FACHHOCHSCHULE KARLSRUHE

Fachbereich Informatik

Studiengang Informatik und Multimedia

Beschreibung von Benutzerschnittstellen mit XML

Master Thesis

---

von:	Jürgen Baier
Zeitraum:	von 1. September 2000 bis 31. Januar 2001
Arbeitsplatz:	Propack Data GmbH, Karlsruhe
Referent:	Prof. Klaus Gremminger
Koreferent:	Prof. Dr. Peter A. Henning
Betreuer:	Dr. August Ludviksson
	Dipl.-Math. Stefan Winzinger

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Erklärung

Hiermit versichere ich, daß ich die vorliegende Arbeit mit dem Titel «Beschreibung von Benutzerschnittstellen mit XML» selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zum Erlangen eines akademischen Grades vorgelegt.

Karlsruhe, 13. Februar 2001

Jürgen Baier

Zusammenfassung

Deutsch

Diese Master Thesis befaßt sich mit der Beschreibung von Benutzerschnittstellen mit XML. Im Grundlagenkapitel werden die Themen «XSLT» und «Jini» behandelt, da diese Technologien für die Arbeit von Bedeutung sind.

Ziel der Thesis ist, eine XML-basierte Sprache zur Beschreibung von Benutzerschnittstellen zu entwickeln. Die Anwendung dieser Sprache soll anhand eines Prototypen gezeigt werden. Dieser Prototyp erlaubt die Steuerung eines Lego Mindstorms Roboters über ein Jini-Netzwerk. Die verschiedenen Clients erhalten ihre Benutzerschnittstelle als XML-Beschreibung vom Jini-Dienst. Der Jini-Dienst stellt mehrere Beschreibungen für verschiedene Zielplattformen (Windows, PalmOS) zur Verfügung. Diese Beschreibungen werden zur Laufzeit auf der Zielplattform interpretiert; d.h. gelesen und entsprechend am Bildschirm dargestellt. Bestandteil des Prototypen sind auch verschiedene Tools zur Transformation und Interpretation der XML-Beschreibungen.

Englisch

The subject of this thesis is the description of user interfaces with XML. In the theoretical chapter technologies are described that are relevant for the thesis and the prototype implementation.

The aim of the thesis is the development of a XML based language that can be used to describe user interfaces. The application of this language is shown in prototype. This prototype allows the control of a Lego Mindstorms Robot over a Jini network. The clients get their user interface as a XML description from the Jini service. The service provides several descriptions for different target platforms (such as Windows or Palm Pilot). These descriptions are interpreted at runtime on the target platform. This means they are read and rendered for the target platform’s display. Part of the prototype are several tools for transforming and interpreting XML descriptions.

Kapitel 1

Einleitung

1.1 Entwicklung von modernen Benutzerschnittstellen

Um Menschen die Bedienung eines Computer zu ermöglichen, muß eine Benutzerschnittstelle (auch: User-Interface, UI) vorhanden sein. Benutzerschnittstellen sind (wie der Name schon sagt) Schnittstellen zu einem Computersystem. Eine Benutzerschnittstelle kann viele Gesichter haben: Die Benutzerschnittstellen der ersten verfügbaren Rechner waren denkbar einfach: Abgeben eines Lochkartenstapels und Abholen des Ergebnisses in Form eines Ausdrucks (wenige Stunden später). Betriebssystem-Tools unter Unix bieten meist auch nur eine Text-Schnittstelle (die aber stets gut durchdacht und verständlich aufgebaut ist). Allerdings gibt es auch hier (vor allem im Rahmen der Projekte KDE und Gnome) zahlreiche Versuche, grafische Schnittstellen zu diesen Tools anzubieten.

Das heutige Spektrum an Benutzerschnittstellen[1] ist viel breiter gefächert. Grafische Benutzerschnittstellen (GUIs) sind die am weitesten verbreiteten Schnittstellen. GUIs verwenden verschiedene Metaphern (Metapher: bildliche Vorstellung), um die Mensch-Maschine-Kommunikation möglichst intuitiv zu machen. Weit verbreitet ist die Desktop-Metapher, die eine Schreibtischoberfläche (mit Papierkorb, verschiedenen Dokumenten, etc.) am Bildschirm darstellt.

Die Kommunikation zwischen - oder besser die Interaktion von - Menschen und Computern kann über verschiedene Kommunikationskanäle erfolgen. Beispiele für Kommunikationskanäle («Modalitäten der Mensch-Maschine-Interaktion»[Schulte96]) sind Sprache (schriftlich oder gesprochen), Gestik und Gesichtsausdrücke. Der Begriff «Modalität» bezeichnet bei der Mensch-Maschine-Kommunikation die Art und Weise, wie eine Benutzerschnittstelle bedient wird. Mo-

[1] In dieser Arbeit werden die Begriffe Benutzerschnittstelle, User-Interface (UI) und Benutzer- oder Bedienoberfläche synonym verwendet.

derne grafische Benutzeroberflächen (GUIs) verwenden gerne mehrere Kommunikationskanäle zur *Ausgabe* von Daten. Zur Eingabe von Befehlen und Daten stehen leider nur wenige Möglichkeiten zur Verfügung. Bei den meisten Anwendungsprogrammen lassen sich Kommandos und Daten nur über die Maus (Anklicken eines Objektes) und über die Tastatur (Kommando-«Shortcuts» und Eingabe von Text) eingeben. Minh Tue Vo schreibt dazu in seiner Dissertation ([Vo98]):

> «The advent of the graphical user interface (GUI) paved the way for computer applications that attempt to take advantage of the rich human communication modes by presenting information over multiple channels, including text, images, sound, video, virtual reality, etc. This form of *output* presentation has become familiar under the designation of *multimedia*. In contrast, applications that take advantage of multiple *input* channels have been appearing much more slowly.»

Im Zeitalter des Mobilfunks und der weiten Verbreitung von mobilen Telefonen («Handys») gewinnt die Modalität der gesprochenen Sprache an Bedeutung. Sprachgesteuerte Applikationen sind zwar noch relativ selten zu finden, werden aber zunehmend wichtiger. Komplexe Auto-Navigationssysteme sollte ein Autofahrer - schon aus Sicherheitsgründen - nicht manuell bedienen (oder bedienen dürfen). Die Modalität der gesprochenen Sprache ist also besonders wichtig, wenn der Benutzer nicht von seiner Aufgabe durch die Steuerung oder Bedienung eines Rechners abgelenkt werden soll (man vergleiche nur Desktop-Routenplansysteme mit Auto-Navigationssystemen).

Das Entwickeln von multimodalen Applikationen ist relativ schwierig, weil es noch kein einheitliches, populäres Framework gibt, das man als Grundlage verwenden könnte [Vo98]. Das hat zur Folge, daß die Entwicklung von Benutzerschnittstellen mit mehreren Kommunikationskanälen sehr kostenintensiv und zeitaufwendig ist.

Das Thema dieser Arbeit ist aber nicht die Analyse von möglichen Benutzerschnittstellen, sondern vielmehr die *Entwicklung* von Benutzerschnittstellen. Die Entwicklung von modernen Benutzerschnittstellen ist sehr aufwendig und erfordert in der Regel gute Programmierkenntnisse, besonders wenn multimodale Benutzerschnittstellen entwickelt werden sollen. Ein großes Problem bei der Entwicklung ist die Portierung einer Applikation (genauer: der Benutzeroberfläche einer Applikation) auf verschiedene Plattformen. Dabei ist es vergleichsweise einfach, eine Benutzeroberfläche auf verschiedene Desktop- Betriebssysteme (Windows, MacOS, Unix) zu portieren. Problematisch wird es, wenn Applikationen auch auf PDAs (Personal Digital Assistants) oder mobile Telefone portiert werden sollen. In den meisten Fällen kommt man nicht um eine komplette Neuentwicklung der Anwendung herum. Der Zeitaufwand dabei ist enorm, da sich die Entwickler erst wieder in neue Bibliotheken und Tools einarbeiten müssen.

Deshalb wird heute versucht, die Entwicklung (vor allem grafischer) Benutzeroberflächen zu automatisieren. GUI-Builder sind in fast jede Entwicklungsumgebung (Integrated Development Environment, IDE) integriert und ermöglichen es, in relativ kurzer Zeit ein GUI «zusammenzuklicken». Problematisch ist allerdings, daß man sich bei Verwendung eines GUI-Builders häufig an eine bestimmte Entwicklungsumgebung bindet, was die Portierungsproblematik noch verschärft.

Eine Benutzerschnittstelle ist immer nur eine *Schnittstelle* zur Applikationslogik, d.h. zu den Funktionen der Software. Es kann aber bei einer größeren Modifikation (oder einem vollständigen Austausch) der Schnittstelle notwendig sein, die Anwendung selbst zu modifizieren. Für die Entwicklung der populären *Wizards* ist beispielsweise eine zusätzliche Speicherung von Zustandsinformation erforderlich, so daß der Benutzer von Maske zu Maske *vor und zurück* navigieren kann. Eine Anpassung der Applikation kann ebenfalls erforderlich sein, wenn sich die Modalität der Benutzerschnittstelle ändert. Bei einer GUI-basierten Software ist es heute üblich (wenn auch selten sinnvoll) überladene Toolbars und Menüs anzubieten, die einer bestimmten Funktion innerhalb der Applikationslogik zugeordnet sind. Um die Schnittstelle eines solchen Programmes zu ändern sind vielfältige Anpassungen notwendig. Oft ist es sogar erforderlich, eine zusätzliche Abstraktionsschicht zwischen Applikationslogik und Benutzerschnittstelle einzuführen, um am Quellcode der Anwendung selbst nichts zu verändern (was vor allem sinnvoll ist, wenn die Basis der Applikation sehr umfangreich ist und Änderungen die Stabilität gefährden würden). Die neue Benutzerschnittstelle nutzt nur Funktionen der Zwischenschicht, während die Zwischenschicht Zustandsinformationen speichert und Funktionen der (Legacy-) Basisapplikation aufruft.

Wünschenswert ist es also, Benutzerschnittstellen in einem *universellen Austauschformat* für Benutzeroberflächen zu speichern. Idealerweise sollte mit diesem Austauschformat die Portierung einer Benutzerschnittstelle auf eine andere Plattform trivial sein. Und, um noch einen Schritt weiter zu gehen, es sollte problemlos möglich sein, dieses Austauschformat in beliebige Programmiersprachen zu transformieren. Problematisch ist dabei, ein Format festzulegen, das die Erstellung multimodaler Benutzerschnittstellen ermöglicht.

1.2 Aufgabenstellung

Diese Master Thesis beschäftigt sich mit der Entwicklung einer einheitlichen, XML-basierten Beschreibungssprache für Benutzerschnittstellen. Diese Beschreibungssprache soll in erster Linie die Beschreibung von GUI-Anwendungen ermöglichen. Untersucht wird jedoch auch, wie sich Benutzerschnittstellen unabhängig vom Kommunikationskanal beschreiben lassen. Insbesondere wird versucht, eine Brücke zwischen sprach- und GUI-basierte Zielsprachen zu schlagen (VoiceXML vs. Java-GUI).

Der Schwerpunkt dieser Arbeit wurde allerdings nicht auf die Entwicklung einer möglichst vielseitigen und vollständigen XML-Sprache gelegt, sondern vielmehr auf die möglichen Anwendungen dieser XML-Sprache.

Was kann man nun mit der Beschreibung einer (grafischen) Benutzerschnittstelle in XML anfangen? Zunächst natürlich gar nichts. Deshalb muß diese Beschreibung in eine verwendbare Form transformiert werden. Es gibt genau zwei mögliche Transformationen (wobei lediglich eine davon eine Transformation im üblichen Sinne ist):

- Die Transformation in compilierbaren (Java) oder interpretierbaren Quellcode (HTML). Diese Transformation wird mit Hilfe von XSLT durchgeführt.
- Die Transformation in eine Menge von Benutzerschnittstellen-Objekten im Kontext einer Java-VM (Virtual Machine). Diese «Transformation»läßt sich nur mit Hilfe eines *Interpreters* realisieren. Also mit einer Komponente, die in der Lage ist, anhand der XML-Beschreibung Objekte zu instanzieren.

Es sind also zunächst zwei Tools zu entwickeln:

- Eine XSLT-Applikation zur Transformation der XML-Beschreibung in verschiedene Zielsprachen für verschiedene Plattformen
- Ein Interpreter, der aus der XML-Beschreibung ein GUI instanzieren kann. Die Implementation soll das AWT-Toolkit von Java verwenden und sowohl auf dem Desktop als auch auf einem PDA laufen.

Die Funktion dieser Tools soll natürlich an einer Beispielapplikation vorgeführt werden. Hier gibt es verschiedenste Möglichkeiten, aus denen eine herausgegriffen wurde: Es soll ein Lego Mindstorms-Roboter mit einem Personal Digital Assistant (PDA) der Firma Palm (Palm Pilot) gesteuert werden. Die Steuerung soll über eine grafische Benutzerschnittstelle erfolgen, die auf dem Palm Pilot von einem Interpreter erzeugt wird. Als Netzwerk-Middleware wird die Jini-Technologie von Sun Microsystems verwendet.

Die Einbindung von Mindstorms-Robotern (als Services) und Palm Pilots (als Clients) in ein Jini-Netzwerk ist wegen der eingeschränkten Leistungsfähigkeit der Clients und Services problematisch. Auf der JavaOne 1999 wurde erstmals von Sun eindrucksvoll demonstriert, daß es möglich ist, über ein Jini-Netzwerk von Palm Pilots aus auf Mindstorms-Roboter zuzugreifen (siehe [Deleo2000]). Die vorgestellte Lösung hatte allerdings einen Nachteil: Die grafische Benutzerschnittstelle der Palm Pilots war hart-codiert, d.h. es wurde keine GUI dynamisch geladen. Jan Newmarch beschreibt in seinem Jini-Tutorial (siehe [Newmarch2000]), wie man die Benutzerschnittstelle der Roboter dynamisch laden kann. Da die Service-UI-Spezifikation (eine Spezifikation im Jini-Umfeld) genau für diese Funktionalität entwickelt wurde, zeigt er in seinem Tutorial eine Beispielapplikation auf dieser Basis. Leider

funktioniert die vorgestellte Lösung nur bei Jini-fähigen Clients. Die aktuellen Java-Implementierungen für Palm Pilots basieren auf der Java 2 Micro Edition (J2ME) und bieten nur einen kleinen Teil der Funktionen der Java 2 Standard Edition (J2SE). Eine wesentliche Einschränkung ist z.B. daß keine Klassen zur Laufzeit nachgeladen werden können.

Hier ist also eine Anwendungsmöglichkeit für die XML-basierte UI-Beschreibung. Da die KVM Socketverbindungen unterstützt, ist es möglich, die XML-Beschreibung über Sockets zu laden und auf dem Client zu interpretieren. Die prototypische Implementierung soll also die Steuerung von Lego Mindstorms-Robotern über Palm Pilots ermöglichen. Die passende Benutzerschnittstelle dafür soll in XML-Form dynamisch geladen und interpretiert werden.

1.3 Die Propack Data GmbH

Die Propack Data GmbH wurde 1984 gegründet und hat sich seitdem zum weltweit führenden Anbieter von Hard- und Software für die Pharma- und Nahrungsmittelindustrie entwickelt. Propack Data hat ein zertifiziertes Qualitätsmanagement und entwickelt nach anerkannten DIN/ISO-Standards. Die PMX-Standardsoftware ist gemäß den FDA/GMP-Vorschriften validierbar und genügt somit den strengen Richtlinien für Software in den Bereichen Pharma, Kosmetik und Nahrungsmittel.

Abbildung 1.1: Propack Data GmbH

Die weitgehende Konfigurierbarkeit der PMX-Software ermöglicht eine unproblematische Anpassung für verschiedene Branchen. Zusätzlich sind individuelle Anpassungen und Erweiterungen auf Kundenwunsch möglich. Umfassende Dienstleistungen runden das Angebot der Propack Data GmbH ab. Weitere Informationen sind unter http://www.propack-data.com verfügbar.

1.4 Das Forschungsprojekt «Morpha»

Morpha ist ein Verbundprojekt das vom Bundesministerium für Bildung und Forschung (http://www.bmbf.de) gefördert wird. Bei diesem Projekt geht es um intelligente anthropomorphe Assistenzsysteme, z.B. um robotische Gehhilfen für ältere

Menschen oder Haushaltsroboter im weitesten Sinne. Mit «anthropomorph» ist dabei nicht nur «menschenähnlich» im Bezug auf das Aussehen gemeint, sondern auch Ähnlichkeit im Bezug auf die *Wahrnehmung* des Menschen.

Abbildung 1.2: Logo des Morpha-Projektes

Ein wichtiger Bereich bei diesem Projekt ist die Schnittstelle zwischen Mensch und Maschine. In [Morpha2000] heißt es

> «Als übergeordnete Anforderung steht die effiziente Nutzung und Integration menschlicher Sinne wie Sprache, Gestik, Haptik und realitätsnahe Grafik in Verbindung mit der ergonomischen und sicheren Kooperation zwischen Nutzer und Assistenzsystem im Vordergrund. »

So sollte z.B. eine robotische Gehhilfe für ältere Menschen in hohem Maße intuitiv bedienbar sein, d.h. insbesondere, daß die Bedienung eines solchen Assistenzsystems nicht «erlernt» werden muß. Die Kanäle der Mensch-Maschine-Kommunikation sind aber nur ein Thema dieses Forschungsprojektes. Folgende Themen sollen bearbeitet werden (aus [Morpha2000]:

- Kanäle der Mensch-Maschine-Kommunikation
- Szenenanalyse und Interpretation
- Belehrung und Adaptivität
- Bewegungsplanung und -koordination
- Sicherheit/Wartung/Diagnose

In ein anthropomorphes Assistenzsystem sollen möglichst viele Forschungsergebnisse dieser Themen einfließen. Als mögliche Produkte werden vor allem der *Produktionsassistent* und der *Haushaltsassistent* genannt. Gerade bei einem Haushaltsassistenten kommt es darauf an, bei der Kommunikation zwischen Mensch und Maschine möglichst viele verschiedene Kommunikationskanäle zu berücksichtigen. Einige Morpha-Projekte sind schon sehr vielversprechend. Es ist möglich, daß in den nächsten Jahren eine ganz neue Generation von intuitiven Benutzerschnittstellen - nicht nur zur Bedienung von Robotern - auf den Markt kommt.

Die Propack Data GmbH beteiligt sich (neben zahlreichen anderen Firmen und Instituten) am Morpha-Projekt. Verschiedene Ergebnisse dieser Master Thesis, vor allem die textuelle Beschreibung von Benutzerschnittstellen mit XML sollen in das Projekt einfließen und am 17. Januar 2001 in einer Präsentation von Propack Data dargestellt werden.

1.5 Gliederung der Thesis

Im *Grundlagenkapitel* (Kapitel 2) werden einige verwendete Technologien näher beschrieben. Dazu gehören:

- XSLT und
- Jini.

XSLT ist eine Sprache zur Transformation von XML-Dokumenten. In dieser Thesis wird XSLT dazu verwendet, aus XML-Beschreibungen von Benutzerschnittstellen ausführbaren Java-Quellcode zu generieren. Jini ist eine dienstbasierte Netzwerk-Technologie, die nützlich ist, wenn ein flexibles, skalierbares System mit der Programmiersprache Java entwickelt werden soll. In dieser Thesis wird Jini verwendet, um verschiedenen Clients (Desktop-Client, PDA-Client) die Steuerung von Lego Mindstorms-Robotern zu ermöglichen. Im Jini-Abschnitt des Grundlagenkapitels wird auch auf spezielle Spezifikationen und Forschungsprojekte im Jini-Umfeld eingegangen.

Das *Analysekapitel* (Kapitel 3) beschreibt die Anforderungen an eine XML-basierte Sprache zur Beschreibung von Benutzerschnittstellen und untersucht, ob diese Anforderungen überhaupt in einer einzigen Sprache realisierbar sind. Außerdem werden verschiedene XML-basierte Sprachen zur Schnittstellenbeschreibung analysiert und auf Praxistauglichkeit getestet.

Das Kapitel 4 (Design und Implementierung) konzentriert sich auf verschiedene Teilkomponenten des Prototypen:

- Tool zur Transformation von UI-Beschreibungen in Java-Quellcode (Entwicklung mit XSLT).
- Tool zur Interpretation von UI-Beschreibungen (Entwicklung mit Java).
- Jini-Service zur Steuerung von Mindstorms-Robotern.
- Jini-Clients (Desktop-Client und PDA-Client), die den Service nutzen.

Die Steuerung der Mindstorms-Roboter soll dabei lediglich eine Anwendung der beiden Tools und der XML-Sprache demonstrieren.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige für diese Master Thesis wichtige Technologien beschrieben. Dazu gehören XSLT, XPath und Jini. Außerdem wird auch die Java 2 Micro Edition (J2ME) mit ihren verschiedenen Implementierungen auf PDAs behandelt.

2.1 XSLT und XPath

2.1.1 XSLT

Die Extensible Stylesheet Language (XML) ist eine Auszeichnungssprache (Markup Language), mit der sich beliebige andere Auszeichnungssprachen definieren lassen. Zahlreiche Beispiele für XML-basierte Sprachen aus den verschiedensten Bereichen von Astronomie bis Workflow findet man unter http://www.xml.org/xmlorg_registry/index.shtml.

Will man ein XML-Dokument anzeigen bzw. formatiert ausgeben (z.B. auf einem Drucker), hat man verschiedene Möglichkeiten:

- Darstellung des Dokumentes in einem Texteditor.
- Darstellung des Dokumentes in einer speziellen Strukturansicht (z.B. Baumstruktur). Der Internet Explorer von Microsoft kann XML-Dokumente ab Version 5 als Baumstruktur darstellen.
- Transformation des Dokumentes in eine Struktur, die sich zur formatierten Anzeige auf dem Bildschirm (oder auf einem Drucker) eignet. Hier bieten sich als Formate (neben zahlreichen anderen) HTML und PDF an.

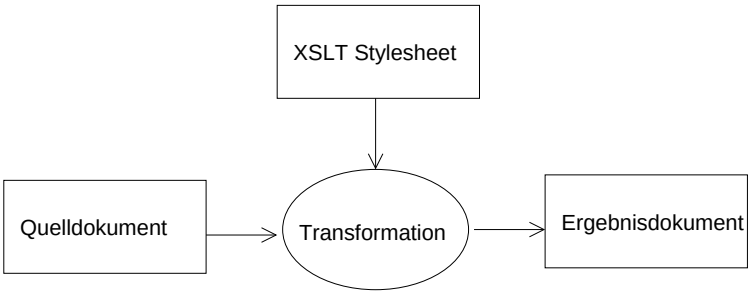


Abbildung 2.1: Transformation mit XSLT

Es ist wichtig, sich klarzumachen, daß XML-Dokumente Struktur und Darstellung streng trennen. Da XML-Dokumente Struktur und Darstellung streng trennen, läßt sich ein XML-Dokument zunächst lediglich im Quelltext darstellen. Der Sinn eines XML-Dokumentes besteht nicht unbedingt darin, es in irgendeiner Form auszugeben oder zu formatieren. Interessant ist vielmehr, das Dokument so zu bearbeiten, daß es für eine bestimmte Aufgabe von Nutzen ist. Die Möglichkeiten, einen Nutzen aus einem XML-Dokument zu ziehen sind zahlreich. Zu den Anwendungsmöglichkeiten gehören die Datenvisualisierung, der Datenaustausch und die Datentransformation.

Für die Bearbeitung von XML-Dokumenten stehen verschiedene Möglichkeiten zur Verfügung:

- Parsen des Dokuments mit einem beliebigen XML-Parser (Verwendung von DOM oder SAX).
- Extraktion bestimmter Informationen mit einem Tool (z.B. mit awk/gawk oder Perl).
- Transformation des Dokuments mit speziellen, für XML und SGML entwickelten (proprietären) Sprachen wie OmniMark.
- Transformation des Dokuments mit XSLT.

Für die Transformation eines XML-Dokumentes in eine andere Struktur bietet sich XSLT als offener Standard an. Eine Kurzdefinition gibt Michael Kay in [Kay2000]:

> «XSLT is a language for transforming the structure of an XML document.»

Hier geht es demnach darum, eine Datenstruktur, die durch eine DTD oder ein Schema - also durch eine Grammatik - festgelegt wurde, in eine andere Datenstruktur zu transformieren. Abbildung 2.1 zeigt das Schema einer XSLT-Transformation.

Am besten eignet sich XSLT, um XML-Dokumente in andere XML-Dokumente zu transformieren oder, genauer ausgedrückt, um die (Daten-)Struktur der Dokumente zu transformieren. Allerdings ist XSLT nicht auf diesen Anwendungsfall eingeschränkt. Im Rahmen dieser Master Thesis z.B. wurden Stylesheets zur Transformation von XML-Dokumenten in Java-Quellcode entwickelt. Die Umkehrung dieser Transformation - d.h. die Generierung von XML-Dokumenten aus Java-Quellcode ist schwieriger, aber möglich. Um diese Transformation mit XSLT zu realisieren gibt es zwei Möglichkeiten:

- Schreiben eines Parsers, der aus dem Textdokument ein XML-Dokument generiert. Danach ist eine Transformation mit XSLT möglich.
- Ein XSLT-Prozessor erwartet streng genommen keine XML-Datei als Eingabe sondern eine Baumrepräsentation der XML-Datei. Damit ist es möglich, einen mitgelieferten XML-Parser durch einen Parser für ein Textdokument auszutauschen, der z.B. SAX-Ereignisse auslöst und sich somit wie ein XML-Parser verhält.

Zusammenfassend kann man sagen, daß XSLT besonders gut geeignet ist, XML-Dokumente in andere XML-Dokumente zu transformieren, daß es sich aber auch gut für die Transformation von XML-Dokumenten in Textdokumente eignet. Die Transformation von Textdokumenten in XML-Dokumente ist nur zu empfehlen, wenn die Textdokumente eine gewisse Struktur aufweisen, d.h. wenn man eine ganze Klasse von Textdokumenten auf gleiche Weise transformieren kann.

2.1.2 XPath - XML Path Language

Die XML Path Language (XPath) ist ein integraler Bestandteil von XSLT. Allerdings ist diese Sprache in einer eigenen Spezifikation ([W3C1999b]) beschrieben, weil sie nicht nur in XSLT sondern auch in XLink und XPointer verwendet wird. XPath ist für XSLT deshalb so wichtig, weil damit auf jeden Knoten der Baumrepräsentation eines XML-Dokumentes zugegriffen werden kann.

Mit XPath kann man einzelne Knoten (oder Knotenmengen) durch Ausdrücke wie «Suche mir alle Elementknoten `AAA`, die einen Kindknoten namens `BBB` mit dem Attributwert `chooseMe` des Attributs `name` haben.» selektieren. Natürlich ist das nicht ganz die XPath-Syntax, aber semantisch ist dieser Ausdruck äquivalent zu `AAA::BBB[attribute::name = "chooseMe"]` (oder zur abgekürzten Schreibweise `AAA::BBB[@name = "chooseMe"]`). Mit diesem Ausdruck wird die Achse `AAA` gesucht, geprüft, ob ein Kindknoten `BBB` vorhanden ist und ob die Bedingung (`BBB` hat ein Attribut `name` mit dem Wert `chooseMe`) wahr ist. Die allgemeine Form eines XPath-Ausdrucks ist

```
axis::node-test[predicate].
```

Die drei Bestandteile dieses Ausdrucks sind:

- Die Achse (axis), optional.
- Ein Knoten-Test (node-test), zwingend.
- Ein Prädikat (predicate), optional.

Die Achse beschreibt einen Pfad relativ zum Kontextknoten. Der Kontextknoten ist der gerade aktuelle Knoten, den der XSLT-Prozessor bearbeitet. Wird z.B. `child` verwendet, so bezieht sich dieser Ausdruck immer auf die Kinder des gerade aktuellen Knotens.

Mögliche Achsen sind `parent` für einen Eltern-Knoten, `ancestor` für einen beliebigen Vorgängerknoten, `child` für einen Kind-Knoten und `self` für den Kontext-Knoten (den gerade aktuellen Knoten). Es gibt verschiedene weitere nützliche Achsen: `descendant`, `following`, `preceding`, `following-sibling`, `preceding-sibling`, `attribute`, `namespace`, `descendant-or-self` und `ancestor-or-self`. Diese Achsen sind z.B. in [Kay2000] näher beschrieben.

Mit dem Knoten-Test wählt man bestimmte Knoten der Achse aus. Ist keine Achse angegeben, wird die Default-Achse `child` vom XSLT-Prozessor nach dem festgelegten Knoten durchsucht. Will man z.B. alle Kind-Knoten auswählen, kann man entweder `select="*"` oder `select="child::*"` schreiben.

Der Prädikat-Ausdruck schränkt die Auswahl (die mit der Achse und dem Knoten-Test getroffen wurde) ein. Damit läßt sich z.B. festlegen, daß man von den Knoten der zurückgegebenen Knoten-Menge lediglich an Knoten interessiert ist, die an einer bestimmten Position stehen oder deren Attribute bestimmte Werte haben. Das Ergebnis des Prädikat-Ausdrucks hat stets einen booleschen Wert. Der Wert ist stets `true`, wenn irgendein Knoten, auf den das Prädikat zutrifft, vorhanden ist. Zum Beispiel werden mit dem Ausdruck `child::*[attribute::name]` (oder `*[@name]` in der abgekürzten Schreibweise) alle Kind-Knoten ausgewählt, die ein Attribut `name` haben.

