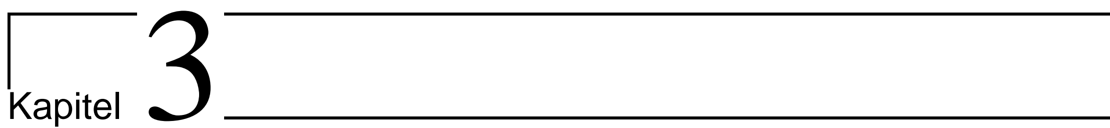
Der Unterschied zwischen mobilem Code und mobilen Agentensystemen wird durch folgende zusätzliche Anforderungen an mobile Agentensysteme sichtbar:

- *Sicherheit:* Da das Ausführen von fremden Code eine große Gefahr für ein Computersystem bedeutet (Beispiel Computerviren), sind Sicherheitsmechanismen nötig, die es erlauben, Agenten zu identifizieren und individuelle Rechte zur Verwendung von Diensten und Ressourcen an sie zu vergeben. Darüberhinaus sind auch Sicherheitsmechanismen notwendig, die Agenten vor dem Ausspähen und Manipulieren ihrer Daten durch bösartige Agenten-Hosts (Malicious Hosts) schützen.
- *Kontextinformation:* Der Agent muss über eine Struktur verfügen, die es ermöglicht, seinen aktuellen Zustand (Befehlszähler, Speicherbereiche) zur Migration zu speichern (einzufrieren), um ihn an anderer Stelle wiederherzustellen. Der jeweilige Zielhost des Agenten muss zumindest über die Möglichkeiten zum Einfrieren sowie zum Auftauen verfügen.

Zu den möglichen Anwendungsbereichen mobiler Agenten zählen die Informationsbeschaffung, etwa in Form von Datenbankrecherchen, und der in Abschnitt 2.2 erwähnte Einsatz zur gemeinsamen Lösung von Problemen, wie etwa der Einsatz von Agenten als aktive Komponenten bei der teamorientierten Softwareentwicklung. Mobile Agenten lassen sich auch für die Wartung von entfernten Systemen einsetzen, wo sie überwachend oder aktiv zur Fehlerbehebung oder für den Support des Systems tätig sein können.

Einen Überblick über laufende Agentenprojekte und im Speziellen mobiler Agenten Projekte bietet [1].

Im Kapitel 6, in dem das Projekt SeMoA vorgestellt wird, finden sich zusätzliche Informationen zur Infrastruktur mobiler Agenten und der Agentenplattform.



Graphische Benutzerschnittstellen

3.1 Entstehung

In diesem Abschnitt wird der Entwicklungsprozess der *Maschine Computer Interaktion* (MCI) anhand von historischen Meilensteinen dargestellt.

Zu Beginn der Computernutzung, als der typische Bediener eines Computers noch der Programmierer selbst war, bestand die Interaktion mit dem Rechner darin, ihn mit Anweisungsfolgen auf Lochkarten oder Magnetband zu speisen. War das Ergebnis nicht ausreichend, mussten Anweisungen geändert und die Prozedur wiederholt werden. Trotz der Weiterentwicklung des Speichermediums und der Ein- und Ausgabegeräte blieben die Benutzerschnittstellen jedoch weitestgehend identisch mit den Programmierschnittstellen [12].

Eine Ausnahme zu dieser Zeit bot das von Ivan Sutherland im Jahre 1963 entworfene *Sketchpad*. Hierbei handelt es sich um ein *Computer Aided Design*-Programm (CAD-Programm), das auf einem für damalige Verhältnisse sehr leistungsstarken Rechner[1] lief. Das Benutzerinterface des Rechners bestand aus einem 9-Inch-Monitor, einigen Knopfreihen und dem sogenannten „lightpen“. Damit liessen sich einfache Vektorgraphiken direkt auf dem Bildschirm erstellen und Menüpunkte zur Manipulation aufrufen. Damit war *Sketchpad* das erste GUI, lange bevor der Begriff geprägt wurde.

Douglas C. Engelbart entwickelte in den 60er Jahren am Stanford Research Institute die Computermaus und zusammen mit seinem Team im Rahmen des Projekts NLS (*oN Line System*) neben vielen anderen heute verwendeten Techniken auch den Vorläufer des Hyperlink und die Fenstertechnik (multiple windows). NLS war eines der ersten Full-Screen-Systeme, bei dem jedes Pixel des Screens per Maus angefahren werden konnte und Anwendungen in unterschiedlichen Bildschirmfenstern zur Ausführung kommen konnten.

[1] der TX-2 Computer am Massachusetts Institute of Technology (MIT) Lincoln Laboratory

Die in den 70er Jahren entstandene Programmierumgebung *Smalltalk* nutzte Engelbarts Innovationen und war eines der ersten kommerziell erfolgreichen Programme mit auf graphischen Bedienelementen gestützter Benutzerschnittstelle.

XEROX entwickelte in den 80er Jahren ein Dokumentverarbeitungssystem mit dem Namen XEROX STAR. Ein großes Entwicklerteam erweiterte die bis dahin erfolgreichen Konzepte zur Benutzerinteraktion. Vergleichbar mit heutigen Benutzerschnittstellen gängiger Betriebssysteme war das zentrale Element bei STAR der Desktop, der einem Schreibtisch nachempfunden war. Dokumente, Programme und physische Geräte wurden als Icons dargestellt. Dateien waren mit ihren Anwendungen verknüpft und konnten somit direkt gestartet werden, was der Zielsetzung der leichten Erlernbarkeit und Handhabung entgegen kam. STAR konnte sich wegen des hohen Preises im Vergleich zu IBM PCs nicht durchsetzen.

Für die Entwicklung des APPLE MACINTOSH wurden viele Ideen des XEROX STAR aufgegriffen und mit eigenen ergänzt. Hierzu zählen das Drag and Drop oder die Zwischenablage für die Kommunikation zwischen Anwendungen.

Das Betriebssystem MS-WINDOWS, das heute zu den weit verbreitetsten zählt, brachte keine wesentlichen Interaktionsideen, sorgte aber für eine weite Verbreitung und Akzeptanz dieser Mensch-Computer-Interaktion. Den quasi Standard für die UNIX Welt bot das in Abschnitt 4.1 vorgestellte X-WINDOW System. Neu für Applikationen war die Möglichkeit, ihre Darstellung an verschiedene Displays im Netzwerk zu delegieren.

Abschließend bleibt die Entwicklung des WWW und die damit verbundene Sprache HTML zu erwähnen. Hyperlinks sind eng mit der Idee des WWW verknüpft und erweiterten das Spektrum an Benutzerinteraktionen.

3.2 JAVA GUI

Für die Entwicklung graphischer Benutzerschnittstellen bietet das von SUN MICROSYSTEMS[2] frei erhältliche *Java Development Kit* (JDK) die *Java Foundation Classes* (JFC). Sie bestehen aus folgenden vier APIs[3]: *Abstract Windows Toolkit* (AWT), *Swing*, *Accessibility* und der *2-D-API*.

Interessant ist hier vor allem die Swing-API, die als verbesserter Nachfolger des AWT gilt. Die Klassennamen von identischen Komponenten beider APIs unterscheiden sich durch ein vorangestelltes `J` bei den Swing Klassen, also beispielsweise `JFrame` für Swing und `Frame` für AWT.

Die folgende Erklärung der Komponenten zur Erstellung von Benutzerschnittstellen gilt jedoch prinzipiell für beide APIs, und wird hier der Verständlichkeit halber anhand des AWTs erläutert

[2] SUN Homepage: http://java.sun.com

[3] API: Application Program Interface

(siehe [7]). Es wird unterschieden zwischen `Component`, `Container`, `LayoutManager` und `Graphics`.

`Component` Etwa die Hälfte aller Klassen im AWT leiten sich von dieser abstrakten Klasse ab, wie beispielsweise `Button`, `Checkbox`, `Label`, `Scrollbar`, `Container`. `Component` bietet eine Reihe von Grundfunktionen und Eigenschaften, hierzu zählen unter anderem:

- Position
- Größe
- übergeordneter Behälter (`Container`)
- Schrifteigenschaften
- Vorder- und Hintergrundfarben
- lokale Bezeichner

`Container` Eine abstrakte Erweiterung der Klasse `Component`, die es ermöglicht mehrere Komponenten (`Component`) zu beinhalten. Zu diesen Behälter Klassen gehören unter anderem:

- `Panel` - ein einfacher Behälter.
- `Applet` - Erweiterung von `Panel`, übergeordnete Klasse aller Applets.
- `Window` - Erweiterung von `Container`, übergeordnete Klasse von `Dialog` und `Frame`.
- `Dialog` - Behälter mit Grundfunktionalität für Dialoge.
- `FileDialog` - ein Dialog zur Auswahl einer Datei.
- `ScrollPane` - Behälter mit Bildlauf Funktionalität für genau ein Element.
- `Frame` - der Behälter einer Anwendung, der im Gegensatz zu `Applet` über eine Menüleiste verfügt.

`LayoutManager` Das Positionieren von Komponenten innerhalb eines Behälters ist Aufgabe der unten aufgeführten Klassen. Neben der gemeinsamen Schnittstelle `LayoutManager` verfügen sie über Methoden zur Darstellung der Komponenten und zur Berechnung der Größe des Behälters. Nachfolgend ist eine Auswahl spezialisierter Layout Manager des AWTs beschrieben.

- `BorderLayout` - Ermöglicht das Zuordnen der Komponenten in nördiche, südliche, östliche, westliche und zentrale Bereiche des Behälters.
- `CardLayout` - damit lassen sich Komponenten hintereinander anordnen, wobei nur jeweils eine aus dem Stapel sichtbar ist.

- `FlowLayout` - Komponenten werden von links nach rechts und oben nach unten angeordnet, ein Umbruch erfolgt bei Erreichen der Containerbreite.
- `GridLayout` - Gestaltung der Komponenten erfolgt innerhalb eines Rasters.

Ereignisse (*Events*) können von Komponenten ausgelöst werden, wie das Drücken eines Buttons oder der Doppelklick der Maus. Die Verarbeitung der Ereignisse erfolgt durch sogenannte *Listener*. Ein oder mehrere solche auf den Ereignistyp spezialisierte Listener werden dazu bei der Komponente registriert. Tritt ein Ereignis ein, wird dieses vom zuständigen Listener durch einen entsprechenden Methodenaufruf abgearbeitet.

3.3 HTML

Tim Berners-Lee, der in den 80er und bis in die 90er Jahre Informatiker am Genfer Hochenergieforschungszentrum CERN war, gilt als Erfinder des *World Wide Web* (WWW). Sein Konzept basierte auf folgenden drei Säulen:

- *Hypertext Transfer Protocol* (HTTP), das die Kommunikation zwischen Web-Clients und Web-Servern spezifiziert.
- *Uniform Resource Identifier* (URI), für die Adressierung von beliebigen Daten im Web.
- *Hypertext Markup Language* (HTML), die Auszeichnungssprache für Web-Dokumente.

Innerhalb weniger Jahre erfuhr dieses Konzept eine nahezu explosionsartige Verbreitung. Tim Berners-Lees ursprüngliche Idee, die es Betrachtern von HTML-Seiten ermöglichen sollte, diese online zu editieren, setzte sich jedoch nicht durch. Für einen gemeinsamen Standard sorgt das 1995 gegründete *World Wide Web Consortium* (W3C).

HTML wird mit Hilfe der *Standard Generalized Markup Language* (SGML) definiert, wie in Abbildung 5.1 skizziert. Es besteht wie andere standardisierte Auszeichnungssprachen aus einer bestimmten Menge von vordefinierten Tags. Zusammen bilden diese Tags den HTML-Quelltext. *Browser* sind Programme die diesen HTML-Quelltext graphisch umsetzen. Enthält der Quelltext Verweise zu Bildern oder anderen HTML-Dateien, ist es Aufgabe des Browsers diese nachzuladen.

Neben Verweisen bietet HTML auch Elemente zur Textstrukturierung, die Möglichkeit Tabellen zu erstellen, Multimedia-Objekte einzubinden oder Benutzereingaben über Formularelemente entgegenzunehmen und darzustellen.

Zu den Formularelementen gehören Eingabefelder, Auswahllisten, Radiobuttons, Checkboxen, Klickbuttons und versteckte Elemente. Es ist desweiteren möglich diese Elemente zu gruppieren, sie zu beschriften oder sie Tastaturkürzeln zuzuordnen.

Die graphische Umsetzung der Formularelemente variiert stark zwischen den verschiedenen Browser-Implementierungen. Die Behandlung von Benutzereingaben verläuft jedoch bei allen gleich. Wird ein Formular etwa durch den Mausklick auf einen Button abgeschickt (*Submit*), werden alle Werte der darin enthaltenen Formularelemente als Key-Value-Paare (*Parameter*) versendet. Die Adresse des Empfängers ist im Formularbereich (*Form Tag*) enthalten[4].

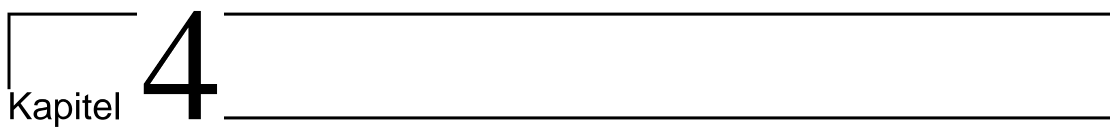
Ein Verwerten der Key-Value-Paare erfolgt mit Hilfe von Zusatzprogrammen für Webserver, die dynamische Webinhalte erzeugen können. Bekannte Vertreter sind Perl, PHP, Servlets oder Java Server Pages (JSP). Wird beispielsweise die *Uniform Resource Locators* (URL) eines Servlets in das Form-Tag einer HTML-Seite geschrieben, löst das Drücken eines Buttons einen Request des Clients aus, den der Webserver an das betreffende Servlet weiterleitet. Das Servlet kann die Parameter aus dem Request auslesen und bei Bedarf an eine Anwendung weitergeben. Die Antwort (*Response*) für den Webclient kann dynamisch erzeugt oder aus einer gespeicherten Datei entnommen sein.

Entscheidend ist, dass ein Webserver einem Webclient nur antworten kann. Er kann also nur reagieren, nie aber agieren. Das hat neben der erhöhten Sicherheit für Clients auch einige Nachteile zur Folge. Es ist nicht möglich den Client zu benachrichtigen, wenn sich auf Seiten des Servers etwas ändert, wie beispielsweise das Eintreffen neuer Börsenkurse.

Abhilfe kann hierbei eine von HTML unterstützte Skriptsprache namens Java-Script schaffen. Moderne Webbrowser können Java-Script interpretieren und ermöglichen dem Webdesigner ein Maß an zusätzlicher Funktionalität auf der Clientseite. Obiges Börsenkurs Beispiel lässt sich in Java-Script mit einer Programmschleife lösen, die in einem festgelegten Zeittakt den Browser zur Aktualisierung der betreffenden Seite auffordert. Java-Script bietet neben der gezielten Manipulation von HTML-Elementen auch die Möglichkeit auf Ereignisse zu reagieren wie etwa beim Anklicken (*onClick*) eines Buttons oder bei gedrückter Maustaste (*onMousedown*) bestimmte Aktionen zu veranlassen. Da sich die Java-Script Unterstützung per Browser auch abwählen lässt, ist es ratsam serverseitig diese Unterstützung abzufragen und entsprechend alternativen Java-Script-freien HTML-Code anzubieten [5].

[4]Definiert durch das Attribut *action* im Form-Tag

[5]Weiterführende Informationen zu HTML und Java-Script finden sich im HTML & Web-Publishing Handbuch der Autoren Stefan Münz und Wolfgang Nefzger [19]



Vergleichbare Arbeiten

Dieses Kapitel gibt einen Überblick über vergleichbare Arbeiten mit anschließender Bewertung zur möglichen Verwendung.

4.1 X-Window-System

Das 1984 vom MIT[1] in Zusammenarbeit mit der Firma DEC[2] entwickelte X-Window-System ist eine graphische Schnittstelle zwischen Computerhardware und UNIX-Anwenderprogramm [16].

Nach dem Erscheinen der Version 11 des X-Window-Systems (kurz X11) im Jahre 1988 gab das MIT die Kontrolle über die Weiterentwicklung des quasi-Standards an das X-Konsortium[3] ab, das von 12 namhaften Computer-Herstellern gegründet wurde.

Die Grundkomponenten der Systemarchitektur sind in der heute aktuellen Version X11R6.6 nachwievor der Server, die Client Programme und der Kommunikationskanal [9].

Server Der Server, auch X-Server genannt, ist für die Darstellung der GUI-Komponenten wie Fenster und deren Inhalt verantwortlich. Jeder Server ist mit genau einem Display gekoppelt. Er reagiert auf Anfragen der Client-Programme, um bestimmte Graphikoperationen wie beispielsweise das Zeichnen einer Linie oder die Darstellung von Text auszuführen. Ereignisse von Eingabegeräten, wie Maus, Tastatur oder Touchscreen, leitet er nach Bedarf an die Client-Programme weiter.

Client-Programme Client-Programme sind GUI-Anwenderprogramme, die zur Darstellung ihrer GUI-Komponenten den X-Server beauftragen. Ein Client-Programm kann sich auf

[1] MIT: Massachusetts Institute of Technology
[2] DEC: Digital Equipment Corporation
[3] Homepage vom X-Konsortium: http://www.x.org

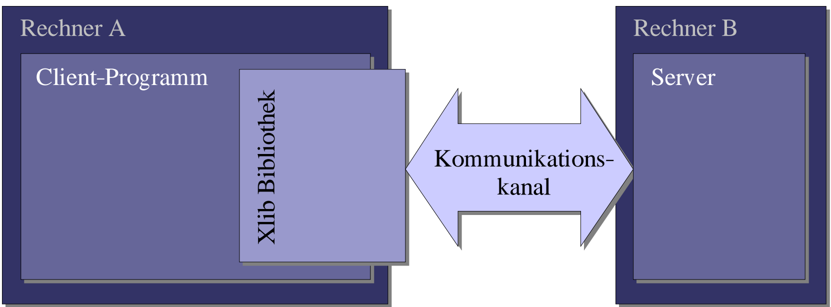


Abbildung 4.1: X-Server und Client auf unterschiedlichen Rechnern

einem anderen Rechner als der Server befinden, siehe Abbildung 4.1. Der Austausch erfolgt über den Kommunikationskanal.

Kommunikationskanal Im Netzwerk wird über ein vereinbartes Protokoll, zum Beispiel TCP/IP kommuniziert. Die Anweisungen, die hierbei übertragen werden, stammen aus der Xlib-Bibliothek, die alle notwendigen Operationen zur Verfügung stellt. Die hardwarespezifische Umsetzung einer Anweisung ist Aufgabe des Servers und bleibt dem Client-Programm verborgen.

Fazit: Wie aus obiger Beschreibung hervorgeht, ist das Vorhandensein eines X-Servers zu jedem Display nötig. Da die in Betracht kommenden Ausgabegeräte üblicherweise nicht über einen X-Server verfügen, wäre eine solche Installation erforderlich. Dies entspricht zum einen nicht der in Abschnitt 1.2 formulierten Zielsetzung, zudem wird es auch bei den meisten Ausgabegeräten nur mit erheblichem Aufwand realisierbar sein.

4.2 XML-Talk

Frank Sauer beschreibt in seinem Artikel *XMLTalk, a framework for automatic GUI rendering from XML specs* [17], XML-Talk als ein Framework, das Java^{TM}-GUI Code automatisch aus einer XML-Beschreibung erstellt. Dazu muss der Benutzer eine bestimmte Syntax verwenden, die zur Beschreibung der Java^{TM}-GUI Komponenten dient. Für das Speichern von Werten der GUI-Klassen und für das Weiterleiten dieser Werte, sowie zur Benachrichtigung von Komponenten verwendet XML-Talk spezielle Klassen, die auch in XML beschrieben werden müssen. XML-Talk unterstützt dabei alle Java^{TM}-Swing-Klassen ab Version 1.3. Weitere Vorteile dieser Technik liegen darin, dass für Änderungen einer GUI lediglich deren XML-Spezifikation angepasst werden muss und in der Wiederverwendbarkeit solcher XML-Spezifikationen.

Fazit: Der Einsatz von XML-Talk ist für solche Applikationen sinnvoll, bei denen der Anteil an GUI-Code besonders hoch ist oder deren GUIs sich häufig ändern. Dies trifft nicht auf die

Agenten-GUIs in SeMoA zu. Desweiteren setzt der Einsatz von XML-Talk ein hohes Maß an Einarbeitungszeit voraus, um die Syntax der XML-Beschreibungssprache zu erlernen.

4.3 Rapid-GUI-Development für das Web

Im Artikel [10] wird ein Framework zur Entwicklung von graphischen Oberflächen vorgeschlagen. Hauptziel der Autoren ist es, dem Entwickler eine Lösung anzubieten, die ein schnelles Erstellen von GUIs ermöglicht. Dazu verwenden sie spezielle Widgets[4] : Input, Output, Action, List of Values, Data und Form. Die Aufgabe dieser speziellen Java-GUI Klassen ist es, eine elementare Funktionalität zur Interaktion mit einem Benutzer zu ermöglichen. Diese Widgets beinhalten keinerlei Angaben zur Darstellung und Positionierung.

Fazit: Der Vorteil der Widgetsammlung, auch Forms-Framework genannt, liegt darin, dass die Programmierung damit einfacher als die mit dem Abstract Windows Toolkit[5] ist und in der schnellen Erstellung der GUIs. Die fehlenden Darstellungsangaben schränken jedoch die Möglichkeiten bei der Anordnung der GUI Komponenten ein.

4.4 Plattformunabhängige Benutzerschnittstellen

Guido de Melo verfolgt in seiner Diplomarbeit [4] mit dem Titel *Plattformunabhängige Benutzerschnittstellen: Beschreibung, Generierung, Anwendung* einen ähnlichen Ansatz wie der Vorschlag aus Abschnitt 4.3 (*Rapid GUI Development*). Das UI wird hier direkt in XML beschrieben, ebenso die Darstellung vom Ablauf der Benutzerinteraktion. Das User Interface Management System (UIMS) erzeugt daraus ein plattformabhängiges Frontend. Für diese Transformation kommt XSLT zum Einsatz.

Fazit: Die Technik für die Transformation, XSLT in Verbindung mit XML fand auch in dieser Arbeit Verwendung.

4.5 Sonstige

Die Firma CreamTec[6] bietet unter dem Namen WebCream ein kommerzielles Produkt an, das Java^{TM}-GUI Komponenten nahezu identisch in HTML darstellen kann. Da sich die Transformation auf HTML beschränkt, das Produkt nicht open Source, somit nicht frei erweiterbar und letztlich kostenpflichtig ist, kam eine Verwendung in SeMoA nicht in Betracht.

[4]Begriff ist aus den englischen Wörtern „window“ und „gadget“ zusammengesetzt. Bestandteil graphischer Fensterelemente.

[5]Das AWT ist eine Klassenbibliothek von Java^{TM} zur Erzeugung von GUIs.

[6]CreamTec Homepage http://creamtec.com:30422

4.6 Bewertung

Die vorgestellten Frameworks halfen bei der Lösungsfindung für die vorliegende Arbeit, sei es durch ihre Konzepte oder durch die dort verwendeten Techniken. Die Beschreibung der GUI in XML, wie von *XML-Talk*, *Rapid-GUI-Development* und *Plattformunabhängige Benutzerschnittstellen* verwendet, gehört dazu, sowie die Transformation durch die Technik XSLT. Das *X-Window-System* war Vorlage für die Client/Server Architektur des Frameworks und die im Abschnitt Sonstiges erwähnte Firma *CreamTec* hat mit ihrem Produkt *WebCream* bewiesen, dass eine Transformation von Java^{TM}-GUIs nach HTML-GUIs möglich und somit machbar ist.

XML und XSLT

In diesem Kapitel werden Grundlagen von XML und XSLT vermittelt. Beide Techniken spielen eine zentrale Rolle in der vorliegenden Arbeit.

5.1 XML

Die in einer Arbeitsgruppe des World Wide Web Consortiums (W3C) unter der Leitung von Jon Bosak (SUN) entwickelte *Extensible Markup Language*, kurz XML, wurde im November 1996 im Rahmen der SGML´96 Konferenz in Boston erstmals der Öffentlichkeit vorgestellt. XML ist eine Untermenge von SGML (Standard Generalized Markup Language) und damit auch eine Metasprache, wie Abbildung 5.1 veranschaulicht. Eine Metasprache definiert selbst keine Tags, auch Elemente genannt. Sprachen wie HTML, WML oder XSLT werden als standardisierte Auszeichnungssprachen bezeichnet, da sie aus einer bestimmten Menge von bekannten Tags bestehen.

XML eignet sich besonders gut dazu, Daten plattformunabhängig zu beschreiben, zu strukturieren und auszutauschen [3]. Ein XML-Element besteht aus einem sich öffnenden und schließenden Tag. Dazwischen kann sich Text, ein oder mehrere andere Elemente aber auch nichts befinden. Innerhalb des öffnenden Tags eines Elements können sich dessen Attribute in Form von Attribute-Wert Paaren befinden. Durch das Verschachteln von Tags können einfache Vater-Kind Beziehungen zwischen Elementen dargestellt werden, siehe Abbildung 5.2.

Ein wohlgeformtes XML-Dokument hat genau ein Wurzel-Element. Elemente müssen korrekt ineinander verschachtelt sein, so dass ein Element vollständig im Inhalt des Vater-Elementes steht. Attribute eines Elements sind durch Gleichheitszeichen von ihren Werten, die in hochgestellten Anführungszeichen stehen müssen, getrennt dargestellt. Die Wohlgeformtheit von XML-Dokumenten ist Voraussetzung vieler XML verarbeitenden Tools (z.B. XML-Prozessoren).

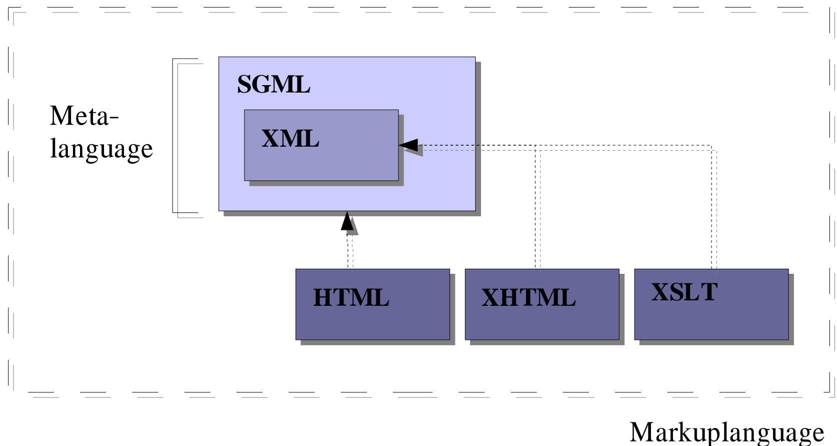


Abbildung 5.1: SGML Stammbaum

Weitere Informationen zu XML-Spezifikation und Wohlgeformtheit finden sich auf der Webseite des W3C[1].

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="html_agent_gui.xslt"?>
<root>
 <class>
  <agentgui.JFrame title="Jumping Agent" height="180.0" width="340.0" x="0.0" y="0.0" id="7438914">
   <class>
    <agentgui.JPanel height="52.0" width="332.0" x="0.0" y="101.0" id="14008569">
     <class>
      <agentgui.JComboBox height="26.0" width="332.0" x="0.0" y="0.0" id="11143287" />
     </class>
     <class>
      <agentgui.JButton label="Jump!" height="26.0" width="332.0" x="0.0" y="26.0" id="6906832" />
     </class>
    </agentgui.JPanel>
   </class>
  </agentgui.JFrame>
 </class>
</root>
```

Abbildung 5.2: Beispiel für ein XML-Dokument

Neben der Wohlgeformtheit kann ein XML-Dokument auch auf Gültigkeit geprüft werden. Dazu wird innerhalb des XML-Dokuments geprüft, ob es eine *Document-Type-Definition* (DTD) besitzt und ob die darin formulierten Regeln eingehalten werden. Es lassen sich per DTD die Namen der Elemente eines XML-Dokuments vorschreiben. Es kann festgelegt werden, wie sich die Kindelemente eines Elements zusammensetzen müssen, die genaue Reihenfolge dieser und die minimale und maximale Anzahl eines jeden einzelnen. Ähnliche Restriktionen lassen sich auch für die Attribute eines Elements treffen.

[1]XML-Spezifikation: http://www.w3.org/TR/REC-xml

Mehr Möglichkeiten zur Validierung als DTD bietet ihr Nachfolger XML-Schema[2].

5.2 XSLT

Extensible Stylesheet Language (XSL) und XSL-Transformations (XSLT) sind zwei unterschiedliche Technologien. XSL dient der Definition von Stilanweisungen, vergleichbar mit der Stilsprache Cascading Style Sheets (CSS) aus dem HTML Umfeld. XSLT wird zur Transformation von XML-Dokumenten in beliebige Textformate eingesetzt.

> „Obwohl XSLT zum Zeitpunkt seiner Entwicklung auf die Generierung von XSL-Formatierungsobjekten ausgelegt wurde, hat es sich zur bevorzugten Technik für alle Arten von Transformationen entwickelt.“ [2]

Im November 1999 wurde XSLT als Empfehlung vom W3C herausgegeben.

5.3 XSLT-Grundlagen

Für eine XSLT-Transformation benötigt man ein XML-Dokument, ein XSLT-Stylesheet und einen XSLT-Prozessor, wie in Skizze 5.3 dargestellt.

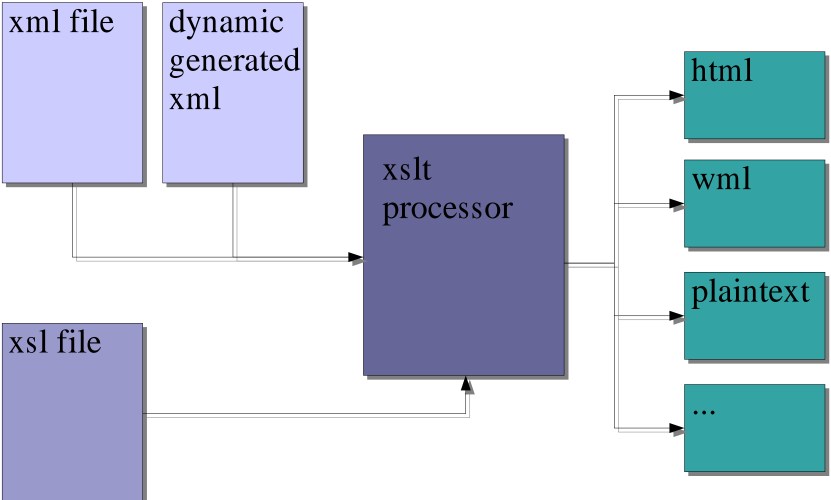


Abbildung 5.3: Schematik einer XSLT-Transformation

Der XSLT-Prozessor ist eine Anwendung, die ohne die originalen XML-Daten zu verändern, diese nach Anleitung des XSLT-Stylesheets transformiert und das Resultat in einem Ergebnisbaum speichert. Der Ergebnisbaum kann in eine Datei geschrieben oder zur weiteren Verarbeitung an andere XSLT-Prozessoren übergeben werden. Zu den bekanntesten Open Source

[2]Weiterführende Informationen zu XML-Schema finden sich unter http://www.w3.org/XML/Schema.

Java^{TM}-Implementierungen der XSLT-Prozessoren gehören *Saxon*[3] und *Xalan*[4] der Apache Software Foundation.

Der XSLT-Prozessor kann sowohl clientseitig als auch serverseitig zum Einsatz kommen. Verwendet der Client einen XSLT fähigen Browser kann dieser nach Aufruf einer XML-Datei, in der die zugehörige XSLT-Datei mit angegeben sein muss, die Transformation selbständig durchführen. Der serverseitige Einsatz bietet einerseits mehr Kontrolle im Hinblick auf eine einheitliche Ausgabe, andererseits ist es Applikationen dadurch möglich (Steuer-)Anweisungen an den XSLT-Prozessor zu übermitteln, und findet deshalb in der Praxis bevorzugt Verwendung.

An dieser Stelle sei noch auf X-Path hingewiesen, das ebenfalls eine Empfehlung des W3C ist und von XSLT verwendet wird. Die Technik X-Path erlaubt es Teile eines XML-Dokuments zu adressieren und bietet Funktionen zum Text- und Zahlen-Vergleich dieser Teile sowie für konditionale Logik. Weiterführende Informationen finden sich auf den Internetseiten des W3C[5] und im folgenden Abschnitt 5.4. Dort werden XSLT-Techniken erklärt, die teilweise X-Path Funktionalitäten verwenden.

5.4 XSLT-Techniken

Ein XSLT-Stylesheet hat neben der XML konformen Deklaration der zu verwendenden XML-Version und Kodierung ein alles weitere umschließendes XSL-Stylesheet Tag mit Versionsangabe und Namespace Attribut, siehe Abbildung 5.4. Das folgende Element

```
<xsl:template match="/"> ... </xsl:template>
```

ist in ähnlicher Form in jedem XSL-Stylesheet zu finden, es ist das initiale *Template*, dessen Aufruf vom XSLT-Prozessor automatisch erfolgt. *Templates* in XSLT lassen sich am einfachsten mit den Methoden einer höheren Programmiersprache vergleichen.

Es gibt zwei Arten von *Templates*, die *named* und die *unnamed Templates*. Beim *named Template* erfolgt der Aufruf über den Wert des Attributs *name*. Beim *unnamed Template* dient das Attribut *match* als Muster, das zum Vergleich mit den Elementen im XML-Dokument verwendet wird.

Die Elemente eines XML-Dokuments liegen dem XSLT-Prozessor in baumartiger Struktur vor. Beziehungen der Elemente in dieser Struktur wie *parent* oder *child*, *ancestor* oder *descendant* werden zum Mustervergleich verwendet. Diese Beziehungen sind relativ zum aktuellen Element, und werden im Zusammenhang mit der Baumstruktur *Achsen* genannt. Für das aktuelle Element gilt auch die Bezeichnung *Kontextknoten*.

[3] Saxon Homepage: http://saxon.sourceforge.net/
[4] Xalan Homepage: http://xml.apache.org/xalan-j/
[5] X-Path Webpage: http://www.w3.org/TR/xpath

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <title> SEMOAs AGENT GUI classes to HTML</title>
      </head>
      <body>
          ...
          <xsl:apply-templates select="root/class"/>
          ...
          <xsl:call-template name="footer">
            <xsl:with-param name="top" select="root/class/agentgui.JFrame/@height"/>
          </xsl:call-template>
          ...
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

Abbildung 5.4: XSLT-Stylesheet Beispieldatei

Beim Aufruf des initialen *Templates* im obigen Beispiel durch den XSLT-Prozessor ist der aktuelle Knoten das Dokument selbst, gesucht wird der *Root-Knoten* („ / “), der daraufhin zum aktuellen Knoten wird u.s.w..

Nachfolgende Tabelle 5.1 gibt eine Übersicht über mögliche Beziehungen der Knoten. Die in Tabelle 5.2 dargestellten Abkürzungen finden üblicherweise in der Praxis und deshalb auch in den hier gezeigten Beispielen Verwendung.

Bezeichnung	Beschreibung
ancestor	Vorfahren des Knotens bis hin zum Root-Knoten
ancestor-or-itself	Wie *ancestor* aber inklusive des Knotens selbst
attribute	Alle Attribute des Knotens
child	Alle unmittelbaren Kindelemente des Knotens
descendant	Alle Nachfahren des Knotens
descendant-or-itself	Alle Nachfahren des Knotens inklusive des Knotens selbst
following	Alle Nachkommen des Knotens bis auf die *descendant* Elemente
following-sibling	Alle Knoten, die dasselbe Elternelement wie der aktuelle Knoten haben
parent	Der Vorfahr des Knotens
preceding	Alle Vorfahren des Knotens bis auf *parent* Elemente
preceding-sibling	Alle Vorfahren des Knotens, die dasselbe Elternelement wie der aktuelle Knoten haben
self	Der Knoten selbst

Tabelle 5.1: Übersicht der Achsen

Abkürzung	Bezeichnung
@	attribute
	child
//	descendant
..	parent
.	self

Tabelle 5.2: Abkürzung der Achsen

Wenn das oben vorgestellte initiale *Template* aufgerufen wird, wird jeglicher Text, der innerhalb des *Templates* steht, in den Ergebnisbaum geschrieben. Findet der XSLT-Prozessor in diesem Text XSLT-Anweisungen, beginnt er diesen zu interpretieren und auszuführen. Der Aufruf eines *unnamed Templates* erfolgt mit folgender Anweisung.

```
<xsl:apply-templates select="root/class"/>
```

Das Attribut *select* liefert das Pattern für den Vergleich. Da zum Zeitpunkt des Aufrufs der aktuelle Kontextknoten der *Root-Knoten* ist, beinhaltet das Ergebnis alle Kindelemente von *root/class*. Ist eines der Kindelemente *agentgui.JFrame*, wird folgendes Template aufgerufen.

```
<xsl:template match="agentgui.JFrame"> ... </xsl:template>
```

Der Aufruf eines *named Templates* liefert unabhängig von der Position des Kontextknotens dasselbe Ergebnis. Der folgende Aufruf

```
<xsl:call-template name="footer">
```

bezieht sich auf dieses Template.

```
<xsl:template name="footer"> ... </xsl:template>
```

Es ist möglich einem *Template* beim Aufruf beliebig viele Parameter zu übergeben. Nachstehende Syntax liefert ein Beispiel.

```
<xsl:call-template name="footer">
  <xsl:with-param name="top" select="agentgui.JFrame/@height"/>
   ...
</xsl:call-template>
```

Die Deklaration der Parameter steht im entsprechenden Template.

```
<xsl:template name="footer">
  <xsl:paramname="top"/>
  ...
</xsl:template>
```

Variablen können in XSLT auf drei verschiedene Arten definiert werden. Die dritte Methode liefert bei Verwendung der Variablen einen leeren String. Anzumerken gilt, dass eine Variable mit der Deklaration gleichzeitig initialisiert wird und dieser Wert anschließend nicht verändert werden kann.

```
<xsl:variable name="JFrameTitleHeight">18</xsl:variable>
<xsl:variable name="JFrameTitleHeight" select="var/jframe_th"/>
<xsl:variable name="JFrameTitleHeight"/>
```

Der Aufruf erfolgt durch Verwendung des Dollarzeichens, das vor den Variablennamen gestellt wird, wie im Folgenden zu sehen ist.

```
<div style="top:{$JFrameTitleHeight};">
  ...
</div>
```

Die `xsl:value-of` Anweisung liefert den Wert eines Elements oder wie in folgendem Beispiel den Wert eines Attributs. Dazu werden alle Attribute des aktuellen Kontextknotens mit dem Pattern des *select* Attributs verglichen, hier *listener*.

```
<xsl:value-of select="@listener"/>
```

XSLT verwendet für konditionale Logik folgende Syntax.

```
<xsl:if test="@resource">
 <img src="resource?type=jpg&amp;id={@resource}&amp;port={$port}"/>
  ...
</xsl:if>
```

Dies entspricht einer if-Bedingung, gleich denen aus den meisten Programmiersprachen. Das Attribut *test* liefert dem XSLT-Prozessor die Bedingung. Mit `test="@resource"` wird die Existenz dieses Attributs geprüft.

Wird ein *else*-Teil benötigt, bietet XSLT keine entsprechende Syntax. Hier hilft es das if-Statement erneut zu verwenden, diesmal mit invertierter Bedingung.

Äquivalent zu dem in Java^{TM} verwendeten *switch* wird nachstehendes XSLT-Statement gebraucht.