2.1.3 XSLT-Transformationsmodell

Mit XSLT können - wie in den Abschnitten 2.1.1 und 2.1.2 beschrieben - beliebige Dokumente (XML oder Text) in beliebige Dokumente (ebenfalls XML oder Text) transformiert werden. Der Kern des XSLT-Prozessors sieht dabei jedoch nicht die Dokumente selbst sondern lediglich eine (temporäre) Repräsentation der Dokumente als Bäume. Aus der Sicht des Prozessors werden also zwei XML-Bäume in einen dritten XML-Baum transformiert. Abbildung 2.2 zeigt das Schema der Transformation.

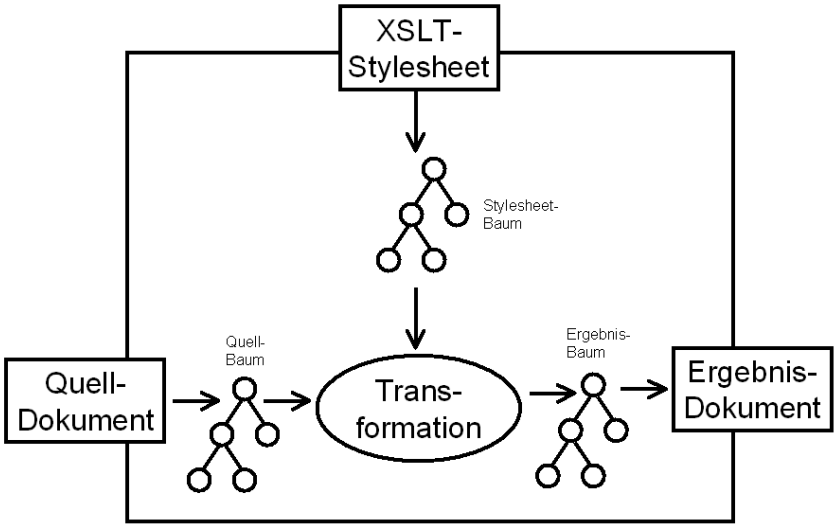


Abbildung 2.2: Transformationsmodell von XSLT

Es wird also der Stylesheetbaum (stylesheet tree) benutzt, um einen Quellbaum (source tree) in einen Ergebnisbaum (result tree) zu transformieren. Die Struktur des Ergebnisbaumes kann dabei völlig verschieden von der Struktur des Quellbaumes sein, da man die Möglichkeit hat, Knoten oder Knotenmengen des Quellbaumes zu löschen und neue Knoten einzufügen. Diese neuen Knoten sind Knoten des Stylesheetbaumes und werden in diesem Kontext als literale Ergebniselemente (literal result elements) bezeichnet. In folgendem Stylesheet wird das Element `<literal-ResultElement>` in den Ergebnisbaum eingefügt.

```
<xsl:template match="AAA">
    <literalResultElement>
        <xsl:apply-templates/>
    </literalResultElement>
</xsl:template>
```

Der Transformation selbst arbeitet nach einem einfachen Prinzip:

> «Durchlaufe alle Knoten des Quellbaumes und suche bei jedem Knoten das Template des Stylesheetbaumes, das am besten paßt. Sind alle Knoten durchlaufen, beende den Transformationsprozeß.»

Um festzustellen, welches Template am besten paßt, wird das Muster (Pattern) des jeweiligen Templates geprüft. Falls mehrere Templates gefunden werden, die die

gleiche Priorität haben, wird entweder eine Fehlermeldung ausgegeben oder das letzte passende Template verwendet. Als Template-Muster wird eine Untermenge von XPath verwendet (man kann als Achsen nur `child` oder `attribute` verwenden). Falls im Stylesheet keine Templates enthalten sind, werden mit den eingebauten Default-Templates aus dem XML-Dokument lediglich die Textknoten ausgegeben.

In XSLT kann man den Typ der Transformationsausgabe über das `method`-Attribut von `<xsl:output method="xxx">` einstellen. Möglich ist dabei `xml`, `text` und `html`. Dieses Attribut ist nützlich, um die Eigenheiten des Ausgabeformats besser behandeln zu können.

2.1.4 XSLT, XPath und SQL

Michael Kay vergleicht in seinem Buch «XSLT Programmer's Reference» die Programmiersprache XSLT mit SQL:

> «[..] when I first saw the XSL transformation language, XSLT, I realized that this was going to be the SQL of the web, the high-level data manipulation language that would turn XML from being merely a storage and transmission format for data into an active information source that could be queried and manipulated in a flexible, declarative way.»[Kay2000].

Diese Analogie zu SQL zeigt auch die Bedeutung von XSLT. Statt für den Zugriff auf verschiedene XML-Datenstrukturen auch verschiedene (SAX- oder DOM-basierte) Applikationen zu schreiben, die entsprechende Transformationen in andere Datenstrukturen durchführen, kann man mit XSLT Transformationen auf deklarative Weise über eine Menge von Regeln beschreiben. Das `SELECT`- Statement von SQL läßt sich mit einem XPath-Ausdruck vergleichen. Mit `SELECT` greift man auf relationale Datenstrukturen zu und mit XPath auf XML-basierte Datenstrukturen. Diese Analogie kann man noch weiter führen. Der Ausdruck

```
SELECT att2 FROM table WHERE att1 = 'chooseMe'
```

entspricht etwa dem folgenden XSLT-Code:

```
<xsl:if test="*::table[@att1 = 'chooseMe']">
    <xsl:value-of select="att2"/>
</xsl:if>
```

XSLT und XPath sind die Lösung (oder besser: das Tool für die Lösung) für eine Vielzahl von Problemen im XML-Bereich. Der XSLT-Standard wird allgemein anerkannt (sogar MSXML von Microsoft ist seit September 2000 weitgehend standardkonform) und erfreut sich breiter Unterstützung durch führende Hersteller. Für die Bearbeitung von XML-Dokumenten gibt es jedenfalls kaum eine Alternative zu XSLT und XPath.

2.1.5 XSLT-Applikationen

Viele gängige Programmiersprachen wie z.B. Java oder C++ sind imperative Programmiersprachen. XSLT dagegen ist eine *deklarative* Programmiersprache. Hauptunterschied zwischen deklarativen und imperativen Programmiersprachen ist, daß der Entwickler bei einer imperativen Programmiersprache im Quellcode festlegt, *wie* eine bestimmte Aufgabe erledigt werden soll. Bei deklarativen Programmiersprachen wird dagegen festgelegt, *was* für eine Ausgabe erwartet wird. In [Kay2000] ist ein schönes Beispiel zur Verdeutlichung abgedruckt:

```
<xsl:variable name="y">
    <xsl:call-template name="f">
        <xsl:with-param name="x">
    </xsl:call-template>
</xsl:variable>
```

Dieser XSLT-Code macht im Prinzip das gleiche wie `y = f(x)` in einer imperativen Sprache. Wozu sollte man diese - eher abschreckende - Syntax verwenden? XSLT wurde entwickelt, um XML-Dokumente (oder, genauer gesagt, Baumrepräsentationen von XML-Dokumenten) auf einfache Weise bearbeiten zu können. Mit XSLT können Applikationen, die komplexe Transformationen erfordern, relativ schnell programmiert werden. Der Aufwand, ähnliche Transformationen etwa in Java zu realisieren ist sehr hoch.

Das Schreiben einer komplexeren XSLT-Applikation ist oft schwierig. Es existieren noch keine allgemeingültigen, verbreiteten Standards für Entwurf und Implementierung. Die Anwendung von Methoden aus der objektorientierten Softwareentwicklung ist hier schwer möglich, da XSLT elementare Konzepte wie Vererbung und Polymorphie nicht kennt. Auch der Begriff «Objekt» ist in dieser Sprache nicht anwendbar. Trotzdem ist eine Aufteilung einer XSLT-Applikation in verschiedene, wiederverwendbare Module nicht nur sinnvoll, sondern auch notwendig, um den Überblick zu behalten. Es gibt auch erste Ansätze für XSLT-Entwurfsmuster (Design Patterns). Michael Kay hat bereits verschiedene Entwurfsmuster gesammelt; mittlerweile wurden auch weitere Patterns wie die «Muench'sche Methode»(siehe http://www.jenitennison.com entwickelt.

XSLT bietet zwei wichtige Konstrukte zur Aufteilung von Stylesheets in verschiedene Module: `<xsl:import>` und `<xsl:include>`. Beide Konstrukte sind eher mit Präprozessor- Direktiven (`#include` in C) zu vergleichen als mit dem `import`-Statement in Java, weil hier ein rein textuelles Einfügen von externen Stylesheets erfolgt.

Es gibt mehrere Unterschiede zwischen `<xsl:include>` und `<xsl:import>`. Der wichtigste Unterschied (aus dem sich alle weiteren Unterschiede ableiten) ist, daß ein Stylesheet, das andere Stylesheet-Module mit `<xsl:import>` einbindet, die eingebundenen Definitionen (d.h. vor allem die eingebundenen Templates) überschrei-

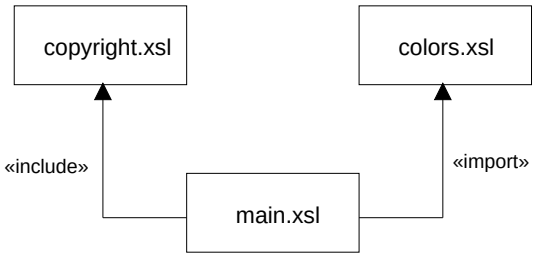


Abbildung 2.3: Verwendung von `<xsl:include>` und `<xsl:import>`

ben kann. Mit `<xsl:import>` eingebundene Module können also entsprechend den Anforderungen des Haupt-Stylesheets geändert werden.

Mit `<xsl:include>` eingebundene Module können nicht überschrieben werden. Die Einbindung mit `<xsl:include>` ist also besonders dann sinnvoll, wenn bestimmte Definitionen (z.B. ein Copyright-Hinweis) global, also für alle Stylesheets einer Firma (oder einer bestimmten Applikation), gelten sollen. Abbildung 2.3 zeigt ein Beispiel für die unterschiedliche Behandlung von `<xsl:include>` und `<xsl:import>`. Hier werden vom Haupt-Stylesheet zwei Module eingebunden: `copyright.xsl` und `colors.xsl`. Es ist nicht wünschenswert, daß ein Entwickler das Firmen-Copyright mit seinem eigenen Copyright überschreibt. Deshalb wird dieses Modul mit `<xsl:include>` eingebunden. Dagegen kann es vorkommen, daß in einer speziellen Applikation das globale Farbschema der Firma überschrieben werden muß. Also werden die Farbdefinitionen (`colors.xsl`) mit `<xsl:import>` eingebunden, z.B. weil einige Farben überschrieben werden müssen.

2.2 Jini

2.2.1 Einführung

Ein bekannter Werbespruch von Sun ist «The network is the computer». Mit Jini ist diese Vision ein Stück näher gerückt. Die Vision ist ein «Zero Administration»-Netzwerk, in dem Dienste angeboten und genutzt werden können. Ein Jini-fähiges Gerät (z.B. ein PDA, ein digitales Thermometer oder ein PC) soll in der Lage sein, sich völlig selbstständig in einen Jini-Verbund zu integrieren, Dienste anzubieten oder Dienste zu nutzen. Es sind verschiedenste Szenarien für den Einsatz von Jini denkbar: vom Videorekorder, der sich über das Netz programmieren läßt und einem Benutzer mitteilen kann, daß ein bestimmter Film leider ausgefallen ist bis zur Fernwartung von Industrierobotern.

«Jini» ist kein Akronym sondern eine etwas eigenwillige Schreibweise von «Genie» (Jee-Nee)[1]. Das Jini-Projekt wird von Bill Joy und Jim Waldo seit seiner Grün-

[1] In einigen Publikationen wird auch versucht, doch noch ein Akronym daraus zu machen («Java

dung 1994 geleitet. Erst fünf Jahre später (am 25. Januar 1999) wurde das Projekt der Öffentlichkeit vorgestellt. Das Ziel bei der Entwicklung war ein selbstorganisierendes Netzwerk, das aus Dienstanbietern und Dienstnutzern besteht (siehe Abbildung 2.4). Die Integration von neuen Diensten in das Netz soll nach dem «Plug and Play»-Prinzip erfolgen: ein Jini-fähiges Gerät wird ins Netz eingesteckt und registriert sich selbstständig bei einem Lookup-Service. Die Ziele von Jini (aus [WirelessDevice2000]) sind also, eine Netzwerkinfrastruktur bereitzustellen, die

- keine Installation erfordern («Zero Installation», d.h. es müssen keine Gerätetreiber installiert werden),
- keine Konfiguration erfordern («Zero Configuration», d.h es sind keine Netzwerkeinstellungen nötig) und die
- keine Wartung erfordern («Zero Maintenance», d.h. keine manuelle (De-)Installation von Diensten).

Jini ist allerdings bei näherem Hinsehen lediglich ein Framework bzw. eine Menge von Spezifikationen, die es Netzwerkkomponenten erlauben, sich selbständig in das Jini-Netz zu integrieren. Die wissenschaftlichen Grundlagen von Jini basieren vor allem auf den Forschungsarbeiten von David Gelernter und Nicholas Carriero, die unter anderem das Koordinationsmodell «Linda» entwickelt haben, das auf sogenannten «tuple spaces» basiert. Auf der Website der Linda Group (http://www.cs.yale.edu/Linda/linda.html) sind zahlreiche Veröffentlichungen dazu zu finden.

Das Jini-System läßt sich nach [Sun1999a] in drei Komponenten aufteilen:

- In eine Netzwerk-Infrastruktur,
- ein Programmiermodell und
- verschiedene Basisdienste.

Die Netzwerk-Infrastruktur stellt die wichtigsten Komponenten eines Jini-Netzes zur Verfügung, darunter einen Lookup-Service, der es ermöglicht, daß sich Clients und Services sich in den Jini-Verbund integrieren können. Dazu werden spezielle Protokolle wie *Discovery*, *Lookup* und *Join* unterstützt.

Das Programmiermodell hilft dem Entwickler dabei, Clients und Services zu entwickeln, die sich in einen Jini-Verbund einbinden lassen. Dazu stehen zahlreiche Schnittstellen (APIs) zur Verfügung, um mit *Leasing*, *Events* und *Transactions* arbeiten zu können.

Intelligent Network Infrastructure»). Ken Arnold von Sun hat allerdings auch festgestellt, daß auch das Anti-Akronym «Jini Is Not Initials» gut paßt.

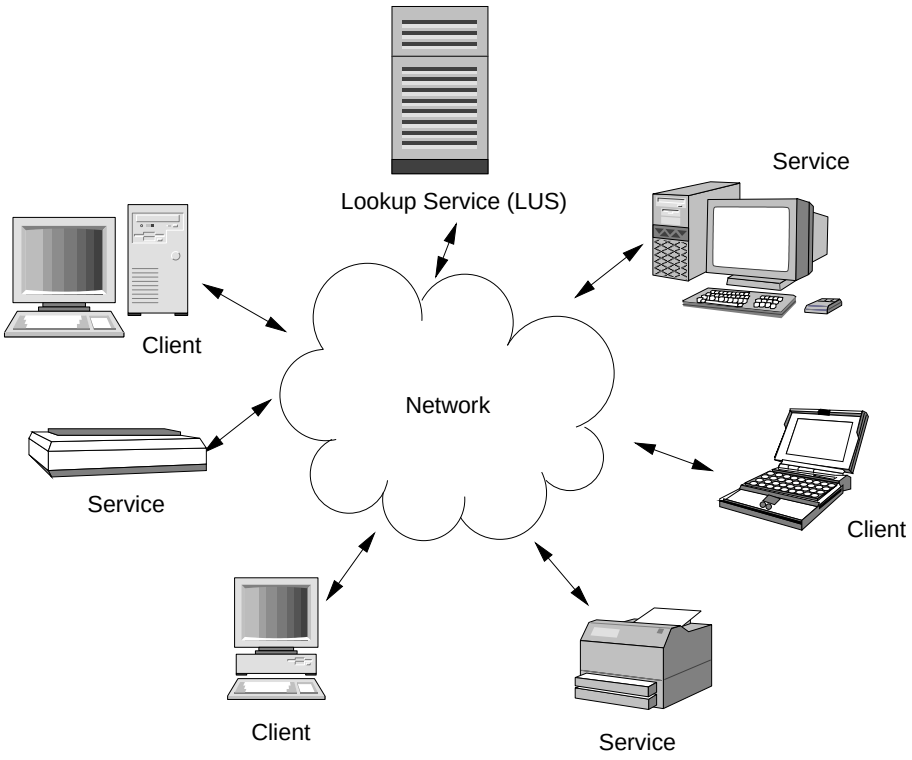


Abbildung 2.4: Aufbau eines Jini-Netzwerks

Außerdem werden noch einige wichtige Dienste zur Verfügung gestellt. Dazu gehören JavaSpaces (persistente Speicherung von Java-Objekten), ein Transaktions-Manager und ein Druckdienst.

Die Zentrale eines Jini-Netzwerks ist der Lookup-Service (LUS), bei dem sich die Dienste registrieren. Jini-Clients greifen auf den Lookup-Service zu, um festzustellen, welche Dienste überhaupt angeboten werden, bzw. welche Dienste gerade verfügbar sind. Technisch gesehen ist Jini eine «lightweight middleware», die die notwendige Infrastruktur für die Client-/Server-Kommunikation zur Verfügung stellt. Abbildung 2.5 zeigt, wie sich die Jini-Middleware einordnen läßt.

Für Jini hat Sun Microsystems eine neue Lizenz entwickelt: die Sun Community Source Licence (SCSL). Diese Lizenz erlaubt die kostenlose Nutzung des Jini-Softwarepaketes (einschließlich des Quellcodes) für Forschungszwecke. Für die kommerzielle Nutzung werden entweder 250.000 Dollar pro Jahr oder 10 Cent pro verkauftes, Jini-fähiges Gerät verlangt.

Interessant ist, daß sich die neue Lizenz an populäre Lizenzen wie die GPL anlehnt. Sun versucht auch, eine «Jini Community» zur Mitarbeit und Weiterentwicklung von Jini zu bewegen. Tatsächlich gibt es eine ganze Reihe interessanter Projekte von Firmen und Privatpersonen auf der Community-Seite http://www.jini.org, an denen man sich auch aktiv beteiligen kann. Zu den bekanntesten Projekten gehörten das ServiceUI-Projekt und das Surrogate- Projekt (die beide weiter unten besprochen werden).

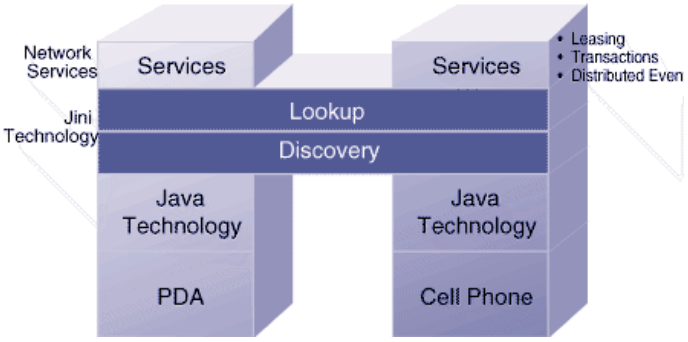


Abbildung 2.5: Das Jini-Schichtenmodell

2.2.2 Die Netzwerkinfrastruktur

Die Jini-Netzwerkinfrastruktur besteht aus Sicherheitskomponenten, Protokollen für Discovery und Join und aus einem Lookup-Service. Ohne diese Infrastruktur kann kein funktionsfähiges Jini-Netzwerk aufgebaut werden. Die folgenden Abschnitte beschreiben die einzelnen Komponenten der Infrastruktur genauer.

Der Lookup-Service: Finden von Diensten

Ein «Plug-and-Play» Netzwerk muß einen Dienst zur Verfügung stellen, der das Auffinden von Diensten (Services) für Clients transparent hält. In einem Jini-Netzwerk ist dafür der Lookup-Service (LUS) zuständig. Jini-Dienste können den Lookup-Service über das *Discovery*-Protokoll ausfindig machen. Dazu schickt der Dienst Uni- oder Multicastpakete ins Netzwerk. Ein Lookup-Service, der diese Pakete empfängt, schickt ein Proxy-Objekt (den sogenannten *Registrar*) zum Dienst zurück. Über diesen Registrar registriert sich der Dienst beim LUS (siehe Abbildung 2.6). Das Protokoll, das bei der Registrierung verwendet wird, ist das *Join*-Protokoll. Normalerweise spricht man bei diesem Protokoll-Paar vom *Discovery-and-Join*-Protokoll. Nach der Registrierung bei einem (oder mehreren) Lookup-Services ist der entsprechende Dienst in den Jini-Verbund integriert. Der LUS ist also eine zentrale Registrierstelle für Jini-Dienste.

Sobald ein Dienst bei einem LUS registriert ist, kann er von einem Client über das *Lookup*-Protokoll gefunden und genutzt werden. Dazu muß der Client zunächst einen LUS auffinden (Discovery-Protokoll). Dann kann er über das Registrar-Objekt Dienste eines bestimmten Typs ausfindig machen. Der Typ eines Dienstes wird durch sein Java-Interface (das dem Client bekannt sein muß) festgelegt. Abbildung 2.7 zeigt den Vorgang.

Eine Implementierung des Lookup-Services wird von Sun mit dem Jini-Paket mitgeliefert. Diese Implementierung heißt *Reggie*. Es ist abzusehen, daß in Zukunft der LUS (und andere Standard-Dienste) auch in kommerziellen Implementierungen

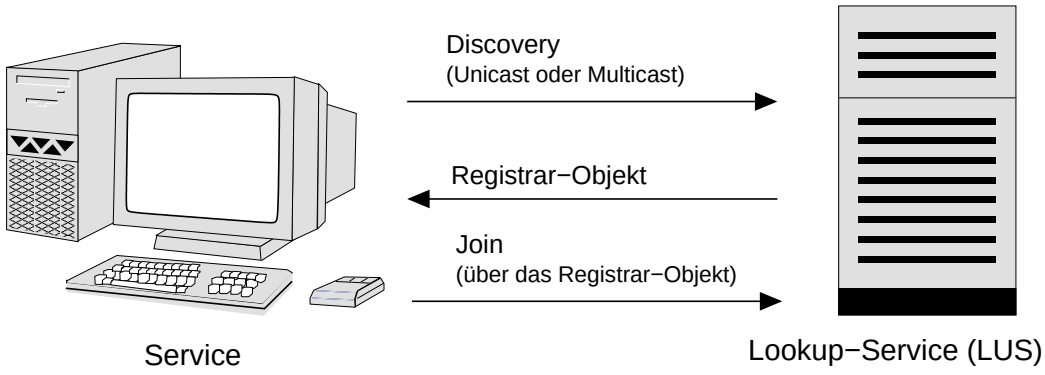


Abbildung 2.6: Discovery- und Join-Protokoll

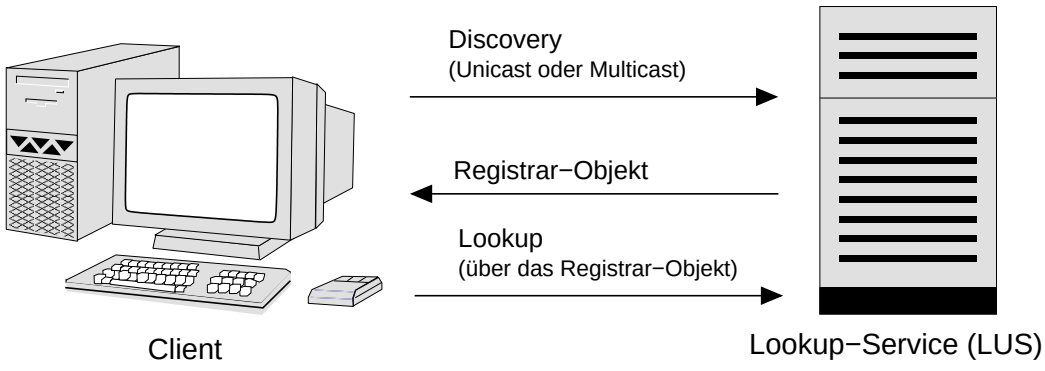


Abbildung 2.7: Lookup-Protokoll

angeboten werden. Die Erfahrungen mit Enterprise Java Beans (EJBs) zeigen, daß Firmen recht schnell mit eigenen Implementierungen auf den Markt drängen.

Discovery

Das Discovery-Protokoll wurde schon im Abschnitt über den Lookup-Service (siehe 2.2.2) kurz angesprochen. Prinzipiell dient das Protokoll dazu, es Clients und Services zu ermöglichen, einen Lookup-Service aufzufinden. Hier geht es nun um die Details des Discovery-Protokolls.

Die Clients bzw. Services versuchen, den LUS selbständig über das Multicast Request Protocol (MRP) oder über das Unicast Discovery Protocol zu finden. Alternativ verschickt der LUS in regelmäßigen Abständen Pakete über das Multicast Announcement Protocol (MAP), auf die Clients und Services auch reagieren (sollten).

Unicast Discovery Protocol Das Unicast Discovery Protocol wird von Clients und Services verwendet, wenn die IP-Adresse des LUS bekannt ist. Falls keine Standard-Portnummer (4160) verwendet wird, muß zusätzlich noch die Portnummer bekannt sein. Dazu wird die Klasse `net.jini.core.discovery.LookupLocator` benötigt, deren Konstruktoren einen gültigen Uniform Resource Locator (URL) bzw. einen gültigen URL und eine gültige Portadresse als Argumente erwarten.

Ein Lookup-Server hört ständig Port 4160 (Standard) auf einkommende Pakete ab (sowohl Unicast als auch Multicast). Empfängt er auf diesem Port ein Paket eines Dienstes, sendet er ein «Registrar»-Objekt an den Dienst zurück.

Das Unicast Discovery Protocol sollte allerdings nicht das einzige Discovery Protokoll sein, das die Jini-Komponenten verwenden können. Wenn der Standard-LUS ausfällt, muß es ohne Probleme möglich sein, einen LUS auf einem anderen Rechner zu starten. Clients und Services sollten dann auch in der Lage sein, selbständig den neuen LUS zu finden.

Multicast Request Protocol Ein Dienst, der sich bei einem LUS registrieren und damit einem Jini-Verbund beitreten will, kennt normalerweise nicht den Standort des LUS. Also sendet er Multicast-Pakete ins Netz, um den LUS ausfindig zu machen. Bei Multicast werden Pakete an spezielle Multicast-Adressen verschickt. Es ist also nicht nötig, Pakete an alle Adressen innerhalb eines lokalen Netzes zu verschicken (Broadcast). Nähere Informationen zu Multicast sind in [Sing2000] und [Tanenbaum1996] zu finden. Abbildung 2.8 verdeutlicht die Abläufe beim Discovery mit dem Multicast Request Protocol.

Ein LUS hört ständig Port 4160 der Multicast-Gruppe 224.0.1.85 ab (das ist eine zwingende Forderung der Spezifikation). Diese Multicast-Adresse ist bei der IANA

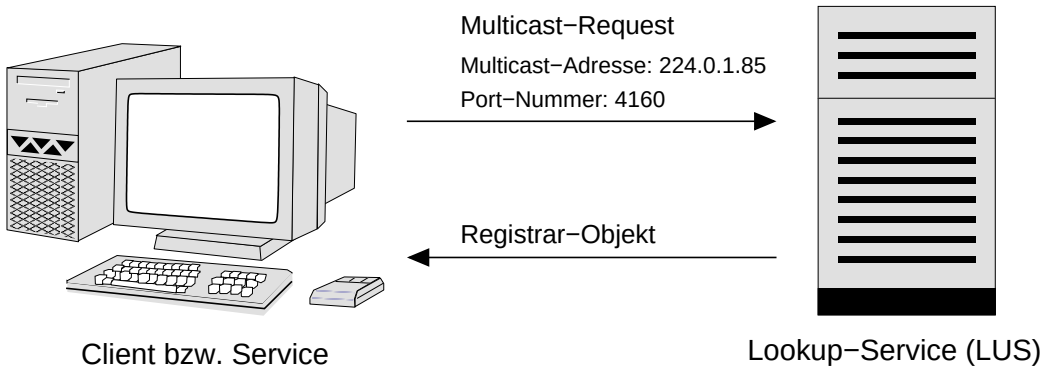


Abbildung 2.8: Multicast Discovery

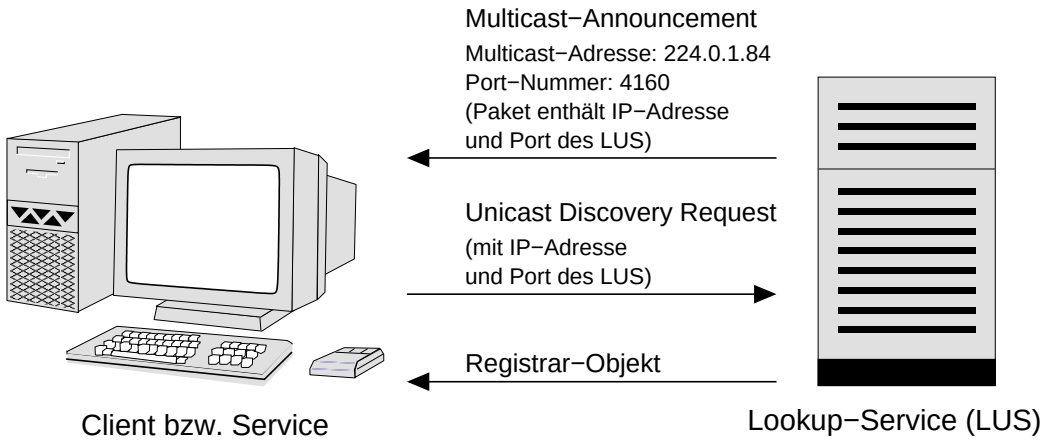


Abbildung 2.9: Multicast Announcement

(http://www.iana.org) als «jini-request» registriert. Weder Adresse noch Port können vom Benutzer geändert werden (siehe `net.jini.discovery.Constants`)[2].