```
<xsl:choose>
 <xsl:when test="farbe = 'rot'">
    rot
 </xsl:when>
 <xsl:when test="farbe = 'blau'">
    blau
 </xsl:when>
 <xsl:otherwise>
    grün
 </xsl:otherwise>
</xsl:choose>
```

Weiterführende Informationen zu XSLT finden sich in Eric M. Burkes Buch *Java und XSLT* [8].

SeMoA

Dieses Kapitel gibt Aufschluss über das Mobile-Agenten-System SeMoA und soll Abschnitt 2.3 ergänzen.

6.1 Intention von SeMoA

SeMoA ist ein Projekt des Instituts für graphische Datenverarbeitung (IGD) der Fraunhofer Gesellschaft in Darmstadt, Abteilung Sicherheitssysteme für Graphik- und Kommunikationssysteme. SeMoA steht für Secure Mobile Agents, und ist eine Java^{TM}-Implementierung für mobile Agenten und Agentenserver. Die Entwickler von SeMoA interessierten sich besonders für das Gebiet der Sicherheit bezüglich folgender Angriffsszenarien:

- *Angreifer zu Agent:* Agenten werden abgefangen, um sie auszuspähen, zu manipulieren oder um sie zu löschen.
- *Agent zu Agent:* Der Angriff erfolgt in Form von Manipulation oder Spionage.
- *Agent zu Agentenserver:* Der Agent führt böswilligen Code aus oder verbraucht die Ressourcen des Hosts (*denial of service attack*).
- *Agentenserver zu Agent:* Die Agentenplattform ist böswillig und versucht Agenten auszuspähen, zu manipulieren, zu eliminieren oder sie in ihrer Ausführung zu behindern.

Detaillierte Hinweise zu den in SeMoA verwendeteten Sicherheitsvorkehrungen finden sich im folgenden Abschnitt und zusammengefasst in Abschnitt 6.3.

Ein weiteres Anliegen der Entwickler galt der einfach zu erweiternden modularen Struktur der Agentenplattform, wie der folgende Abschnitt 6.2 zeigt. Diese Struktur gestattet zum Beispiel

auch das Einbinden eines Wrappers[1], um Agenten von fremden Agentenplattformen empfangen zu können.

6.2 Systemarchitektur

In diesem Abschnitt werden die wichtigsten Komponenten der SeMoA-Plattform vorgestellt. Hierzu zählt die `Shell`, das `Environment`, verschiedene Dienste, sowie die Agenten.

Die `Shell` ist die Laufzeitumgebung für Dienste und Agenten und erlaubt deren Administration. Der im Zusammenhang mit SeMoA häufig verwendete Begriff *Agenten-Server* ist eine Sammlung von Basisdiensten zur Migration, Kommunikation und zum Tracking sowie zu Sicherheitsmechanismen. Die Konfiguration des Agenten-Servers erfolgt beim Start von SeMoA über das Laden verschiedener Shellskripte.

Die bereits erwähnte modulare Bauweise der Plattform ist besonders deutlich im Zusammenhang mit den Diensten (*Service*) erkennbar. Sie können beispielsweise den Zugriff auf eine Datenbank oder auf Bereiche des Filesystems ermöglichen, oder die Funktionalität eines Webservers anbieten. Um einen Dienst nutzen zu können, muss er zuvor im `Environment` registriert werden. Es können auch Dienste zur Laufzeit registriert und entfernt werden. Dies ist dem Administrator über die `Shell`, sowie Agenten und anderen Diensten gegebenenfalls eingeschränkt durch eine entsprechende Java-API möglich.

Der Zugriff auf Dienste ist nur über das `Environment` möglich. Es liefert jedem Nutzer eine individuelle Sicht auf die Dienste, die seinen Zugriffsrechten entspricht. Einen weiteren Sicherheitsaspekt stellt die Kapselung der Dienste in Proxyobjekte dar. Das `Environment` liefert beim Zugriff auf einen Dienst ein solches Proxyobjekt, das alle Schnittstellen des Dienstes implementiert und damit einen direkten Zugriff auf die Diensteinstanz ausschließt.

Die `Shell` gestattet neben dem bereits erwähnten Registrieren von Diensten die Interaktion mit der Plattform in einer Weise, die den gängigen UNIX Shells nachempfunden wurde. Damit lassen sich Java^{TM}-Klassen, beispielsweise SeMoA-Agenten, direkt mit dem bekannten Befehl *java* starten. Ein Benutzer kann über die `Shell` auch Informationen über alle ihm zur Verfügung stehenden Dienste und Agenten erhalten. Hier hat er zudem die Möglichkeit einen Befehlszeilen-orientierten Dialog mit einem Agenten zu führen und kann sich über Status und Fehlermeldungen der Plattform informieren.

Die Anforderungen an eine Agentenklasse sind minimal. Durch das Implementieren der Schnittstellen `Runnable` (bzw. dessen SeMoA-Erweiterung `Resumable`) und `Serializable` entsteht bereits ein einfacher Agent. Mobil wird er durch das Setzen eines `Tickets`, einer

[1] Setzt die Aufrufe des Plattform-fremden Agenten in geeigneter Weise um.

Klasse, die den Eintrag von Zieladressen erlaubt und die beim Terminieren eines Agenten auf mögliche Einträge untersucht wird. Ist das Ticket gesetzt, veranlasst der Agenten-Server den Transport des Agenten. Dazu wird der Agent serialisiert und zusammen mit seinen Ressourcen in ein *Java Archive* (JAR-File) geschrieben. Die Wahl fiel auf JAR-Files, weil sie die digitale Signatur nach PKCS7 [11] unterstützen. Die von SeMoA erweiterten JAR-Files gestatten darüberhinaus das selektive Signieren und Verschlüsseln von Inhalten. JAR-Files eignen sich auch deshalb besonders gut zum Transport von Agenten, weil der verzeichnisartige Aufbau eines JAR-Files die Abbildung von Agenten unterstützt. Agenten setzen sich aus ihren mitgeführten benötigten Klassen, den Konfigurationsparametern und den veränderlichen Daten zusammen.

Vor dem eigentlichen Transport durch den Dienst `OutGate` durchläuft der Agent eine Reihe von Filtern. Der `EncryptFilter` kann Teile des Agenten verschlüsseln, der `SignFilter` signiert abschließend den gesamten Agenten.

Ankommende Agenten werden von `InGate` an die Filter `VerifyFilter` und `Decrypt-Filter` und falls vorhanden, auch an optionale Filter übergeben. Der `VerifyFilter` prüft die Signaturen und kann bei Bedarf, wie die anderen Filter auch, einen Agenten ablehnen. Nach erfolgreicher Prüfung wird der Agent deserialisiert. Anschließend wird er vom Server in einer eigenen `ThreadGroup` gestartet und erhält einen eigenen `Classloader`. In dieser Umgebung hat er Zugriff auf alle von ihm mitgeführten Klassen, Konfigurationsparameter, Ressourcen, etc. (vgl. [15]). Die Schnittstelle zur Plattform stellt für den Agenten das bereits erwähnte `Environment` dar.

6.3 Sicherheitsmechanismen

In diesem Abschnitt werden die bereits teilweise angesprochenen Sicherheitsmechanismen von SeMoA zusammengefasst und den in Abschnitt 6.1 vorgestellten Angriffsszenarien zugeordnet.

Wie in Abbildung 6.1 zu erkennen ist, durchläuft ein mobiler Agent, in der Abbildung mit M.A. bezeichnet, zwei verschiedene Layer (Schichten), bevor er in der dritten Schicht seinen Code ausführen kann.

Der äußere Layer repräsentiert die Verschlüsselung von Agenten beim Transport (*transport layer security*), zum Beispiel durch die Verwendung von *Secure Socket Layer* (SSL). Dieser Layer bietet Schutz vor dem Ausspähen und Manipulieren der Agenten.

Die im vorigen Abschnitt angesprochenen Filter sind Teil des mittleren Layers. Hier erfolgt anhand der festgestellten Herkunft eines Agenten die Vergabe von Rechten auf der Plattform (*agent verification and authentication*). Dieser Layer minimiert das Risiko, das von böswilligen Agenten ausgeht, indem beispielsweise unbekannte Agenten abgelehnt werden. Er bietet aber auch Agenten selbst Schutz, die ihre Daten verschlüsseln können, um ein Ausspionieren zu verhindern.

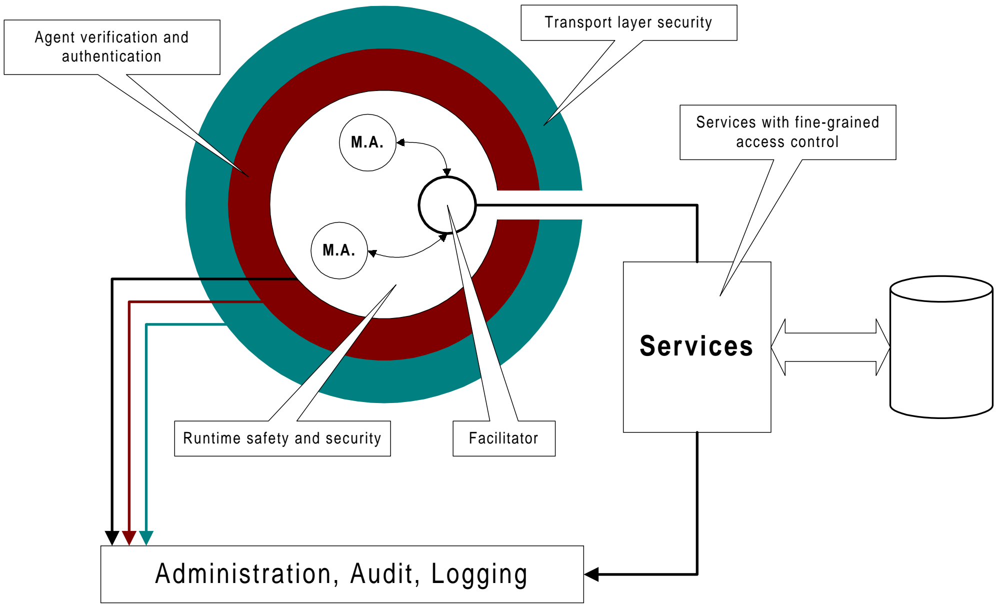


Abbildung 6.1: Sicherheitskonzept von SeMoA

Der innere Bereich (*runtime safety and security*) steht für die Ausführung der Agenten in der eigenen `ThreadGroup` mit eigenem `Classloader`. Die ebenfalls bereits angesprochene individuelle Sicht der Agenten auf die Dienste ist in der Abbildung mit *Services with fine-grained access control* beschriftet. Diese Sicherheitsmechanismen richten sich vor allem gegen die Gefahren, die von böswilligen Agenten ausgehen. Das Ausführen in einer Art *Sandbox* vermindert dieses Risiko und gestattet den administrativen Eingriff.

Teil II

Systementwicklung

Kapitel 7

Problemdefinition

Die zu Beginn dieser Arbeit vorgestellten Ziele werden im Folgenden etwas ausführlicher dargestellt, im Abschnitt Einsatzbereich werden mögliche Anwendungsszenarien diskutiert.

7.1 Anforderungen

Neben den in Abschnitt 1.2 dargestellten Anforderungen folgen hier die SeMoA-spezifischen. Es soll nicht nur möglich sein, mit Agenten auf dem lokalen SeMoA-Server zu kommunizieren, sondern auch mit Agenten auf entfernten Rechnern, wobei der Kontakt wahlweise direkt erfolgen kann oder über den lokalen SeMoA-Server, der dann die Rolle des Vermittlers einnimmt. Die Realisierung dieser offenen Architektur soll eine größere Vielfalt zukünftiger Anwendungsfälle ermöglichen.

SeMoA-Agenten können persistente Daten mit sich führen, auch Ressourcen genannt. Das entfernte Darstellen dieser Ressourcen ist eine weitere Anforderung an das Framework.

7.2 Einsatzbereich

Folgendes Beispiel soll den praktischen Nutzen der vorliegenden Arbeit verdeutlichen. Eine Wintersportlerin befindet sich im Sportgeschäft ihres Urlaubsdorfes und möchte einen Preisvergleich tätigen. Dazu verwendet sie ihren Palm PC, der sie mit ihrem SeMoA-Server zu Hause verbindet[1]. Sie beauftragt einen dafür konzeptionierten SeMoA-Agenten mit dem Preisvergleich. Der Agent startet seine Reise zu verschiedenen anderen geeigneten SeMoA-Servern[2]

[1] Beispielsweise mit Mobilfunkerweiterungskarte für Palm PC, oder mit Infrarotverbindung zu Mobilfunkgerät möglich.

[2] Beispielsweise SeMoA-Server bei Sportartikelvertrieben, die Agenten den Datenbankzugriff gestatten.

um die Preisinformationen zu sammeln. Die Urlauberin trennt die Verbindung, sie weiß, dass es abhängig vom Umfang der Recherche eine Weile dauern kann bis sie die Ergebnisse vom Agenten geliefert bekommt. Später kontaktiert sie erneut ihren SeMoA-Server und stellt fest, dass der Agent mit den Ergebnissen zurückgekehrt ist und verwendet sie. Sie hätte den Agenten ebenfalls per Mobiltelefon, oder über den frei zugänglichen Computer im Fremdenverkehrsbüro ihres Urlaubdorfes kontaktieren können.

Ein derartiges Szenario ist mit den bisherigen SeMoA-Bestandteilen nicht direkt möglich, da ein Agent nur innerhalb der SeMoA-Infrastruktur (also nur auf einen SeMoA-Host) migrieren und durch seine Java-GUI auf dem jeweiligen Host momentan nur lokal mit dem Benutzer interagieren kann. Die vorgestellte SeMoA-Erweiterung schafft hier Abhilfe und ermöglicht die entfernte Interaktion mit Agenten.

Die Erweiterung bietet dabei einen Mechanismus, der die Darstellung auf die entfernten Geräte transportiert. Die Entkoppelung von Anwendung und Darstellung bietet den Vorteil, dass auch solche Geräte verwendet werden können, die aus Gründen der Performance für die Anwendung ungeeignet wären, für die reine Darstellung jedoch genügen. Die unterschiedlichen Display-Eigenschaften dieser Geräte wie Größe und Anzahl der darstellbaren Farben gilt es dabei zu berücksichtigen.

Lösungsansatz

Da SeMoA selbst in Java^{TM}implementiert ist, liegt es nahe die vorgestellten Ergänzungen selbst auch in Java^{TM}zu realisieren. Zur Verwendung kommende Techniken wie beispielsweise der XSLT-Prozessor sind ebenfalls in Java^{TM}implementiert und erlauben die nahtlose Integration in das Framework.

Die Kenntnis über die Funktionsweise eines Webservers und dazu gehörender Servlet-Technologie [20] wird an dieser Stelle vorausgesetzt und nicht weiter erläutert.

8.1 Darstellung

Um der Anforderung gerecht zu werden bestehende Java^{TM}-GUIs möglichst ohne Änderung zu übernehmen (siehe Abschnitt 1.2), fiel die Entscheidung darauf, die Erstellung dieser Benutzerschnittstellen weiterhin mit Java^{TM}-Klassen zu realisieren, zusätzlich aber deren Aufbau zu erfassen und für die verschiedenen Clients oder Anzeigegeräte in geeigneter Form zu transformieren.

Für das Erfassen von Java^{TM}-Instanzen, hier speziell von Java^{TM}-GUI Instanzen gibt es in Java^{TM}eine Technik namens *Reflection*. Hierbei handelt es sich um eine Funktionssammlung, die es erlaubt Informationen über den Aufbau einer Klasse oder deren Instanz zu erfahren. So können beispielsweise die Methodennamen einer Klasse erfragt werden und anhand eines Methodennamens und einer Klasseninstanz ein Methodenaufruf auf diese Klasseninstanz erzeugt werden.

Der hierarchische Aufbau eines Java^{TM}-GUI Fensters und dessen Komponenten (siehe Abschnitt 3.2) erlaubt das rekursive Traversieren von der Wurzel-Komponente bis zur letzten Kind-Komponente.

Auf der Suche nach einer geeigneten Form um diese Struktur zu beschreiben, scheint XML durch die hierarchische Anordnung der XML-Tags (siehe Abschnitt 5.1) am geeignetsten und im Hinblick auf die Transformation (siehe nächster Abschnitt) die ideale Lösung zu sein.

Die meisten internettauglichen Geräte besitzen bereits die Möglichkeit Inhalte aus dem WWW über integrierte Browser darzustellen. Die dafür verwendeten Protokolle sind derzeit HTTP und WAP[1]. Ziel ist es daher, die Java^{TM}-GUI Komponenten in den entsprechenden Sprachen HTML und WML abzubilden, welche über die Transportprotokolle HTTP bzw. WAP transportiert werden.

Beinhaltet ein Java^{TM}-GUI Fenster auch Bildelemente oder andere Ressourcen, gilt es diese ebenfalls zu übertragen.

8.2 Transformation

Das in Abschnitt 5.2 vorgestellte XSLT bietet die Möglichkeit vorhandene XML-Inhalte in beliebige Ausgabeformate zu transformieren. Der dazu benötigte XSLT-Prozessor bekommt den XML-Input direkt aus den entsprechenden Java^{TM}-GUI Klassen. Um eine höhere Performance des XSLT-Prozessors beim Transformieren zu erhalten, ist es sinnvoll die XML-Daten in einer passenden Form zu erstellen und so dem Prozessor zu übergeben. Hilfreich ist hierbei das Java^{TM}-Packet *org.w3c.dom*.

Die XSLT-Stylesheets liegen den verschiedenen Ausgabeformaten und Browserversionen entsprechend in einem Verzeichnis bereit. Hier können sie, ohne dass erneut Java^{TM}-Code kompiliert werden muss, zur Laufzeit verändert werden. Diese Stylesheets berücksichtigen die Eigenheiten der Ausgabegeräte, wie beispielsweise die Anpassung an die Display-Größe oder die Verteilung der GUI-Elemente auf mehrere Seiten (WML).

8.3 Kontrolle und Verwaltung

Jeder Agent, der auf einem SeMoA-Host gestartet wird oder dorthin migriert, wird automatisch bei einer Kontrollinstanz registriert, sobald er durch den Aufruf von Java^{TM}-Fensterelementen in den Benutzerdialog tritt. Verlässt der Agent den SeMoA-Server oder beendet er den Benutzerdialog, wird er von dieser Kontrollinstanz deregistriert.

Eine weitere Aufgabe der Verwaltung ist das Überprüfen der Benutzerrechte zum Zugriff auf einen Agenten, sowie die Steuerung der Darstellungsvarianten eines Agenten. Möglich sind folgende Szenarien:

- der Dialog ausschließlich mit Java^{TM}-GUI Elementen.

[1] WAP: Wireless Application Protocol

- das Unterdrücken der Darstellung dieser Elemente für eine exklusive alternative Darstellung.
- die Kombination beider mit der Gefahr der gegenseitigen Beeinflussung.

8.4 Transport und Weiterleitung

SeMoA-Hosts sollen nach Abschnitt 7.1 die Möglichkeit erhalten, die GUI-Daten ihrer Agenten auszutauschen. Für die Weiterleitung ist eine Client-Server Architektur notwendig. Die Kommunikation soll über ein geeignetes Protokoll erfolgen. Eine übergeordnete Instanz delegiert die notwendigen Abläufe. Jeder SeMoA-Host erhält einen Server für den Transport, der an einem festgelegten Port auf Anfragen reagiert.

Will beispielsweise ein Benutzer auf Host A mit einem Agenten auf Host B kommunizieren, wird die Anfrage von der Weiterleitungs-Komponente des Hosts A entgegengenommen. Dieser startet einen Transport-Client, der sich mit dem Server auf Host B in Verbindung setzt. Der Transport-Server auf Host B leitet dann die Anfrage an die Kontroll- und Verwaltungsinstanz seines Hosts, siehe Abbildung 8.1. Folgende Anfragen sind möglich:

- zeige alle verfügbaren SeMoA-Hosts.
- zeige alle Agenten eines bestimmten Hosts.
- zeige die GUI-Elemente eines bestimmten Agenten.
- zeige eine bestimmte Ressource eines bestimmten Agenten.

Die Antworten sind XML-codiert mit Ausnahme der Ressourcen. Hier wird eine Kopie des Originals in Form eines Bytestroms übertragen.

8.5 Systemarchitektur

Die Architektur von SeMoA bietet die Möglichkeit so genannte Dienste zu integrieren, vgl. Abschnitt 6.2. Die Kontrollinstanz, der Exchanger und der Server für den Austausch werden als solche in SeMoA eingefügt. Bereits vorhanden ist ein Webserver (auch als SeMoA-Dienst realisiert), der auch Servlets unterstützt. Die Verwendung von Servlets ist unumgänglich im Zusammenhang mit Java^{TM}-Anwendungen und dynamischen Webinhalten. Webanfragen erhält das Hauptservlet, das diese Anfragen an den Exchanger weiterleitet, u.s.w.. Die XSLT-Transformation wird ebenfalls von diesem Servlet veranlasst. Ein zweites Servlet ist für die Zustellung der Ressourcen zuständig.

Abbildung 8.1 zeigt das Zusammenspiel der einzelnen Komponenten.

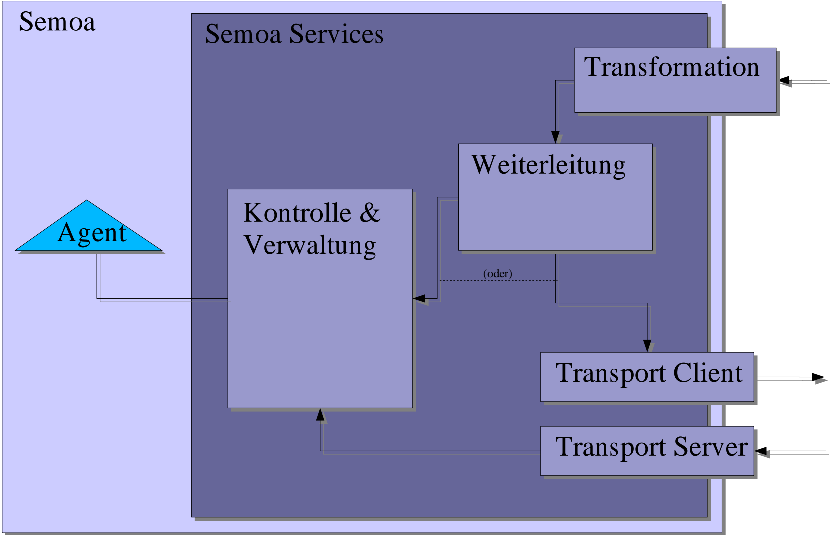