Multicast Announcement Protocol Mit Hilfe des Multicast Announcement Protocol können sich Lookup-Services im Jini-Netz bekannt machen. Beim Neustart eines LUS (danach normalerweise alle 120 Sekunden) werden Pakete zum Port 4160 der Multicast-Gruppe 224.0.1.84 geschickt. Bei der IANA ist diese Gruppe als «jini-announcement» registriert.

Clients und Services hören diese Multicast-Adresse auf Pakete von Lookup-Services ab und registrieren sich, wenn sie Pakete von einem unbekannten LUS empfangen. Das Multicast Announcement Protocol ist also sehr wichtig, damit Jini-Komponenten auf den Ausfall und die Wiederverfügbarkeit eines LUS entsprechend reagieren können.

[2] Das ist auch nicht notwendig, da mehrere Prozesse eines Betriebssystems gleichzeitig auf den Multicast-Port zugreifen können.

Join

Nach dem *Discovery* (siehe Abschnitt 2.2.2) registriert sich ein Dienst üblicherweise beim Lookup-Service. Diese Registrierung entspricht einer Integration in den Jini-Verbund und wird über das *Join*-Protokoll abgewickelt. Das Join-Protokoll legt dabei nicht nur die Details fest, wie ein Dienst in ein Jini-Netz aufgenommen werden soll, sondern beschreibt auch konkrete Anforderungen, die ein Dienst erfüllen soll, damit das Jini-System dauerhaft stabil läuft. So soll ein Dienst beispielsweise die Service-ID, die ihm vom LUS zugewiesen wurde, persistent abspeichern, um sich bei einem Absturz und anschließendem Neustart wieder korrekt beim LUS anmelden zu können. Um die Details des Join-Protokolls für den Benutzer transparent zu halten, hat Sun die Klasse `net.jini.lookup.JoinManager` zur Verfügung gestellt. Allerdings ist auch so noch etwas Handarbeit notwendig, um das Join-Protokoll vollständig zu unterstützen. Das ist zwar nicht zwingend erforderlich, sollte jedoch in Produktionssystemen korrekt implementiert sein.

Lookup

Das Lookup-Protokoll wird von Clients benutzt, um gewünschte Dienste ausfindig zu machen. Dabei hat ein Client weitreichende Möglichkeiten, einen bestimmten Dienst ausfindig zu machen. Bei der Registrierung von Diensten werden die folgenden Einträge gespeichert:

- Das Proxy-Objekt des Dienstes.
- Die eindeutige ID des Dienstes.
- Attribute des Dienstes.

Ein Client kann nun nach genau diesen Einträgen suchen. Das geschieht durch ein Template, das der Client selbst zusammenstellen kann. Etwas technischer: Der Client konstruiert ein `ServiceTemplate` mit den Argumenten Service-ID, Typ des Dienstes und mit einem Array aus `Entry`-Attributen. Ist eines dieser Argumente nicht bekannt oder nicht relevant, kann stattdessen `null` angegeben werden. Interessant ist das Array aus `Entry`- Attributen.

Entries oder *Entry*-Objekte sind Attribute oder Eigenschaften eines Dienstes bzw. dessen Proxy-Objekts. Ein Arbeitsgruppen-Dienst könnte z.B. Attribute haben, die den Namen, die Abteilung und den Standort festlegen. Auch bei einem Druckdienst ist es sinnvoll, in Attributen die Art des Druckers (Schwarzweiß oder Farbe, einseitig oder doppelseitig, ...) und den Standort zu beschreiben. Technisch gesehen implementieren Attribute (z.B. `Location`) das Interface `net.jini.core.entry.Entry`, das jedoch keine Methoden oder Attribute festlegt sondern ausschließlich als Markierungs-Interface dient.

Dieses einfache, aber mächtige Modell erlaubt vielfältige Suchmöglichkeiten nach bestimmten Diensten. Vermutlich werden in Kürze die ersten Service-Browser auftauchen, die eine leicht bedienbare Schnittstelle zum Suchen von Diensten bieten.

2.2.3 Das Programmiermodell

Das Jini-Programmiermodell besteht grundsätzlich aus einer Menge von Schnittstellen, die dem Entwickler zur Verfügung stehen, um Jini-Clients und Jini-Services programmieren zu können. Die wichtigsten Schnittstellen sind

- die Leasing-Schnittstelle,
- die Transaktionsschnittstelle und
- die Event- und Notification-Schnittstelle.

Diese Schnittstellen ermöglichen die Entwicklung von robusten und stabilen Jini-Anwendungen. Insbesondere die Leasing-Schnittstelle ist essentiell für die Selbstheilungsfähigkeiten des Jini-Netzes. Die Transaktionsschnittstelle wird im folgenden nicht mehr behandelt.

Leasing

Wenn sich ein Jini-Dienst bei einem Lookup-Service registriert, ist die Registrierung stets nur eine bestimmte Zeit gültig. Bei einer Registrierung muß also zwischen dem LUS und dem Dienst eine Zeitzusicherung ausgehandelt werden[3]. Diese Zeitzusicherung wird als *Leasing* bezeichnet. Der Dienst muß sich selbst darum kümmern, daß die Registrierung in regelmäßigen Abständen aufgefrischt wird. Grund für diese zeitliche Restriktion ist, daß das Jini-Netzwerk dauerhaft stabil sein soll und dieser stabile Zustand ohne Administrator-Eingriffe erhalten bleibt. Bei einem Absturz eines Jini-Dienstes, eines Netzwerkfehlers oder einem Hardwarausfalles ist lediglich der entsprechende Dienst betroffen. Kann der Dienst seine Registrierung nicht mehr auffrischen, wird er nicht mehr als Teil des Jini-Netzes angesehen. Der Lookup-Service gibt dann keine Service-Objekte mehr an Clients weiter und der Dienst ist im Netz nicht mehr sichtbar. Nach einem Neustart kann der Dienst sich erneut beim LUS registrieren und am Jini-Verbund teilnehmen.

Damit wird eine Referenz auf ein Objekt nicht - wie sonst üblich - unbegrenzt lange gehalten, sondern nur für einen begrenzten Zeitraum. Im Falle eines Netzwerkfehlers oder des Ausfalls eines Dienstes kann die Referenz des LUS auf den Dienst (bzw. auf das Proxy-Objekt des Dienstes) jederzeit wieder hergestellt werden.

[3] «Aushandeln» ist eigentlich das falsche Wort: Der Dienst schlägt eine bestimmte Zeit vor und der LUS nimmt die Zeit an, lehnt sie ab oder vergibt eine geringere Zeitzusicherung.

Verteilte Events

Das Standard-Eventmodell von Java ist auf lokale Events beschränkt. Das heißt, es ist nicht möglich, Events, die im Kontext einer VM ausgelöst wurden an andere Virtuelle Maschinen weiterzugegeben. Jini erweitert das Standardmodell um einen Verteilungsaspekt. Es wird damit Objekten ermöglicht, sich für *distributed events* zu registrieren. Wenn ein Event in der VM der Eventquelle ausgelöst wird, leitet der `EventGenerator` den Event an den `EventListener`, der sich registriert hat, weiter. Im Detail sieht das so aus, daß die `notify()`-Methode des für Events registrierten Objekts aufgerufen wird.

Die verteilten Events werden von der Jini-Infrastruktur ausgiebig verwendet. Wenn ein Dienst einen LUS ausfindig machen will, kann er z.B. einen `DiscoveryListener` implementieren, um von aufgefundenen Lookup-Services automatisch zurückgerufen zu werden. Die neuen verteilten Events sind dabei sehr fehlertolerant (ein verteilter Event kann z.B. verzögert oder auch gar nicht ankommen).

2.2.4 Die Service-UI-Spezifikation

Die Service-UI-Spezifikation ist aus einem Jini-Community-Projekt hervorgegangen und beschreibt einen Ansatz, wie sich Jini-Diensten Benutzerschnittstellen (Service-UIs) auf einfache Weise zuordnen lassen. Bisher war die clientseitige Darstellung der Benutzerschnittstelle eines Dienstes applikations- oder dienstspezifisch. Das heißt, daß der Dienst das UI in binärer Form bereitstellt und bei Anforderung zum Client schickt. Problematisch bei diesem Ansatz sind folgende Punkte:

- Ein Dienst kann unter Umständen nicht die Benutzerschnittstellen für alle möglichen Zielplattformen erzeugen, z.B. weil er nicht über bestimmte Bibliotheken verfügt.
- Die Lokalisierung eines UI kann nur clientseitig erfolgen, da auf dem Server z.B. nicht immer alle möglichen Zeichensätze installiert sind (asiatische Schriftarten sind selten auf westlichen Windows-Installationen zu finden).

Mit der Service-UI-Spezifikation (die hoffentlich in das nächste offizielle Jini-Release einfließt) ist es möglich, die Applikationslogik eines Dienstes von seiner Benutzerschnittstelle zu entkoppeln.

Abbildung 2.10 zeigt die Interaktion eines Benutzers mit einem Dienst bei Verwendung der Service UI-Spezifikation. Man sieht deutlich, daß das UI-Objekt vom Proxy des Dienstes vollständig getrennt ist. Das hat für den Entwickler den Vorteil, daß bei einer Änderung des UI-Toolkits (AWT, Swing), der Plattform (Desktop, PDA, Mobiltelefon) oder der Modalität (GUI, Sprache) nur das UI-Objekt geändert werden muß. Für den Benutzer liegt der Vorteil dieser Trennung von Benutzerschnittstelle und

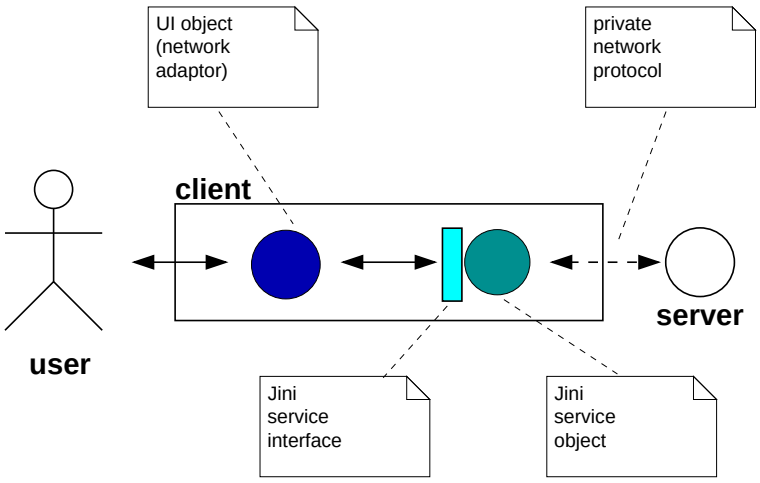


Abbildung 2.10: Interaktion eines Benutzers mit einem Dienst (aus [Venners2000])

Applikationslogik darin, daß er auf jeder unterstützten Plattform für die Nutzung des Dienstes eine angemessene Benutzerschnittstelle zur Verfügung gestellt bekommt.

Die Service-UI-Spezifikation versucht also, die Entkopplung von UI und Dienst zu standardisieren. Die Spezifikation definiert einheitliche Schnittstellen, die ein Client ohne vorheriges Wissen über den Dienst oder das User Interface ansprechen kann. Außerdem zeigt die Spezifikation Möglichkeiten, wie ein Client nach einem bestimmten User Interface suchen kann. Denkbar ist z.B. daß auf einer Plattform nur das AWT-Toolkit zur Verfügung steht und Swing-GUIs deshalb nicht dargestellt werden können. Der Client sollte in diesem Fall einfach angeben können, daß er eine AWT-basierte Benutzerschnittstelle benötigt.

Nach der Spezifikation muß ein Dienst-Anbieter dem Client drei Komponenten zur Verfügung stellen (aus [Venners2000]):

- die Benutzerschnittstelle selbst
- eine Benutzerschnittstellen-Fabrik, die die Schnittstelle generiert und
- einen UI-Deskriptor, der die Benutzerschnittstelle beschreibt und eine serialisierte Instanz der Fabrik enthält.

Ein Dienst, der die Service-UI-Spezifikation implementiert, hat ein `UIDescriptor`-Attribut. Dieses Attribut ist vom Typ `Entry` und beschreibt die Benutzerschnittstelle des Dienstes. Außerdem enthält dieses Attribut vier Felder, mit denen man die Benutzerschnittstelle genau beschreiben kann. Tabelle 2.1 (aus [Sing2000]) gibt einen Überblick über die einzelnen Felder.

Die Idee hinter dieser Architektur ist klar. Zum einen muß sich der Lookup-Service nicht um die (üblichen) zahlreichen GUI-Klassen kümmern und zum anderen kann

Feldname	Beschreibung
`role`	Ein `String`, der die Rolle der Benutzerschnittstelle beschreibt
`toolkit`	Ein `String`, der die Bibliothek eines für das UI erforderlichen Toolkits beschreibt (`javax.swing`, `java.awt`)
`attributes`	Ein `Set` von Attributen, die das UI beschreiben.
`factory`	Ein `MarshalledObject`, das die UI-Fabrik enthält, die verwendet werden kann, um ein UI zu instanzieren

Tabelle 2.1: Felder eines UI-Deskriptors (aus [Sing2000])

ein Client *vor* dem «Auspacken» des `factory`-Objektes anhand der einzelnen Felder des UI-Deskriptors feststellen, ob das enthaltene User-Interface überhaupt einsetzbar ist.

Ein `Factory`-Objekt (also die Fabrik, die das UI erzeugt) hat eine oder mehrere Methoden, die eine Benutzerschnittstelle instanzieren und zurückgeben. Die Klasse `FrameFactory` im Paket `net.jini.lookup.ui.factory` z.B. bietet nur eine Methode an: `getFrame()`. Die Fabrik kann sich über den UI-Deskriptor unter anderem über die Rolle (MainUI, AdminUI, AboutUI) und über das Toolkit informieren. Der UI-Deskriptor enthält neben den entsprechenden Attributen auch eine serialisierte Instanz des Fabrik-Objektes.

Mit einem Dienst, dessen Benutzerschnittstelle in Form von Entries beschrieben ist, hat ein Client die Möglichkeit, direkt nach einem Dienst zu suchen, der eine passende Benutzerschnittstelle anbietet. Das ist auch eine Herausforderung für die Dienstanbieter, weil es von Vorteil für Kunden ist, ein Gerät (z.B. einen Drucker) zu benutzen, der über zahlreiche Benutzerschnittstellen verfügt und deshalb auch z.B. von einem Palm Pilot aus bedient werden kann.

2.2.5 Jini auf mobilen Geräten

Um mobile Kleingeräte (z.B. Palm PDA, Mobiltelefon) an einen Jini-Verbund anzuschließen, muß auf den Geräten eine VM installiert sein, welche die J2SE (Java 2 Standard Edition) unterstützt. Leider ist die J2SE zu umfangreich, um sie vollständig auf Geräten mit beschränkten Ressourcen implementieren zu können. Deshalb wurde von Sun die Java 2 Micro Edition (J2ME) entwickelt, die auch auf Geräten mit eingeschränkter Prozessorleistung und eingeschränktem Speicher laufen kann.

Der folgende Abschnitt befaßt sich kurz mit der Java 2 Micro Edition, geht allerdings nur auf Aspekte ein, die im Rahmen dieser Master Thesis und des zu entwickelnden Prototypen interessieren. In den weiteren Abschnitten geht es um verschiedene Ansätze, wie mobile Kleingeräte in einen Jini-Verbund eingebunden werden können. Näher betrachtet werden dabei die Jini Surrogate Spezifikation und ein Forschungsprojekt an der Universität Hamburg.

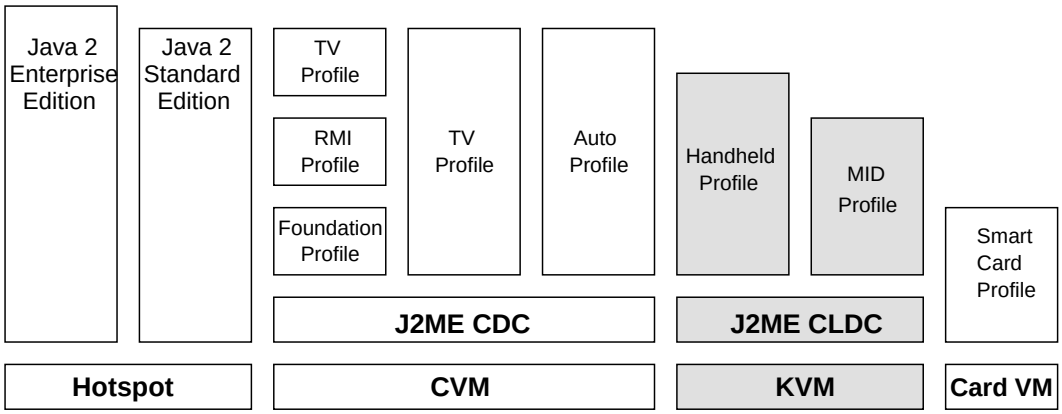


Abbildung 2.11: Konfigurationen und Profile der J2ME (aus [Day2000])

Die Java 2 Micro Edition (J2ME)

Die Java 2 Micro Edition (J2ME) wurde entwickelt, weil Sun Microsystems erkannt hat, daß die Standard-VM nicht allen Ansprüchen genügt. Es gibt jetzt insgesamt drei Java-Editionen:

- Die Java 2 Enterprise Edition (J2EE) für die Entwicklung von großen, skalierbaren Enterprise-Applikationen.
- Die Java 2 Standard Edition (J2SE) für die Entwicklung von Desktop-Applikationen.
- Die Java 2 Micro Edition (J2ME) für die Entwicklung von Applikationen für Unterhaltungselektronik und für eingebettete Systeme.

Die J2ME wird auch für mobile Telefone und PDAs verwendet. In diesem Markt gibt es jedoch zahlreiche verschiedene Geräte, so daß sich z.B. für Benutzerschnittstellen-APIs nur schlecht brauchbare Schnittmengen definieren lassen. Das Prinzip des «kleinsten gemeinsamen Nenners» ist hier nicht anwendbar, da APIs für Displays von mobilen Telefonen und APIs für farbige Displays von PDAs wie dem Palm Pilot sehr unterschiedlich gestaltet werden müssen.

Deshalb wurde die J2ME in Konfigurationen (*Configurations*) und Profile (*Profiles*) unterteilt. Abbildung 2.11 zeigt einen Überblick über verschiedene Konfigurationen und Profile.

Die Core-APIs für bestimmte Geräteklassen werden in den Konfigurations-Spezifikationen festgelegt. Es gibt hier vor allem zwei Konfigurationen: die CLDC (Connected Limited Device Configuration) und die CDC (Connected Device Configuration). Die CLDC-Konfiguration wurde für Geräte mit 128 bis 512k RAM entwickelt, die CDC-Konfiguration erwartet mindestens 512k RAM. Die Referenzimplementierung der CLDC-Konfiguration ist die KVM (K Virtual Machine). Das «K» in KVM steht für «Kilobyte». Die CLDC wurde primär für Geräte entwickelt, die

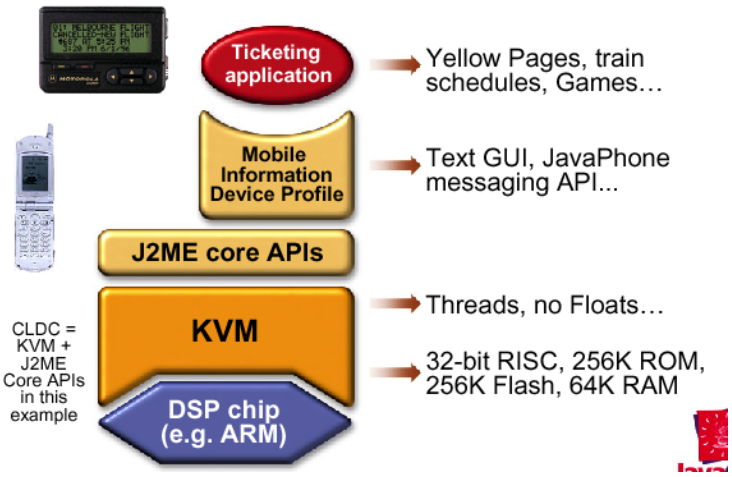


Abbildung 2.12: Beispielapplikation für KVM/CLDC/MIDP (aus [Day2000])

- keine schnelle Netzwerkverbindung,
- ein eingeschränktes Display,
- eine eingeschränkte Prozessorleistung und
- eine geringe Speicherkapazität haben.

Wegen den Einschränkungen dieser Geräte wurden dann verschiedene Fähigkeiten der Standard-VM aus der CLDC-Spezifikation entfernt. So fehlt z.B. die Fließkomma-Unterstützung, das Java Native Interface (JNI) und die Reflection API. Da die Reflection-API komplett fehlt, ist auch weder Remote Method Invocation (RMI) noch Objektserialisierung verfügbar. Es wurde weder eine Benutzerschnittstelle noch eine Datenbank-API (z.B. JDBC) spezifiziert. Solche APIs werden von den Profilen (MIDP, PDAP) zur Verfügung gestellt. Abbildung 2.12 veranschaulicht das Zusammenspiel von Konfiguration und Profil bei einer Beispielapplikation.

Für Details zur CLDC sei hier auf die CLDC-Spezifikation (siehe [Sun2000d]) verwiesen. Die Profile bauen auf den Konfigurationen auf und sind eher anwendungs- als gerätebezogen (obwohl hier die Grenzen fließend sind). Ein Profil für Set-Top-Boxen (TV-Profil) könnte in den APIs Methoden wie `setVolume()` oder `setChannel()` anbieten (siehe [Wong2000]). Set-Top-Boxen sind nun der CDC-Konfiguration zugeordnet, das TV-Profil könnte aber auch für CLDC-Geräte wie ein (zukünftiges) mobiles UMTS-Telefon angewandt werden. Zur Zeit aktuell (d.h. in einer Implementierung vorliegend) ist lediglich das MIDP-Profil (Mobile Information Device Profile), das sich vor allem für mobile Telefone eignet. Nähere Informationen findet man unter http://java.sun.com/products/midp/index.html.

Die «Jini Surrogate Specification»

Die Jini Surrogate Spezifikation ist aus einem Jini-Community-Projekt (siehe http://www.jini.org) entstanden. Diese Spezifikation befaßt sich mit dem Problem, Geräte mit eingeschränkten Ressourcen in ein Jini-Netzwerk einzubinden. Für PDAs mit dem PalmOS- Betriebssystem z.B. existieren zwar verschiedene Virtuelle Maschinen[4]; leider implementieren diese VMs zumeist nur die CLDC-Konfiguration der Java 2 Micro Edition. Das heißt, daß viele Features der Standard-VM hier nicht vorhanden sind. Es ist z.B. nicht möglich, über RMI Methoden eines Dienstes aufzurufen oder dynamisch Klassen über ein Netzwerk zu laden. Die Jini-Architektur ist zwar nicht unbedingt auf RMI angewiesen (obwohl es schwierig ist, darauf zu verzichten), aber das dynamische Nachladen von Klassen ist für die Einbindung eines Gerätes in einen Jini-Verbund notwendig. Wie bindet man nun, z.B. einen Palm Pilot in einen Jini-Verbund ein? Dazu ist ein Rechner erforderlich, auf dem eine Standard-VM laufen kann. Auf diesem Rechner läuft dann ein Proxy, der sich stellvertretend für das nicht-jinifähige Gerät im Jini-Verbund registriert. Für andere Jini-Clients und -Services ist dabei völlig transparent, daß sie nur mit einem Proxy kommunizieren. Der Proxy leitet Anfragen bzw. Ergebnisse über proprietäre Protokolle an das nicht-jinifähige Gerät weiter.

Bei der Entwicklung der Surrogate-Spezifikation wurden verschiedene Anforderungen festgelegt (aus [Sun2000c]). Die Architektur soll:

- unabhängig vom Gerätetyp sein,
- unabhängig vom Netzwerktyp sein und
- das «Plug-and-Work» von Jini erhalten.

In der Surrogate Spezifikation wird ein generischer Ansatz beschrieben, wie Geräte mit eingeschränkter oder gar keiner VM in den Jini-Verbund integriert werden. Prinzip dieser Architektur ist, daß ein Jini-fähiges Gerät (z.B. ein PC) eine Schnittstelle anbietet, über die das Gerät mit der eingeschränkten VM am Jini-Netzwerk teilnehmen kann.

Abbildung 2.13 zeigt einen Überblick über die Surrogate-Architektur. Der *surrogate host* ist ein Dienst, der eine Umgebung für die Ausführung des *surrogate* (des Proxy-Objektes) bietet. Es wird angenommen, daß der Surrogate Host in einer *host capable machine* ausgeführt wird, also in einem Java- und Jini-fähigen Rechner, der sowohl mit dem Jini-Netz als auch mit dem Gerät verbunden ist. Das Surrogate ist binärer Java-Code, der das Gerät repräsentiert. Dieser Code vom *interconnect adaptor* geladen, wenn das Gerät entdeckt wurde (Discovery). Es ist auch möglich, daß das Gerät das Surrogate selbst zum Interconnect Adaptor schickt oder daß das Surrogate z.B. von einem Webserver geladen wird. Der Interconnect Adaptor

[4] z.B. J9 von IBM, Waba von Wabasoft, KVM von Sun oder die VM von Esmertec

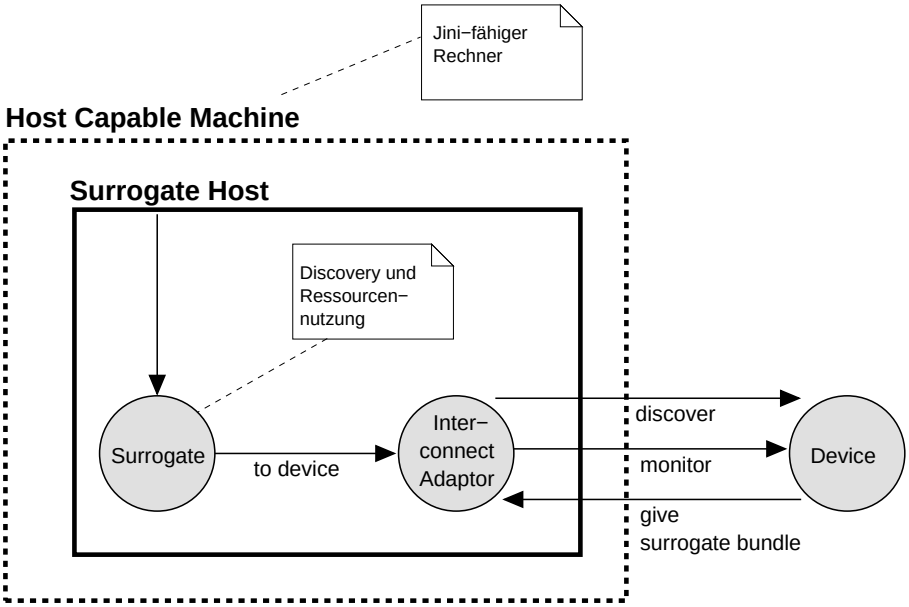


Abbildung 2.13: Die Surrogate-Architektur (aus [Sun2000c])

ist auch für das Discovery und die Überwachung des Gerätes zuständig. Wenn das Gerät ausfällt, wird das Surrogate vom Surrogate Host deaktiviert. Wenn der Interconnect Adaptor festellt, daß das Gerät wieder verfügbar ist (Discovery), wird das Surrogate wieder aktiviert.

Leider sind einige Teile der Surrogate Architektur noch nicht vollständig spezifiziert. Außerdem gibt es zur Zeit (Anfang 2001) noch keine öffentlich zugänglichen Implementierungen. Die Spezifikation bietet allerdings viele gute Ansätze, so daß erste Implementierungen wohl nicht viel länger auf sich warten lassen werden. Im Rahmen dieser Master Thesis wurde die Spezifikation allerdings nicht implementiert. Nähere Informationen zur Surrogate Architektur sind in [Sing2000] und [Sun2000c] zu finden.