Abbildung 8.1: Systemarchitektur

8.6 Vorabbewertung des Lösungsansatzes

Die in Kapitel 4 vorgestellten Techniken erlauben teilweise das schnellere und einfachere Erstellen von Benutzerschnittstellen als das mit Java^{TM}möglich wäre. Neben den in diesem Kapitel aufgeführten Gründen, die eine Verwendung dieser Techniken in dieser Arbeit ausschließen, gibt es einen weiteren Grund. Es ist nicht die Absicht dieser Arbeit das Erstellen von Java^{TM}-GUIs zu erleichtern oder zu beschleunigen, vielmehr soll eine Lösung gefunden werden, die dem Agentenprogrammierer weiterhin die uneingeschränkte Verwendung aller Java^{TM}-GUI Komponenten erlaubt.

Zu bedenken gilt, dass die Möglichkeiten der identischen Darstellung auf den Anzeigegeräten sprachabhängig ist, so kennt WML weniger graphische Elemente als HTML. Und selbst HTML kennt weitaus weniger Elemente zur Anzeige wie sie Java^{TM}kennt. Bei der Erstellung der XSLT-Stylesheets kann mit entsprechendem Aufwand eine brauchbare Umsetzung dieser Elemente (z.B JTree[2]) nach HTML erfolgen, zum Beispiel durch die Verwendung von JavaScript. Ein Agentenprogrammierer, der die vorliegende Technik für seine Agenten einsetzen möchte, sollte deshalb nur solche Java^{TM}-GUI Komponenten verwenden, für die es bereits eine Entsprechung im XSLT-Stylesheet gibt. Möchte der Programmierer eine neue Komponente verwenden, kann er das Framework auch entsprechend einfach erweitern.

Die oben vorgestellten Komponenten zum Thema Transport (siehe Abschnitt 8.4) sind gemäß Aufgabenstellung (Abschnitt 1.2) nicht zwingend erforderlich. Besonders dann nicht, wenn jeder SeMoA-Host über einen aktiven Webserver verfügen würde. Genau diese Voraussetzung

[2]Eine Java^{TM}-Swing Klasse, die eine baumartige Struktur erzeugt, vergleichbar mit der baumartigen Darstellung von Verzeichnissen und Unterverzeichnissen von graphischen Datei-Browsern.

soll aber mit der vorgestellten Lösung umgangen werden. Gründe für den Verzicht des Einsatzes eines Webservers könnten beispielsweise im erhöhten Arbeitsspeicherbedarf oder im erhöhten Sicherheitsrisiko für die Plattform liegen.

Das Framework ist im Rahmen des Projekts SeMoA entstanden und dementsprechend damit verflochten. Grundsätzlich ist eine Wiederverwendung an anderer Stelle möglich. Der Aufwand hierfür ist als gering einzustufen.

Analysemodell

In diesem Kapitel soll anhand von Use-Case-Diagrammen, Szenarien und Zustandsdiagrammen eine Verfeinerung des Lösungsansatzes stattfinden. Die verwendeten Diagramme wurden als UML-Diagramme realisiert und versuchen den Empfehlungen der *Object Management Group* (OMG)[1] zu genügen.

9.1 Use-Case-Diagramme

Anhand nachfolgender Use-Case-Diagramme sollen die Entscheidungen aus Kapitel 8 mitbegründet werden. Desweiteren lassen sich hier bereits funktionale Einheiten erkennen, die beim Bilden des Klassenmodells berücksichtigt werden.

In Abbildung 9.1 ist der bisherige Anwendungsfall dargestellt. Der Dialog mit Agenten ist nur auf Java^{TM}-Plattformen möglich.

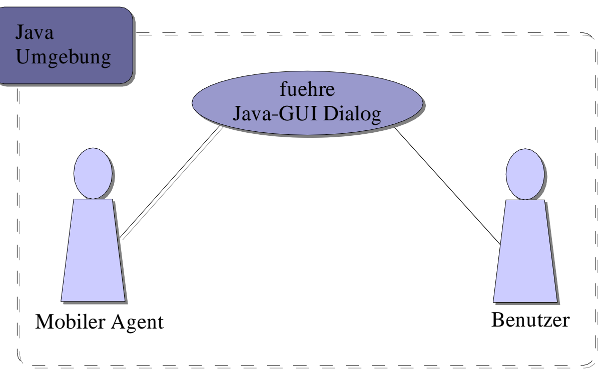


Abbildung 9.1: Use Case Diagramm vom Agenten-Dialog unter Verwendung von Java

[1] Informationen zu UML finden sich auf der Homepage der OMG: http://www.omg.org/uml/ oder unter [18].

Das Übersetzen des Dialogs, von der Java^{TM}-GUI über XML in ein anderes Ausgabeformat, hebt diese Einschränkung auf, wie Abbildung 9.2 zeigt. Das Übersetzen ist grundsätzlich an keine spezielle Programmiersprache gebunden. Ein Entkoppeln der Übersetzungseinheit macht im Zusammenspiel mit SeMoA keinen Sinn, da es eine Vielzahl von frei verfügbaren, in Java^{TM} implementierten Übersetzungseinheiten gibt (siehe Abschnitt 5.2).

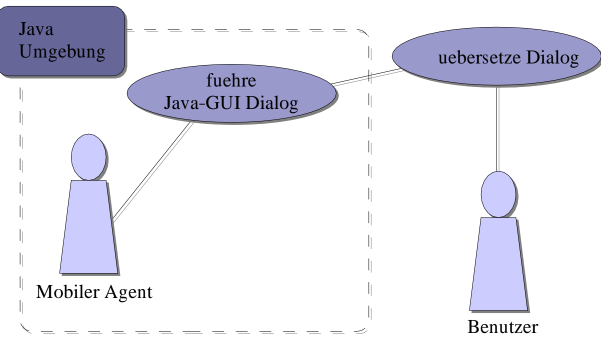


Abbildung 9.2: Use Case Diagramm vom Übersetzten des Agenten-Dialogs

In Abbildung 9.3 ist das Weiterleiten einer Anfrage zwischen zwei SeMoA-Hosts dargestellt. Neben der Verwaltung der Agenten-GUIs soll auch die Kontrolle über die Zugriffsrechte stattfinden.

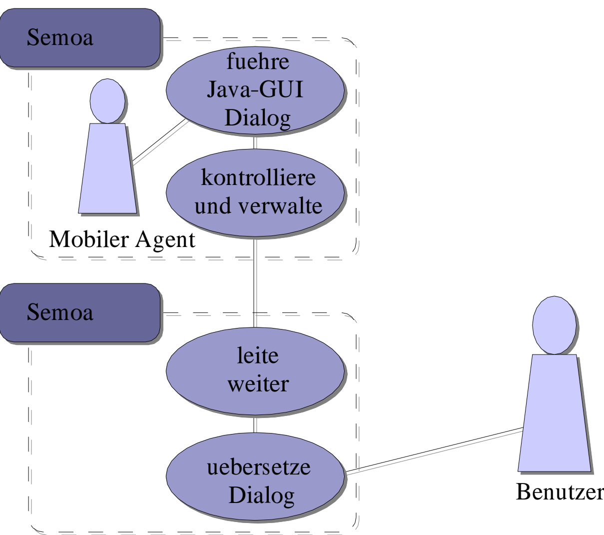


Abbildung 9.3: Use Case Diagramm vom Übersetzten und Weiterleiten des Agenten-Dialogs

Der Teil, der für die Weiterleitung zuständig ist, ist in Abbildung 9.4 detaillierter dargestellt. Die Komponenten sollen neben dem Weiterleiten der GUI-Beschreibung auch alle im Netzverbund verfügbaren SeMoA-Hosts liefern.

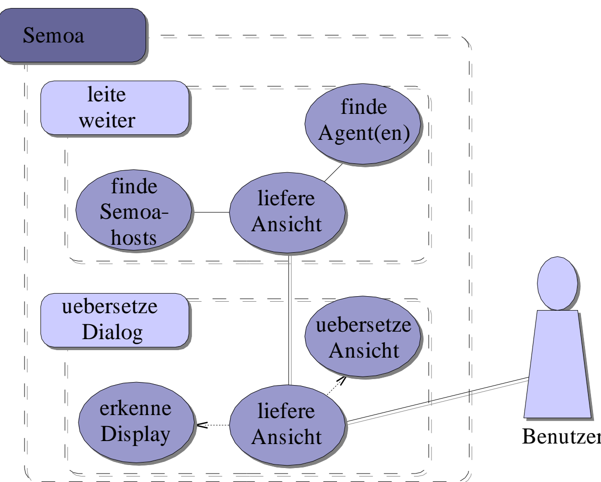


Abbildung 9.4: Detailliertes Use Case Diagramm vom Weiterleiten des Agenten-Dialogs

Zu den Aufgaben der Übersetzerkomponenten gehört neben dem Übersetzen der GUI-Beschreibung auch das Erkennen der Display-Eigenschaften und die damit verbundene Anpassung der Übersetzung. Die Einheiten zur Übersetzung der GUI samt Kontrolle ist in Abbildung 9.5 zu sehen.

Damit die Kontroll- und Verwaltungseinheit auf Agenten zugreifen kann, ist es nötig, dass sich diese beim erstmaligen Start, beim Migrieren oder beim Beenden an- und abmelden, also registrieren und deregistrieren.

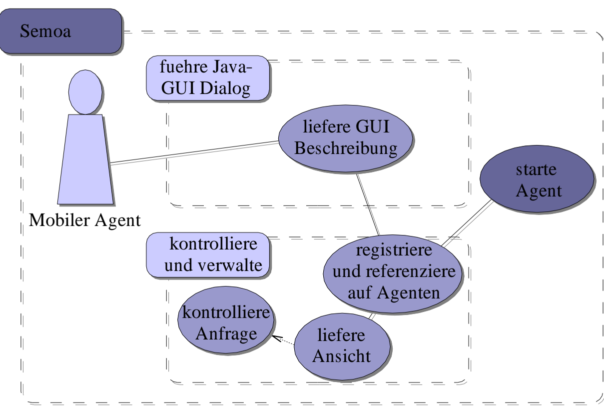


Abbildung 9.5: Detailliertes Use Case Diagramm vom Übersetzten des Agenten-Dialogs

Neben dem Darstellen der aktuellen GUI ist es nötig auch auf Benutzereingaben zu reagieren. Dazu müssen diese der Java^{TM}-GUI zugestellt werden. Abbildung 9.6 zeigt den Use Case der die Eingaben eines Benutzers über die entsprechenden Module bis zur Java^{TM}-GUI weiterreicht. In diesem speziellen Fall befinden sich Agent und die Einheit zum Übersetzen auf dem gleichen SeMoA-Host.

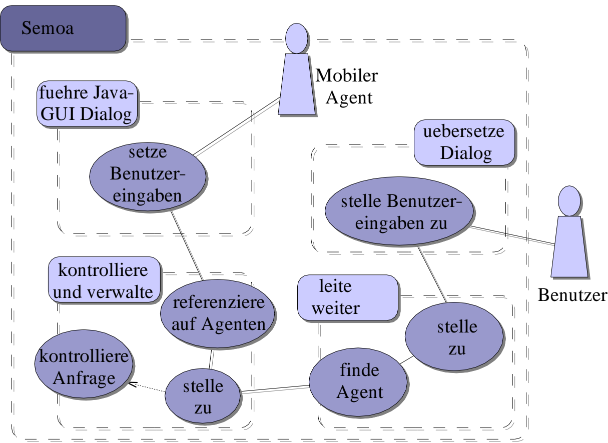


Abbildung 9.6: Detailliertes Use-Case-Diagramm vom Weiterleiten der Benutzereingaben

Für die Java^{TM}-GUI ist es unwesentlich woher die Benutzereingabe kommt. Es ist möglich das Ausfüllen von Input-Feldern oder das Drücken von Buttons einer Java^{TM}-GUI per Methodenaufruf zu simulieren. Die Logik der Weiterverarbeitung dieser Eingaben ist durch die bestehende Java^{TM}-GUI vorgegeben. Ändert sich beispielsweise das aktuelle Fenster, so erfährt das der Benutzer durch ein erneutes Erfragen der aktuellen Agenten-GUI. Sinnvollerweise sollte dies automatisch im Anschluss an das Übermitteln der Benutzerdaten erfolgen. Ein Benachrichtigen des Benutzers, genauer seines Anzeigegerätes, ist zumindest bei Verwendung der Protokolle HTTP und WAP nicht möglich, siehe hierzu auch den letzten Absatz in Abschnitt 3.3.

9.2 Klassendiagramme

In diesem Abschnitt soll durch Klassendiagramme eine weitere Annäherung an die Implementierung erfolgen. Grundlage bilden die Erkenntnisse aus den in Abschnitt 9.1 erarbeiteten Use-Case-Diagrammen.

Die in Abbildung 9.6 dargestellten funktionalen Bereiche *fuehre Java- GUI Dialog*, *kontrolliere und verwalte*, *leite weiter* und *uebersetze Dialog* werden im Folgenden englischen Begriffen zugeordnet, die in dieser Form auch die Java^{TM}-*Package*-Namen bilden werden.

Anhand von Interfaces, deren Namen ebenfalls an englische Begriffe angelehnt sind, sollen nun für jede funktionale Einheit die wichtigsten Komponenten dargestellt werden. In Tabelle 1.1 wird die Bedeutung der Symbole für nachfolgende UML-Diagramme festgelegt.

Wie in Abschnitt 8.1 bereits angesprochen, soll es möglich sein, Benutzerschnittstellen wie gewohnt in Java^{TM}zu erstellen. Es soll aber auch möglich sein, den Aufbau dieser GUI-Klassen

Use-case Name	*package* Name
fuehre Java- GUI Dialog	`agentgui`
kontrolliere und verwalte	`control`
leite weiter	`guiExchange`
uebersetze Dialog	`servlet`

Tabelle 9.1: Zuordnung der *package* Namen

in Form einer XML-Beschreibung zu erhalten. Aus dem Abschnitt 3.2 geht hervor, dass es bei Java^{TM}-GUI Klassen immer eine Komponente gibt, die die restlichen Komponenten enthält. Diese *Top*-Komponente soll Auskunft über den eigenen Inhalt geben und erhält deshalb das Interface `I_TopAgentGUI`, siehe Abbildung 9.7. Der Methodenaufruf `getJDOMDocument()` liefert die erwähnte XML-Beschreibung in einer Form, die vom XML-Prozessor verwendet werden kann.

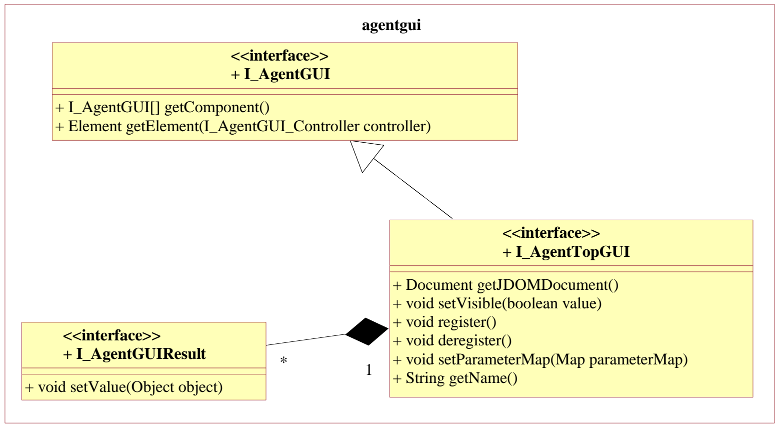


Abbildung 9.7: Klassendiagramm der `agentgui` Interfaces

Das Interface `I_AgentGUI` muss jede GUI-Klasse implementieren, die für die vorliegende Arbeit verwendet werden soll. Die Methode `getComponent()` liefert ein Array von `I_AgentGUI`-Klassen, falls die betreffende Komponente Elternelement von einer oder mehreren Komponenten des Typs `I_AgentGUI` ist. Die Methode `getElement()` liefert eine XML-Beschreibung der Komponente selbst und ihrer Attribute vom Typ `org.jdom.Element`.

Die Klassen aus `org.jdom` bieten Unterstützung im Umgang mit XML. So steht beispielsweise die Klasse `org.jdom.Element` für ein XML-Element das Teil einer XML-Datei sein kann (vgl. Abschnitt 5.1).

Das Interface `I_AgentGUIResult` beschreibt solche GUI Klassen, die eine Werteeingabe oder eine Aktion erwarten, wie das Drücken eines Buttons oder die Texteingabe in ein Texteingabefeld. `I_AgentGUIResult` Objekte erhalten die Werte von ihrer `I_TopAgentGUI`

Instanz, die ihrerseits die Werte über die Methode `setParameterMap()` erhält und an die betreffenden `I_AgentGUIResult` Komponenten durch Verwendung der Methode `setValue()` weiterverteilt.

Die Methoden `register()` und `deregister()` von `I_TopAgentGUI` stellen die Verbindung zum Interface `I_AgentGUI_Collector` dar. Deshalb muss `register()` die Methode `makeController()` von `I_AgentGUI_Collector` aufrufen und `deregister()` sein Äquivalent aus Abbildung 9.8. Beim Registrieren erstellt `I_AgentGUI_Collector` ein neues Object vom Typ `I_AgentGUI_Controller`, den individuellen Vertreter eines GUI Agenten. Die Argumente dienen zur Identifikation des Agenten durch dessen `AgentCard`, einer Klasse die als Visitenkarte eines Agenten zu verstehen ist, und dem Hashcode der aktuellen GUI.

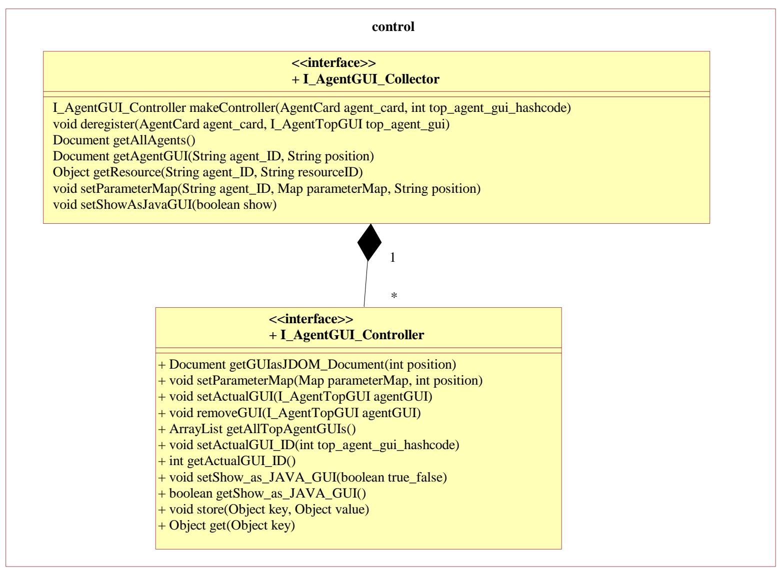


Abbildung 9.8: Klassendiagramm der `control` Interfaces

An dieser Stelle sei angemerkt, dass ein Agent mehrere `I_TopAgentGUI` Klassen zeitgleich betreiben kann, beispielsweise indem er mehrere GUI Fenster öffnet. Allerdings ist immer nur ein Fenster das Aktive. Die Inhalte in den Fenstern können sich jedoch beliebig ändern, während das Fenster, also das `I_TopAgentGUI` bis zum Schließen immer das Gleiche bleibt.

Der `I_AgentGUI_Collector` erstellt wie bereits erwähnt die `I_AgentGUI_Controller` und kann deshalb auch Auskunft über alle registrierten Agenten geben - `getAllAgents()`. Der Rückgabewert vom Typ `org.jdom.Document` aus dem oben beschriebenen Paket entspricht dem Java^{TM}-Abbild einer vollständigen und wohlgeformten XML-Datei. Die aktuelle GUI liefert `getAgentGUI(String agent_ID, String position)`. Das Argument `position` dient zur Identifizierung der GUI Fenster, falls der Agent mehrere betreibt.

Die Ressourcen eines Agenten in diesem Zusammenhang sind solche, die der Agent seinem Be-

nutzer über die GUI mitteilen kann, üblicherweise `Image` Objekte. Diese Ressourcen können mit `getResource()` beim `I_AgentGUI_Collector` erfragt werden. Dieser ruft dazu die Methode `get()` vom entsprechenden `I_AgentGUI_Controller` auf. Ein Agent speichert seine Ressourcen durch den Aufruf der Methode `store()` seines Assistenten `I_AgentGUI_-Controller`. Dies geschieht während der Selbstbeschreibung eines `I_TopAgentGUI`, ausgelöst durch den Aufruf von `getJDOMDocument()`. Als Ressource gilt dabei alles, das sich nicht in XML beschreiben lässt, beispielsweise ein Bild.

Mit `setShowAsJavaGUI()` vom `I_AgentGUI_Collector` lässt sich festlegen, ob die Darstellung der Java^{TM}-GUI für alle Agenten unterdrückt wird, die Interaktion mit Agenten also nur anhand der transformierten Darstellung möglich ist. Die gleiche Methode beim Interface `I_AgentGUI_Controller` gestattet diese Steuerung für jeden einzelnen Agenten. Der Agent ruft dafür die Methode `getShow_as_JAVA_GUI()` von `I_AgentGUI_Collector` auf.

Das *package* `guiExchange` besitzt mit `I_GUI_Exchanger` eine Komponente, die ankommende Anfragen weiterleitet oder wie bei `getAllHosts()`, das alle SeMoA-Hosts in XML-Form liefert, auch selbst beantwortet. Vor dem Weiterleiten prüft `I_GUI_Exchanger`, ob sich die Anfrage auf den lokalen Host oder einen entfernten Host bezieht. Für Letzteres startet er einen Client, `I_GUI_Exchanger_Client`, der eine Verbindung zu `I_GUI_Exchange_-Server`, dem Server, aufbaut. Der Server startet seinerseits für jede ankommende Verbindung eine Instanz vom Typ `I_Protocol_Server`, wie in Abbildung 9.9 dargestellt. Diese Instanz leitet die Anfragen an `I_AgentGUI_Collector` weiter wie aus Abbildung 9.10 hervorgeht.

`I_GUI_Exchanger_Client` kann beim Versuch eine Verbindung aufzubauen die folgenden Fehlermeldungen (*Exceptions*) erzeugen:

- `UnknownHostException` - Es existiert keine Gegenstelle zu den Verbindungsparametern.
- `IOException` - Es tritt ein Fehler während der Verbindung auf, der dem Verbindungskanal zugeordnet werden kann.
- `ProtocolException` - Der Fehler wurde vom eigenen Übertragungsprotokoll erzeugt (siehe Anhang A.1).

Diese Meldungen werden von `I_GUI_Exchanger` abgefangen und an den Aufrufer seiner Methoden weitergeleitet, mit Ausnahme des Methodenaufrufs `getAllHosts()`, der keine `Exception` erzeugt. Der Aufrufer, das `AgentServlet` fängt die `Exceptions` ab, protokolliert sie und erzeugt gegebenenfalls eine Fehlerseite.

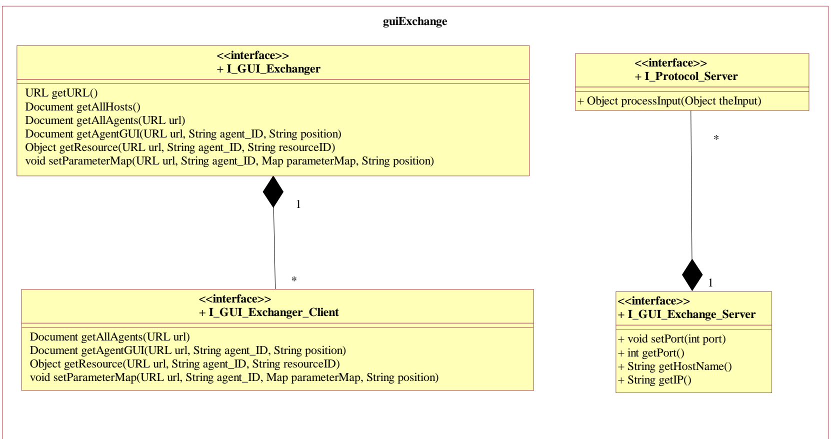


Abbildung 9.9: Klassendiagramm der `guiExchange` Interfaces

Im *package* `servlet`, siehe Abbildung 9.10, erfolgt die Transformation der XML-Daten in das entsprechende Ausgabeformat und die Bearbeitung von Benutzeranfragen (`Request` und `Response`) unter Zuhilfenahme der *Servlet* Technologie.

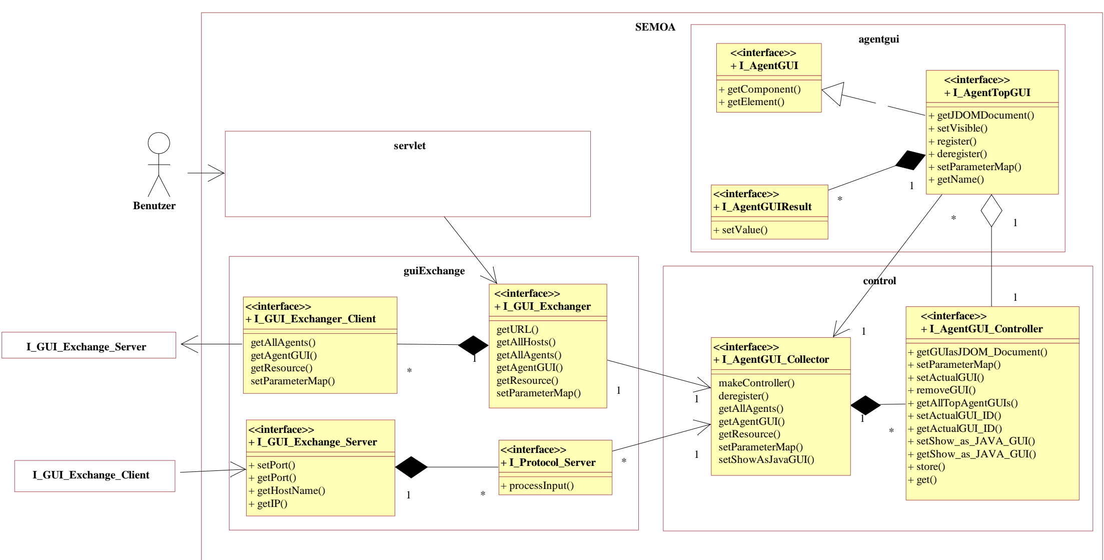


Abbildung 9.10: Klassendiagramm aller Interfaces

Abbildung 9.10 zeigt zudem das Zusammenspiel aller vorgestellten Pakete. Weitere Informationen hierzu und im Speziellen zur Implementierung finden sich im gleichnamigen Kapitel 10.

Implementierung

Dieses Kapitel behandelt die Umsetzung der im letzten Kapitel (9) gewonnenen Kenntnisse.

10.1 Prototyp

Als Entwicklungsumgebung kam *Eclipse*[1] in Version 2.0 zum Einsatz. Die Java^{TM}-Programmierumgebung wurde von IBM entwickelt und ist kostenlos. Zahlreiche nützliche Features erlauben das komfortable Programmieren, wie beispielsweise die automatische Code-Vervollständigung. Eine Schnittstelle unterstützt das Einbinden externer Plugins. Für die Erstellung der UML-Diagramme habe ich das *Slim Modeling Environment UML Plugin* (*Slime UML*)[2] verwendet.

10.2 Aufbau des Prototyps

In diesem Abschnitt werden die Erkenntnisse und Probleme aus der Implementierphase aufgezeigt.

Damit die Umstellung auf das neue Framework für den Agentenprogrammierer so einfach wie möglich ist, eine Weiterverwendung der Java^{TM}-Swing Klassen wegen fehlender Eigenschaften jedoch nicht in Frage kam, bestand die beste Lösung darin, von diesen Klassen abzuleiten. Der Name bleibt dabei erhalten, nur das Package ändert sich. Die neuen Klassen implementieren die bereits vorgestellten Interfaces, wie Abbildung 10.1 zeigt. Gegenwärtig implementiert nur die Klasse `JFrame` das Interface `I_AgentTopGUI`, da sie die am meisten verwendete oberste Behälterkomponente beim Erstellen von GUIs ist.

[1] Eclipse Homepage: http://www.eclipse.org/platform
[2] Slime UML Homepage: http://www.mvmsoft.de/content/plugins/slime/slime.htm

Der Agent, der ebenfalls in der Abbildung 10.1 zu sehen ist, muss beim Aufruf des `JFrame` Konstruktors ein zusätzliches Argument vom Typ `AgentCard` verwenden. Die Übergabe dieser Agenten-*Visitenkarte* ist nötig, um eine Verbindung zwischen Agent und seinen GUI-Klassen herzustellen. Das Ändern des Packagenamens von `javax.swing` nach `agentgui` und das Anpassen der `JFrame`-Konstruktoraufrufe sind Aufgaben des Agentenprogrammierers, um bestehenden Agenten-Code auf das vorgestellte Framework umzustellen.

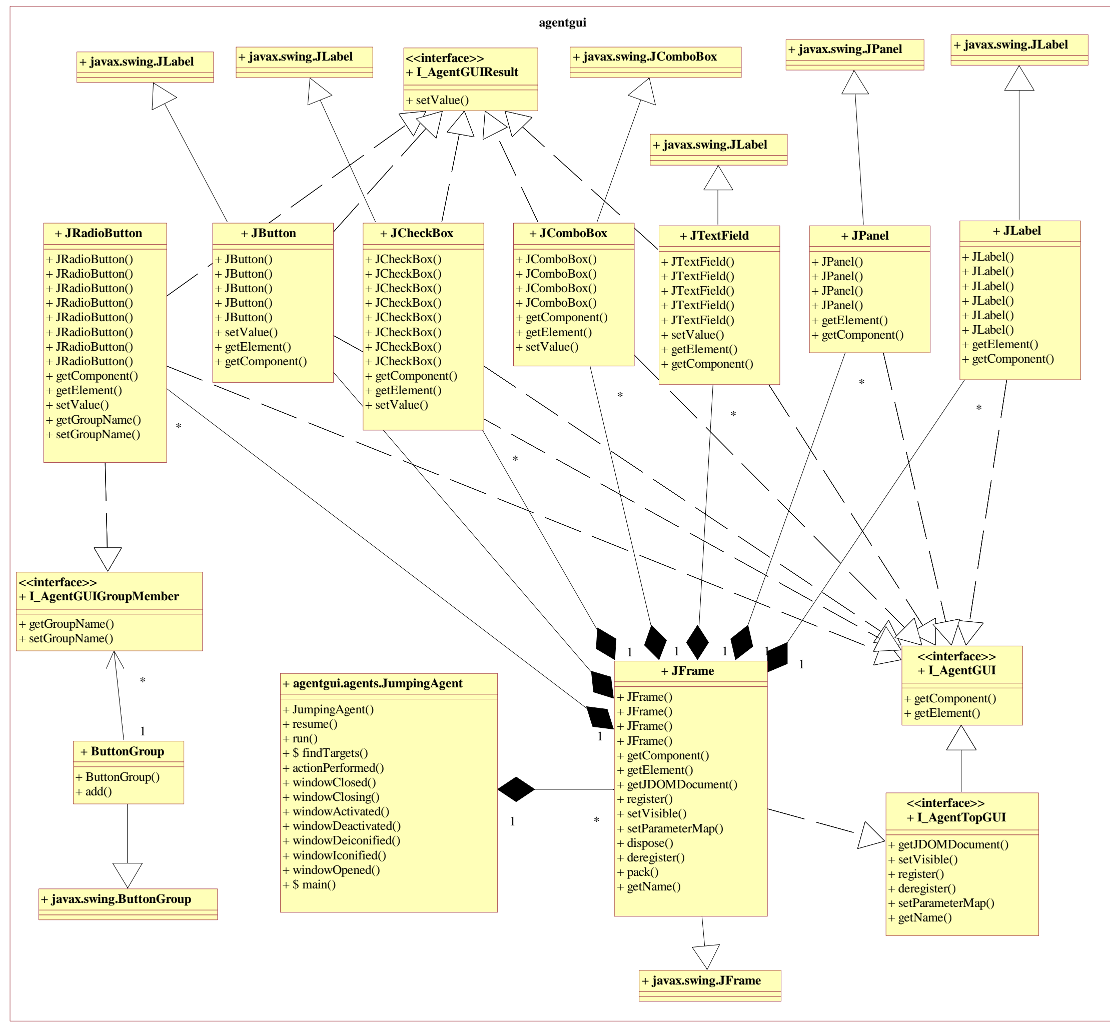


Abbildung 10.1: Klassen aus dem Packet `agentgui`

Die Phase der Systementwicklung war stark iterativ, und führte zu mancher unvorhersehbaren Erweiterung des Analysemodells. Das Interface `I_AgentGUIGroupMember` aus Abbildung 10.1 gehört dazu. In einer Radiobutton-Gruppe kann immer nur ein Button gleichzeitig selektiert sein (*toggle*[3]). Zuvor müssen sich die Buttons dafür bei `javax.swing.ButtonGroup` registrieren. Leider hat ein `javax.swing.JRadioButton` keine Methode, die eine Abfrage der

[3] Aus dem Englischen: to toggle - hin und her schalten, umschalten

Gruppenzugehörigkeit erlaubt. Diese Funktionalität ist aber wichtig. In HTML beispielsweise, ist der oben beschriebene Effekt des Umschaltens nur durch das Zuweisen des selben Gruppennamens bei Radiobuttons zu erreichen. Die von `javax.swing.ButtonGroup` abgeleitete Klasse `agentgui.ButtonGroup` setzt deshalb den Gruppennamen bei allen Klassen des Typs `I_AgentGUIGroupMember`, sobald sich diese bei ihr registrieren.

```
JRadioButton birdButton = new JRadioButton("Bird");
// Group the radio buttons.
ButtonGroup group = new ButtonGroup();
group.add(birdButton);
```

Die Methode `deselectAllJCheckboxes()` der Klasse `JFrame` war nötig, weil HTML nur die Information über die gesetzten Checkboxen eines HTML-Dokuments überträgt. Der umständliche Weg, erst alle Checkboxen kurz vor Erhalt dieser Informationen zu deselektieren um sie dann (vereinzelt) erneut zu selektieren, ist bisher die einfachste Lösung[4].

Die Implementierung der Klassen `AgentGUI_Collector` und `AgentGUIController` verlief ohne nennenswerte Abweichung vom Analysemodell. In Abbildung 10.2 sind alle Komponenten des Packets `control` dargestellt.

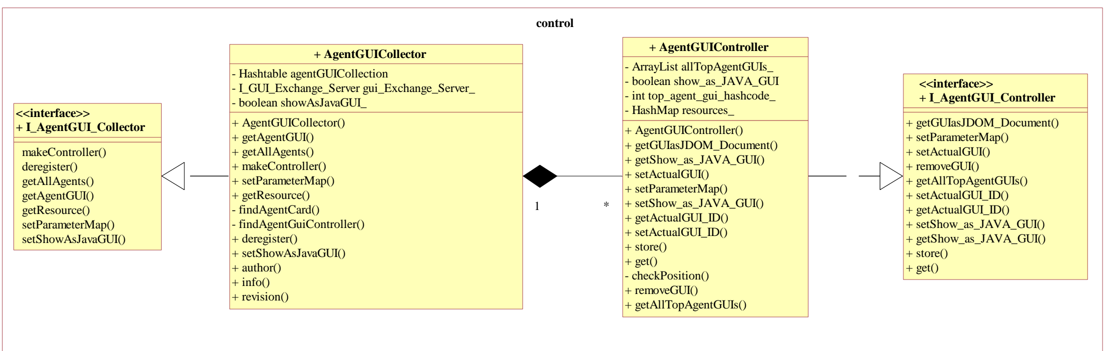


Abbildung 10.2: Klassen aus dem Packet `control`

Die Klasse `GUI_Exchanger` aus Abbildung 10.3 delegiert, wie bereits in Abschnitt 9.2 angesprochen, die an ihn gerichteten Anfragen entweder an `I_AgentGUI_Collector` oder an eine von ihm erstellte neue Instanz der Klasse `GUI_Exchanger_Client`. Diese stellt eine Verbindungsanfrage an die Gegenstelle vom Typ `GUI_Exchanger_Server`, der die Anfrage ablehnen oder annehmen kann. Java^{TM} stellt hierzu die Klassen `java.net.ServerSocket`

[4]Das Framework sieht es bisher nicht vor, Werte von Java^{TM}-GUI Komponenten abzufragen, wie beispielsweise den Status einer Checkbox. Andernfalls könnte eine elegantere Lösung gefunden werden.

und `java.net.Socket` zur Verfügung. `ServerSocket` lauscht an einem frei wählbaren Port und seiner lokalen IP-Adresse auf ankommende Verbindungsanfragen. `Socket` wird verwendet um diese Anfragen zu stellen. Akzeptiert `ServerSocket` eine Anfrage, können über die entstandene Verbindung Daten ausgetauscht werden. Damit der `GUI_Exchanger_Server` mehrere Verbindungen zeitgleich bewältigen kann, war es wichtig ihn *multithreading* fähig zu machen. Für jede akzeptierte Anfrage erzeugt er deshalb einen neuen Thread.

Für den Austausch der Daten zwischen Client und Server war die Entwicklung eines gemeinsamen Übertragungsprotokolls nötig.Die Klassen `Protocol_Client_Impl` und `Protocol_Server_Impl` kapseln diese Funktionalität. Eine detaillierte Beschreibung darüber liefert Anhang A.1, Protokollspezifikation.

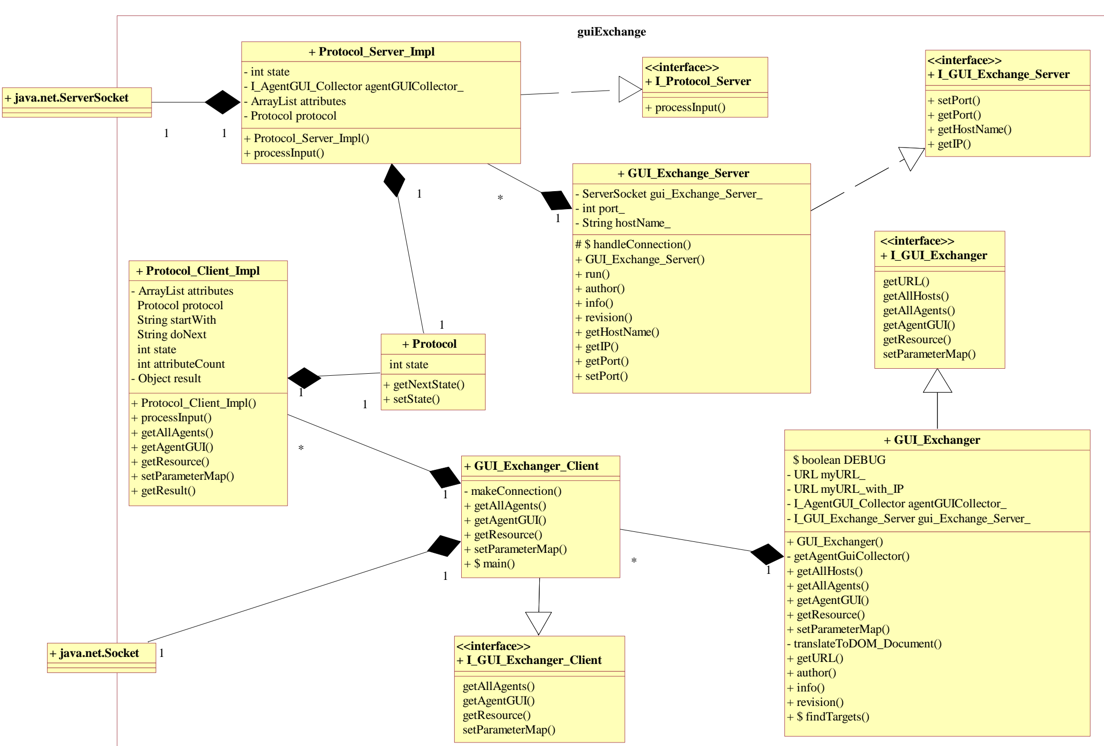


Abbildung 10.3: Klassen aus dem Packet `guiExchange`

Abbildung 10.4 zeigt die Klassen aus dem Packet *servlet*. Die Klasse `AgentServlet` ist von `HttpServlet` abgeleitet. Das Servlet ist in Verbindung mit dem Webserver-Dienst von SeMoA über eine fest definierte URL zu erreichen. Es stellt die graphische Schnittstelle zum Benutzer dar, der dazu einen Browser benötigt. `AgentServlet` untersucht die Anfrage (`HttpServletRequest`) des Benutzers nach Parametern, die ihm Aufschluss über die gewünschte Aktion geben, beispielsweise das Anzeigen aller verfügbaren SeMoA-Hosts [5]. Diese Anfrage leitet er an `I_GUI_Exchanger` weiter. Das Ergebnis in XML-Form übergibt er dem XSLT-

[5] Screenshots finden sich in Abschnitt 11.1

Prozessor (`Transformer`), das passende Stylesheet liefert die Helferklasse `StyleSheet-Finder`. Das Ergebnis der Transformation wird anschließend als Antwort (`HttpServlet-Response`) an den Browser gesendet.

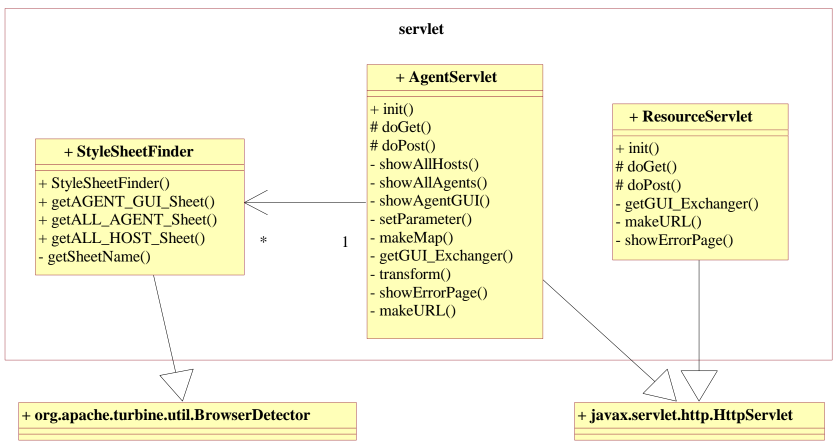


Abbildung 10.4: Klassen aus dem Packet `servlet`

Die Klasse `ResourceServlet` ist ebenfalls ein Servlet und hat die Aufgabe Anfragen an die Ressourcen eines Agenten weiterzuleiten. Der Aufruf dieses Servlets erfolgt automatisch durch den Browser beim Nachladen aller, im angeforderten Meta Code, beispielsweise HTML, enthaltenen Referenzen. Folgender Ausschnitt aus einer HTML-Datei zeigt diese Referenz in Form eines *image tags*.