Das Forschungsprojekt «Hydepark»

«Hydepark» steht für «Hyper Distributed Environment for Personal Appliances». Es handelt sich dabei um ein Forschungsprojekt der «Distributed Systems Group» des Instituts für Informatik an der Universität Hamburg (http://vsys-www.informatik.uni-hamburg.de/. Ziel des Projekts ist es, «eine generische Integrationsplattform für mobile Geräte zu realisieren» ([VSYSa]).

Es soll also die Möglichkeit geschaffen werden, mobile Geräte ohne Java-Unterstützung in ein Jini-Netzwerk zu integrieren ([VSYSb]). Dazu wurde eine C-Implementierung der Jini-Protokolle entwickelt. Als wichtiges Teilprojekt wird eine «Java Border Service Architecture» (JBSA) entwickelt, die es ermöglichen soll, Benutzerschnittstellen für Jini-Dienste auch für Geräte ohne Java-Unterstützung zur

Verfügung zu stellen. Dazu wird - ähnlich anderen Ansätzen - eine zusätzliche Abstraktionsschicht zwischen Applikationsschicht und Präsentationschicht eingefügt. Diese zusätzliche Abstraktionsschicht basiert auf einer XML-Beschreibung der Benutzerschnittstelle. Diese XML-Beschreibung wird - abhängig von den Möglichkeiten des Gerätes, das einen Dienst nutzen will - in eine konkrete Repräsentation transformiert. Als mögliche konkrete Repräsentationen der Benutzerschnittstelle werden von der Forschungsgruppe die Sprachen XHTML, WML, VoiceXML und VRML angesehen. Das Prinzip von Hydepark ist in [VSYSa] und [VSYSb] knapp beschrieben.

Interessant ist der Ansatz, eine UI-Beschreibung in XML zu generieren. Dabei scannt ein «application shadow», der in derselben VM wie die eigentliche Applikation läuft, die Objekthierarchie, die die Benutzerschnittstelle darstellt. Aus dem Scan wird in regelmäßigen Abständen die XML-Beschreibung generiert ([VSYSb].

In [VSYSc] wird auf Details zu Jini und auf die Probleme mit dem Anschluß von mobilen Geräten ohne Java-Unterstützung näher eingegangen. Dabei wird eine Modifikation des Jini Discovery-Protokolls vorgeschlagen, die die Kommunikation zwischen Jini-Diensten und -Clients und dem Lookup-Service auch ohne RMI ermöglicht. Dabei wird ein TCP-Socket geöffnet, über den der Service-Proxy an den Client übertragen wird. Dieser Socket wird bleibt für die gesamte Dauer der Sitzung geöffnet und dient als Kommunikationskanal. Auch der gesamte Leasing-Mechanismus wird über diese Socket-Verbindung abgewickelt. Der Prototyp der Forschungsgruppe enthält einen modifizierten Lookup-Service, auf den mit RMI (für Java-Clients) bzw. über eine Socket-Verbindung zugegriffen werden kann. Die «Java Border Service Architecture» (JBSA) wird in [VSYSd] genauer beschrieben.

2.2.6 Alternativen zu Jini

Es gibt natürlich auch Alternativen zu Jini. Die Unterschiede zwischen Jini und den alternativen Technologien sind manchmal sehr subtil und oft ist ein direkter Vergleich nicht möglich. Sun Microsystems schreibt deshalb in [Sun2000a] (den Jini-FAQs) auf die Frage «What other technologies compete with the Jini architecture?» auch einfach «*None.*». Das ist natürlich nicht ganz korrekt. Es gibt durchaus vergleichbare Technologien:

- UPnP: Universal Plug and Play (http://www.upnp.org)
- Salutation (http://www.salutation.org)
- Inferno (http://www.vitanuova.com/)
- HAVi: Home Audio/Video interoperability (http://www.havi.org)

Diese Technologien werden von Firmen bzw. Konsortien unterschiedlich positioniert, haben aber ähnliche Designziele. Ein direkter Vergleich ist nur schwer möglich.

Inferno

Inferno von Vita Nuova[5] ist ein komplettes Netzwerk-Betriebssystem, das speziell für netzwerkfähige Geräte wie Mobiltelefone, PDAs und Set-Top-Boxen entwickelt wurde. Inferno kann sowohl nativ auf vielen Plattformen laufen, als auch als Applikation unter den gängigsten Betriebssystemen (Windows, Solaris, Linux, AIX, FreeBSD, ...). Es existiert sogar ein Plug-In für den Microsoft Internet Explorer, das es ermöglicht im Web-Browser Inferno-Applikationen auszuführen. Inferno-Applikationen werden in «Limbo» einer plattformunabhängen Programmiersprache geschrieben. Von daher läßt sich Inferno durchaus mit der Virtuellen Maschine von Java vergleichen; das Plug-In für den Internet Explorer (für Netscape ist ebenfalls eines angekündigt) erinnert sofort an das Java-Plug-In, das die meisten Web-Browser zur Verfügung stellen. Inferno Applikationen werden - wie Java-Applets - im Kontext dieses Plug-Ins bzw. des Web-Browsers ausgeführt. Vita Nuova versucht auch, Inferno und die Inferno-Programmiersprache Limbo als Java-Alternative zu positionieren (vgl. die Presseveröffentlichung «Inferno Plug-In Upsets Java's Monopoly» vom 6. Dezember 2000 unter http://www.vitanuova.com/press/pr_5.html). Leider ist Inferno relativ unbekannt, aber das könnte sich schnell ändern: Inferno läuft - im Gegensatz zu Java - auch nativ auf zahlreichen Hardware-Plattformen.

Universal Plug and Play (UPnP)

UPnP von Microsoft ist ebenfalls als Konkurrenz zu Jini zu sehen (obwohl Sun Microsystems UPnP als komplementäre Technologie einstuft). Microsoft hat aus den Fehlern der Vergangenheit gelernt und das UPnP-Form (www.upnp.org) gegründet, um UPnP als offenen Standard zur Verfügung zu stellen. Auf der Mitgliedsliste des Forums stehen zahlreiche (um nicht zu sagen sämtliche) bekannte Hard- und Softwarefirmen; somit dürfte einer Verbreitung von UPnP nichts mehr im Wege stehen. Allerdings muß man dazu sagen, daß viele Firmen (Sun, IBM, Nokia, ...) auch in Foren bzw. Organisationen von Konkurrenzprodukten (Jini, Salutation, ...) vertreten sind. Universal Plug and Play verwendet ein eigenes Protokoll (Simple Service Discovery Protocol - SSDP) um UPnP-Geräte ausfindig zu machen. Ein Client, der ein bestimmtes Gerät (z.B. einen Drucker) nutzen will, macht dieses Gerät über SSDP ausfindig. Anschließend wird eine XML-basierte Beschreibung geladen, in der unter anderem Beschreibung des Service Control Protocol (SCP) enthalten ist. Der Client muß dann das Gerät über dieses Protokoll ansprechen. Mit der General Event Notification Architecture (GENA) sind verteilte Events möglich. Ausführbarer Code wird in UPnP nicht heruntergeladen; stattdessen wird über SOAP (Simple Object Access Protocol) auf serverseitige Objekte zugegriffen. Unter

[5] Vita Nuova wurde 1996 gegründet, um das von Lucent Technologies (Bell Labs) entwickelte Inferno-Betriebssystem zu vermarkten. Die Firma vertreibt auch das Plan9- Betriebssystem, das ebenfalls von den Bell Labs stammt. Einen Überblick über die Firma Vita Nuova gibt deren Website: http://www.vitanuova.com/company/vnuova.html.

http://msdn.microsoft.com/library/psdk/upnp/upnpport_6zz9.htm lassen sich weitergehende Informationen finden.

Salutation

Auch *Salutation* ist eine direkte Konkurrenz zu Jini. Das Salutation-Konsortium besteht aus zahlreichen bekannten Firmen (AOL, Canon, Fuji, IBM, Toshiba, ...), die die Salutation-Technologie weiterentwickeln. Ein Salutation-fähiges Produkt ist NuOffice, ein «[..] networked office system based on Lotus Notes Domino» [IBM1999a]. Mit dieser Software ist es möglich, einfach auf Salutation-fähige Geräte zuzugreifen. Weitere Informationen zu diesem «add-on» zu Lotus Notes Domino Server findet man u.a. in einer Broschüre von IBM (siehe [IBM1999b]). Mehrere Firmen, darunter Canon, haben bereits Produkte mit Salutation-Unterstützung angekündigt. Wird beispielsweise ein Salutation- Drucker oder ein Salutation-Faxgerät ins Netzwerk eingesteckt, ist es möglich, das Gerät aus NuOffice sofort anzusprechen. Es ist also keine umständliche Treiberinstallation mehr nötig. Salutation hat also einen großen Vorsprung vor der Jini-Technologie, wird allerdings weniger aggressiv vermarktet und ist deshalb auch weniger bekannt als Jini.

HAVi

HAVi wurde gemeinsam von Sony, Philips, Grundig, Matsushita, Hitachi, Philips, Sharp und Thomson entwickelt und ist der Versuch, einen gemeinsamen Standard für die Vernetzung von digitalen Audio- und Videogeräten zu entwickeln. Die Basis von HAVi ist der IEEE1394-Standard, der auch unter «Firewire» und unter «i.LINK» bekannt ist (siehe [HAVi2000]). Die HAVi-Technologie wird von der HAVi-Organisation lediglich als Technologie zur Vernetzung von Unterhaltungselektronik positioniert. Die Verbindung zum PC oder anderen Geräten der Heimautomation soll über Brücken zu anderen Netzwerktechnologien erfolgen. Es existieren Pläne, HAVi und Jini zu kombinieren (siehe [Sun1999b], [Sun1999c] und [Sun2000b]). Die Idee ist, HAVi-fähige Geräte als Jini-Dienste über ein Netzwerk zur Verfügung zu stellen. Möglich ist damit die bequeme Steuerung von HAVi-fähiger Unterhaltungselektronik (DVD-Player, Stereoanlage, ...) über einen PC oder einen PDA. Damit könnte schon bald eine universelle Fernbedienung in Form eines PDA zur Verfügung stehen. Denkbar ist auch, daß ein Jini-fähiger DVD-Player automatisch aktuelle Filme von verschiedenen Servern (Jini-Diensten) herunterladen und abspielen kann. Wie weit die Pläne zur Kombination von Jini und HAVi schon fortgeschritten sind, ist allerdings noch ein Geschäftsgeheimnis der Firmen. Und eine Kombination von UPnP und HAVi ist ebenfalls denkbar ...

Fazit

Als Fazit kann man festhalten, daß für die Vernetzung von PCs, PDAs und Unterhaltungselekronik noch kein gemeinsamer Standard existiert und verschiedenste Kombinationen existierender (und zukünftiger) Technologien möglich sind. Es empfiehlt sich jedenfalls nicht, darauf zu warten, daß sich eine bestimmte Technologie durchsetzt. Zum einen wird es vermutlich keinen Standard geben, der alle anderen Technologien verdrängt, und zum anderen ist es für Firmen einfacher, *jetzt* Produkte auf der Basis einer Technologie zu entwickeln und später - falls nötig - Brücken zu anderen Vernetzungsstandards zu entwickeln oder zuzukaufen.

Kapitel 3

Analyse

3.1 Anforderungen an die Benutzerschnittstellen-Sprache

Im Rahmen dieser Masterarbeit soll eine XML-Sprache entwickelt werden, mit der sich auf einfache Weise komplexe Benutzerschnittstellen realisieren lassen. In diesem Abschnitt werden nun die Anforderungen an eine Schnittstellen-Sprache formuliert und näher untersucht.

Das wichtigste Designziel ist, die *Vereinfachung* der Entwicklung von Benutzerschnittstellen. Damit ist nicht nur gemeint, dem Programmierer die Arbeit zu erleichtern sondern auch, Anwendern und UI-Designern die Entwicklung von Benutzerschnittstellen zu ermöglichen. Eng mit dieser Anforderung verbunden ist die *Trennung von Benutzerschnittstelle und Applikationslogik*. Zur Zeit existiert lediglich ein Tool, das grafische Java-Benutzeroberflächen im XML-Format abspeichert: der GUI-Builder der Java-IDE «Forté for Java». Diese XML-Beschreibung soll sich in die zu entwickelnde XML-Sprache transformieren lassen.

Mit der XML-Sprache soll es auch möglich sein, *komplexe Benutzerschnittstellen* zu entwickeln. Im Zusammenhang mit der Programmiersprache Java heißt das, daß auch GUIs beschrieben werden sollen, die sich nur über komplexe Verschachtelungen von Containern und Komponenten realisieren lassen.

Wie bereits in der Einleitung (siehe Abschnitt 1.1) kurz beschrieben, sollen Tools für Transformation und Interpretation dieser XML-Sprache entwickelt werden. Das heißt vor allem, daß die Sprache so gestaltet werden muß, daß die *Tool-Entwicklung* nicht übermäßig erschwert wird.

Eine zusätzliche Anforderung ist, daß die XML-Sprache eine rudimentäre *Verhaltensbeschreibung* unterstützt. Damit ist gemeint, daß sich bestimmten Ereignissen (z.B. Anklicken eines Buttons) eine oder mehrere Aktionen zuordnen lassen. Denkbar ist hier z.B. ein entfernter Methodenaufruf (Remote Procedure Call, in Java mit

RMI realisiert) oder der Wechsel einer Dialogmaske (bei Java: automatischer Wechsel von GUI-Panels bei Verwendung des `CardLayout`-Layout-Managers).

Zwei weitere wichtige Anforderungen sind die *Zielsprachenunabhängigkeit* und die *Plattformunabhängigkeit*. Als Zielsprachen kommen alle Sprachen in Frage, mit denen sich Benutzerschnittstellen entwickeln lassen. Bei Sprachen ohne eine eigene UI-Bibliothek müssen dann natürlich Tools für verschiedene UI-Bibliotheken entwickelt werden. Mit der Unabhängigkeit von einer bestimmten Plattform sind hier nicht primär verschiedene *Desktop-Betriebssysteme* (Linux, Windows, MacOS, ..) gemeint sondern verschiedene Plattform*typen*. Es macht kaum einen Unterschied, ob eine Java-Desktop-Applikation unter Unix oder unter Windows läuft. Dagegen muß die Benutzerschnittstelle für eine Applikation auf einem Desktop-Betriebssystem unter Umständen völlig anders aussehen als z.B. auf PalmOS oder einem proprietären Betriebssystem für Mobiltelefone.

Die Anforderungen sind im folgenden nochmals kurz aufgelistet:

- Vereinfachung der Entwicklung von komplexen Benutzerschnittstellen.
- Unterstützung bei der Trennung von Benutzerschnittstelle und Applikationslogik.
- Es soll möglich sein, die XML-Beschreibung eines GUI-Builders (z.B. Forté für Java) weiter zu verwenden.
- Die Tool-Entwicklung soll nicht übermäßig erschwert sein.
- Eine rudimentäre Verhaltensbeschreibung soll unterstützt werden.
- Die XML-Beschreibung soll plattformunabhängig sein.
- Die XML-Beschreibung soll zielsprachenunabhängig sein.

Auf diese Anforderungen wird in den folgenden Abschnitten näher eingegangen.

3.1.1 Trennung von Benutzerschnittstelle und Applikationslogik

Es ist bei der Entwicklung von Applikationen mit einer Benutzerschnittstelle oft ratsam, die Applikations- von der Schnittstellen-Entwicklung zu trennen. Dafür gibt es mehrere Gründe:

- Die UI-Entwicklung findet oft schon in der Analysephase statt, d.h. parallel zur Pflichtenheft-Entwicklung.
- Für die Entwicklung von UIs sind gut ausgebildete Screen-Designer oft besser geeignet als Softwareentwickler.

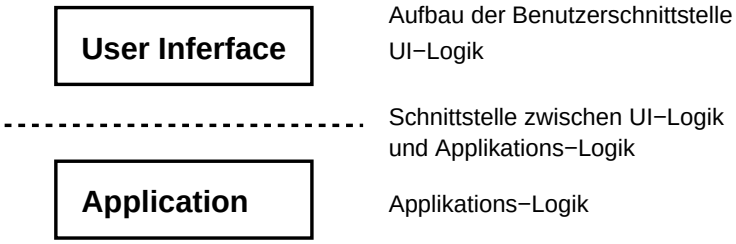


Abbildung 3.1: Trennung von Applikationslogik und Benutzerschnittstelle

- Bei der Anpassung einer Applikation an andere Sprachen und Kulturen müssen evt. große Teile des GUI neu entwickelt werden[1].
- Falls während der Entwicklungsphase ein Re-Design des GUI notwendig ist, sollten keine Änderungen an der Applikationslogik notwendig sein.
- Die Plattformunabhängigkeit einer Applikation gewinnt heute immer mehr an Gewicht.

Normalerweise sind in der Applikation verschiedene Callback-Funktionen definiert, die von der Benutzerschnittstelle bei bestimmten Events aufgerufen werden. Wird z.B. ein Button angeklickt, wird ein Event ausgelöst und der Event-Handler der Benutzerschnittstelle ruft eine entsprechende Methode in der Applikation auf. Abbildung 3.1 veranschaulicht das Prinzip der Trennung von Benutzerschnittstelle und Applikationslogik.

Die Trennung von Benutzerschnittstelle und Applikation findet sich auch in den *Rollen* bei der Entwicklung wieder. Abbildung 3.2 zeigt die Rollen des Applikations-Entwicklers (Application Developer), des Schnittstellen-Designers (Screen Designer oder UI Designer) und des Endbenutzers (End User) und deren jeweilige Sicht auf die Applikation.

Der Screen Designer[2] ist also nicht nur für die Entwicklung des GUI verantwortlich, sondern muß auch die Sicht des Applikations-Entwicklers *und* des Endbenutzers verstehen.

3.1.2 Einfache Entwicklung von komplexen Benutzerschnittstellen

Moderne grafische oder sprachbasierte Benutzerschnittstellen zu entwickeln ist häufig aufwendiger als die Entwicklung der Applikationslogik. Um die Entwicklungszeit

[1] Man vergleiche z.B. ein chinesisches Windows NT mit einer westlichen Version

[2] Im Deutschen hat sich dieser Begriff inzwischen etabliert und wird deshalb in dieser Arbeit dem Begriff «Schnittstellen-Designer» o.ä. vorgezogen.

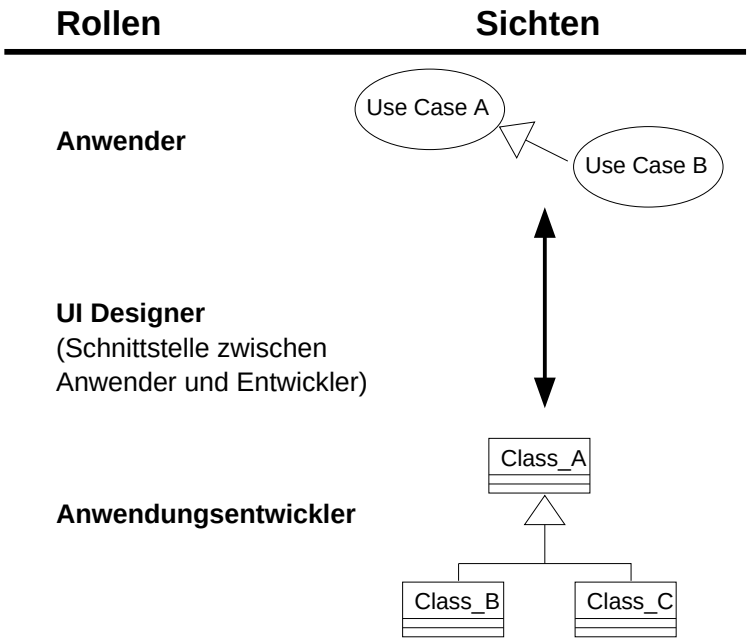


Abbildung 3.2: Verschiedene Sichten auf die Applikation

zu verkürzen werden oft GUI-Builder eingesetzt. GUI-Builder sind in der Regel Bestandteil einer modernen Entwicklungsumgebung (z.B. Forté für Java). Leider fehlt bei vielen Entwicklungsumgebungen ein stimmiges Konzept, um Applikationslogik und Benutzerschnittstelle sinnvoll zu trennen. Es wird zwar funktionsfähiger Quellcode generiert, aber die Integration von Benutzerschnittstelle und Applikationslogik bleibt dem Entwickler überlassen. So wird häufig die Applikationslogik direkt in die Event-Handler-Methoden des generierten GUI-Codes geschrieben. Sinnvoller wäre es, den einzelnen Event-Handlern direkt Callback-Methoden der Applikation zuzuweisen, um die Trennung zwischen Applikation und Benutzerschnittstelle konsistent zu halten.

Allerdings ist es bei den heutigen, sehr kurzen, Entwicklungszyklen kaum möglich, auf die Vorteile einer Entwicklungsumgebung (IDE: Integrated Development Environment) zu verzichten. Einige Entwicklungsumgebungen (Forté) speichern die «zusammengeklickte» Benutzerschnittstelle in Form einer XML-Datei ab. Es bietet sich nun an, die XML-Ausgabe solcher Entwicklungsumgebungen in eine gemeinsame Struktur zu transformieren. Die Idee ist also ein gemeinsames Austauschformat für grafische (Java-)-Benutzerschnittstellen. Abbildung 3.3 zeigt dieses Szenario.

Mit diesem Modell ist es also möglich, daß mehrere Entwickler mit verschiedenen Entwicklungsumgebungen gemeinsam eine Benutzerschnittstelle realisieren und diese dann über das gemeinsame Austauschformat synchronisieren.

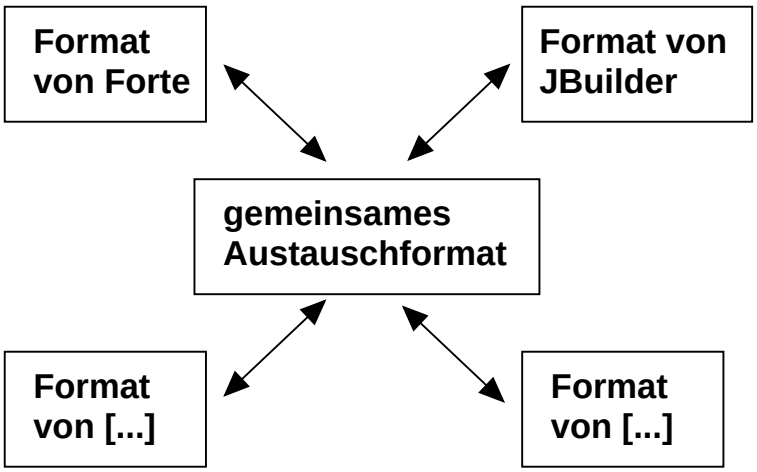


Abbildung 3.3: Gemeinsames Austauschformat für Benutzerschnittstellen

3.1.3 Plattformunabhängigkeit

Plattformunabhängigkeit ist heute ein wichtiges Kriterium bei der Softwareentwicklung. Ein großer Teil des Erfolgs von Java ist auf die Plattformunabhängigkeit dieser Programmiersprache zurückzuführen. In dieser Master Thesis wird der Begriff der Plattformunabhängigkeit allerdings etwas anders definiert. Es wird davon ausgegangen, daß die XML-Beschreibung einer Benutzerschnittstelle in plattformunabhängige Sprachen übersetzt wird bzw. ohnehin von einer (plattformunabhängigen) Java-Applikation interpretiert wird. Mit «Plattformunabhängigkeit» ist hier gemeint, daß entweder aus einer einzigen XML-Beschreibung Quellcode für verschiedene Plattformen (PC, Palm, mobiles Telefon) generiert wird oder der XML-Interpreter abhängig von der Client-Plattform entsprechend angepaßte Benutzerschnittstellen rendert. Abbildung 3.4 zeigt das Prinzip.

Die verschiedenen Zielplattformen bieten (aufgrund von Hardwarebeschränkungen) sehr unterschiedliche Möglichkeiten der Schnittstellenrealisierung. Eine typische Desktop-Applikation mit zahlreichen Menüs, verschiedenen Toolbars und kontextsensitiven Hilfsmenüs läßt sich in dieser Form sicherlich schlecht etwa auf PDAs oder gar auf mobile Telefone portieren. Wenn die Benutzerschnittstelle portierbar sein soll, dann muß sie auf einem abstrakteren Niveau beschrieben werden. Ein gutes Beispiel, wie eine abstrakte Schnittstellenbeschreibung aussehen könnte, liefert die Wireless Markup Language (WML). Hier besteht die Benutzerschnittstelle aus einer Folge von Dialogen, die entweder (auf WAP-Handys) sequentiell darstellt werden oder (auf PCs) gleichzeitig. Das heißt, daß die WAP-Applikation primär für die Darstellung auf einem WAP-Handy entwickelt wird, daß aber Geräte mit einem größeren Bildschirm die Dialoge sehr wohl auf einmal darstellen können (WML wird in Abschnitt 3.2.3 näher beschrieben).

Abbildung 3.4: Plattformunabhängigkeit der XML-Beschreibung

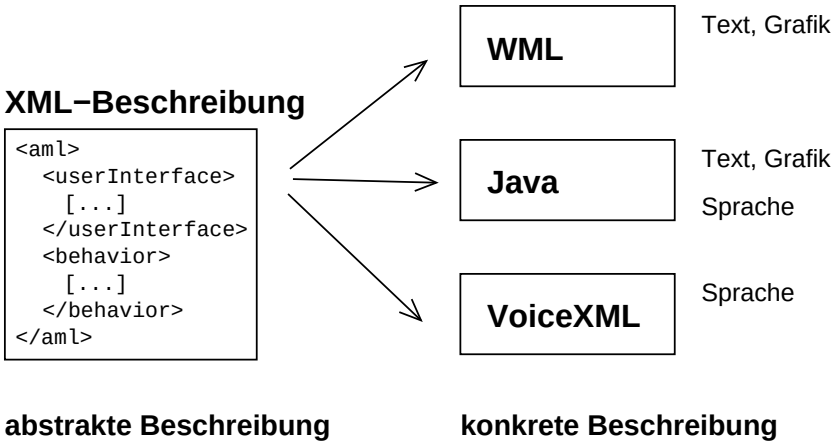


Abbildung 3.5: UI-Beschreibung für verschiedene Zielsprachen

3.1.4 Zielsprachenunabhängigkeit

Die XML-Beschreibung einer Benutzerschnittstelle soll unabhängig von der Zielsprache (Java, WML, VoiceXML) sein. Diese Anforderung ist schwierig zu erfüllen, da die anvisierten Zielsprachen völlig unterschiedliche Möglichkeiten der Repräsentation von Benutzerschnittstellen kennen. VoiceXML beispielsweise hat keine grafische sondern eine sprachbasierte Schnittstelle. Hier muß also eine Sprache entwickelt werden, die eine Schnittmenge von verschiedenen Programmiersprachen implementiert.

Bei der Entwicklung dieser Sprache müssen also programmiersprachenspezifische Details abstrahiert werden. Das heißt auch, daß die XML-Beschreibung einer Schnittstelle keine Toolkit-spezifischen Komponenten enthalten soll. Bei der Zielsprache Java soll also aus einer einzigen Schnittstellenbeschreibung Code für die Toolkits Swing und AWT generiert werden können. Abbildung 3.5 zeigt das angedachte Prinzip:

Leider wird schnell klar, daß dieser Ansatz problematisch ist, da die Zielsprachen sehr unterschiedlich sind:

- VoiceXML und WML sind deklarative Sprachen, Java ist eine prozedurale/objektorientierte Sprache.
- WML verwendet zur Darstellung von Benutzerschnittstellen Text und Grafik, VoiceXML dagegen Sprache und mit Java stehen sowohl Sprache (JavaSpeech) als auch Text und Grafik (AWT, Swing) zur Verfügung.

Aus Implementierungssicht besteht kein großer Unterschied zwischen WML (textbasiert) und VoiceXML (sprachbasiert), da beide Sprachen XML-Anwendungen sind und nach einem ähnlichen Prinzip arbeiten. Die Unterschiede zwischen WML und VoiceXML auf der einen Seite und Java auf der anderen Seite sind dagegen enorm. Das hat zwei verschiedene Gründe:

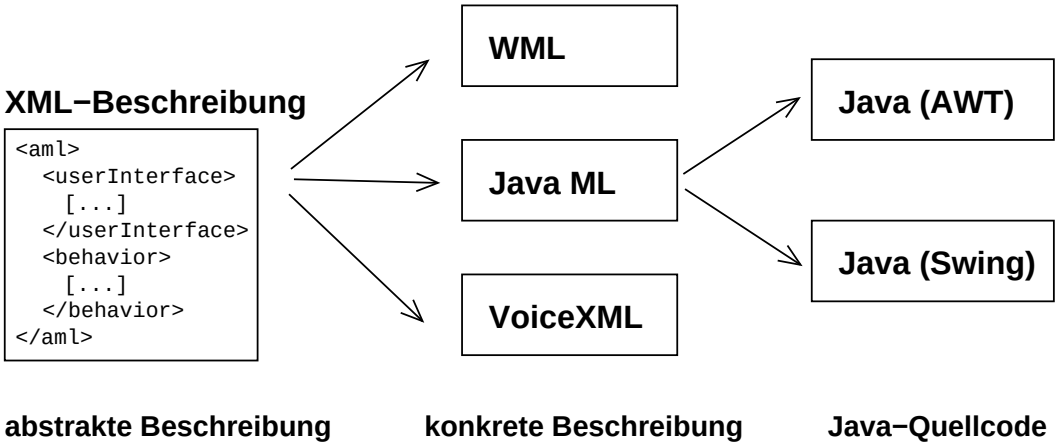


Abbildung 3.6: UI-Beschreibung für interpretierte und compilierte Sprachen

1. Es ist viel aufwendiger, ein GUI für Java zu entwickeln als für WML oder HTML.
   - Die Java-Toolkits AWT und Swing verfügen über zahlreiche GUI-Komponenten.
   - Java verfügt über mehrere Layout-Manager, mit denen die Anordnung der GUI-Komponenten auf der Oberfläche sehr genau kontrolliert werden kann.
2. WML und VoiceXML werden interpretiert, Java wird compiliert.