```
<img src="servlet.ResourceServlet?type=jpg&id=14823211
 &agent_id=DJHFBLAIAHBMGGFMIECJDPAEBNNKCHPPOHNLCODL
 &host=192.168.0.101&port=20002">
```

Die Parameter *type, id, agent_id, host* und *port* verwendet das `ResourceServlet` als Argumente für den Methodenaufruf `getResource()` von `I_GUI_Exchanger`.

Abbildung 10.6 zeigt das Klassendiagramm des gesamten Frameworks mit allen Paketen und dazugehörenden Klassen.

10.3 Integration in SeMoA

Das in Abschnitt 6.2 vorgestellte Konzept der Dienste erlaubte das problemlose Einbinden der Klassen `GUI_Exchanger`, `GUI_Exchange_Server` und `AgentGUICollector`. Voraussetzung ist das Ableiten von der abstrakten Klasse `AbstractService`, wie Abbildung

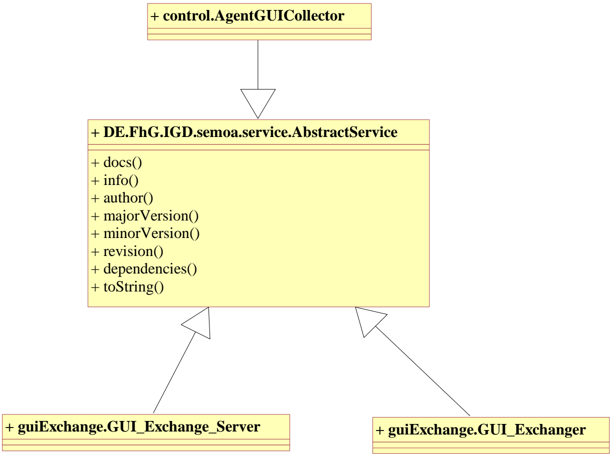


Abbildung 10.5: Integration als Dienst in SeMoA

10.5 zeigt. Der Zugriff auf einen Dienst kann nur über die Klasse `Environment` erfolgen. Die Methode `lookup()` liefert hierfür die Referenz auf den betreffenden Dienst.

Durch Skripte lässt sich in SeMoA der Start von Diensten automatisieren. Im Anhang B finden sich alle Skripte, innerhalb derer die für diesen Prototypen benötigten Dienste gestartet und konfiguriert werden.

Abschließend bleibt noch zu bemerken, dass die Erstellung des Quellcodes nach den SeMoA *Code Conventions* [14] erfolgt und englisch dokumentiert ist.

Abbildung 10.6: Alle Klassen des Frameworks

Teil III

Resümee

Evaluation und Ausblick

Dieses Kapitel stellt den Abschluss der vorliegenden Arbeit dar. Ihre Funktionsweise wird aus Anwendersicht dargestellt, offene Punkte der Frameworks werden beschrieben und bisherige Ergebnisse zusammengefasst.

11.1 Funktionsweise des Frameworks

Anhand einiger Screenshots wird im Folgenden die Funktionsweise des Frameworks beschrieben. Der Benutzer kann nun über die verschiedensten Browser auf den unterschiedlichsten Plattformen das *Agenten Servlet* anwählen. Er erhält als Startseite eine Übersicht aller verfügbaren SeMoA-Hosts, wie Abbildung 11.1 zeigt.

Abbildung 11.1: Browserdarstellung aller SeMoA-Hosts

Entscheidet er sich für einen Host durch Anklicken des Hyperlinks, erhält er eine Übersicht aller Agenten dieses Hosts, siehe Abbildung 11.2.

Abbildung 11.2: Browserdarstellung aller SeMoA-Agenten eines Hosts

Wählt er einen davon aus, sieht er die, ins entsprechende Endgeräte-abhängige Ausgabeformat (hier HTML) transformierte, aktuelle Benutzerschnittstelle und kann diese wie gewohnt bedienen. Eingaben und Aktionen werden an den Agenten weitergeleitet. Auf Benutzeraktionen folgende Veränderungen der GUI sind sofort sichtbar.

Abbildung 11.3 zeigt die HTML Umsetzung der Java^{TM}-GUI aus Abbildung 11.4.

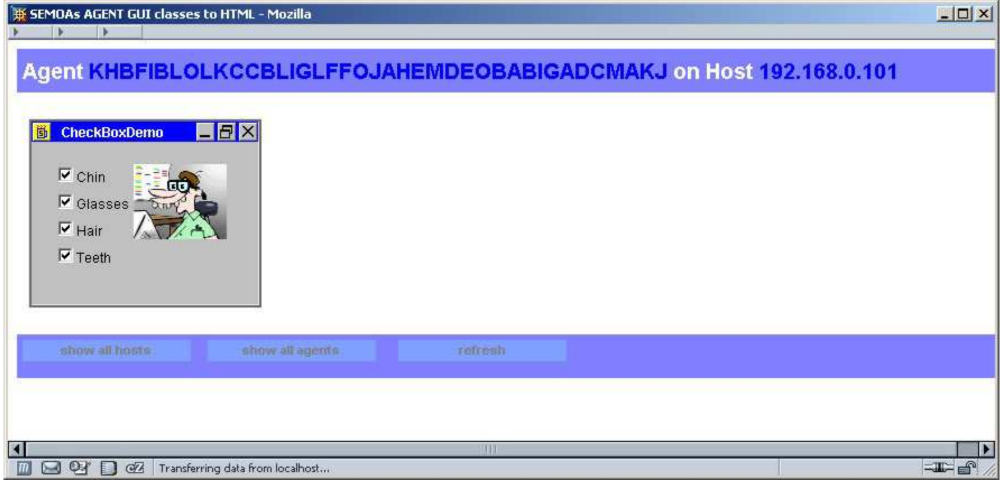


Abbildung 11.3: Browserdarstellung einer Agent-GUI mit Checkboxen

Abbildung 11.5 zeigt den `JumpingAgent`, einen Testagenten, der auf Knopfdruck auf entfernte Hosts migriert. Seine Java^{TM}-GUI ist in Abbildung 11.6 zu sehen. Drückt der Benutzer den Jump-Button, versucht der Agent auf die angegebene SeMoAPlattform zu migrieren und

Abbildung 11.4: Java^{TM}-Darstellung einer Agent-GUI mit Checkboxen

ist im Erfolgsfall nicht mehr in der Liste der lokal verfügbaren Agenten zu sehen. Verfügt der Zielhost ebenfalls über das Agent-GUI-Framework, so kann der Benutzer die Interaktion mit seinem Jumping-Agent dort fortsetzen.

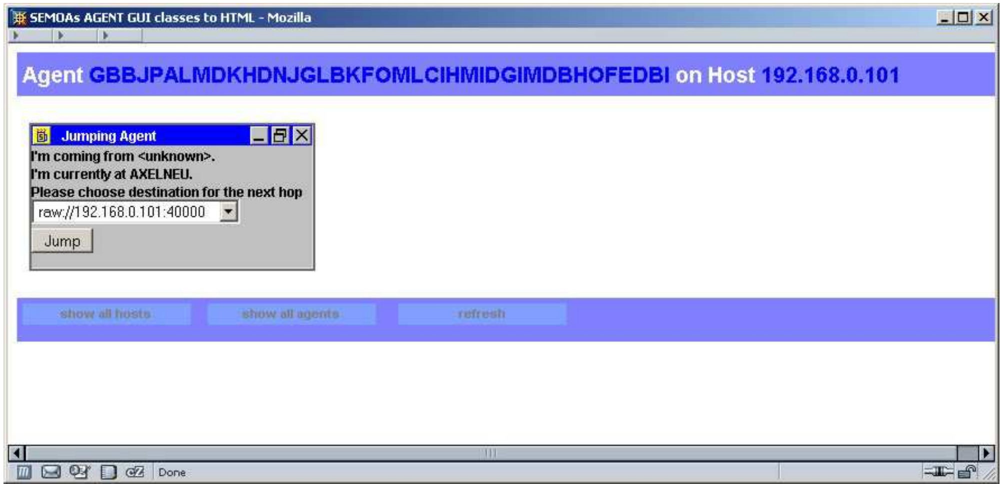


Abbildung 11.5: Browserdarstellung der `JumpingAgent`-GUI

Abbildung 11.6: Java^{TM}-Darstellung der `JumpingAgent`-GUI

11.2 Kurze Evaluation

Die Performance bei der Umsetzung von Java^{TM}-GUI in HTML lässt sich augenscheinlich beurteilen, indem beide GUIs parallel betrieben werden. Eine Benutzeraktion innerhalb der

HTML-GUI löst unmittelbar die gleiche Aktion wie bei der Java^{TM}-GUI-Variante aus, beispielsweise das Drücken eines Buttons. Die Zeit die es nun dauert, bis die neue HTML-GUI im Browser erscheint, gibt Aufschluss über die Performance. Keinen Einfluss hat das Framework dabei auf die Zeiten, die durch die Übertragung im Netzwerk und durch das Rendern der Graphik durch den Browser entstehen. Die Zeit, die beim GUI-Framework von der Anfrage am Servlet bis zum Liefern der (HTML-) Daten durch das Servlet vergeht, betrug im Mittel 300 Millisekunden. Eine Verzögerung in dieser Größenordnung ist vertretbar, zumal keine Anforderungen an das System hinsichtlich besonders kurzer Zugriffszeiten bestehen.

11.3 Ausblick

Im Folgenden sind einige Ergänzungen aufgeführt, die ich aus zeitlichen Gründen nicht realisieren konnte, die das Framework allerdings noch verbessern würden.

Die im Use-Case Diagramm 9.3 dargestellte Kontrolle der Zugriffsrechte (Autorisation des Nutzers gegenüber dem GUI-Framework) ist bisher nicht integriert. Eine mögliche Lösung könnte ein Authentifizierungs-Servlet sein, das den Benutzernamen und Passwort an SeMoA zur Überprüfung weiterleitet. Wird der Zugriff gestattet, gilt dieser für die aktive Sitzung (*Session*) zwischen Benutzer und Servlet, gekennzeichnet durch eine Session-Id[1]. Zusätzlich muss der Dienst `AgentGUI_Collector` um die Funktionalität erweitert werden, eine benutzerindividuelle Sicht auf die verfügbaren Agenten zu gewähren.

Der Dienst `GUI_Exchanger` liefert bei der Anfrage nach den erreichbaren SeMoA-Hosts momentan nur die statischen Einträge aus einem XML-File. In SeMoA existiert ein Dienst mit dem Namen *Vicinity* der eine aktuelle Liste der verfügbaren SeMoA-Hosts liefert. Da sich die Zusammensetzung von Agentenplattformen im Netzwerk ständig verändert, schlage ich das Einbinden dieses Dienstes vor.

Es ist außerdem notwendig die vorhandenen Stylesheets konsequenter an die verschiedenen Browserversionen anzugleichen. Die Anpassung an kleine Display-Grössen stellt hierbei sicherlich eine große Herausforderung dar. Anspruchsvoll ist hierbei die Logik, die für das übersichtliche Neuanordnen von graphischen Komponenten notwendig ist, um automatisch erscheinende Scrollbalken bei der Browseransicht zu vermeiden. Die Erstellung eines Stylesheets für das Ausgabeformat WML, das gerade kleine Display-Grössen unterstützt, kann diesbezüglich neue Erkenntnisse liefern.

Die Verwendung einer DTD habe ich vermieden, da innerhalb von SeMoA die Erzeugung der betreffenden Agent-GUI XML-Tags und deren Verwendung in den XSLT-Stylesheets eng ge-

[1] Eine temporär vereinbarte Zeichenfolge, die zur Identifikation der Session, gekapselt in den Anfragen und Antworten zwischen Browser und Servlet, übermittelt wird.

koppelt ist. Trotzdem könnte der Einsatz einer DTD in bestimmten Fällen, beispielsweise im Kontext des Zusammenspiels verschiedener Agentensysteme, Sinn machen.

Die im Anhang A.1 vorgestellten Client/Server-Klassen mit dazugehöriger Protokollspezifikation benötigen einer Überarbeitung. Sie vollführen zwar fehlerfrei ihre Aufgabe, der Code ist jedoch teilweise redundant und schwer verständlich.

Bei Verwendung der überschriebenen Java^{TM}-Swing-Klasse `JFrame` muss der Benutzer ein zusätzliches Argument, `AgentCard` mit angeben. Die Notwendigkeit entstand, trotz gemeinsamer `ThreadGroup` und der SeMoA-Erweiterung, die es Klassen innerhalb der `ThreadGroup` gestattet auf den gleichen Kontext zuzugreifen. Eine Instanz von `JFrame` kann deshalb auf `AgentCard` zugreifen. Das Problem ist aber folgendes: wenn der Benutzer über das Servlet beispielsweise auf einen Button drückt, der einen Event auslöst, der wiederum die Erzeugung einer neuen Instanz von `JFrame` zur Folge hat, so wird diese neue Instanz nicht in der Agenten-`ThreadGroup` ausgeführt und kann deshalb nicht auf den Agentenkontext und die darin enthaltene `AgentCard` zugreifen.

Der Vorteil einer Lösung dieses Problems, kombiniert mit dem automatischen Laden der Agenten-GUI-Klassen anstelle der Swing-Klasse über den SeMoA eigenen Classloader, liegt darin, dass dann eine Verwendung der Agenten-GUI für den Agentenprogrammierer vollkommen transparent wäre.

Es ist mit dem vorgestellten Framework nicht nur möglich, die XML-Beschreibung der Agenten-GUI in HTML oder WML zu transformieren, es kann an anderer Stelle durch die Nachbildung der ursprünglichen Klassen auch wieder Java(TM)-GUI-Code generiert und ausgeführt werden. Allerdings würden die gebildeten Klassen die Möglichkeiten (beispielsweise) einer HTML-GUI nicht übertreffen. Ähnlich wie einem Browser beruht die Interaktion mit dem Agenten auf der Übermittlung der GUI-Beschreibung und Events vom bzw. zum Agent-Servlet.

11.4 Schlussbemerkung

SeMoA ist wieder um einen Baustein reicher geworden. Das vorgestellte Framework ermöglicht nun die Interaktion mit SeMoA-Agenten von den verschiedensten Geräten aus. Die einzige Voraussetzung für solche Geräte ist die Fähigkeit, Internet-Inhalte darstellen zu können, etwa durch einen Internet-Browser. Das Framework transformiert dafür GUI-Daten in Ausgabeformate wie HTML oder WML. Es ist aber bezüglich der Ausgabeformate durch Verwendung der Technik XSLT frei erweiterbar. Die mit diesem Framework realisierte Web-Anbindung erweitert damit das Anwendungsspektrum der SeMoA-Agenten-Technologie.

Teil IV

Anhang

Protokollspezifikation

A.1 Protokollspezifikation

Im Folgenden wird anhand von UML-Sequenz-Diagrammen der Ablauf einer Client-Server Transaktion dargestellt.

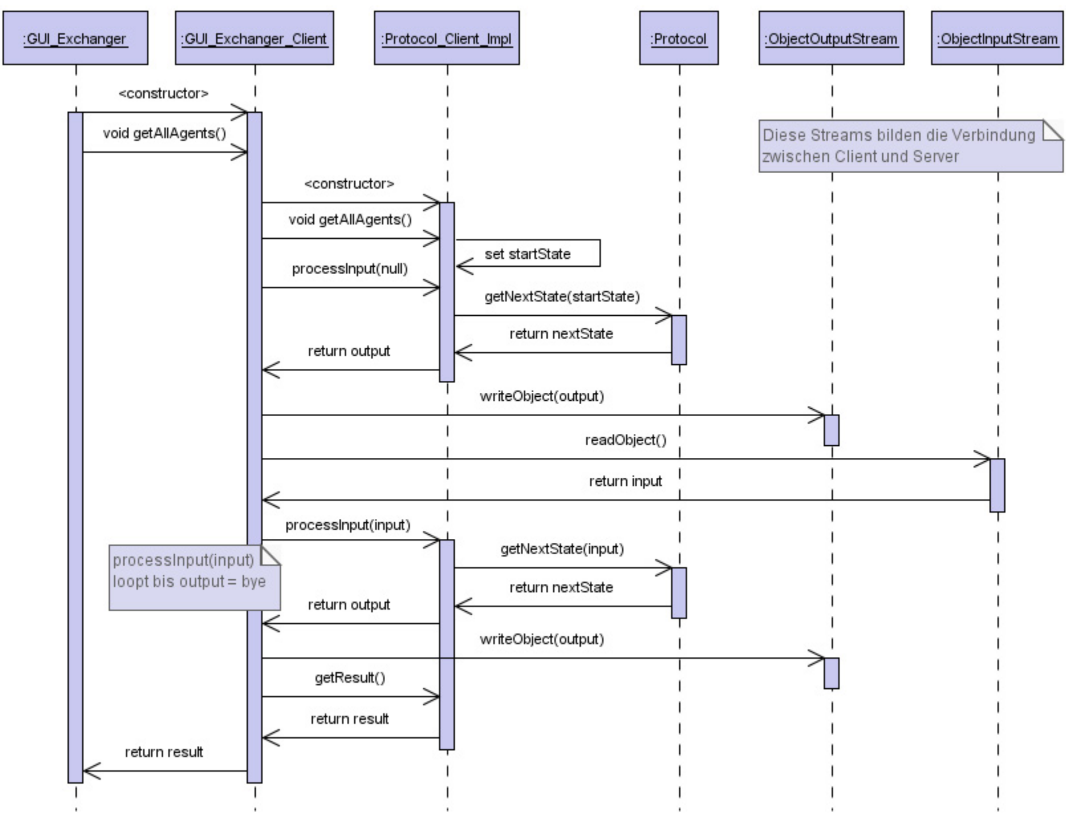


Abbildung A.1: Sequenz-Diagramm der `guiExchange.protocol` Client-Klassen

Abbildung A.1 zeigt, exemplarisch am Methodenaufruf `getAllAgents()` von `GUI_Exchanger`, den Ablauf einer Client-Server-Anfrage. Der Ablauf ist dabei analog zu den Aufrufen von `getAgentGUI()`, `getResource()` und `setParameterMap()`. Unterschiedlich ist der Anfangszustand (*startState*), der durch den namensgleichen Methodenaufruf von `GUI_Exchanger_Client` bei `Protocol_Client_Impl` festgelegt wird. Die Helfer-Klasse `Protocol` ordnet jedem *Input* einen entsprechenden Zustand zu. Den Zustand liefert sie als Rückgabewert beim Aufruf der Methode `getNextState()`. `Protocol_Client_Impl` merkt sich diesen Zustand und reicht ihn als Rückgabewert beim erstmaligen Methodenaufruf von `processInput()` an `GUI_Exchanger_Client` weiter. Von dort wird er der Methode `writeObject()` als Argument übergeben und kommt so auf der Server-Seite an. (Der `ObjectInputStream` des Servers entspricht dem `ObjectOutputStream` des Clients, und umgekehrt.) Der Client wartet auf eine Antwort des Servers. Sobald er sie erhält, übergibt er diese an die Klasse `Protocol_Client_Impl` durch erneuten Aufruf der Methode `processInput()`. Dieser Antwort ordnet die Klasse `Protocol` erneut einen Zustand zu. Dem Zustand entsprechend bestimmt `Protocol_Client_Impl` Rückgabewerte von `processInput()`. War der Rückgabewert keine Abbruchsequenz (hier *bye*), wiederholt sich das Schreiben und Lesen auf den Stream-Objekten gefolgt vom Aufruf der Methode `processInput()`, u.s.w.. Bei Erhalt der Abbruchsequenz quittiert `GUI_Exchanger_Client` diese beim Server und holt sich das Ergebnis der Transaktion von `Protocol_Client_Impl`. Dieses Ergebnis ist gleichzeitig der Rückgabewert auf den Methodenaufruf `getAllAgents()` von `GUI_Exchanger`.

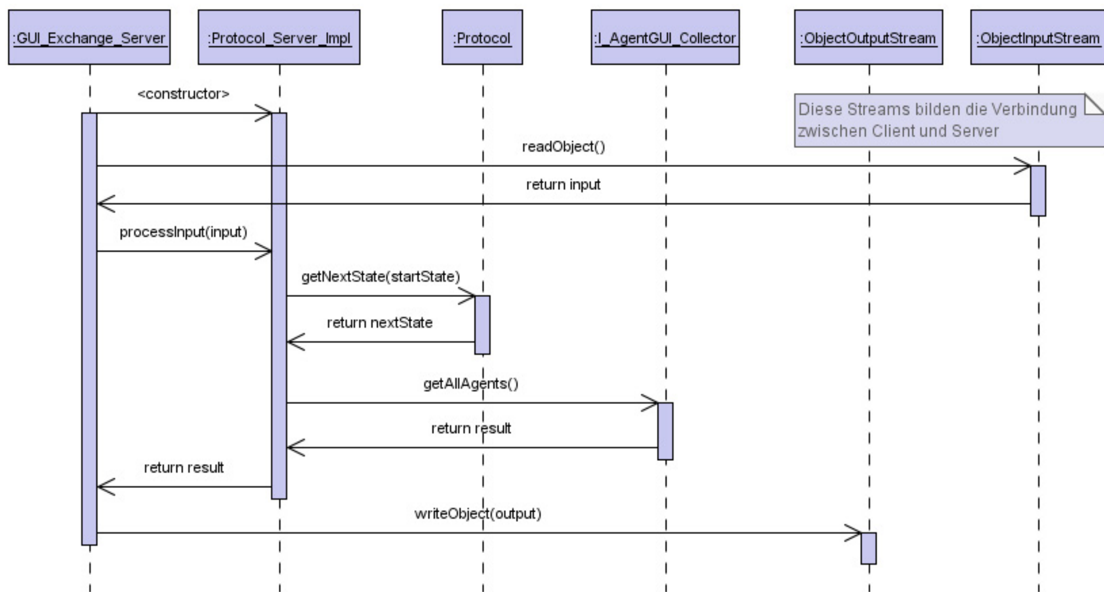


Abbildung A.2: Sequenz-Diagramm der `guiExchange.protocol` Server-Klassen

Der Ablauf auf der Serverseite beim `GUI_Exchange_Server` verhält sich ähnlich. Der Ser-

ver (siehe Abbildung A.2) lauscht am `ObjectInputStream` auf Anweisungen. Diese leitet er an die Klasse `Protocol_Server_Impl` weiter. Von `Protocol` erhält er auch einen dem Input zugeordneten Zustand, der ihn entweder dazu veranlasst, gleich den entsprechenden Methodenaufruf auf den Typ `I_AgentGUI_Collector` abzusetzen, oder erst die Anzahl nötiger Argumente für einen darauf folgenden Methodenaufruf zu sammeln. Beim Sammeln der Argumente muss der in Abbildung A.3 dargestellte Ablauf wiederholt werden, bis schließlich die Anweisung für den Methodenaufruf folgt. `Protocol_Server_Impl` generiert daraus den gewünschten Aufruf und übergibt das Ergebnis an `GUI_Exchange_Server`, der dieses dann in den `ObjectOutputStream` schreibt.

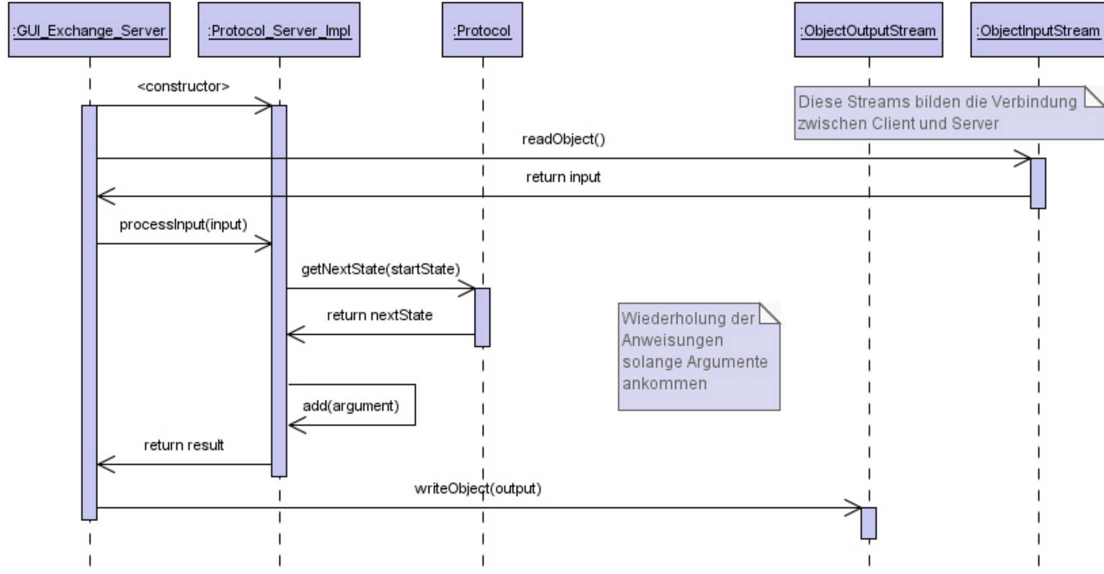


Abbildung A.3: Sequenz-Diagramm der Server-Klassen: Sammeln der Methoden Argumente

Das gemeinsame Protokoll legt den Ablauf der Transaktion fest. Der Client initiiert den Aufruf, und teilt dem Server die Anforderung mit. Der Server kennt die vereinbarten Zeichenfolgen und ordnet sie bestimmten Anweisungen zu. Die bereits genannte Anweisung zum Sammeln der Argumente geht dabei all denen Methodenaufrufen voraus, die auch Argumente benötigen.

A.2 Übersicht der Agent-GUI-Klassen

Die nachfolgende Tabelle zeigt den gegenwärtigen Stand der GUI Klassen im Packet `agentgui`. Sie sind alle von ihren namensgleichen Swing-Klassen abgeleitet.

Name
`JButton`
`JCheckBox`
`JComboBox`
`JFrame`
`JLabel`
`JPanel`
`JRadioButton`
`JTextField`
`ButtonGroup`

Tabelle A.1: Bisherige Agent-GUI-Klassen

An dieser Stelle sei auf die einfache Erweiterung dieser Auswahl hingewiesen. Neben dem Ableiten von der Swing-Klasse ist gegebenenfalls die Anpassung der Helfer-Klasse `AgentGUI_Helper` notwendig. Der `AgentGUI_Helper` ruft bestimmte Methoden der Swing-Klassen auf, um deren Attribute zu erfahren. Enthält eine neue Swing-Klasse zusätzliche, neue Attribute, muss `AgentGUI_Helper` angepasst werden. Zusätzlich ist auch eine Anpassung der vorhandenen XSLT-Stylesheets notwendig, beispielsweise von dem, das die HTML-Ausgabe bewirkt (abgebildet in Anhang C.4).

Initialisierungsdateien für SeMoA

B.1 demo1.conf

```
#
# Original Version
#
# Author  : Volker Roth
# Revision: $Revision: 1.2 $
# Date    : $Date: 2001/07/06 21:03:47 $
#

setenv SSL_PORT          40010
setenv RAW_PORT          40000
setenv HTTPD_PORT        8080
setenv GUI_EXCH_SRV_PORT    20002
setenv SHOW_AS_JAVA_GUI     false
```

B.2 rc.network

```
#
# Original Version
#
# Boots and publishes vital network services such as
# the ingates and outgates as well as the vicinity
# service (optional).
```

```
#
# Launching the vicinity daemons can be disabled by
# setting ${VICINITY} to 'disable', and enabled
# by setting it to 'enable'. SSL gateways can be
# enabled and disabled by means of ${SSL}. Both
# variables should be set in your 'rc.conf' file.
#
# Author  : Volker Roth
# Revision: $Revision: 1.15 $
# Date    : $Date: 2002/03/28 09:07:16 $
#

alias ingate java DE.FhG.IGD.semoa.bin.Publish -detach
                                          -spawn -proxy none

echo
echo "Network services"
echo "----------------"
echo -n "Publishing gateways:"

echo -n " outgate"
publish DE.FhG.IGD.semoa.server.OutGate ${WhatIs:OUTGATE}

echo -n " ingate"
publish DE.FhG.IGD.semoa.server.InGate ${WhatIs:INGATE}

echo -n " raw"
publish DE.FhG.IGD.semoa.net.RawOutGate ${WhatIs:OUTGATE}/raw

${SSL} echo -n " raws"
${SSL} publish DE.FhG.IGD.semoa.net.SSLRawOutGate
                                       ${WhatIs:OUTGATE}/raws

echo "."

echo -n "Launching ingates:"

echo -n " raw"
ingate -key ${WhatIs:INGATE}/raw
        -class DE.FhG.IGD.semoa.net.RawInGate ;
```

```
        -setPort ${RAW_PORT} -setCapacity 3
        -setMax 65535 -run

${SSL} echo -n " raws"
${SSL} ingate -key ${WhatIs:INGATE}/raws
        -class DE.FhG.IGD.semoa.net.SSLRawInGate ;
        -setPort ${SSL_PORT} -setCapacity 3 -setMax 0 -run

echo "."

echo -n "Network:"
${VICINITY} echo -n " vicinity"
${VICINITY} Publish -proxy plain -spawn -detach
        -class DE.FhG.IGD.semoa.net.Multicastd
        -key ${WhatIs:VICINITY} ;
        -setGateway ${WhatIs:INGATE}/raw -run

${WEB} echo -n " httpd"
${WEB} Publish -detach -spawn -proxy plain
        -key ${WhatIs:HTTP_SERVER}
        -class DE.FhG.IGD.semoa.web.HttpServer ;
        -setPort ${HTTPD_PORT}
        -setLog ${semoa.etc}/${semoa.conf}/httpd.log
        -setMimeTypes ${semoa.etc}/mime.types -run

echo "."

echo -n "Atlas:"

# NOTE: DO NOT CHANGE THE PROXY PORT 50505 YET!
# It is also hardcoded and changing it leads to
# problems. This will be fixed soon.
#
${ATLAS_PROXY} echo -n " proxy"
${ATLAS_PROXY} java DE.FhG.IGD.atlas.core.LSProxy
        -port 50505 -capacity 10
        -maxsize 65536 -serverconf  ${semoa.etc}/atlas.conf
        -loglevel ${ATLAS_LOG_LEVEL}
        -logfile ${semoa.etc}/${semoa.conf}/atlas-proxy.log
```

```
${ATLAS_CLIENT} echo -n " client"
${ATLAS_CLIENT} java DE.FhG.IGD.atlas.core.LSClient
        -serverconf ${semoa.etc}/atlas.conf
        -loglevel ${ATLAS_LOG_LEVEL}
        -logfile ${semoa.etc}/${semoa.conf}/atlas.log

${ATLAS_AGENT_TRACER} echo -n " agent_tracer"
${ATLAS_AGENT_TRACER} publish DE.FhG.IGD.atlas.core.AgentTracerImpl
        ${WhatIs:AGENT_TRACER} PLAIN_PROXY

echo "."

${WEB} echo -n "Registering Servlets:"

  ${WEB} echo -n " agentservlet"
  ${WEB} java DE.FhG.IGD.semoa.bin.PublishServlet
    -class servlet.AgentServlet -path agentservlet
  ${WEB} echo -n " resourceservlet"
  ${WEB} java DE.FhG.IGD.semoa.bin.PublishServlet
        -class servlet.ResourceServlet -path resource

${WEB} echo "."

echo -n "Gui Exchange:"
# Achtung - Aufruf unbedingt in dieser Reihenfolge
${GUIEXCHANGE} echo -n " Gui Exchange Server "
${GUIEXCHANGE} Publish -detach -proxy plain -spawn
        -class guiExchange.GUI_Exchange_Server
        -key ${WhatIs:GUI_EXCH_SRV} ;
        -setPort ${GUI_EXCH_SRV_PORT} -run

${GUIEXCHANGE} echo -n " Gui Collector "
${GUIEXCHANGE} Publish -detach -proxy plain
        -class control.AgentGUICollector
        -key ${WhatIs:GUI_COLL} ;
        -setShowAsJavaGUI ${SHOW_AS_JAVA_GUI}

${GUIEXCHANGE} echo -n " Gui Exchanger "
${GUIEXCHANGE} Publish -proxy plain
        -spawn -class guiExchange.GUI_Exchanger
```

```
        -key ${WhatIs:GUI_EXCH}
echo "."
```

B.3 whatis.conf

```
#
# The list of static global definitions in the SeMoA server.
# This list must be loaded into 'WhatIs' using a call such
# as 'java DE.FhG.IGD.semoa.server.WhatIs -init <url>' where
# <url> is a file URL to this file.
#
# This file defines the name space structure of important
# objects in a SeMoA server. Programs depend on the integrity
# of the keys, so don't change them unless you *exactly* know
# what you are doing. The values can be changed to suit the
# needs of your server configuration. It goes without saying
# that changes of values can be fatal if a server configuration
# or classes hardcode paths instead of using WhatIs, because
# inconsistencies are preprogrammed.
#
# Author : Volker Roth
# Version: $Id: whatis.conf,v 1.16 2002/03/04 14:38:02 upinsdor Exp $
#

# The storage resources used by the server.
#
SERVER_RESOURCE = /private/resource
TMP_RESOURCE = /private/tmp

# Definitions for transportation related constants.
#
INGATE  = /transport/ingate
OUTGATE = /transport/outgate

# Security related constants.
#
AGENT_LAUNCHER = /security/launcher
KEYMASTER = /security/keymaster
SECURITY_FILTERS_IN = /security/in
```

```
SECURITY_FILTERS_OUT = /security/out
THREADS_FILTER = /security/threads
SSL_MASTER = /security/sslmaster
BYTECODE_FILTER = /security/bytecode
POLICY = /security/policy
CERTFINDER = /public/certfinder

# Miscellaneous network services
#
VICINITY = /network/vicinity
HTTP_SERVER = /network/httpd
GUI_EXCH = /network/gui_exchanger
GUI_EXCH_SRV = /network/gui_exchanger/server
GUI_COLL = /network/gui_collector

# ATLAS related constants.
#
ATLAS = /atlas
AGENT_TRACER = /atlas/tracer

# The path under which agent contexts are published. The path
# must not have a trailing slash because it is added automatically
# by the ingate when agents are published.
#
AGENTS = /agents/active

# The event reflector that is used to signal state changes in
# agents. This should be a child path of ${WhatIs:AGENTS}
#
AGENT_EVENTS = /agents/event_reflector

# The event reflector that is used to signal state changes in
# agents. The events distributed via this reflector are
# restricted in terms of the information they provide. In
# contrast to the events distributed via ${WhatIs:AGENT_EVENTS}
# these events do not provide a reference to the agent's context.
#
RESTRICTED_EVENTS = /restricted/event_reflector

# The event reflector that is used to communicate server
```

```
# events such as shutdown.
#
PUBLIC_EVENTS = /public/event_reflector

# This path is used as the prefix of the URL string returned
# by DE.FhG.IGD.semoa.service.AbstractService.docs()
#
SEMOA_DOCS_URL = http://www.semoa.org/docs/api/

# Variables for test purposes, do not rely on these! They
# may change without prior notice.
#
AGLET_PROPS          = /compat/aglet/props
JADE_MESSAGE_ROUTER = /compat/jade/router
TRACY_BLACKBOARD     = /compat/tracy/blackboard
TRACY_MSGDISPOSER    = /compat/tracy/mqd

# Commuication related constants
#
OUTBOX               = /comm/outbox
INBOX                = /comm/inbox
SPOOL                = /comm/spool
```

XSLT-Stylesheets

C.1 global.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:param name="servletname"/>
 <xsl:param name="javascriptOK"/>
 <xsl:param name="cssOK"/>

 <xsl:param name="agent_id"/>
 <xsl:param name="host"/>
 <xsl:param name="port"/>
 <xsl:param name="gui_pos"/>
 <xsl:variable name="JFrameTitleHeight">18</xsl:variable>
 <xsl:variable name="title_height">22</xsl:variable>
 <xsl:variable name="title_style"> font-weight:bold; color:white;
  width:100%; font-size:14pt; line-height:22pt;
  background-color:#8080FF; padding:5px;</xsl:variable>
 <xsl:variable name="footer_style"> position:absolute;
  height:22pt; width:100%; color:white;
  font-size:8pt;line-height:14pt;
  background-color:#8080FF; padding:5px;</xsl:variable>
 <xsl:variable name="space_style"> height:22pt;</xsl:variable>
 <xsl:variable name="table_background">#C0C0C0</xsl:variable>
 <xsl:variable name="table_overview_background">#8080FF
```

```
 </xsl:variable>
<xsl:variable name="table_content_background">#FFFFFF
 </xsl:variable>

<xsl:template name="css">
 body {font-size:10pt; line-height:14pt;
       font-family:Arial;
       letter-spacing:0.2mm;
       word-spacing:0.8mm;
       color:black; }

       a:link { text-decoration:none; font-weight:bold;
                color:blue; }
       a:visited { text-decoration:none; font-weight:bold;
                   color:grey; }
       a:hover { text-decoration:none; font-weight:bold;
                 background-color:yellow; }
       a:active { text-decoration:none; font-weight:bold;
                  background-color:white; }
</xsl:template>

<xsl:template name="footer">
 <xsl:param name="top"/>
  <div style="{$footer_style} top:{$top + 60}; ">
   <div style="position:absolute; text-align:center;
    background-color:#80A0FF; width:150;">
    <a href="{$servletname}">
     show all hosts
    </a>
   </div>
   <div style="position:absolute; text-align:center;
    background-color:#80A0FF; left:170; width:150;">
    <a href="{$servletname}?&amp;
     host={$host}&amp;port={$port}">
     <xsl:choose>
      <xsl:when test="$agent_id">
       show all agents
      </xsl:when>
      <xsl:otherwise>
       refresh
```

```
       </xsl:otherwise>
      </xsl:choose>
     </a>
    </div>
    <xsl:if test="$gui_pos">
     <div style="position:absolute; text-align:center;
      background-color:#80A0FF; left:340; width:150;">
      <a href="{$servletname}?&amp;agent_id={$agent_id}
       &amp;position={$gui_pos}&amp;
       host={$host}&amp;port={$port}">
       refresh
      </a>
     </div>
    </xsl:if>
   </div>
  </xsl:template>
</xsl:stylesheet>
```

C.2 allhosts_ms.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:import href="global.xslt"/>

 <xsl:template match="/">
  <html>
   <head>
    <title><xsl:value-of select="/root/title"/></title>
    <style type="text/css">
     <xsl:call-template name="css"/>
    </style>
   </head>
   <body>
    <div style="{$title_style}">
     Alle verfügbaren SEMOA Hosts
    </div>
```

```
    <div style="{$space_style}"/>
    <form action="servlet.Start" method="get">
     <xsl:apply-templates select="root/hosts"/>
    </form>
   </body>
  </html>
</xsl:template>
<xsl:template match="hosts" >
  <table border="0" bgcolor="{$table_background}"
   cellspacing="0" cellpadding="0">
   <tr>
    <td>
     <table border="1" cellspacing="0"
      cellpadding="4" width="100%">
      <tr bgcolor="{$table_overview_background}" border="0">
       <td align="left">
        <span style="font-weight:bold;">
         Name
        </span>
       </td>
       <td align="left">
        <span style="font-weight:bold;">
         Beschreibung
        </span>
       </td>
      </tr>
      <xsl:for-each select="host">
       <tr bgcolor="{$table_content_background}" border="0">
        <td align="left">
         <a href="{$servletname}?&amp;host={@ip}
          &amp;port={@port}">
          <xsl:value-of select="@name"/>
         </a>
        </td>
        <td align="left">
         <xsl:value-of select="@description"/>
        </td>
       </tr>
      </xsl:for-each>
     </table>
```