Damit ist eine Erweiterung des Modells (siehe Abbildung 3.5) notwendig.

Das neue Modell enthält jetzt einen weiteren Zwischenschritt auf dem Weg zum Java-Quellcode, der es vereinfacht, aus XML-Beschreibungen Code zu erstellen. Außerdem hat dieses Zwischenformat (wir nennen es jetzt einfach JML - Java Markup Language) noch einen weiteren Vorteil. Mit diesem Format ist es möglich, Benutzerschnittstelle und Applikationslogik in XML zu codieren. Das heißt, der Entwickler kann unabhängig vom verwendeten Toolkit und von der verwendeten Java-Version programmieren (vorausgesetzt, das XSLT-Stylesheet für die Transformation in Quellcode wird gepflegt und neuen Versionen angepaßt).

Mit diesem Modell ist es möglich, die XML-Sprache für Java-Benutzerschnittstellen enger an das Ausgabeformat der verwendeten Entwicklungsumgebungen anzupassen. Damit hat man ein Standard-Format, das als Basis für die Quellcodegenerierung dient. So kann jeder Entwickler zunächst die Benutzerschnittstelle mit seiner Entwicklungsumgebung (oder mit seinem Lieblings-Editor) erstellen und nach der Konvertierung in JML weiteren Quellcode (Applikationslogik) hinzufügen.

Zu diskutieren ist noch, ob in die Java Markup Language die Möglichkeit, echten Quellcode in die Beschreibung zu integrieren, eingebaut werden soll. Der Vorteil wäre, daß eine komplette Applikation in XML codiert werden könnte. Denkbar ist

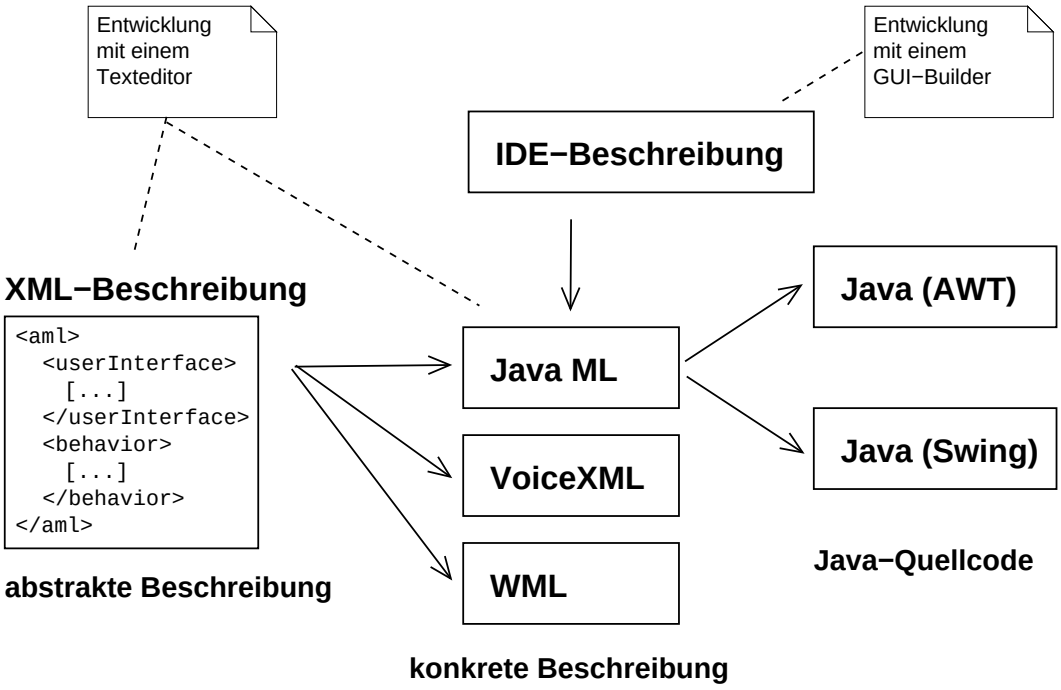


Abbildung 3.7: UI-Beschreibung für verschiedene Zielsprachen (finale Version)

dann, für einzelne Felder des GUI eine Feldvalidierung zu programmieren. Andererseits ist die XML-Beschreibung dann nur noch zur Quellcodegenerierung zu verwenden. Der Programmcode kann nämlich nicht mehr (zumindest nicht mehr ohne sehr großen Aufwand) interpretiert werden. Es ist natürlich auch möglich, hier eine Scriptsprache (ECMAScript bietet sich an) zu verwenden. Allerdings muß dann auf der entsprechenden Zielplattform ein ECMAScript-Interpreter vorhanden sein, weil die notwendigen Bibliotheken sehr umfangreich sind.

Alternativ ist es auch möglich, in der XML-Beschreibung festzulegen, welche Methode bei einem bestimmten Event aufgerufen werden soll. Der Vorteil hierbei wäre, daß die Applikation, die die XML-Beschreibung interpretiert, die entsprechenden Methodenaufrufe z.B. an einen Jini-Dienst weiterleiten könnte. Auf diese Details wird in Kapitel 4 näher eingegangen.

3.1.5 Quellcodegenerierung und Interpretation von Benutzerschnittstellenbeschreibungen

Es gibt zwei grundsätzlich verschiedene Ziele beim Design einer XML-Sprache zur Beschreibung von Benutzerschnittstellen:

1. Entwicklung einer Sprache zur Generierung von Quellcode und
2. Entwicklung einer Sprache zur Interpretation zur Laufzeit.

Betrachtet man diese Ziele näher, stellt man fest, daß sie sich zwar nicht gegenseitig ausschließen, aber dennoch unterschiedliche Anforderungen stellen. Bei interpretierten Sprachen (z.B. XML-Sprachen wie WML oder XHTML) sind die beiden Ziele identisch: der generierte Quellcode wird zur Laufzeit dynamisch gerendert. Bei compilierten Sprachen wie Java[3] oder C++ macht es einen großen Unterschied, ob Quellcode generiert werden soll oder dynamisch ein GUI gerendert werden soll. Ist eine Quellcodegenerierung gewünscht, tritt eine weitere Anforderung auf: Es soll möglich sein, Applikationslogik in die Beschreibung der Benutzerschnittstelle zu integrieren. Allerdings wird dann aus einer XML-Beschreibungssprache für Benutzerschnittstellen eine Beschreibungssprache für Applikationen. Hier ist Vorsicht geboten, damit keine Vermischung von Benutzerschnittstelle und Applikationslogik auftritt. Die beiden folgenden Abschnitte diskutieren Anwendungsszenarien, Vor- und Nachteile der beiden Designziele «Quellcodegenerierung» und «Interpretation».

Designziel: Quellcodegenerierung

Unter Quellcodegenerierung ist das Erzeugen von Quellcode einer bestimmten Programmiersprache zu verstehen. Wichtig ist dabei, daß sich der generierte Quellcode ohne Nachbearbeitung compilieren läßt. In dieser Master Thesis ist als Zielsprache Java vorgesehen.

Der Nutzen dieses Ansatzes erschließt sich bei der Betrachtung eines Beispielszenarios. Das Beispielszenario läuft wie folgt ab:

- Ein Anwender fordert einen Dienst von einem Server an.
- Der Server verschickt eine `.class`-Datei mit der entsprechenden Benutzerschnittstelle.
- Der Anwender (Client) verwendet die ausgelieferte Benutzerschnittstelle, um den angebotenen Dienst zu nutzen.

Man muß hier unterscheiden, ob die `.class`-Datei den gesamten Code der Applikation, also Benutzerschnittstelle *und* Applikationslogik enthält, oder ob die `.class`-Datei nur die Benutzerschnittstelle enthält. Letzterer Fall ist bei einer Client/Server-Umgebung üblich (und sinnvoll), da der Sinn von über das Netz angebotenen Diensten entweder darin liegt, daß auf eine große, serverseitige, Datenbank zugegriffen werden kann oder für komplexe und aufwendige Berechnungen die Ressourcen eines leistungsfähigen Servers genutzt werden können.

[3] Java wird hier nicht als interpretierte Sprache angesehen, obwohl der entsprechende Bytecode sehr wohl interpretiert oder zur Laufzeit in nativen Code übersetzt wird. Letztendlich ist ein compilieren notwendig, und das ist hier der relevante Punkt.

In der prototypischen Implementierung soll Jini als Netzwerk-Middleware zum Einsatz kommen. Die Jini-Dienste laufen hier auf einem Server, der bei Bedarf `.class`-Dateien über das Netz verschickt. Die eigentlich Applikation läuft auf dem Rechner, der den Jini-Dienst anbietet. Es sollen nun zu diesem Dienst Benutzerschnittstellen ausgeliefert werden - *abhängig von der Zielplattform*. Das heißt, daß ein Dienst in der Lage sein muß, eine Benutzerschnittstelle an Clients der gängigsten Plattformen (PCs, PDAs, mobile Telefone) auszuliefern. Die Benutzerschnittstellen für die verschiedenen Clients sollen nach Möglichkeit aus derselben XML-Beschreibung generiert werden. Dazu muß serverseitig ein Batch-Prozeß laufen, der in regelmäßigen Abständen oder nach Anforderung aus der XML-Beschreibung Quellcode generiert und compiliert. Der Batch-Prozeß muß also folgende Aufgaben erledigen:

1. Generierung des Quellcodes aus der XML-Beschreibung (Transformation mit XSLT).
2. Übersetzen des Quellcodes mit `javac`.
3. Auslieferung der Binärdateien (z.B. als `.jar`-Datei).

Dieser Ansatz hat verschiedene Vorteile, aber auch Nachteile. Zu den Vorteilen gehört, daß ein lauffähiger (d.h. getesteter) Dienst kaum gepflegt werden muß. Bei Änderungen an der Benutzerschnittstelle wird der Batch-Prozeß automatisch oder manuell gestartet und das Deployment der geänderten Benutzerschnittstelle erfolgt automatisch. Problematisch ist der Ansatz, wenn Benutzerschnittstellen an Geräte ausgeliefert werden sollen, die keine Klassen nachladen können. Dann müßten die Benutzerschnittstellen für diese Geräte separat gepflegt und in einer alternativen Deployment-Schnittstelle zur Verfügung gestellt werden.

Designziel: Interpretation

Es gibt einen zweiten Ansatz für die Verwendung von Benutzerschnittstellenbeschreibungen in XML: die direkte Interpretation der XML-Beschreibung. Bei einer Interpretation wird die Benutzerschnittstelle zur Laufzeit gerendert. Das heißt, es ist keine Codegenerierung und keine Compilierung mehr nötig. Da die XML-Beschreibung direkt interpretiert wird, ist kein Batch-Prozeß zur Weiter- bzw. Nachbearbeitung der Beschreibung notwendig. Ein weiterer Vorteil ist, daß XML-Dateien per Definition Textdateien sind, die einfach über beliebige Netzwerkprotokolle verschickt werden können. Damit können also Benutzerschnittstellen auch auf Geräten laufen, die nicht über die Möglichkeit verfügen, Klassen dynamisch aus dem Netz nachzuladen (Die KVM von Sun bietet z.B. nicht die Möglichkeit des dynamischen Nachladens von Klassen).

Allerdings gibt es auch Nachteile. Zum einen ist für die Interpretation der XML-Beschreibungen eine Applikation mit eingebautem XML-Parser notwendig, die ständig als Hintergrundprozeß laufen müßte. Zum anderen kann es Performanceprobleme geben. PDAs und mobile Telefone haben nur eingeschränkte Ressourcen

(Speicher und Prozessorleistung), die dazu führen können, daß die Interpretation der XML-Beschreibung sehr langsam ist. Der Performanceaspekt muß also anhand einer Implementierung überprüft werden.

Ein weiterer Nachteil ist, daß bei einer Interpretation der XML-Beschreibung keine Applikationslogik möglich ist. Es ist zwar möglich, einen Skript-Interpreter zu schreiben, aber erstens ist der Implementierungsaufwand sehr hoch und zweitens erhöht sich der Ressourcenverbrauch auf dem Clientgeräte enorm.

In Abschnitt 3.1.4 wird diskutiert, ob überhaupt die Möglichkeit bestehen soll, Quellcode in die XML-Beschreibung einzubauen. Wenn man den Aufwand für die Implementierung und den Ressourcenverbrauch auf den Clientgeräten berücksichtigt, sollte auf die Möglichkeit der direkten Einbindung von Quellcode in der XML-Beschreibung verzichtet werden. Das schließt allerdings nicht aus, daß in einer abstrakteren Benutzerschnittstellenbeschreibung (also nicht Java-spezifisch) Methoden bzw. Funktionen definiert werden, die der Serverteil der Applikation (der Jini-Dienst) implementieren muß. Tabelle 3.1 faßt die Ergebnisse nochmals zusammen.

Zielsprache	Codegenerierung	Interpretation
Java	wünschenswert	Nur bei reiner Schnittstellenbeschreibung.
WML	wünschenswert	Applikationslogik wird interpretiert (WMLScript).
VoiceXML	wünschenswert	Applikationslogik wird interpretiert (VoiceXML-Tags).

Tabelle 3.1: Codegenerierung vs. Interpretation

Der Prototyp, der im Rahmen dieser Arbeit entwickelt wird, soll mit Java und Jini implementiert werden. Deshalb wird hier auf die Möglichkeit, Applikationslogik in die XML-Beschreibung einzubauen, verzichtet.

3.2 XML-Sprachen für Benutzerschnittstellenbeschreibungen

Das Thema dieser Master Thesis ist (unter anderem) die abstrakte Beschreibung von Benutzerschnittstellen mit XML. Es gibt bereits verschiedene Sprachen für die Beschreibung von (meist grafischen) Benutzerschnittstellen. Einige bekannte Sprachen sind

- das GTK-Interface Glade/Libglade,
- die XML-based User Interface Language (XUL),
- die Wireless Markup Language (WML),
- die Voice Markup Language (VoiceXML),

- und die User Interface Markup Language (UIML).

Diese XML-Sprachen unterscheiden sich in zahlreichen Punkten und implementieren verschiedene Designziele. Einige Sprachen orientieren sich sehr stark an bestimmten Toolkits (das Tool Glade z.B. generiert ausschließlich C-Quellcode für das GTK-Toolkit). Andere Sprachen wiederum beschreiben Benutzerschnittstellen für bestimmte Zielgeräte und -plattformen (WML, VoiceXML). Die User Interface Markup Language (UIML) ist dagegen als Allzwecksprache für Benutzerschnittstellen ausgelegt. Beim Design von UIML wurde viel Wert darauf gelegt, daß Benutzerschnittstellen sowohl unabhängig von der Zielplattform als auch von der Zielsprache sind. Alle diese Sprachen haben ihre eigenen Vor- und Nachteile und werden im folgenden näher untersucht. Dabei wurden bewußt Anwendungen außen vor gelassen, deren Benutzeroberfläche nach einer textuellen Beschreibung aufgebaut wird, die aber kein XML verwenden. Ein Beispiel dafür ist der «VI iMproved»(VIM) von Bryan Moolenaar (http://www.vim.org.

3.2.1 Glade und Libglade

Was sind Glade und Libglade?

Das GTK-Interface ist ein sehr spezieller Ansatz zur Beschreibung einer Benutzeroberfläche. Zielplattform ist ausschließlich das GTK-Toolkit, das für die Programmierung von Anwendungen für das Desktop-Environment Gnome unter Unix verwendet wird. Zur Entwicklung von Gnome-Anwendungen wird häufig der GUI-Builder «Glade» eingesetzt. Mit Glade kann man einfach seine Benutzeroberfläche mit einem Satz von Standard-Widgets zusammenklicken. Diese Benutzeroberfläche wird in XML gespeichert. Aus der XML-Beschreibung generiert Glade dann den C-Sourcecode. Abbildung 3.8 zeigt einen Screenshot von Glade bei der Arbeit.

Eine interessante Erweiterung zu Glade ist die Bibliothek «libglade». Diese Bibliothek kann aus den von Glade generierten XML- Dateien zur Laufzeit eine Benutzeroberfläche erstellen. Das heißt, es ist möglich, eine Applikation zu schreiben, deren Benutzeroberfläche sich problemlos ohne Neucompilieren ändern läßt. Damit kann man dem Anwender ermöglichen, sich eine eigene Benutzeroberfläche zusammenzustellen.

Glade's XML-Sprache

Die von Glade verwendete XML-Sprache (im folgenden Glade-XML genannt) ist sehr stark an das verwendete Toolkit (GTK) angelehnt. Folgender Quelltext zeigt ein typisches Beispiel für eine Benutzerschnittstellenbeschreibung in Glade-XML (aus [Fischer2000]):

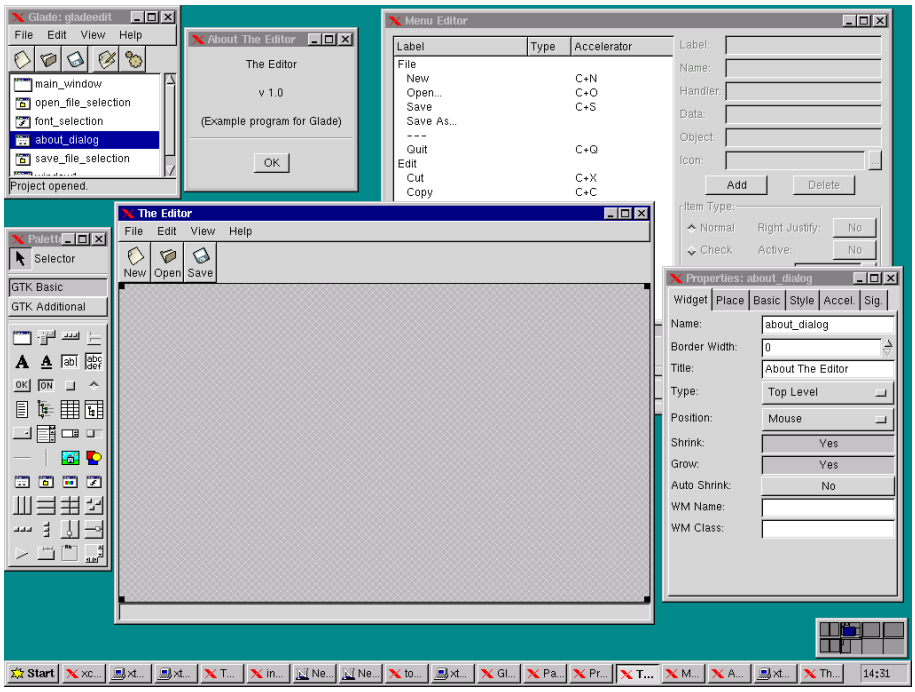


Abbildung 3.8: Screenshot von Glade (aus [Glade2000])

```
<?xml version="1.0"?>
<GTK-Interface>

<widget>
  <class>GtkWindow</class>
  <name>main_window</name>
  <signal>
    <name>destroy_event</name>
    <handler>on_destroy_event</handler>
    <data>NULL</data>
    <last_modification_time>
        Sun, 09 Jul 2000 10:20:16 GMT
    </last_modification_time>
  </signal>
  <title>Erster Versuch</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <modal>False</modal>
  <allow_shrink>False</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>

  <widget>
    <class>Placeholder</class>
```

```
  </widget>
</widget>

</GTK-Interface>
```

Interessant sind die Kindelemente von `<signal>`. Das Element `<name>` gibt natürlich den Namen des Signals an. Ein Signal wird vom X-Server an die Applikation geschickt, wenn der Anwender ein bestimmtes Ereignis ausgelöst hat (z.B. eine Mausbewegung oder einen Klick auf einen Button). Erhält die Applikation ein solches Signal, muß die mit dem Signal assoziierte Callback-Methode aufgerufen werden. Der Name der Callback-Methode ist im `<handler>`-Element angegeben.

Fazit

Glade-XML ist keine akademische Fingerübung, sondern vielmehr aus dem Bedürfnis heraus entstanden, ein Format für die Beschreibung grafischer Benutzeroberflächen zu definieren. Die Sprache ist nicht besonders elegant, aber leicht weiterzuverarbeiten und leicht zu verstehen. Interessant ist dieser Ansatz in jedem Fall, gerade weil er sich an den Bedürfnissen der Praxis orientiert und damit großen Erfolg hat.

3.2.2 XUL - XML-based User Interface Language

XUL und das XPToolkit

Die XML-based User Interface Language (XUL) wurde für den Open-Source Web-Browser Mozilla (Netscape 6) im Rahmen des XPToolkit-Projekts entwickelt. Das XP in XPToolkit steht für «Cross Platform» («[..] because X and C look similar if you beat them long and hard with a hammer», wie das Mozilla-Team in [Mozilla2000] schreibt). Synonym wird für dieses Toolkit auch das Akronym XPFE (für «Cross Platform Front End») verwendet. Die Struktur einer Applikation (also hier der Mozilla-Browser) wird mit XUL beschrieben, für das Layout wird CSS (1 und 2) verwendet und für die Logik JavaScript. Wenn Mozilla startet, wird die XUL-Datei mit der Oberflächenbeschreibung gelesen und interpretiert, d.h. es werden die GUI-Komponenten (Widgets) angezeigt und die entsprechenden Event-Handler initialisiert.

Die Idee hinter XUL ist, die Benutzeroberfläche unabhängig vom Toolkit der Zielplattform zu machen. Entwickelt man z.B. mit den Microsoft Foundation Classes (MFC) unter Windows, ist die Portierung auf andere Plattformen nur mit einer Bibliothekssemulation möglich. In den meisten Fällen kommen die Entwickler nicht um eine komplette Neuprogrammierung der Benutzerschnittstelle herum.

Interessant ist, daß man auch eigene Applikationen mit der XUL-Engine realisieren kann. Das XUL-Tutorial auf http://zvon.org/mozilla/XUL-reference.html ([Zvon2k])

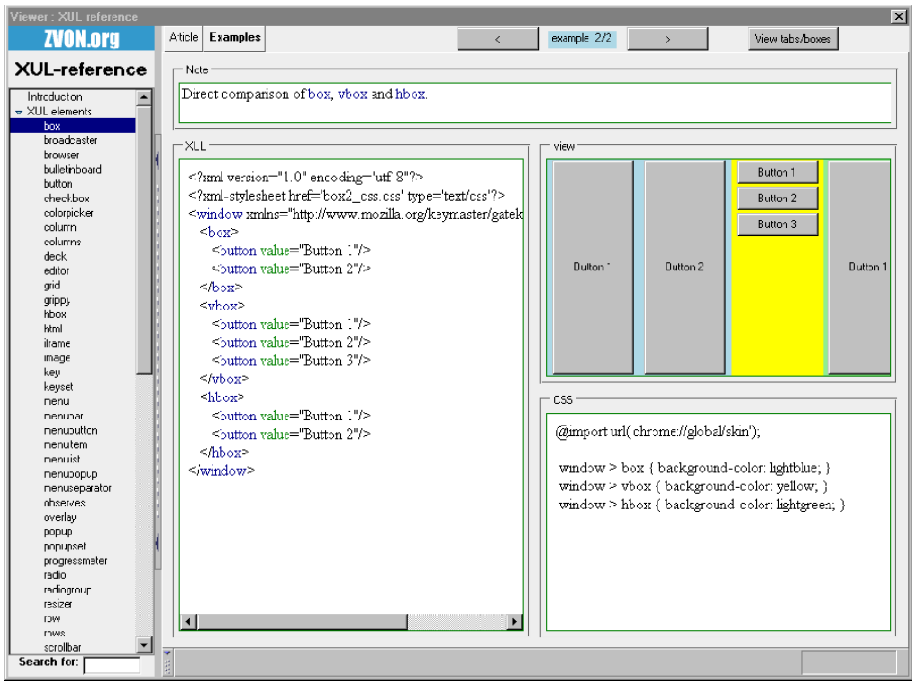


Abbildung 3.9: XUL-Tutorial auf Zvon.org

bietet nicht nur eine sehr gute Einführung sondern zeigt auch die Möglichkeiten von XUL. Beispielsweise kann man mit dem Button «View tabs/boxes» bequem zwischen verschiedenen Sichten (siehe Abbildung 3.9) umschalten.

Der Mozilla-Browser selbst ist natürlich das beste Beispiel für die Mächtigkeit von XUL. Die Plattformunabhängigkeit des GUI ermöglicht eine schnelle Portierung auf andere Plattformen - ein Vorteil für die Entwickler. Die Dynamik des GUI ist besonders ein Feature für die Benutzer des Browsers, da die Benutzeroberfläche recht frei konfigurierbar ist. Es gibt verschiedene Themen (Skins), zwischen denen zur Laufzeit umgeschaltet werden kann. Die Abbildungen 3.10 und 3.11 zeigen das Proxy-Panel in zwei verschiedene Themen. Es ist auch relativ leicht, eigene Themen zu erstellen.

Das Prinzip von XUL ist denkbar einfach. Es existiert ein `<window>`-Element, in dem alle anderen Elemente enthalten sind. Diese Elemente (z.B. `<button>`) werden mit `<box>` im `<window>` positioniert. Folgender - vereinfachter[4] - Code dient zur Anzeige des «Proxy»-Panels (eine Teil des «Preferences»-Panels, siehe Abbildung 3.10 und 3.11).

```
<?xml version="1.0"?>
<window>
  <box class="box-smallheader" title="&lHeader;"/>
  <titledbox orient="vertical">
```

[4]Die Originaldatei heißt `pref-proxies.xul` und ist nach der Installation von Mozilla im entsprechenden Themenverzeichnis zu finden. Beim hier abgedruckten Quelltext fehlen nicht nur komplexere Tags und Skripte sondern auch die Attribute einiger Tags.

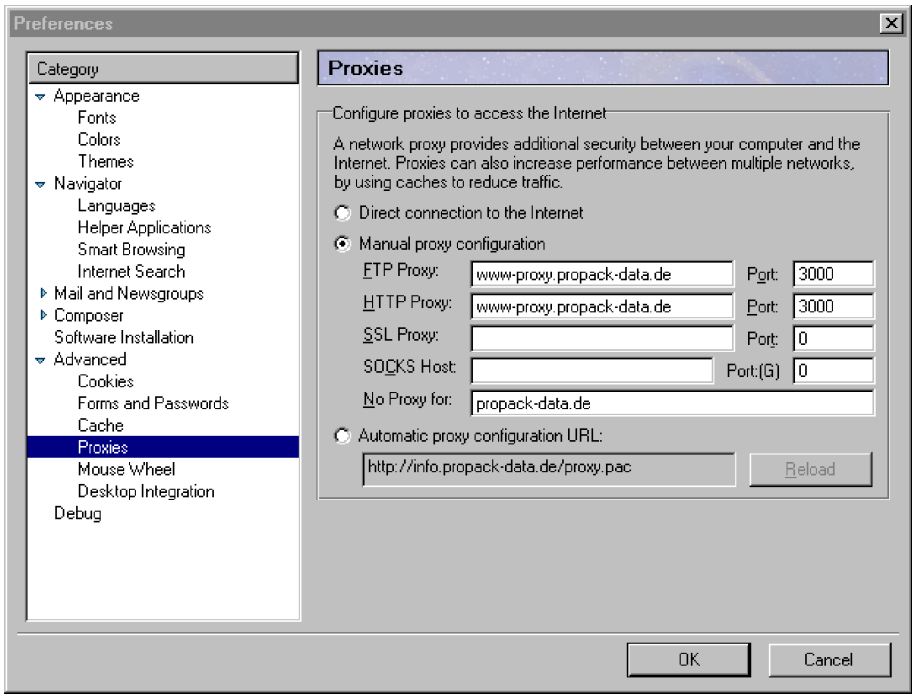


Abbildung 3.10: Das Proxy-Panel im «Classic»-Stil

```
    <title><text value="&proxyTitle.label;"/></title>
6   <html>&networkHeader.label;</html>
    <radiogroup>
8     <radio group="networkProxyType" value="&directTypeRadio.label;"
        oncommand="DoEnabling();"/>
10    <radio group="networkProxyType" value="&manualTypeRadio.label;"
        oncommand="DoEnabling();"/>
12    <grid class="indent" flex="1">
      <columns>
14      <column/>
        <column flex="1"/>
16    </columns>
      <rows>
18      <row>
          <box autostretch="never">
20          <text class="label" value="&ftp.label;" for="networkProxyFTP"/>
            <textfield id="networkProxyFTP" prefstring="network.proxy.ftp"/>
22          <text class="label" value="&port.label;"
                for="networkProxyFTP_Port"/>
24          <textfield id="networkProxyFTP_Port"
                prefstring="network.proxy.ftp_port" size="5"/>
26        </box>
        </row>
28      <row>
          <box autostretch="never">
```

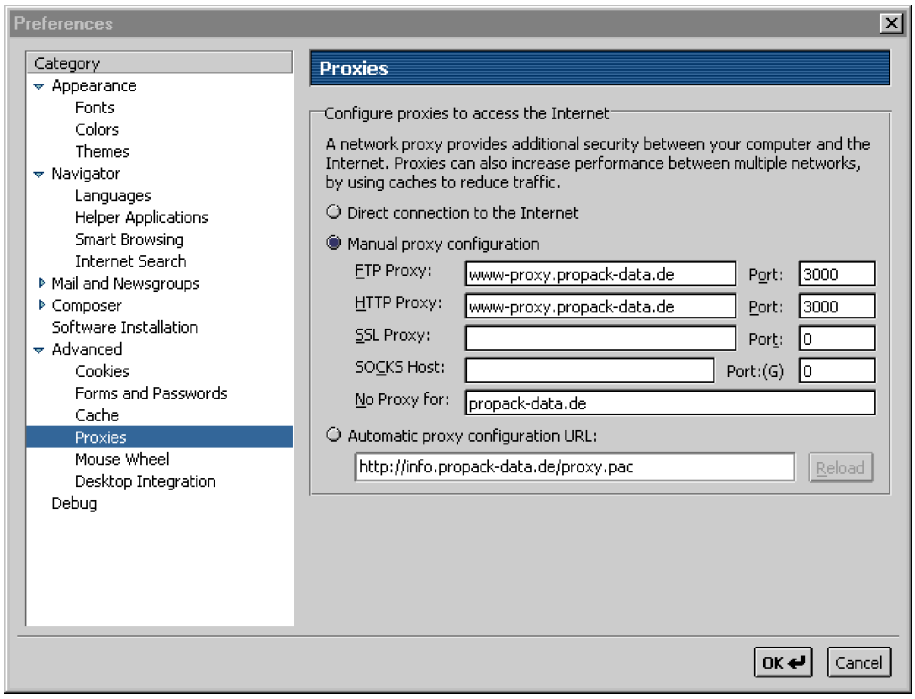


Abbildung 3.11: Das Proxy-Panel im «Modern»-Stil

```
30          <text class="label" value="&http.label;" for="networkProxyHTTP"/>
            <textfield id="networkProxyHTTP"
32              prefstring="network.proxy.http"/>
            <text class="label" value="&port.label;"
34              for="networkProxyHTTP_Port"/>
            <textfield id="networkProxyHTTP_Port"
36              prefstring="network.proxy.http_port" size="5"/>
          </box>
38      </row>
        [...]
40    </rows>
      </grid>
42    <radio group="networkProxyType" value="&autoTypeRadio.label;"
        oncommand="DoEnabling();"/>
44    <box class="indent" flex="1">
        <textfield id="networkProxyAutoconfigURL"
46        prefstring="network.proxy.autoconfig_url"/>
        <button id="autoReload" class="dialog" value="&reload.label;" />
48    </box>
    </radiogroup>
50  </titledbox>
</window>
```

Auch eine Verhaltensbeschreibung (also die Integration von Applikationslogik) ist möglich. Man legt bei jeder Komponente (z.B. Button) fest, was passieren soll,

wenn sie aktiviert wird. Es läßt sich z.B. festlegen, daß eine bestimmte JavaScript-Funktion aufgerufen wird. Diese Funktion kann aus einer externen Datei aufgerufen werden, sie kann aber auch direkt in `<html:script>`-Sektionen stehen. Der Inhalt dieser Skript-Sektionen ist stets CDATA, damit der XML-Parser sich nicht über Operatoren wie «<» beschwert:

```
<html:script>
    <![CDATA[
        function lesser(a,b)
        {
            return a < b ? a : b;
        }
    ]]>
</html:script>
```

Das Problem bei diesem Ansatz ist allerdings, daß XUL-Dateien nicht *sprachunabhängig* sind. Will man andere Sprachen verwenden (z.B. Visual Basic) muß der Code neu geschrieben werden (wobei es natürlich nicht Sinn der Sache ist, statt einem Standard (ECMAScript) proprietäre Sprachen wie Visual Basic zu verwenden).

Fazit

Interessant für diese Master Thesis ist, wie das Verhalten der Applikation beschrieben wird. Dabei wird einer bestimmten Komponente (z.B. einem Button) eine Funktion einer ECMAScript-Bibliothek zugewiesen, die vom Interpreter bei der Aktivierung der Komponente aufgerufen wird.

Das XPToolkit-Projekt wurde natürlich primär für den Mozilla-Browser gestartet. Das heißt, es war nicht geplant, ein generisches Framework für Cross-Platform-Benutzerschnittstellen zu entwerfen, sondern lediglich eine Untermenge davon, die in Netzwerkapplikationen (z.B. Browsern) nützlich ist [Mozilla2000].

3.2.3 WML - Wireless Markup Language

WAP und WML

Das Akronym WAP steht für «Wireless Application Protocol» und ist ein Industriestandard für die Entwicklung von Applikationen für drahtlose Geräte wie z.B. mobile Telefone. Für die Entwicklung der WAP-Spezifikation ist das WAP-Forum (http://www.wapforum.org) zuständig. Gegründet wurde das Forum von den Firmen Nokia, Ericsson, Motorola und Unwired Planet (heute: Phone.com). Die Version 1.0 der WAP-Spezifikation erschien 1997. Heute sind mehr als 500 Firmen im WAP-Forum vertreten (siehe http://www.wapforum.org/who/members.htm). Das

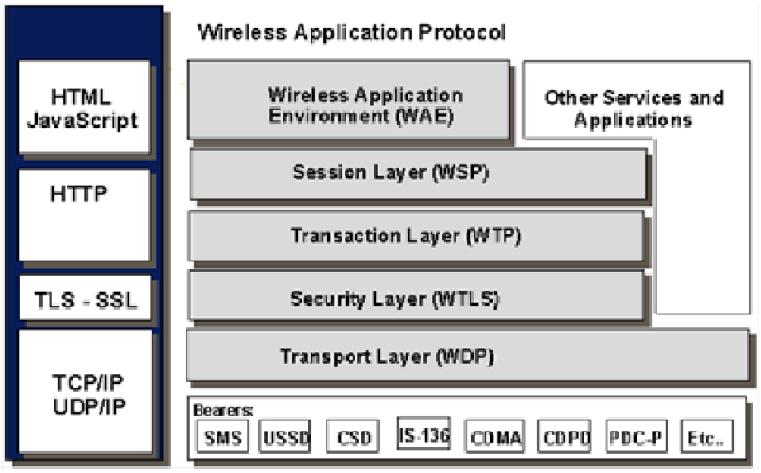


Abbildung 3.12: WAP-Schichtenmodell (aus [Ortner2000]

WAP-Forum spricht davon, daß seine Mitglieder 95 % des globalen «Handset»-Marktes repräsentieren.

Die WAP-Spezifikation ist sehr umfangreich. Sie legt unter anderem Sicherheitsfeatures, Netzwerkprotokolle und eine Sprache für WAP-Anwendungen fest. Diese Sprache heißt «Wireless Markup Language»und ist bekannt unter dem Akronym WML. WML ist ein Teil des «Wireless Application Environment»(WAE). Diese Umgebung besteht aus verschiedenen Spezifikationen, die eher den Anwendungsentwickler interessieren. Abbildung 3.12 zeigt das WAP-Schichtenmodell. Wie man sieht, liegt das WAE - wie HTML und JavaScript - auf der Applikationsschicht.

WML wurde speziell für Anwendungen auf (drahtlosen) mobilen Geräten (Mobiltelefone, Pager, ...) entwickelt. Dabei wurden verschiedene Einschränkungen dieser Geräte bzw. prinzipielle Einschränkungen der drahtlosen Kommunikation berücksichtigt (aus [WAPForum2000a]):

- Mobile Geräte haben nur ein kleines Display und bieten nur eingeschränkte Eingabemöglichkeiten.
- Die Bandbreite von drahtlosen Netzen ist gering.
- Die Rechenleistung und der Speicher von mobilen Geräten ist sehr gering.

Laut Spezifikation ([WAPForum2000a]) besteht WML aus vier funktionalen Bereichen:

1. Repräsentation von Text und Layout.
2. WML-Dokumente werden in Decks und Cards aufgeteilt. Eine Card spezifiziert eine oder mehrere Benutzerinteraktionen, Cards werden zu Decks zusammengefaßt.

3. Möglichkeit der Navigation bzw. des Linkings zwischen Cards.

4. String-Parametrisierung und Zustandsmanagement: WMLDecks können mit Hilfe eines Zustandsmodells parametrisiert werden. Es ist möglich, statt Strings mit Variablen zu arbeiten. Die Variablen werden dann zur Laufzeit durch entsprechende Strings ersetzt, wodurch wiederum Bandbreite eingespart wird.

Die Repräsentation von Text und Layout ähnelt HTML. Es stehen z.B. Elemente für die Hervorhebung von Text (<b>, <e>) zur Verfügung. Neu ist die Struktur von WML-Dokumenten, die es ermöglicht komplette Applikationen zu schreiben. Da WML ein Zustandsmodell unterstützt steht der Entwicklung von WAP-Applikationen nichts mehr im Wege.

WAP-Applikationen

WAP-Applikationen werden in WML geschrieben. WML-Dokumente (auch *Deck* genannt) enthalten eine oder mehrere *Cards*. Eine Card enthält die Beschreibung einer einfachen Benutzerschnittstelle und beschreibt wie die Interaktion mit dem Benutzer abläuft. Man kann sich eine WML-Applikation also als eine Abfolge von Benutzerinteraktionen vorstellen. Etwas formaler: Die WML-Metapher repräsentiert einen endlichen Automaten. Jede Card entspricht einem Zustand, ein Deck ist die Menge aller Zustände. Folgender WML-Code aus [Wireless2000b] zeigt eine einfache Applikation:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
  <card id="Login" title="Login">
    <do type="accept" label="Password">
      <go href="#Password"/>
    </do>
    <p>
    UserName:
    <select name="name" title="Name:">
      <option value="John Doe">John Doe</option>
      <option value="Paul Smith">Paul Smith</option>
      <option value="Joe Dean">Joe Dean</option>
      <option value="Bill Todd">Bill Todd</option>
    </select>
    </p>
  </card>

```

Abbildung 3.13: WML-Beispiel: Login-Card ([Wireless2000b]

```
  <card id="Password" title="Password:">
22  <do type="accept" label="Results">
      <go href="#Results"/>
24  </do>
    <p>
26  Password: <input type="text" name="password"/>
    </p>
28  </card>

30  <card id="Results" title="Results:">
    <p>
32  You entered:<br/>
    Name: $(name)<br/>
34  Password: $(password)<br/>
    </p>
36  </card>
</wml>
```

Diese Applikation besteht aus drei Cards: Login, Password und Results. In der Login-Card muß ein Benutzername ausgewählt werden (in der Praxis keine sehr sinnvolle Anwendung). Mit dem `do`-Element wird einer Card ein Event zugewiesen. Dabei wird kein spezielles Widget für den Event festgelegt. Es bleibt vielmehr dem WML-Browser überlassen, wie das Widget dargestellt wird. Möglich ist z.B. die Darstellung als Menüpunkt oder als grafischer Button. Auch das Mapping auf bestimmte Funktionstasten oder Sprach-Kommandos ist möglich. Die einzige Voraussetzung ist, daß eine Aktivierungsoperation mit dem entsprechenden Widget verbunden ist [WAPForum2000a].

Kindelemente von `<do>` legen die Aktionen, die ausgeführt werden sollen, fest. In obigem Beispiel wird mit dem `<go>`-Element auf eine weitere Card innerhalb des aktuellen Decks verwiesen. In der nächsten Card kann ein Paßwort eingegeben werden; wird auf der entsprechende Event ausgelöst, springt der WML-Browser zur

letzten Card dieser Applikation, wo der ausgewählte Benutzername und das eingegebene Paßwort angezeigt werden. Abbildung 3.13 zeigt die einzelnen Dialogschritte, wie sie auf einem WAP-Handy ausgeführt werden.

WAP-Applikationen werden geräteabhängig gerendert. Es ist z.B. möglich, auf einem größeren, farbigen Display wie auf einem PDA mehrere Cards gleichzeitig am Bildschirm darzustellen. In der WAP-Spezifikation heißt es ausdrücklich

> «The card element indicates the general layout and required input fields, but does not overly constrain the user agent implementation in the areas of layout or user input. For example, a card can be presented as a single page on a large-screen device and as a series of smaller pages on a small-screen device.»

Beim Design von WAP-Applikationen sollte trotzdem stets darauf geachtet werden, daß sich die Software auch auf einem vier- oder fünfzeiligen Display gut bedienen läßt.

WMLScript

Die Wireless Markup Language verfügt über keine Elemente, mit denen sich prozedurale Logik ausdrücken läßt. Obwohl sich über das eingebaute Zustandsmodell WAP-Applikationen entwickeln lassen, sind etwa Berechnungen oder Feldvalidierungen in WML nicht möglich. Deshalb wurde die WMLScript-Spezifikation ([WAPForum2000b] als Ergänzung zu WML und Teil der WAP-Spezifikation entwickelt. WMLScript basiert auf ECMAScript, um HTML/ECMAScript-Entwicklern einen schnellen Einstieg in die Sprache zu ermöglichen. Zusätzlich wurde WMLScript um eine Standardbibliothek erweitert. Die Funktionen der Standardbibliothek sind in der «WMLScript Standard Library Specification»(siehe [WAPForum2000c]) näher beschrieben.

Es gibt verschiedene Unterschiede zwischen ECMAScript und WMLScript. Der vielleicht wichtigste Unterschied ist, daß WMLScript compiliert werden muß. Das Ergebnis ist plattformunabhängiger Bytecode, der bei der Client/Server-Kommunikation wenig Bandbreite verbraucht. Der «Core» der ECMAScript-Spezifikation ist größtenteils auch in WMLScript enthalten. In der WMLScript-Spezifikation sind die Unterschiede zwischen den beiden Skriptsprachen detailliert aufgeführt. Wichtige Unterschiede sind:

- Arrays werden nicht unterstützt.
- Die Operatoren `new` und `delete` werden nicht unterstützt.
- Ganzzahlige Division (`div`-Operator) ist - im Gegensatz zu ECMAScript - möglich.

Einige Funktionen von ECMAScript (z.B. `isFloat()`) wurden in die Standardbibliothek ausgelagert. Diese Standardbibliothek ist in [WAPForum2000c] näher beschrieben. Konforme WMLScript-Interpreter müssen diese Bibliothek implementieren («Static Conformance Requirement» Nummer WMLS-002, siehe [WAPForum2000b]). Daraus folgt natürlich, daß diese Bibliothek auf dem Clientgerät vorhanden sein muß.

Fazit

WML wird vor allem als HTML-Ersatz für mobile Geräte angesehen. Durch das eingebaute Zustandsmodell bietet WML die Möglichkeit, kleine Applikationen zu entwickeln. Da WML nur über statische Konstrukte verfügt, ist WMLScript die ideale Ergänzung, um komplexere Anwendungen zu schreiben. Insgesamt ist der Sprachumfang von WML etwas enttäuschend. Schon für einfache Aufgaben - z.B. eine Feldvalidierung - muß WMLScript eingesetzt werden. Interessant ist dagegen das Zustandsmodell. Die Zustandsübergänge (Transitionen) werden mit relativen Links modelliert, die nur unter bestimmten Bedingungen aktiviert werden.

Für diese Master Thesis sind zwei Aspekte interessant:

- Das eingebaute Zustandsmodell und
- das geräteabhängige Rendern der WML-Applikation.

Letzerer Punkt ist deshalb interessant, weil das WML-Modell (Aufteilung der Benutzerschnittstelle in einzelne Dialoge) sehr flexibel ist. WML-Applikationen können also sowohl auf den fünfzeiligen Displays von mobilen Telefonen als auch auf größeren Farbdisplays gut dargestellt werden. Dieser "kleinste gemeinsame Nenner" ist ein gutes Konzept, das bei der Entwicklung einer Sprache für geräteunabhängige Benutzerschnittstellen unbedingt berücksichtigt werden muß.

3.2.4 VoiceXML

Überblick über VoiceXML

VoiceXML ist eine XML-Sprache, die vom VoiceXML-Forum entwickelt und geplegt wird. Das Forum wurde von IBM, Motorola, AT&T und Lucent Technologies mit dem Ziel gegründet , einen Standard für den Zugriff auf Netz-Dienste über das Telefon zu etablieren [Wireless2000a]. Die Version 1.0 der VoiceXML-Spezifikation [VoiceXML2k] ist seit dem 7. März 2000 öffentlich verfügbar.

VoiceXML wurde entwickelt, um auf einfache Weise sprachbasierte Applikationen entwickeln zu können. Sprachbasierte Applikationen machen hauptsächlich auf Geräten Sinn, die nur ein sehr kleines (oder überhaupt kein) Display haben und die oft

nur über eine numerische Tastatur verfügen, wie z.B. Mobiltelefone. Dazu kommt noch begrenzter Speicher und begrenzte Rechenleistung. Mit VoiceXML ist es nun möglich, sprachgesteuerte Applikationen zu entwickeln, die alle VoiceXML-fähigen Geräte ausführen können.

VoiceXML-Dokumente bestehen zunächst aus einer Anzahl von Dialogen (`<form>` oder `<menu>`). Diese Dialog-Elemente beschreiben den Ablauf von Interaktionen mit dem Benutzer und sind demnach als Applikation anzusehen. Das `menu`-Element ist eine spezielle Form des `<form>`- Elementes. Es ist nützlich, um dem Benutzer eine Auswahl von Möglichkeiten zu geben und entsprechend der Auswahl weiter im Programm zu verzweigen (oder ein neues VoiceXML-Dokument zu laden). Ein `<form>`-Element ist etwas komplexer. Es kann beinhalten:

- Feld-Elemente (`<field>`) und Kontroll-Elemente (`<control>`),
- Variablendeklarationen
- Event-Handler
- Blöcke mit prozeduraler Logik

Folgendes Beispiel aus der Spezifikation [VoiceXML2k] zeigt den Aufbau einer einfachen Applikation:

```
<form id="weather_info">
    <block>Welcome to the weather information service.</block>
    <field name="state">
        <prompt>What state?</prompt>
        <grammar src="state.gram" type="application/x-jsgf"/>
        <catch event="help">
            Please speak the state for which you want the weather.
        </catch>
    </field>
    <field name="city">
        <prompt>What city?</prompt>
        <grammar src="city.gram" type="application/x-jsgf"/>
        <catch event="help">
            Please speak the city for which you want the weather.
        </catch>
    </field>
    <block>
        <submit next="/servlet/weather" namelist="city state"/>
    </block>
</form>
```

Ein denkbares Szenario wäre dann (ebenfalls aus [VoiceXML2k]):

C (computer): Welcome to the weather information service. What state?
H (human): Help
C: Please speak the state for which you want the weather.
H: Georgia
C: What city?
H: Tblisi
C: I did not understand what you said. What city?
H: Macon
C: The conditions in Macon Georgia are sunny and clear at 11 AM ...

Die `<field>`-Elemente stehen also für die einzelnen Dialogschritte. Mit dem `<prompt>`-Element wird eine Sprachausgabe generiert. Das `<grammar>` -Element inkludiert eine Grammatik. Diese Grammatik beschreibt lediglich, welche Spracheingaben erwartet werden. In `city.gram` z.B. sind alle Städte aufgelistet, die von der Applikation erkannt werden. Die Applikation wird beendet, wenn ein gültiger Staat und eine gültige Stadt eingegeben wurden. Dann wird der Inhalt des `<block>`-Elements ausgeführt. In diesem Beispiel werden die eingegebenen Daten an einen Server geschickt (der hoffentlich eine Antwort zurückliefert). Interessant sind hier noch die Event-Handler. Mit `<catch event="help">` wird ein Hilfstext ausgegeben, wenn der Benutzer das Wort «help» ausspricht.

Verhaltensbeschreibung mit VoiceXML

Richtig interessant sind bei VoiceXML Blöcke mit prozeduraler Logik. Folgendes Beispiel (aus [VoiceXML2k]) führt eine Validierung von Monatseingaben durch:

```
<filled>
    <!-- validate the mmyy -->
    <var name="mm"/>
    <var name="i" expr="expiry_date.length"/>
    <if cond="i == 3">
        <assign name="mm" expr="expiry_date.substring(0,1)"/>
    <elseif cond="i == 4"/>
        <assign name="mm" expr="expiry_date.substring(0,2)"/>
    </if>
    <if cond="mm == '' || mm < 1 || mm > 12">
        <clear namelist="expiry_date"/>
        <throw event="nomatch"/>
    </if>
</filled>
```

Dieser Ausschnitt definiert zunächst die Variablen `mm` und `i`. Der Variablen `i` wird als Wert die Länge des eingegebenen Datums (Form: 1-2-0-1 für Dezember 2001) zugewiesen. Das Jahr muß zweistellig eingegeben werden, der Monat kann ein-

oder zweistellig eingegeben werden (0-1 oder 1 für Januar). Abhängig vom Wert der Variablen `i` wird nun der Monats-Substring extrahiert und in der Variablen `mm` abgespeichert. Jetzt wird nur noch nachgeprüft, ob sich `mm` im Bereich [1..12] befindet. Ist das nicht der Fall, wird ein passender Event generiert (der dann eine entsprechende Fehlermeldung anzeigt).

Solche Logik-Blöcke werden vom VoiceXML-Interpreter über einen ECMAScript-Interpreter ausgeführt, d.h. es ist kein Compilieren notwendig.

Fazit

Insgesamt macht VoiceXML einen sehr guten Eindruck. Die Sprache ist gut durchdacht und wird sich dank der Unterstützung von Firmen wie IBM und Motorola sicher als Standard durchsetzen. VoiceXML ist für diese Masterarbeit besonders interessant, weil es die einzige XML-Sprache für Benutzerschnittstellen ist, die als Metapher kein GUI verwendet sondern ausschließlich mit Sprachsynthese bzw. Spracherkennung arbeitet[5].

3.2.5 UIML - User Interface Markup Language

Einführung in UIML

Die User Interface Markup Language (UIML) hat ähnliche Designziele wie die Beschreibungssprache, die im Rahmen dieser Thesis entwickelt werden soll. UIML wurde von «Universal Interface Technologies» entwickelt, um einen offenen Standard für die Beschreibung von Benutzerschnittstellen zu etablieren. Die Spezifikation soll nach der Fertigstellung der endgültigen Version dem W3C zur Standardisierung vorgelegt werden. Universal Interface Technologies heißt mittlerweile «Harmonia, Inc.» und hat sich auf die Entwicklung von Tools, die Teile der Spezifikation implementieren, konzentriert.

In [UIML2000a] wird das Ziel bei der Entwicklung von UIML kurz umrissen:

> «An objective of UIML is to permit a UIML document to be mapped to any type of user interface, from graphical to speech, and even to those not yet invented.»

Die konkreten Designziele von UIML sind in [UIML2000b] näher beschrieben. UIML soll

[5] Es ist auch möglich, Eingaben über eine (alpha-)numerische Tastatur einzugeben. Das sollte aber eher die Ausnahme als die Regel sein.

- es erlauben, Benutzerschnittstellen für beliebige Geräte zu entwickeln ohne Sprachen und gerätespezifische APIs zu lernen,
- die Entwicklungszeit für die Entwicklung von Benutzerschnittstellen für Gerätefamilien zu verkürzen,
- eine natürliche Trennung zwischen Benutzerschnittstelle und Applikationslogik zur Verfügung stellen,
- es Menschen ohne Programmierkenntnisse ermöglichen, Benutzerschnittstellen zu implementieren,
- «Rapid Prototyping» ermöglichen,
- Internationalisierung und Lokalisierung vereinfachen,
- effizientes Herunterladen von Benutzerschnittsellen über ein Netz ermöglichen,
- die Durchsetzung von Sicherheit erlauben und
- für zukünftige Benutzerschnittstellentechnologien erweiterbar sein.

Es geht also darum, eine Beschreibungssprache für Benutzerschnittstellen verschiedenster Zielsprachen und Zielplattformen zu entwickeln. Auch die Entwicklung komplexer Benutzerschnittstellen, die z.B. Internationalisierung und Lokalisierung unterstützen soll stark vereinfacht werden.

Es existieren bereits verschiedene Tools, die Teile der Spezifikation implementieren, z.B. ein Renderer für Java und ein Codegenerator für WML. Das Virginia Polytech Institute (http://www.vt.edu ist zur Zeit dabei, eine Entwicklungsumgebung für UIML zu erstellen (http://csgrad.cs.vt.edu/ asathyam/IDE.html). Harmonia (http://www.harmonia.com) entwickelt verschiedene UIML-Produkte (darunter auch eine Entwicklungsumgebung). Bis auf ein Tool zur Generierung von WML-Quellcode und ein Tool zur Interpretation von UIML-Beschreibungen (für das Java-Toolkit Swing) sind allerdings noch keine Produkte realisiert.

Entwicklung mit UIML

UIML ist eine der wenigen XML-Sprachen, die als Allzwecksprache für Benutzerschnittstellen entwickelt wurde und nimmt deshalb eine Sonderstellung unter den bisher betrachteten Sprachen ein. Folgender Quellcode aus der Spezifikation ([UIML2000b]) zeigt eine typische UIML-Applikation:

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0a Draft//EN"
    "http://uiml.org/dtds/UIML2_0a.dtd"
>
<uiml>
    <interface>
        <structure>
            <part name="TopHello">
                <part name="hello" class="helloC"/>
            </part>
        </structure>
        <style>
            <property part-name="TopHello" name="rendering">
                Container
            </property>
            <property part-name="TopHello" name="content">
                Hello
            </property>
            <property part-class="helloC" name="rendering">
                String
            </property>
            <property part-name="hello" name="content">
                Hello World!
            </property>
        </style>
    </interface>
    <peers>
        <presentation name="VoiceXML">
            <component name="Container" maps-to="vxml:form"/>
            <component name="String" maps-to="vxml:block">
                <attribute name="content" maps-to="PCDATA"/>
            </component>
        </presentation>
    </peers>
</uiml>
```

Die Applikation ist prinzipiell in `<interface>` und `<peers>` aufgeteilt. Das `<interface>` besteht aus einer Struktur-Beschreibung und einer Stil-Beschreibung. Die entsprechenden Elemente sind `<structure>` und `<style>`. Die Kindelemente von `<peers>` beschreiben das Mapping auf die entsprechende Zielsprache (hier: VoiceXML). Die VoiceXML-Applikation, die aus obigem UIML-Code generiert wird, sieht so aus:

```
<?xml version="1.0"?>
<vxml>
    <form>
        <block>Hello World!</block>
```

```
    </form>
</vxml>
```

Zunächst fällt auf, daß der UIML-Code erheblich komplizierter ist als der generierte VoiceXML-Code. Grund dafür ist, daß die Benutzerschnittstelle nur sehr abstrakt beschrieben werden kann, um sowohl sprach- als auch text- oder GUI-basierte Applikationen daraus zu generieren.

Allgemein betrachtet besteht eine UIML-Benutzerschnittstelle aus folgenden Teilbeschreibungen:

- Strukturbeschreibung (`<structure>`)
- Stilbeschreibung (`<style>`)
- Inhaltsbeschreibung (`<content>`)
- Verhaltensbeschreibung (`<behavior>`)
- Beschreibung des zielsprachenspezifischen Mappings (`<peer>`)

Eine Verhaltensbeschreibung ist in obigem Beispiel nicht vorhanden. Das Konzept ist allerdings leicht verständlich. Es basiert auf verschiedenen Regeln (`<rule>`) und zugehörigen Bedingungen (`<condition>`) und Aktionen (`<action>`). Wenn die Bedingung einer Regel wahr ist, dann werden die entsprechenden Aktionen ausgeführt. Folgender Quellcode zeigt die Struktur einer Verhaltensbeschreibung:

```
<behavior>
    <rule>
        <condition>
            <event class="ButtonSelected" part-name="b1">
        </condition>
        <action>
            <call name="storeData">
                <param name="a1">5</param> <!--arg is constant -->
                <param name="a2"> <!--arg is reference to constant-->
                    <reference constant-name="green"/>
                </param>
            </call>
        </action>
    </rule>
</behavior>
```

Als Bedingung sind Ereignisse, die bei der Aktivierung von Komponenten ausgelöst werden, zugelassen. Es ist auch möglich, zusätzlich festzulegen, daß Daten, die mit einem Event in Verbindung stehen, einen bestimmten Wert haben müssen. Mögliche Aktionen sind Wertzuweisungen an bestimmte Komponenten, externe Methodenaufrufe und die Generierung eines Events.