```
    </td>
   </tr>
  </table>
 </xsl:template>
</xsl:stylesheet>
```

C.3 allagents_ms.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>

 <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:import href="global.xslt"/>

 <xsl:template match="/">
  <html>
   <head>
    <title><xsl:value-of select="/root/title"/></title>
     <style type="text/css">
      <xsl:call-template name="css"/>
     </style>
   </head>
   <body>
    <div style="{$title_style}">
      Alle Agenten auf Host
      <a style="color:blue">
        <xsl:value-of select="root/host/@url"/>
      </a>
    </div>
    <div style="{$space_style}"/>
    <form action="servlet.Start" method="get">
     <xsl:apply-templates select="root/agents"/>
    </form>
    <div style="{$space_style}"/>
    <xsl:call-template name="footer"/>
   </body>
  </html>
```

```
</xsl:template>

<xsl:template match="agents" >
 <table border="0" bgcolor="{$table_background}"
   cellspacing="0" cellpadding="0">
  <tr>
   <td>
    <table border="1" cellspacing="0" cellpadding="4"
     width="100%">
     <tr bgcolor="{$table_overview_background}" border="0">
      <td align="left">
       <span style="font-weight:bold;">
        Name
       </span>
      </td>
      <td align="left">
       <span style="font-weight:bold;">
        Nickname
       </span>
      </td>
      <td align="left">
       <span style="font-weight:bold;">
        Framename
       </span>
      </td>
     </tr>
     <xsl:for-each select="agent">
      <tr bgcolor="{$table_content_background}" border="0">
       <td align="left">
        <xsl:value-of select="@agentid"/>
       </td>
       <td align="left">
        <xsl:value-of select="@name"/>
       </td>
       <td align="left">
        <xsl:for-each select="./agentGUI">
         <a href="{$servletname}?&amp;agent_id={../@agentid}
          &amp;position={position()}&amp;host={/root/host/@ip}
          &amp;port={/root/host/@port}">
          <xsl:value-of select="@name"/>
```

```
          </a>
          <xsl:text> </xsl:text>
         </xsl:for-each>
        </td>
       </tr>
      </xsl:for-each>
     </table>
    </td>
   </tr>
  </table>
 </xsl:template>
</xsl:stylesheet>
```

C.4 html_agent_gui_ms.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:import href="global.xslt"/>

 <xsl:template match="/">
  <html>
   <head>
    <title> SEMOAs AGENT GUI classes to HTML</title>
    <meta http-equiv="cache-control" content="no-cache"/>

    <style type="text/css">
     body {  font-size:12; line-height:14pt;
     font-family:Arial;
     color:black; }

     a:link { text-decoration:none; font-weight:bold;
              color:blue; }
     a:visited { text-decoration:none; font-weight:bold;
                 color:grey; }
    </style>

    <script type="text/javascript">
     /* erste Standard-Graphik */
```

```
    minimize = new Image();
    minimize.src = "resource?id=windows_minimize_small.gif";
    /* erste Highlight-Graphik */
    minimize_inv = new Image();
    minimize_inv.src =
              "resource?id=windows_minimize_small_inv.gif";
    /* zweite Standard-Graphik */
    winsize = new Image();
    winsize.src = "resource?id=windows_winsize_small.gif";
    /* zweite Highlight-Graphik */
    winsize_inv = new Image();
    winsize_inv.src =
              "resource?id=windows_winsize_small_inv.gif";
    /* dritte Standard-Graphik */
    abort = new Image();
    abort.src = "resource?id=windows_abort_small.gif";
    /* dritte Highlight-Graphik */
    abort_inv = new Image();
    abort_inv.src = "resource?id=windows_abort_small_inv.gif";

    function press(image_object, source_image) {
     image_object.src = source_image.src;
    }
    function release(image_object, source_image) {
     image_object.src = source_image.src;
    }
    function release_submit(image_object, source_image) {
     image_object.src = source_image.src;
     document.agentform.submit();
    }

   </script>
  </head>
  <body>
   <form action="{$servletname}" name="agentform" method="get">

    <input type="hidden" name="firstcall" value="false"/>
    <input type="hidden" name="host" value="{$host}"/>
    <input type="hidden" name="port" value="{$port}"/>
    <input type="hidden" name="agent_id" value="{$agent_id}"/>
```

```
    <input type="hidden" name="position" value="{$gui_pos}"/>

    <div style="{$title_style}">
     Agent
     <a style="color:blue">
      <xsl:value-of select="$agent_id"/>
     </a>
     on Host
     <a style="color:blue">
      <xsl:value-of select="$host"/>
     </a>
    </div>

    <div style="position:absolute; top:{$JFrameTitleHeight
     +$title_height + 30 }; left:{$JFrameTitleHeight};
     z-index:1;" info="body">
     <xsl:apply-templates select="root/class"/>
    </div>
    <!--
     <xsl:if test="$javascriptOK = 'false'">
      <input type="submit" name="nojavascript"
       value="{$javascriptOK}"/>
     </xsl:if>
    -->
    <xsl:call-template name="footer">
     <xsl:with-param name="top"
      select="root/class/agentgui.JFrame/@height
      + $JFrameTitleHeight + $title_height"/>
    </xsl:call-template>
   </form>
  </body>
 </html>
</xsl:template>

<xsl:template match="agentgui.JFrame" >
 <div style="background:#C0C0C0; border-style:ridge;
  border-width:3; border-color:#FFFFFF;  position:absolute;
  left:{@x}; height:{@height}; width:{@width}; top:{@y};
  z-index:1; " info="agentgui.JFrame">
  <div style="background:blue; width:{@width};
```

```
height:{$JFrameTitleHeight};">
<div style="position:absolute; left:{@x};
 width:{$JFrameTitleHeight + 3}; height:{$JFrameTitleHeight};
 margin-left:1; margin-top:1;">
 <xsl:if test="@resource">
  <img src="resource?type=jpg&amp;id={@resource}&amp;
   agent_id={$agent_id}&amp;host={$host}&amp;port={$port}"/>
 </xsl:if>
</div>
<div style="position:absolute;  vertical-align:middle;
 font-weight:bold; color:white; left:{@x +
 $JFrameTitleHeight + 9 }; height:{$JFrameTitleHeight};">
 <xsl:value-of select="@title"/>
</div>
<div style="position:absolute; vertical-align:middle;
 left:{@x}; margin-left:{@width - 55}; margin-top:1;
 height:{$JFrameTitleHeight};">
 <a onMousedown="press(mini, minimize_inv)"
  onMouseUp="release(mini, minimize)"
  onMouseout="release(mini, minimize)" >
  <image name="mini" value="minimize_window"
   src="resource?&amp;
   id=windows_minimize_small.gif" alt="Fake"/>
 </a>
 <xsl:text> </xsl:text>
 <a onMousedown="press(size, winsize_inv)"
  onMouseUp="release(size, winsize)"
  onMouseout="release(size, winsize)" >
  <image name="size" value="size_window"
   src="resource?&amp;
   id=windows_winsize_small.gif" alt="Fake"/>
 </a>
 <xsl:text> </xsl:text>
 <xsl:text> </xsl:text>

 <a href="{$servletname}" onMousedown="press(close, abort_inv)"
  onMouseUp="release_submit(close, abort)"
  onMouseout="release(close, abort)" >
  <image name="close" value="close_window"src="resource?&amp;
   id=windows_abort_small.gif" border="0" alt="Close"/>
```

```
    </a>
   </div>
  </div>
  <div style="position:absolute; background:yellow;
   top:{@y + $JFrameTitleHeight }; z-index:1;">
   <xsl:apply-templates select="class"/>
  </div>
 </div>
</xsl:template>

<xsl:template match="agentgui.JPanel">
 <div style=" position:absolute;  left:{@x}; height:{@height};
  width:{@width}; top:{@y}; z-index:1;" info="agentgui.JPanel">
  <xsl:apply-templates select="class"/>
 </div>
</xsl:template>

<xsl:template match="agentgui.JLabel">
 <div style="position:absolute; font-weight:bold;
 left:{@x};height:{@height}; width:{@width};
 top:{@y}; z-index:1;" info="agentgui.JLabel">
  <xsl:value-of select="@text"/>
  <xsl:if test="@resource">
   <img src="resource?type=jpg&amp;id={@resource}&amp;
    agent_id={$agent_id}&amp;host={$host}&amp;port={$port}"/>
  </xsl:if>
 </div>
</xsl:template>

<xsl:template match="agentgui.JTextField">
 <div style=" position:absolute; left:{@x};
  height:{@height}; width:{@width}; top:{@y};
  z-index:1;" info="agentgui.JTextField">
  <input name="{@identifier}" type="text" size="{@height}"
   maxlength="{@width}" value="{@text}"/>
 </div>
</xsl:template>

<xsl:template match="agentgui.JComboBox">
 <div style=" position:absolute; left:{@x};
```

```
   height:{@height}; width:{@width}; top:{@y};
   z-index:1;" info="agentgui.JComboBox">
   <select name="{@identifier}" size="1">
    <xsl:for-each select="item">
     <option><xsl:value-of select="."/></option>
    </xsl:for-each>
   </select>
  </div>
</xsl:template>

<xsl:template match="agentgui.JButton">
  <div style=" position:absolute;  left:{@x}; height:{@height};
   width:{@width}; top:{@y}; z-index:1;">
   <input type="submit" name="{@identifier}" value="{@text}"/>
  </div>
</xsl:template>

<xsl:template match="agentgui.JCheckBox">
  <div style=" position:absolute;  left:{@x}; height:{@height};
   width:{@width}; top:{@y}; z-index:1;">
   <input>
    <xsl:attribute name="type">checkbox</xsl:attribute>
    <xsl:attribute name="name">
     <xsl:value-of select="@identifier"/>
    </xsl:attribute>
    <xsl:attribute name="value">
     <xsl:value-of select="@text"/>
    </xsl:attribute>

    <xsl:if test="@selected = 'true'">
     <xsl:attribute name="checked">checked</xsl:attribute>
    </xsl:if>

    <xsl:attribute name="listener">
     <xsl:value-of select="@listener"/>
    </xsl:attribute>

    <xsl:if test="@listener">
     <xsl:attribute name="onClick">
      <xsl:text>document.agentform.submit()</xsl:text>
```

```
    </xsl:attribute>
   </xsl:if>
  </input>
  <xsl:value-of select="@text"/>
 </div>
</xsl:template>

<xsl:template match="agentgui.JRadioButton">
 <div style=" position:absolute;  left:{@x}; height:{@height};
  width:{@width}; top:{@y}; z-index:1;">
  <input>
   <xsl:attribute name="type">radio</xsl:attribute>
   <xsl:choose>
    <xsl:when test="@groupname">
     <xsl:attribute name="name">
      <xsl:value-of select="@groupname"/>
     </xsl:attribute>
     <xsl:attribute name="value">
      <xsl:value-of select="@identifier"/>
     </xsl:attribute>
    </xsl:when>
    <xsl:otherwise>
     <xsl:attribute name="name">
      <xsl:value-of select="@identifier"/>
     </xsl:attribute>
     <xsl:attribute name="value">
      <xsl:value-of select="@text"/>
     </xsl:attribute>
    </xsl:otherwise>
   </xsl:choose>

   <xsl:if test="@selected = 'true'">
    <xsl:attribute name="checked">checked</xsl:attribute>
   </xsl:if>
   <xsl:attribute name="listener">
    <xsl:value-of select="@listener"/>
   </xsl:attribute>
   <xsl:if test="@listener">
    <xsl:attribute name="onClick">
     <xsl:text>document.agentform.submit()</xsl:text>
```

```
      </xsl:attribute>
     </xsl:if>
    </input>
    <xsl:value-of select="@text"/>
   </div>
  </xsl:template>

  <xsl:template match="agentgui.JButton" mode="img">
   <div style=" position:absolute;  left:{@x}; height:{@height};
    width:{@width}; top:{@y}; z-index:1;">
    <input type="submit" name="{@identifier}" value="{@text}"/>
   </div>
  </xsl:template>
</xsl:stylesheet>
```

Beispiel-GUI mit XML-Datei

D.1 Java^{TM}-GUI und HTML-GUI

Abbildung D.1: Java^{TM}-Darstellung einer Agent-GUI mit Radiobuttons

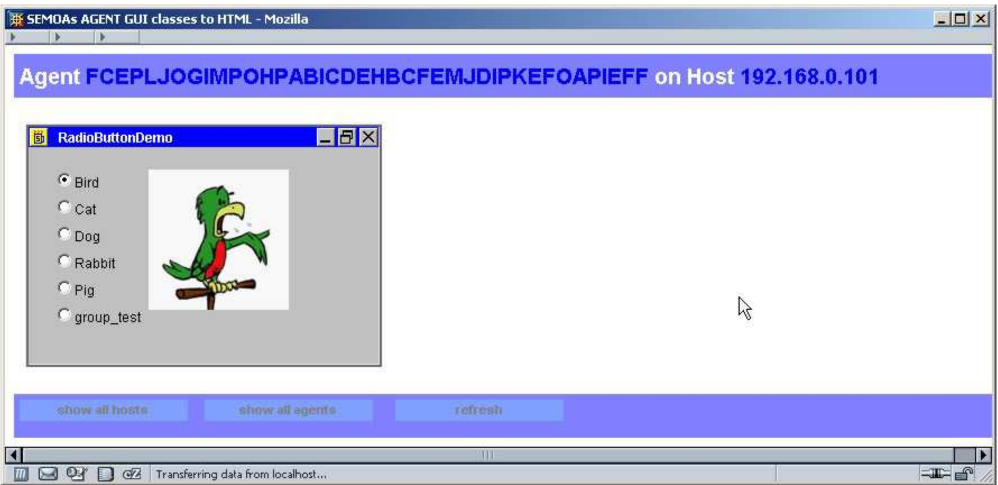


Abbildung D.2: Browserdarstellung einer Agent-GUI mit Radiobuttons

D.2 Beispiel der dazugehörigen GUI-Beschreibung in XML

```
<?xml version="1.0" encoding="ISO-8859-1"?> <root>
 <class>
  <agentgui.JFrame title="RadioButtonDemo" height="211.0"
   width="312.0" x="0.0" y="0.0" resource="13011608"
   identifier="agentgui.JFrame_31637242">
   <class>
    <agentgui.JPanel height="144.0" width="87.0" x="20.0"
     y="20.0" identifier="agentgui.JPanel_12524859">
     <class>
      <agentgui.JRadioButton text="Bird" label="Bird"
       height="24.0" width="87.0" x="0.0" y="0.0"
       selected="true" listener="true"
       groupname="groupname_9716945"
       identifier="agentgui.JRadioButton_9867226" />
     </class>
     <class>
      <agentgui.JRadioButton text="Cat"
       label="Cat" height="24.0" width="87.0"
       x="0.0" y="24.0" selected="false"
       listener="true" groupname="groupname_9716945"
       identifier="agentgui.JRadioButton_12468716" />
     </class>
     <class>
      <agentgui.JRadioButton text="Dog" label="Dog" height="24.0"
       width="87.0" x="0.0" y="48.0" selected="false"
       listener="true" groupname="groupname_9716945"
       identifier="agentgui.JRadioButton_14298351" />
     </class>
     <class>
      <agentgui.JRadioButton text="Rabbit" label="Rabbit"
       height="24.0" width="87.0" x="0.0" y="72.0"
       selected="false" listener="true"
       groupname="groupname_9716945"
       identifier="agentgui.JRadioButton_29103856" />
     </class>
     <class>
      <agentgui.JRadioButton text="Pig" label="Pig" height="24.0"
       width="87.0" x="0.0" y="96.0" selected="false"
```

```
       listener="true" groupname="groupname_9716945"
       identifier="agentgui.JRadioButton_25567987" />
     </class>
     <class>
      <agentgui.JRadioButton text="group_test"
       label="group_test" height="24.0" width="87.0" x="0.0"
       y="120.0" selected="false" listener="true"
       identifier="agentgui.JRadioButton_22518320" />
     </class>
    </agentgui.JPanel>
   </class>
   <class>
    <agentgui.JLabel height="144.0" width="177.0"
     x="107.0" y="20.0" resource="4980331"
     identifier="agentgui.JLabel_19682353" />
   </class>
  </agentgui.JFrame>
 </class>
</root>
```

Akronyme

API	Application Program Interface
AWT	Abstract Windows Toolkit
CAD	Computer Aided Design
DEC	Digital Equipment Corporation
DTD	Document Type Definition
GIF	Graphics Interchange Format
GUI	Graphic User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IGD	Institut für Graphische Datenverarbeitung
IP	Internet Protocol
JAR	Java Archive
JFC	Java Foundation Classes
JPEG	Joint Photographic Experts Group

JPG	siehe JPEG
JDK	Java Development Kit
LAN	Local Area Network
MCI	Maschine Computer Interaktion
MIT	Massachusetts Institute of Technology
MVC	Model View Controller
NLS	oN Line System
OMG	Object Management Group
PC	Personal Computer
PKCS	Public-Key Cryptography Standard
RMI	Remote Method Invocation
SGML	Standard Generalized Markup Language
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UI	User Interface
UIMS	User Interface Management System
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WWW	World Wide Web
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations

Literaturverzeichnis

[1] Umbc agentweb, Stand März 2003. http://agents.umbc.edu/Agents_for_.../index.shtml.

[2] Eric M. Burke. *Java und XSLT*. O'Reilly, 2002. ISBN 3-89721-295-1.

[3] Paul Prescod Charles F. Goldfarb. *The XML Handbook*. Prentice Hall PTR, 3 edition, 2001. ISBN 0-13-055068-x.

[4] Guido de Melo. Plattformunabhängige Benutzerschnittstellen: Beschreibung, Generierung, Anwendung, 2001.

[5] J. Müller (Ed.). *Verteilte Künstliche Intelligenz: Methoden und Anwendungen*. BI Wissenschaftsverlag, 1993. ISBN 3-411-16181-7.

[6] Jacques Ferber. *Multiagenten-Systeme, Eine Einführung in die Verteilte Künstliche Intelligenz*. Addison-Wesley, 2001. ISBN 3-8273-1679-0.

[7] David M. Geary. *Graphic Java 2.0, Die JFC beherschen (AWT)*. Prentice Hall, 3 edition, 1999. ISBN 3-8272-9585-8.

[8] Stefan Mintert Henning Behme. *XML in der Praxis*. Addison Wesley, 2 edition, 2001. ISBN 3-8273-1636-7.

[9] Niall Mansfield. *Das Benutzerhandbuch zum X- WINDOW- System*. Addison-Wesley, 1991. ISBN 3-89319-210-7.

[10] Sven Ehrke Mike Mannion. Einsatz von Java, XML und XSLT für grafische Oberflächen. *Java Spektrum*, September Oktober 2000.

[11] Technical Note. PKCS #7: Cryptographic Message Syntax Standaerd. Technical report, RSA Laboratories, 1993. http://www.rsalabs.com/pkcs/pkcs-7/.

[12] Bernhard Preim. *Entwicklung Interaktiver Systeme*. Springer, 1999. ISBN 3-540-65648-0.

[13] Tony Johnson Richard Murch. *Agententechnologie: Die Einführung, Intelligente Software-Agenten auf Informationssuche im Internet.* Addison-Wesley, 2000. ISBN 3-8273-1652-9.

[14] Volker Roth. *The $SeMoA^{TM}$ Code Conventions.* Fraunhofer IGD, Darmstadt, Germany, November 2000. Version 0.1.

[15] Volker Roth and Mehrdad Jalali. Access Control and Key Management for Mobile Agents. *Computers & Graphics, Special Issue on Data Security in Image Communication and Networks*, 22(4):457–461, 1998.

[16] Ewald Salcher. *Das X-Window-System und MS-Windows im Vergleich.* BI-Wissenschaftsverlag, 1993. ISBN 3-411-16201-5.

[17] Frank Sauer. XMLTalk, a framework for automatic GUI rendering from XML specs. *Java Report*, Oktober 2001.

[18] Joseph Schmuller. *Sams Teach Yourself UML in 24 Hours(2nd Edition).* Sams, 2001. ISBN 0-672-32238-2.

[19] Wolfgang Nefzger Stefan Münz. *HTML & Web-Publishing Handbuch (Band 1), HTML - JavaScript - CSS - DHTML.* Franzis Verlag, 2002. ISBN 3-7723-7517-0.

[20] Jason Hunter und William Crawford. *Java Servlet-Programmierung.* O'Reilly, Dezember 2001. ISBN 3-89721-282-X.