Fazit

UIML ist zwar eine gut durchdachte Sprache, aber gerade bei der Betrachtung von komplexeren Beispielen (siehe Spezifikation) fällt auf, daß die Sprache etwas überambitioniert ist. Es ist zwar möglich, Schnittstellenbeschreibungen für verschiedenste Zielsprachen und -plattformen zu entwickeln, aber es ist fraglich, ob mit UIML wirklich Entwicklungszeit eingespart werden kann. UIML-Benutzerschnittstellen zu entwickeln ist eine schwierige Aufgabe; die Entwicklung wird damit keineswegs vereinfacht. Problematisch ist auch, daß bislang sehr wenige Tools existieren, die UIML-Dateien interpretieren oder daraus Quellcode generieren können. Ist das verwendete Tool nicht in der Lage, eine bestimmte Komponente zu verarbeiten, bleibt nur die zeitaufwendige Anpassung des UIML-Codes für die entsprechende Sprache oder Plattform. Unangenehm fällt auch auf, daß keine Tools existieren, um XML-Beschreibungen von GUI-Buildern wie Forté nach UIML zu konvertieren. Dabei wäre gerade das interessant, weil Entwicklungsumgebungen wie JBuilder oder Forté über aufwendig entwickelte, gut durchdachte GUI-Builder verfügen. Bis GUI-Builder ähnlicher Qualität auch für UIML auf den Markt kommen wird wohl noch einige Zeit vergehen.

Kapitel 4

Design und Implementierung

4.1 Die Abstract Markup Language (AML)

Im Abschnitt 3.1 (Anforderungen an die Benutzerschnittstellensprache) wird als wichtigstes Designziel die *einfache Entwicklung komplexer Schnittstellen* genannt. Beim Design der Sprache wurden deshalb die anderen Anforderungen zunächst ignoriert. Die Überlegung ist also, wie die Sprache gestaltet werden soll, damit auch der Programmierunkundige damit eine Benutzerschnittstelle entwerfen kann. Da heutige Benutzerschnittstellen zumeist grafisch oder textuell sind, soll besonders hier die Entwicklung einfach sein.

4.1.1 Struktureller Aufbau

Der strukturelle Aufbau von AML ist relativ einfach: Es gibt zwei Elemente, die unterhalb des Wurzelelementes stehen: `<userInterface>` und `<behavior>`. Die Kindelemente von `<userInterface>` und `<behavior>` beschreiben die Benutzerschnittstelle (die Anordnung der Komponenten) bzw. das Verhalten der Benutzerschnittstelle.

```
<userInterface>
    <dialog class="frame">
        [...]
    </dialog>
    <dialog class="popup">
        [...]
    </dialog>
</userInterface>

<behavior>
    [...]
</behavior>
```

Die Benutzerschnittstelle (`<userInterface>`) besteht aus einer Folge von Dialogen. Für Dialoge ist das Element `<dialog>` vorgesehen. Ein `<dialog>`-Element hat ein einziges Attribut: `class`. Dieses Attribut legt fest, ob der Dialog ein Popup-Fenster (`class="popup"`) oder ein Frame (`class="frame"`) beschreibt. Die Verhaltens-Elemente werden in Abschnitt 4.1.4 näher beschrieben.

4.1.2 UI-Komponenten

Bei der Analyse verschiedener Sprachen zur UI-Beschreibung (Abschnitt 3.2) fällt auf, daß viele dieser Sprachen zur Beschreibung von Komponenten den Namen der Komponente als Element in der DTD definieren (bekannt durch HTML). Das heißt, daß z.B. ein Text-Eingabefeld im XML-Dokument als `<textfield id="someID">` beschrieben wird. Der Vorteil bei dieser Art der Beschreibung ist, daß das XML-Dokument für Menschen leichter lesbar ist. Problematisch ist dieser Ansatz dann, wenn weitere Komponenten hinzukommen. In diesem Fall ist eine Erweiterung der DTD nötig, was bedeuten kann, daß Dokumente, die nach der neuen Version der DTD erstellt wurden, nicht mehr konform zu der alten Version sind.

Aus diesem Grund wird in der DTD dieser Master Thesis das Element `<component>` verwendet, um Komponenten zu beschreiben. Um die Klasse einer bestimmten Komponente festzulegen, wird mit dem `class`-Attribut gearbeitet. In einer XML-Instanz wird eine Button-Komponente dann so beschrieben: `<component class="button"name="myButton"/>`.

4.1.3 Layoutbeschreibung

Betrachtet man die Entwicklung des Internet in den letzten Jahren, stellt man fest, daß inzwischen zahlreiche private Websites existieren, die von Nutzern ohne oder mit geringen Programmierkenntnissen erstellt wurden. Obwohl viele dieser Websites mit Toolunterstützung (Frontpage, Dreamweaver, ...) erstellt wurden, haben zahlreiche Benutzer die Web-Markup-Sprache HTML gelernt und sind in der Lage, auch von Hand eine Website zu entwickeln. Die Syntax und Semantik von HTML scheint also für viele Benutzer relativ leicht verständlich zu sein. Eines der wichtigsten Layout-Elemente in HTML ist das `<table>`-Element. Mit diesem Element ist es leicht, verschiedene Texte, Grafiken oder GUI-Komponenten wie Buttons oder Texteingabefelder auf einer Seite zu plazieren. Es ist bemerkenswert, wie einfach es ist, mit HTML-Tabellen ein Seitenlayout zu entwickeln, besonders wenn man sich die Layout-Manager der Java-GUI-Toolkits oder anderer GUI-Bibliotheken ansieht. Aus diesem Grund ist es naheliegend, daß für die Anordnung von Komponenten innerhalb eines AML-Dialoges mit HTML-ähnlichen Tabellen gearbeitet wird. Damit könnte eine typische Layout-Beschreibung etwa so aussehen:

```
<table rows="5" cols="3">
```

```
    <row>
        <col></col>
        <col>
            <component class="button" name="forwardButton" value="Forward"/>
        </col>
        <col></col>
    </row>
    <row>
        <col>
            <component class="button" name="leftButton" value="Left"/>
        </col>
        <col>
            <component class="button" name="stopButton" value="Stop"/>
        </col>
        <col>
            <component class="button" name="rightButton" value="Right"/>
        </col>
    </row>
    <row>
        <col></col>
        <col>
            <component class="button" name="backButton" value="Back"/>
        </col>
        <col></col>
    </row>
</table>
```

In diesem Beispiel werden fünf Buttons in einem 3x3-Gitternetz am Bildschirm angeordnet. Oben in der Mitte befindet sich der «Forward»-Button, in der Zeile darunter befinden sich die Buttons «Left», «Stop» und «Right». In der untersten Zeile in der Mitte ist der «Back»-Button plaziert. Aus der abstrakten Beschreibung der Benutzerschnittstelle läßt sich das Layout, d.h. die Plazierung der einzelnen Komponenten gut ablesen. Umgekehrt dürfte es auch kein Problem sein, auf diese Weise ein ansehnliches Layout selbst zu entwickeln.

4.1.4 Verhaltensbeschreibung

Es ist problematisch, auf einfache Weise das komplexe Verhalten von modernen Benutzerschnittstellen zu beschreiben. Dabei gibt es prinzipiell zwei Möglichkeiten, eine Verhaltensbeschreibung zu modellieren:

- Über ein Zustandsmodell oder
- über eine Menge von Regeln.

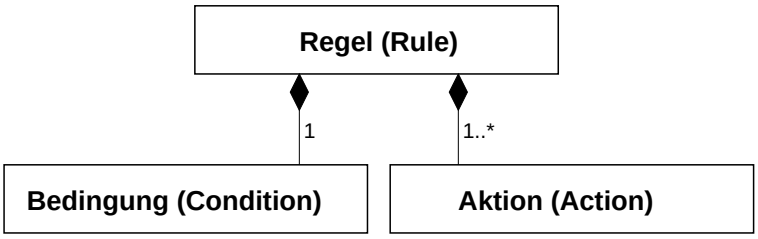


Abbildung 4.1: Verhaltensbeschreibung mit Regeln

Da es schwierig ist, Zustände einer Benutzerschnittstelle zu definieren und die Zustandsübergänge oft nichttrivial sind, wurde in der Abstract Markup Language mit Regeln gearbeitet. Eine Regel besteht aus einer Bedingung und einer Aktion oder mehreren Aktionen (siehe Abbildung 4.1).

Das *Verhalten* einer Benutzerschnittstelle wird also mit einer Menge von *Regeln* beschrieben. Eine Regel besteht aus genau einer Bedingung und einer oder mehreren Aktionen. Wenn eine Verhaltensbeschreibung keine Regeln enthält (was durchaus erlaubt ist), dann ist das Verhalten der Benutzerschnittstelle statisch (was selten sinnvoll ist). Die Struktur einer Verhaltensbeschreibung sieht demnach aus wie folgt:

```
<behavior>
    <rule>
        <condition></condition>
        <action></action>
    </rule>
</behavior>
```

Damit stellt sich eigentlich nur noch die Frage, welche Bedingungen und Aktionen zugelassen werden sollen und wie sich diese ausdrücken lassen. Bei den möglichen Bedingungen gibt es zwei verschiedene Ansätze. Entweder man läßt als Bedingung prinzipiell nur Ereignisse oder Events zu, die von einer Komponente ausgelöst wurden (z.B. ein Button-Klick) oder man erlaubt komplexe Bedingungen (z.B. ein Button wurde angeklickt und ein zuvor eingegebener Benutzername hat einen bestimmten Wert)[1]. Da solche komplexen Bedingungen zur UI-Logik und nicht zur Applikationslogik gehören, wurde diese Möglichkeit in die DTD aufgenommen[2]. Für komplexe Bedingungen wird das Element `<evaluate>` eingeführt, das mindestens zwei Kindelemente besitzt: `<event>` und ein konditionales Element (`<if>` oder `<if-not>`) besitzt. Es ist zu beachten, daß die primäre Bedingung ein *Event* ist. Wird kein Event ausgelöst, werden vom Interpreter keine Bedingungen evaluiert. Das Transformations-Tool für Java transformiert zusätzliche Bedingungen als `if`-Anweisung in die Event-Handler-Methoden der UI-Klasse. Folgender Code-

[1] In UIML (vgl. Abschnitt 3.2.5) wurden komplexe Bedingungen integriert, damit UIML-Beschreibungen als echter Ersatz für sprachspezifische Benutzerschnittstellen dienen können.

[2] Anmerkung: Komplexe Bedingungen wurden aus Zeitgründen weder im Transformations-Tool (siehe Abschnitt 4.3) noch im Interpreter (siehe 4.4) implementiert.

ausschnitt zeigt eine XML-Instanz einer Verhaltensbeschreibung mit einer einfachen und zwei komplexen Bedingungen:

```
<behavior>
    <rule>
        <condition>
            <event class="ActionPerformed" comp-name="forwardButton"/>
        </condition>
        <action>
            <call method="forward">
                <param name=""><value-of name="userName"/></param>
            </call>
        </action>
    </rule>
    <rule>
        <condition>
            <evaluate>
                <event class="ActionPerformed" comp-name="forwardButton"/>
                <if comp-name="userName" value="Juergen"/>
            </evaluate>
        </condition>
        <action>
            <call method="forward">
                <param name=""><value-of comp-name="userName"/></param>
            </call>
        </action>
    </rule>
    <rule>
        <condition>
            <evaluate>
                <event class="ActionPerformed" comp-name="forwardButton"/>
                <if-not comp-name="userName" value=""/>
            </evaluate>
        </condition>
        <action>
            <call method="forward">
                <param name=""><value-of comp-name="userName"/></param>
            </call>
            <change-property
                comp-name="statusLine"
                value="Trying to call method 'forward' "
            />
        </action>
    </rule>
</behavior>
```

Die einfache Bedingung enthält nur ein `<event>`-Element. Mit den beiden Attributen `class` und `comp-name` wird die Art des Events bzw. der Name der Komponente, die

für diesen Event verantwortlich ist, angegeben.

Bei der Bedingung der zweiten Regel ist neben dem `<event>`-Element noch ein `<if>`-Element mit den Attributen `comp-name` und `value` angegeben. Der `comp-name` ist der Name der Komponente, deren Wert ausgelesen und mit dem `value` verglichen wird. Die Bedingung der dritten Regel arbeitet mit `<if-not>`; im Codebeispiel wird damit abgefragt, ob die Komponente `userName` (also ein Texteingabefeld) einen Leerstring (`value=""`) als Wert hat.

Ist die Bedingung einer Regel erfüllt, werden eine oder mehrere Aktionen ausgeführt. Welche Aktionen sollen nun ausgeführt werden können? Es kommen im wesentlichen folgende Aktionen in Frage:

- Methoden-/Funktionsaufrufe
- Eigenschaftsänderungen bei UI-Komponenten
- Wechseln zu einem anderen Dialog

Für Methodenaufrufe wird in AML das `<call>`-Element verwendet. Das einzige Attribut dieses Elementes ist `method` und bezeichnet den Methodennamen, der aufgerufen werden soll. Falls bei dem Methodenaufruf keine Parameter übergeben werden sollen, genügt etwa `<call method="saveChanges"/>` um eine Methode mit dem Namen `saveChanges()` aufzurufen. Bei einer Parameterübergabe sind zwei Fälle zu beachten: die Übergabe von statischen und von dynamischen Parametern. Statische Parameter haben die Form `<call name="test"><param name="version"> <value>1.0</value></param>`, dienen also zur Übergabe von konstanten Werten, die normalerweise für Versionsnummern u.ä. verwendet werden. Dynamische Parameter werden zum Zeitpunkt des Methodenaufrufs evaluiert. Als dynamische Parameter sind nur Werte erlaubt, die aus einer UI-Komponente ausgelesen werden können. Will man beispielsweise den eingegebenen Benutzernamen auslesen, sieht der Methodenaufruf so aus: `<call name="test"><value-of comp-name="userName"/></call>`. Damit wird die methode `test(String)` mit dem Wert, der in der Komponente `userName` (ein Texteingabefeld) gespeichert ist, aufgerufen. *Wie* die Methoden aufgerufen werden, wird in AML nicht weiter spezifiziert. Üblicherweise sollte der sprachspezifische Interpreter in der Lage sein, den Methodenaufruf entweder lokal oder entfernt durchzuführen. Im der Client-Applikation des Prototypen wird z.B. eine Methode des Jini-Dienstes aufgerufen; allerdings nicht über RMI sondern über ein eigenes Protokoll.

Eine weitere mögliche Aktion ist die Änderung der Eigenschaft einer Komponente. Mit dem Element `<change-property>` ist eine solche Eigenschaftsänderung problemlos möglich. Das Element hat die Attribute `comp-name`, `prop-name` und `value`. Dabei gibt `comp-name` den Namen der entsprechenden Komponente an, `prop-name` den Namen der Eigenschaft und `value` deren neuen Wert. Mit `<change-property comp-name="statusLabel" prop-name="text"value="Ok."/>` wird z.B. der Text des Labels `statusLabel` zu «Ok.» geändert.

Die letzte Aktion, die AML unterstützt ist der Wechsel von einem Dialog zu einem anderen. Mit `<go dialog="robotControl" class="frame"/>` wird z.B. zu dem Dialog `robotControl` umgeschaltet. Das `class`-Attribut des `<go>`-Elementes bezeichnet die Art des Dialogs. Mit `frame` wird das aktuelle (Dialog-)Fenster ausgeblendet und stattdessen das neue Fenster angezeigt. Soll lediglich ein Popup-Fenster (z.B. ein Bestätigungsdialog) eingeblendet werden, nimmt man als Attributwert `popup`.

4.1.5 Fazit und Zusammenfassung

AML ist eine leicht erlernbare Sprache zur Beschreibung von Benutzerschnittstellen. Die Grundstruktur der Benutzerschnittstelle orientiert sich mit ihren `<dialog>`-Elementen an der Wireless Markup Language und an VoiceXML. Die Verhaltensbeschreibung von AML wurde zum Teil aus UIML übernommen.

Ein Nachteil von AML ist, daß bei der Entwicklung vor allem an grafische Benutzeroberflächen gedacht wurde. Es lassen sich zwar mit Konstrukten wie `<component class="prompt">` auch abstrakte Beschreibungen für sprachgesteuerte Applikationen entwickeln, aber eine parallele Entwicklung von grafischen und sprachgesteuerten Benutzerschnittstellen ist mit der Abstract Markup Language schwierig. Dabei ist fraglich, ob es in der Praxis sinnvoll ist, mit einer einzigen UI-Beschreibung beide Modalitäten abzudecken. Die UI-Beschreibung für die Robotersteuerung des Prototypen ist allerdings ein Ausnahmefall: Hier ist es tatsächlich möglich, die Beschriftung der Button-Komponenten in eine applikationsspezifische Grammatik für eine sprachgesteuerte Applikation umzusetzen. Es läßt sich jedoch nicht immer festlegen, daß ein Button-Klick genau einem Kommando (`<prompt>` in VoiceXML) entspricht.

Die vorhergehenden Abschnitte haben das Design von AML näher erklärt, ohne auf sämtliche Anforderungen aus Abschnitt 3.1 einzugehen. Es wurde also eine Sprache entwickelt, die relativ leicht zu erlernen ist und viele Möglichkeiten bietet. Allerdings werden verschiedene Anforderungen nicht von dieser Sprache abgedeckt. In ihrer derzeitigen Form ist AML nicht für wirklich komplexe Benutzerschnittstellen geeignet. So ist z.B. die Gliederung des UI in Dialoge sinnvoll, wenn die Benutzerschnittstelle auch auf Geräten mit kleineren Displays darstellbar sein soll oder wenn das UI aus einer sequentiellen Folge von Dialogen besteht (z.B. die Wizards unter MS Windows). Für andere Zwecke, z.B. die Oberfläche einer Textverarbeitung mit zahlreichen Toolbars[3] ist das nicht ausreichend.

Auch ist es sehr schwierig, Tools für die Interpretation von AML-Beschreibungen zu entwickeln, weil zahlreiche Eigenschaften bestimmter Komponenten hier nicht ausgedrückt werden können. Das heißt vor allem, daß spezielle Anforderungen, wie z.B. eine bestimmte Schriftart und -größe in Java-Buttons nicht beschrieben und damit auch nicht interpretiert werden können.

[3] Wobei man hier über Sinn von hoffnungslos überladenen Toolbars streiten kann.

Statt AML weiter an die Anforderungen anzupassen, habe ich mich entschieden, eine weitere Sprache für die Java-Programmiersprache zu entwickeln. Die Java Markup Language (JML) bietet die Möglichkeit, *beliebig komplexe* Benutzerschnittstellen zu entwickeln, da sich der strukturelle Aufbau am Objektgraphen des UI orientiert und nicht am der Anforderung der Einfachheit der Entwicklung. Dabei sollen Tools entwickelt werden, die AML-Instanzen in JML-Instanzen transformieren, um die Entwicklung einfach zu halten und gleichzeit auch komplexere Benutzerschnittstellen zu ermöglichen.

4.2 Die Java Markup Language (JML)

Wie schon im Abschnitt 4.1 (über die AML) beschrieben, reicht eine XML-Sprache nicht aus, um allen Anforderungen, die in 3.1 formuliert wurden, gerecht zu werden. Insbesondere ist die Anforderung der «Einfachheit» schwierig umzusetzen, wenn gleichzeitig eine umfassende Funktionalität gefordert wird. Deshalb wurde speziell für die Zielsprache Java eine weitere XML-basierte Sprache entwickelt: die Java Markup Language (JML).

4.2.1 Struktureller Aufbau

Die prinzipielle Aufteilung in Benutzerschnittstelle (`<userInterface>`) und Verhalten der Benutzerschnittstelle (`<behavior>`) der AML bleibt auch hier erhalten. Die Elementhierarchie der Verhaltensbeschreibung ist identisch zu der entsprechenden Elementhierarchie der AML. Die Elementhierarchie der JML-Benutzerschnittstelle (`<userInterface>`) orientiert sich am Objektgraphen der Toolkits AWT und Swing. Dieser Objektgraph wird in verschiedenen IDEs (z.B. JBuilder von Borland) als Baum dargestellt. Damit läßt sich mit JML eine beliebig komplexe Benutzerschnittstelle *äquivalent* zur Objekthierarchie eines Java-GUI beschreiben.

Die Java-IDE *Forté for Java* ist die zur Zeit einzige IDE, deren GUI-Builder den Objektgraphen als XML-Datei abspeichert. Leider hat dieses XML-Format einige Schwächen:

- Es ist keine DTD verfügbar.
- Bei einigen Attributen und Attributwerten ist die Bedeutung unklar.
- Oft werden statt einfachen Bezeichnungen sehr lange, Forté -spezifische Paket- und Klassennamen verwendet (z.B.`"org.netbeans.modules.form.compat2.layouts.DesignBorderLayout"` statt `"BorderLayout"`).

Deshalb habe ich mich entschlossen, das Ausgabeformat von Forté als Basis für JML zu nehmen und es entsprechend den Anforderungen anzupassen. Im Folgenden werden die Unterschiede zwischen JML und dem Forté-Format nicht mehr weiter erwähnt; stattdessen werden Syntax und Semantik von JML näher beschrieben.

4.2.2 Details

Die Verhaltensbeschreibung (`<behavior>`) aus AML wird unverändert in JML übernommen. Dafür wurde die UI-Beschreibung stark geändert; insbesondere ist die Dialogstruktur nur noch in der Verhaltensbeschreibung und (etwas versteckt) in bestimmten Layout-Managern vorhanden. Doch zunächst zur Struktur der `<userInterface>`-Kindelemente.

Normalerweise ist das oberste Element der UI-Hierarchie ein Container wie z.B. ein `Frame`, ein `Panel` oder ein `Window`. Container sind in Java - per Definition - auch Komponenten. Trotzdem ist es sinnvoll, den Unterschied zwischen typischen Containern (`Frame`, `Panel`, ...) und typischen Komponenten (`Button`, `TextField`, ...) durch zwei verschiedene Elemente hervorzuheben. Die JML-Elemente für Container und Komponente heißen - aus naheliegenden Gründen - `<container>` und `<component>`. Sowohl `<container>` als auch `<component>` haben zwei Attribute: `class` und `name`. Die `class` gibt dabei den genauen (Klassen-)Namen der entsprechenden Komponente an. Es ist normalerweise nicht notwendig, die gesamte Paketstruktur der Komponente aufzuzählen (z.B. `java.awt.TextField`), da die notwendigen Pakete ohnehin erkannt und importiert werden müssen. Der `name` ist eine frei wählbare Bezeichnung für die Komponte, die aber *eindeutig* sein muß.

Sowohl `<container>` als auch `<component>` haben folgende Kindelemente:

- `<constraints>`,
- `<events>` und
- `<properties>`;

wobei `<container>`-Elemente noch `<layout>`-Kindelemente haben können. Außerdem können Container - im Gegensatz zu Komponenten - weitere Komponenten und Container als Kindelemente haben. Folgendes Listing zeigt eine einfache Komponentenhierarchie (aus dem UI des Prototypen):

```
<userInterface>
  <container class="Frame" name="javaUI">
    <events>
      <eventHandler event="windowClosing" handler="exitForm"/>
    </events>
    <layout class="BorderLayout"/>
```

```
<container class="Panel" name="cardPanel">
   <constraints>
      <constraint layoutClass="BorderLayout" value="">
         <borderConstraints direction="CENTER"/>
      </constraint>
   </constraints>
   <layout class="CardLayout"/>
   <container class="Panel" name="overview">
      <constraints>
         <constraint layoutClass="CardLayout" value="">
            <cardConstraints cardName="overview"
               cardString="OVERVIEWPANEL"/>
         </constraint>
      </constraints>
      <layout class="FlowLayout"/>
      <component class="Label" name="overviewLabel">
         <properties>
            <property name="background" type="Color">
               <color id="" red="0" blue="0" green="127"/>
            </property>
            <property name="foreground" type="Color">
               <color id="white" red="" blue="" green=""/>
            </property>
            <property name="text" type="String"
               value="PD MINDSTORMS CONTROL"/>
         </properties>
         <constraints>
            <constraint layoutClass="FlowLayout" value=""/>
         </constraints>
      </component>
      [...]
   </container>
   <container class="Panel" name="robotControl">
      <properties>[...]</properties>
      <constraints>
         <constraint layoutClass="CardLayout" value="">
            <cardConstraints cardName="robotControl"
               cardString="ROBOTCONTROLPANEL"/>
         </constraint>
      </constraints>
      <layout class="GridLayout">
         <properties>
            <property name="rows" value="5"/>
            <property name="columns" value="3"/>
         </properties>
      </layout>
      <component class="Label" name="robotLabel">
```

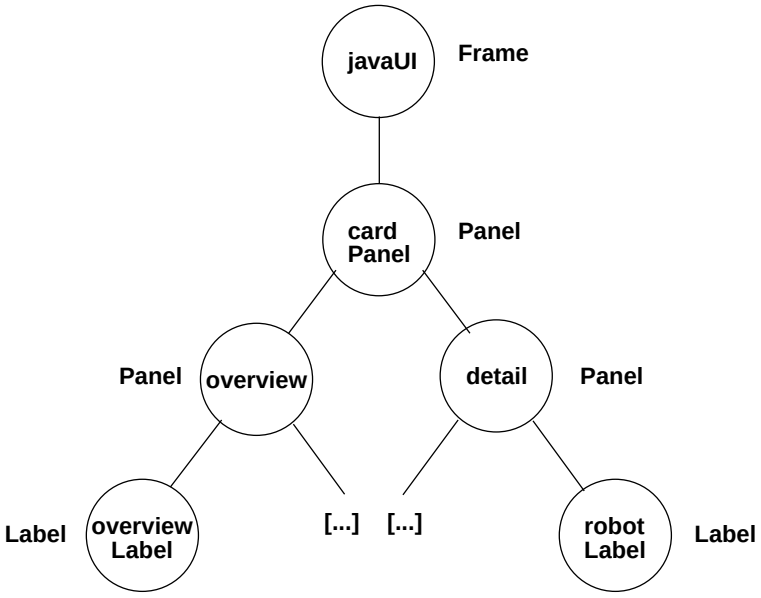


Abbildung 4.2: Komponentenhierarchie

```
54              <properties>[...]</properties>
                <constraints>
56                 <constraint layoutClass="GridLayout" value=""/>
                </constraints>
58            </component>
              [...]
60          </container>
        </container>
62    </container>
</userInterface>
```

Abbildung 4.2 zeigt die Komponenten/Container-Hierarchie als Baum.

Die Verbindung zwischen `<layout>` und `<constraints>` ist nicht sofort offensichtlich. Layout-Elemente treten nur als Kindelemente von Containern auf. Sie spezifizieren den Layout-Manager des entsprechenden Containers. In obigem Listing ist das Frame mit dem Namen «javaUI» das oberste Element in der Hierarchie. Das Layout dieses Containers ist `BorderLayout`. Sollen weitere Komponenten zu einem Container mit einem `BorderLayout` als Layout-Manager hinzugefügt werden, muß zusätzlich eine Richtungsangabe (NORTH, WEST, CENTER, ...) gemacht werden, damit die Komponente korrekt im Kontext des Layout-Managers plaziert wird. Diese zusätzliche Angabe (die von Layout-Manager zu Layout-Manager verschieden sein kann), wird im `<constraints>`-Element der entsprechenden Kindkomponente gemacht. Hier wird das Panel «cardPanel» mit `CENTER` in das Eltern-Frame gesetzt.

4.2.3 Fazit und Zusammenfassung

Die Java Markup Language ermöglicht eine vollständige Beschreibung von beliebig komplexen Benutzeroberflächen. Da sich das Design der Sprache stark an dem Java-Objektgraphen orientiert, dürfte es kein Problem sein, JML in andere (textuelle) Ausgabeformate zu transformieren. Das Ausgabeformat von Forté basiert ebenfalls auf dem Graph der GUI-Objekte und ist deshalb JML sehr ähnlich; d.h. eine Transformation von JML in Forté-XML und umgekehrt ist unproblematisch. Da eine JML-Beschreibung semantisch äquivalent zum entsprechenden Java-Quellcode ist, ist es nicht schwierig, Tools für die Transformation bzw. Interpretation von JML zu schreiben.

Die Kombination AML/JML deckt den größten Teil der Anforderungen aus Abschnitt 3.1 ab. Leider wurde die Zielsprachenunabhängigkeit nicht zufriedenstellend gelöst. Es ist zwar einfach, aus der Abstract Markup Language z.B. HTML oder WML zu generieren aber es ist problematisch, AML in VoiceXML zu transformieren. Es ist zwar möglich, in AML mit speziellen VoiceXML-Konstrukten (`<prompt>`) zu arbeiten, aber dann kann daraus kein GUI generiert werden. Bei speziellen Anwendungen ist es möglich, z.B. Button-Komponenten in `<prompt>`-Elemente zu transformieren, aber meistens entsteht daraus keine besonders komfortable VoiceXML-Anwendung. Eine Erweiterung von JML könnte das Problem beseitigen. Da die Java Speech API die Entwicklung sprachbasierter Applikationen ermöglicht, könnten mit einer JML-Erweiterung sprachorientierte AML-Beschreibungen in JML transformiert werden. Eine zufriedenstellende Lösung bei der parallelen Entwicklung von sprach- und GUI-basierten Applikationen ist allerdings nur schwer möglich.

4.3 Die Transformations-Tools

Es sollen Tools entwickelt werden, die Transformationen

- von AML in JML und andere Zielsprachen (VoiceXML, WML, ...) und
- von JML in Java-Quellcode

vornehmen. Das heißt, daß mindestens zwei verschiedene Stylesheets entwickelt werden müssen. Mit XSLT ist es allerdings möglich, Stylesheets modular aufzubauen. Die folgenden Abschnitte beschreiben die entwickelten Tools und deren Module.

4.3.1 Transformation von AML zu JML

Dialogstruktur und Layout-Manager

Das Hauptproblem bei der Transformation von AML zu JML ist die Abbildung der Dialog-Struktur von AML auf die Objekthierarchie von JML. Es gibt hier keine opti-

male Lösung, aber speziell für den Palm-Client des Prototypen ist es sinnvoll, den Layout-Manager `CardLayout` zu verwenden. Dieser Layout-Manager ermöglicht es, einem Fenster verschiedene Masken zuzuweisen und bei Bedarf zwischen diesen Masken umzuschalten. Bei der Benutzeroberfläche des Prototypen (siehe Abbildung 4.11 im Abschnitt 4.5.3) kann man zwischen zwei Masken umschalten: der Startmaske, auf der man seinen Namen eintragen kann und der Hauptmaske, auf der die Buttons zur Robotersteuerung plaziert sind.

Alle Komponenten eines Dialoges werden also in ein eigenes `Panel` gesetzt, das wiederum ein Kindelement eines `Panel`-Objektes mit dem `CardLayout` ist. Folgender Codeauschnitt zeigt die AML-Beschreibung und die entsprechende JML-Beschreibung:

```
<aml>
    <dialog></dialog>
    <dialog></dialog>
</aml>

<jml>
    <container class="Panel" name="cardPanel">
        <container class="Panel" name="cont1">
            <constraints>
                <constraint layoutClass="CardLayout" value="">
                    <cardConstraints cardName="cont1" cardString="CONT1PANEL"/>
                </constraint>
            </constraints>
        </container>
        <container class="Panel" name="cont1">
            <constraints>
                <constraint layoutClass="CardLayout" value="">
                    <cardConstraints cardName="cont2" cardString="CONT2PANEL"/>
                </constraint>
            </constraints>
        </container>
    </container>
</jml>
```

Für die Plazierung von Komponenten innerhalb eines `<dialog>`-Elementes werden zwei Layout-Manager eingesetzt: das `FlowLayout` und das `GridLayout`. Der Layout-Manager `FlowLayout` wird verwendet, um Komponenten ohne explizite Layout-Information anzuordnen:

```
<aml>
    <dialog>
        <component class="button" name="button1"/>
        <component class="button" name="button2"/>
    </dialog>
</aml>
```

```
<jml>
    <container class="Panel" name="cont1">
        <layout class="FlowLayout"/>
        <component class="button" name="button1">
            <constraints>
                <constraint layoutClass="FlowLayout" value=""/>
            </constraints>
        </component>
        <component class="button" name="button2"/>
            <constraints>
                <constraint layoutClass="FlowLayout" value=""/>
            </constraints>
        </component>
    </container>
</jml>
```

Die Sprache AML bietet auch die Möglichkeit, Komponenten in Tabellenform anzuordnen. Für diese Anordnung ist das `GridLayout` nützlich:

```
<aml>
    <dialog>
        <table rows="2" cols="2">
            <row>
                <col><component class="button" name="button1"/></col>
                <col><component class="button" name="button2"/></col>
            </row>
            <row>
                <col><component class="button" name="button3"/></col>
                <col><component class="button" name="button4"/></col>
            </row>
        </table>
    </dialog>
</aml>

<jml>
    <container class="Panel" name="cont1">
        <layout class="GridLayout">
            <properties>
                <property name="rows" value="2"/>
                <property name="columns" value="2"/>
            </properties>
        </layout>
        <component class="button" name="button1">
            <constraints>
                <constraint layoutClass="GridLayout" value=""/>
            </constraints>
        </component>
```

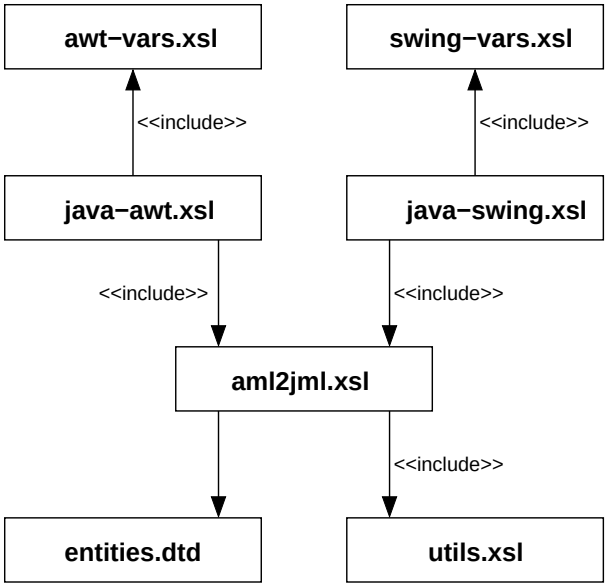


Abbildung 4.3: Module des Tools «aml2jml»

```
        <component class="button" name="button2"/>
30          <constraints>
                <constraint layoutClass="GridLayout" value=""/>
32          </constraints>
        </component>
34      [...]
    </container>
36 </jml>
```

Es sieht zunächst seltsam aus, daß die `<constraints>` der Komponenten keinen Wert (`value`) haben. Denkbar wäre hier z.B. `<constraint layoutClass="GridLayout" row="1" col="1"/>` um eine Komponente in Position (1,1) im Gitternetz des Layout-Managers zu plazieren. Leider erlaubt die derzeitige Version des `GridLayout`-Managers keine absolute Positionierung.

Module

Obwohl im Rahmen dieser Master Thesis nur ein Tool zur Transformation von AML-Beschreibungen in JML-Beschreibungen entwickelt wird, soll das entsprechende Tool (`aml2jml.xsl`) modular aufgebaut sein. Es müssen also Allzweck-Module entwickelt werden, die auch z.B. von einem Tool zur Transformation von AML in WML verwendet werden können. Abbildung 4.3 zeigt die Module des Tools «aml2jml»[4].

Die Kernfunktionalität ist in dem Modul `aml2jml.xsl` enthalten. Das Modul `utils.xsl` ist als Allzweck-Utility-Modul gedacht; enthält zur Zeit aber nur ver-

[4] Implementiert wurden lediglich die AWT-Module.

schiedene Variablen für String-Manipulationen. Die Dokumenttyp-Definition `entities.dtd` enthält verschiedene, häufig benötigte Entities wie z.B. `&n;` für das Einfügen einer neuen Zeile oder `&tab;` für das Einfügen eines Tabulator-Zeichens. Das Stylesheet `awt-vars.xsl` enthält verschiedene Variablen-Definitionen, die das Hauptmodul `aml2jml.xsl` frei von toolkitspezifischen Definitionen halten. Etwas ungewohnt ist es, daß `aml2jml.xsl` von `java-awt.xsl` eingebunden wird und nicht umgekehrt. Der Grund dafür ist, daß in XSLT die `<import>`- und `<include>`-Statements zum Compilierzeitpunkt ausgeführt werden; es also nicht möglich ist, zur Laufzeit dynamisch Module zu laden. Es wird stets das allgemeine Stylesheet vom speziellen Stylesheet eingebunden. Um eine Transformation auszuführen wird das spezielle Stylesheet (z.B. `java-awt.xsl`) aufgerufen.

Ein Beispiel für die AWT-Definition eines Buttons ist `<xsl:variable name="button"> Button </xsl:variable>`. In `aml2jml.xsl` wird diese Variable mit `$button` eingebunden. Wird `java-awt.xsl` z.B. durch `java-swing.xsl` ausgetauscht, kann sofort JML für das Swing-Toolkit generiert werden. Eine Variablendefinition für einen Swing-Button sieht dann so aus: `<xsl:variable name="button">JButton</xsl:variable>`. Dieses Designkonzept ermöglicht also durch einen einfachen Austausch von bestimmten Modulen die Verwendung eines anderen Toolkits.

4.3.2 Transformation von JML zu Java-Quellcode

Es gibt aus Designsicht keine großen Unterschiede zwischen der Transformation von AML nach JML und der Transformation von JML nach Java. Das allgemeine Stylesheet (`java-codegen.xsl`) wird von den speziellen Stylesheets (`awt-codegen.xsl`, ...) eingebunden. Diese speziellen Stylesheets wiederum binden die Stylesheets mit den Variablendefinitionen ein. Abbildung 4.4 zeigt den Aufbau des Tools «java-codegen».

Bei der Transformation von AML zu JML sind sowohl Quell- als auch Zielsprache XML-Anwendungen. Dagegen ist das Ergebnis der Transformation von JML zu Java-Quellcode reiner Text. Das hat verschiedene Auswirkungen auf die Implementierung. Zum einen muß `<xsl:output method="text"/>` gesetzt werden und zum anderen muß darauf geachtet werden, daß kontextbezogene Einrückungen, Zeilenwechsel u.ä. korrekt erkannt werden. Schwierig ist es auch, für die Events passende Methoden zu generieren. Ein Problem dabei ist z.B. daß es in Java nicht erlaubt ist, daß zwei Methoden den gleichen Namen tragen. Es kann aber durchaus vorkommen, daß eine bestimmte Methode sowohl bei einem Klick auf einen Button als auch bei der Anwahl eines Menüpunktes aufgerufen werden soll. Mit XSLT ist es relativ schwierig, solche Fälle korrekt zu erkennen, da in XSLT eine Ausgabe als Funktion der Eingabe beschrieben werden muß. Hier muß mit dem Konstrukt `<xsl:key>` gearbeitet werden. Folgender Ausschnitt aus dem Quellcode zeigt, wie `<eventHandler>`-Elemente erkannt und mit `<xsl:key>` abgespeichert werden:

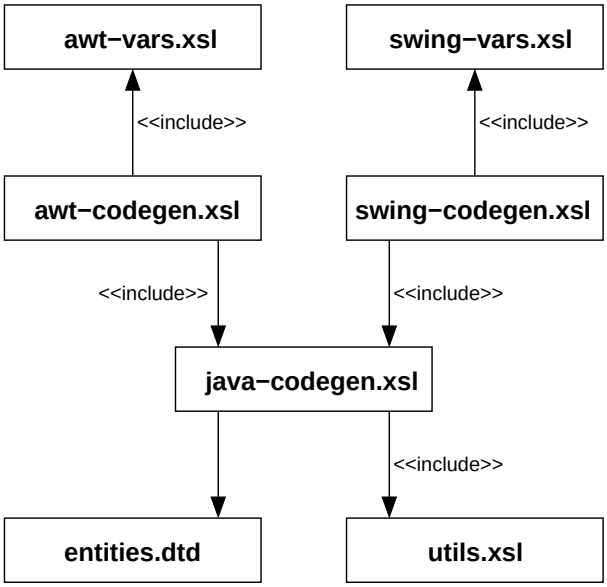


Abbildung 4.4: Aufbau von «java-codegen»

```
<!--
    This key produces something like "eventHandler::saveFileActionPerformed.
    You could select specific event handlers by using:

    <xsl:for-each select=
        "key('event-handlers', 'eventHandler::saveFileActionPerformed')">
-->
<xsl:key name="event-handlers" match="eventHandler"
         use="concat(name(), '::', @handler)"/>

<!--
    This template generates methods for each eventHandler. The trick is
    that two or more event handlers with the same name get mapped to only
    one method. A call to the template "behavior-rules" includes some
    code in the body of the methods.
-->
<xsl:template match="events" mode="methods">
    <!--
        The following XPath-Expression does the following:
        It checks whether the set that results from comparing the two
        nodes has one or more nodes in it. If the set consists of only
        one node, then the two compared nodes are identical.
        This means, the current eventHandler node is the first node
        in the list returned by the key.
        If this is the case, then we add a new event handler method,
        otherwise we skip the
        method generation.
```

```
    -->
    <xsl:for-each select="eventHandler[count(. | key('event-handlers',
        concat(name(), '::', @handler))[1]) = 1]">
        <xsl:choose>
            <!-- actionPerformed -->
            <xsl:when test="@event = 'actionPerformed'">
                &tab;private void&space;
                <xsl:value-of select="@handler"/> (ActionEvent event)&n;
                &tab;{&n;
                &tab2;// Put your code here&n;
                <!--
                    Here we insert the action that results from the
                    condition in the behavior section of the document.
                -->
                <xsl:call-template name="behavior-rules">
                    <xsl:with-param name="handler" select="@handler"/>
                </xsl:call-template>
                &tab;}&n2;
            </xsl:when>
        </xsl:choose>
    </xsl:for-each>
</xsl:template>
```

Zum Glück muß bei den meisten Elementen kein derartiger Aufwand getrieben werden. Meistens genügt es, ein Template zu schreiben, das auf eine bestimmte Eigenschaft paßt. Im folgenden Beispiel wird die Eigenschaft «Font» bearbeitet:

```
<xsl:template match="property[@type = 'Font']">
    &tab2;<xsl:value-of select="../parent::node()/@name"/>
    &e;.setFont(new Font ("&e;
    <xsl:value-of select="font/@name"/>&e;", &e;
    <xsl:value-of select="font/@style"/>&e;, &e;
    <xsl:value-of select="font/@size"/>));&n;
</xsl:template>
```

Manchmal ist es auch sinnvoll, ein allgemeineres Template zu schreiben und mit `<xsl:if>` oder `<xsl:choose>` spezielle Fälle abzudecken. Das Template für das `<layout>`-Element muß zunächst ermitteln, ob sich das Layout auf den Root-Container bezieht oder auf einen Sub-Container. Abhängig davon, wird dann entweder `this.setLayout(...)` oder `subContainer.setLayout(...)` generiert. Um das Layout zu setzen muß ein Layout-Manager instanziert werden. Abhängig vom Layout-Manager wird dann der Konstruktur erzeugt. In folgendem Quellcode wird der Konstruktor für den `GridLayout`-Layout-Manager generiert. Falls die Werte der Attribute `rows` und `columns` Leerstrings sind, wird ein `GridLayout` mit «0» Zeilen und «0» Spalten erzeugt; ansonsten werden hier die entsprechenden Werte eingetragen.

```
<xsl:template match="layout">
    <!--
        Is the corresponding node the rootcontainer? Then add the
        current component with 'this'.
    -->
    <xsl:choose>
        <xsl:when test="name(../parent::node())='userInterface'">
            &tab2;this&e;
        </xsl:when>
        <xsl:otherwise>
            &tab2;<xsl:value-of select="parent::node()/@name"/>
        </xsl:otherwise>
    </xsl:choose>

    &e;.setLayout(new <xsl:value-of select="@class"/> (&e;
        <xsl:choose>
            <xsl:when test="@class = 'GridLayout'">
                <xsl:choose>
                    <xsl:when test="not(properties/property
                        [@name = 'rows']/@value´´ = )">
                        <xsl:value-of select="properties/property
                            [@name = 'rows']/@value"/>
                    </xsl:when>
                    <xsl:otherwise>0</xsl:otherwise>
                </xsl:choose>
                &e;,&space;
                <xsl:choose>
                    <xsl:when test="not(properties/property
                        [@name = 'columns']/@value´´ = )">
                        <xsl:value-of select="properties/property
                            [@name = 'columns']/@value"/>
                    </xsl:when>
                    <xsl:otherwise>0</xsl:otherwise>
                </xsl:choose>
            </xsl:when>
            [...]
        </xsl:choose>
    &e;));&n;
</xsl:template>
```

Mittlerweile sind im Netz verschiedene Websites zu finden, die Tips, Techniken und Entwurfsmuster sammeln. Empfehlenswert sind http://www.jenitennison.com/xslt/ und http://www.dpawson.co.uk/.

4.4 Der Interpreter

Die Interpretation der UI-Beschreibung ist ein wesentliches Feature des Prototypen, da es sonst nicht möglich ist, auf Palm-PDAs die Benutzerschnittstelle für die Steuerung des Lego Mindstorms Roboters zur Laufzeit über das Netz nachzuladen. Also muß der Interpreter bereits auf dem PDA installiert sein, um die UI-Beschreibung interpretieren zu können. Bei einem Desktop-Client lassen sich die UI-Klassen zwar problemlos nachladen, aber zu Demonstrationszwecken wird auch hier die UI-Beschreibung interpretiert. Der Unterschied zum PDA-Client ist, daß die UI-Beschreibung *einschließlich* des Interpreters zur Laufzeit geladen wird.

Der Interpreter arbeitet plattform- und toolkitspezifisch. Hier soll ein Interpreter für das AWT-Toolkit der Programmiersprache Java entwickelt werden. Eine wichtige Anforderung an den Interpreter ist, daß er sowohl auf Virtuellen Maschinen der J2SE als auch auf Virtuellen Maschinen der J2ME (CLDC-Konfiguration) läuft. Da J2ME/CLDC zunächst nicht das AWT unterstützen, muß die Zusatzbibliothek *kAWT*, zu finden unter http://www.kawt.de installiert werden. Für den Interpreter ist es nicht relevant, ob AWT oder kAWT verwendet werden; entscheidend ist, daß AWT-Objekte instanziert werden können. Der Interpreter, der im Rahmen dieser Master Thesis implementiert wurde, unterstützt folgende Plattformen, Toolkits und Java-Versionen:

System	Toolkit	Java-Version
Desktop-PC	AWT	J2SE (1.3.0)
Palm-PDA	kAWT	J2ME CLDC1.0/KVM Palm

Tabelle 4.1: Unterstützte Hardware, Toolkits und Java-Versionen

Um die XML-Beschreibung verarbeiten zu können, ist ein XML-Parser notwendig. Als XML-Parser kommen nur Parser in Frage, die auf der KVM lauffähig sind. Für den Interpreter wurde eine modifizierte Version des *TinyXML*-Parsers verwendet. Die TinyXML-Website liegt unter http://www.gibaradunn.srac.org/tiny/index.shtml, die modifizierte Version ist auf der KVMWorld-Website (übrigens eine empfehlenswerte Seite für den Einstieg in die KVM-Programmierung) unter http://www.kvmworld.com/Downloads/Utilities/TinyXML.shtml zu finden.

Abbildung 4.5 zeigt das Klassendiagramm des Interpreters (einschließlich der TinyXML-Klassen).

Der `SaxHandler` (abgeleitet von `HandlerBase`) erkennt die Elemente der XML-Beschreibung und ruft bei deren Auftreten die entsprechenden Methoden der Klasse `Interpreter` auf. Dabei wird intern der gerade aktuelle Teilbaum des XML-Dokumentes abgespeichert. Das Sequenzdiagramm in Abbildung 4.6 zeigt den typischen Ablauf beim Parsen einer XML-Beschreibung. Wenn ein `<component>`-Element geparst wird, so wird das entsprechende UI-Element (z.B. ein Button) instanziert und intern als `_currentComponent` abgespeichert. Alle auftretenden `<property>`-Tags werden automatisch der aktuellen Komponente zugewiesen. Folgender Quellcode aus `Interpreter.java` zeigt, wie die Properties zugewiesen werden:

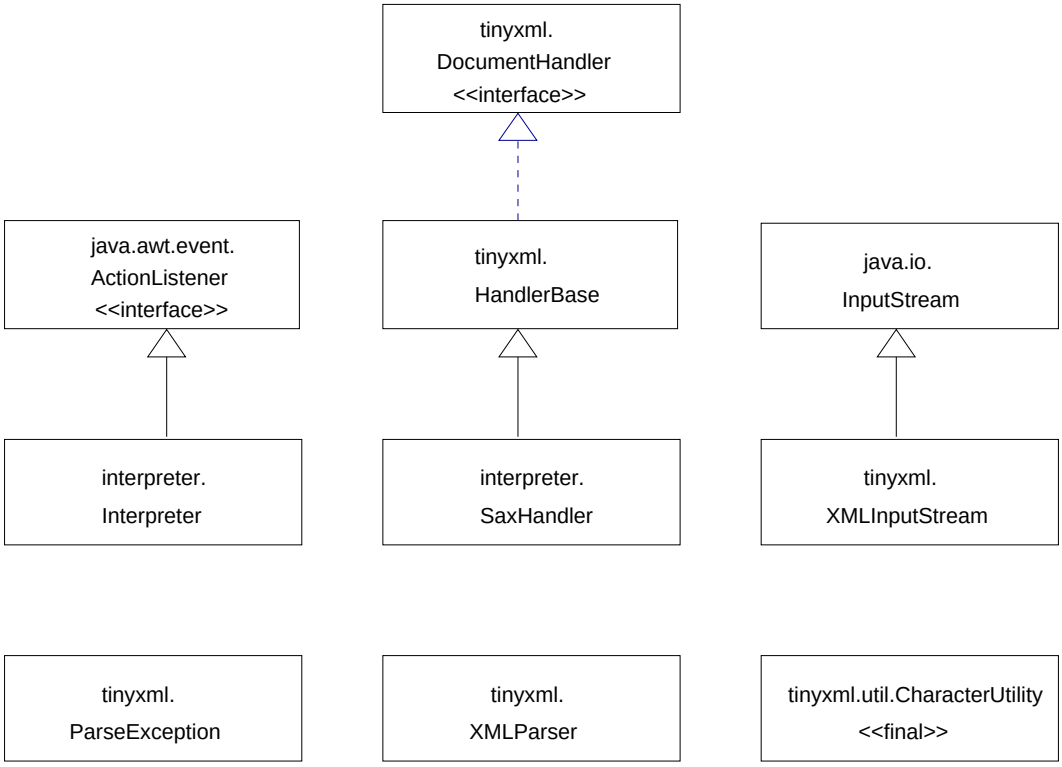


Abbildung 4.5: Klassendiagramm des Interpreters

```
if (propertyName.equals("text")) {
    String propertyValue = (String)attList.get("value");
    if (_currentComponent instanceof Label) {
        Label myLabel = (Label)_currentComponent;
        myLabel.setText(propertyValue);
    }
    if (_currentComponent instanceof TextArea) {
        TextArea myTextArea = (TextArea)_currentComponent;
        myTextArea.setText(propertyValue);
    }
}
```

Ist der Name der Eigenschaft «text» (`<property name="text">`), dann wird abhängig von der aktuellen Komponente der entsprechende Text gesetzt.

Die aktuelle Komponente wird zum aktuellen Container (z.B. einem Frame) hinzugefügt, wenn ein `<constraint>`-Tag geparst wurde und somit bekannt ist, *wie* die Komponente zum Container hinzugefügt werden soll. Der folgende Quellcode zeigt einen Ausschnitt aus den Methoden `createComponent()` und `addComponent()`.

```
public void createComponent(Hashtable attList)
{
    String componentName = (String)attList.get("name");
```

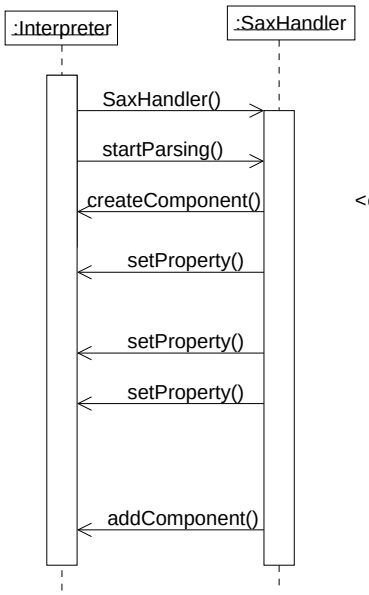
```
4       String componentClass = (String)attList.get("class");

6       if (componentClass.equals("Button")) {
            Button myButton = new Button();
8           _currentComponent = myButton;
        }
10      else if (componentClass.equals("Label")) {
            Label myLabel = new Label();
12          _currentComponent = myLabel;
        }
14      [...]
        _currentComponent.setName(componentName);
16  }

18
    public void addComponent(String special, Hashtable attList)
20  {
        Container parentContainer = null;
22      if (_currentComponent instanceof Container) {
            parentContainer = _model.getParent((Container)_currentComponent);
24      }

26      if (DEBUG) System.out.println("### Adding \""+_currentComponent.getName()
                +"\" to \""+parentContainer.getName()+"\".");
28
        if (special.equals("borderConstraints")) {
30          String direction = (String)attList.get("direction");

32          if (direction.equals("CENTER")) {
                parentContainer.add(_currentComponent, BorderLayout.CENTER);
34          }
            else if (direction.equals("EAST")) {
36              parentContainer.add(_currentComponent, BorderLayout.EAST);
            }
38          else if (direction.equals("NORTH")) {
                parentContainer.add(_currentComponent, BorderLayout.NORTH);
40          }
            else if (direction.equals("SOUTH")) {
42              parentContainer.add(_currentComponent, BorderLayout.SOUTH);
            }
44          else if (direction.equals("WEST")) {
                parentContainer.add(_currentComponent, BorderLayout.WEST);
46          }
            else {
48              parentContainer.add(_currentComponent);
            }
50
```




```
<component class="button">
  <properties>
    <property name="background" type="Color">
      <color id="black" red="" blue="" green=""/>
    </property>
    <property name="font" type="Font">
      <font name="dialog" size="11" style="bold"/>
    </property>
    <property name="visible" type="boolean" value="true" >
  </properties>
  [...]
  <constraints>
    <constraint layoutClass="BorderLayout" value="">
      <borderConstraints direction="CENTER"/>
    </constraint>
  </constraints>
</component>
```

Abbildung 4.6: Sequenzdiagramm: Interpretation einer XML-Beschreibung

```
    }
52  [...]
    _model.addComponent(_currentComponent);
54 }
```

Die Methode `createComponent()` wird aufgerufen, wenn in der XML-Beschreibung ein `<component>`-Element gefunden wurde. Dann werden Komponentenname und -klasse gelesen und es wird eine entsprechende Komponente erzeugt. Oft ist es beim Hinzufügen einer Komponente zu einem Container erforderlich, ein zusätzliches, beschreibendes Argument anzugeben. Wenn der aktuelle Container z.B. ein `BorderLayout` hat, dann muß eine Richtung (NORTH, SOUTH, WEST, EAST, CENTER) angegeben werden. Deshalb wird die aktuelle Komponente erst nach dem Lesen der `<constraints>` zu einem Container hinzugefügt. In `addComponent()` wird zunächst der übergeordnete Container mit `_model.getParent()` ermittelt. Der abgedruckte Quellcode zeigt, wie die aktuelle Komponente zum Elterncontainer hinzugefügt wird, wenn der Elterncontainer das `BorderLayout` verwendet. Abhängig von dem Attribut `direction` wird die Komponente im Elterncontainer plaziert.

Komfortabler wäre es, mit einem DOM-basierten XML-Parser das Dokument zu parsen und als DOM-Repräsentation an den Interpreter zu übergeben. Leider ist der Speicher von typischen CLDC-Geräten zu begrenzt, um einen vollständigen DOM-Baum aufzubauen. Der Verzicht auf eine Speicherung der DOM-Repräsentation führt allerdings auch dazu, daß der `Interpreter` zahlreiche Zustandsinformationen abspeichern muß, um z.B. Eigenschaften auch den richtigen GUI-Komponenten zuweisen zu können.

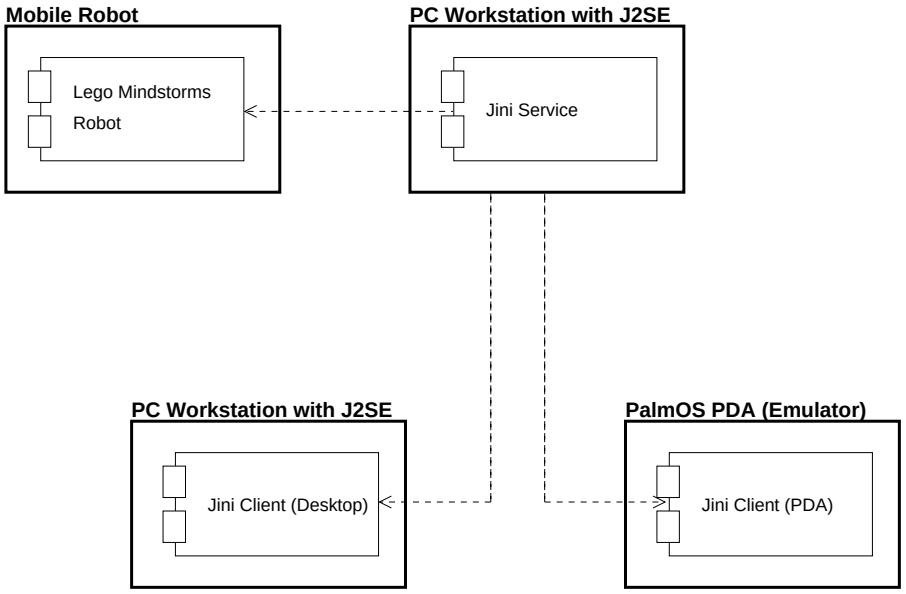


Abbildung 4.7: Komponenten des Prototypen

4.5 Der Prototyp

4.5.1 Statisches Modell

Als Prototyp soll eine Jini-basierte Applikation entwickelt werden, die die Steuerung von Lego Mindstorms Robotern über verschiedene Clients erlaubt. Abbildung 4.7 (Komponenten des Prototypen) gibt eine Übersicht über die beteiligten Komponenten.

Wie man sieht, muß der Jini-Dienst zur Robotersteuerung Benutzerschnittstellen für verschiedene Typen von Clients ausliefern. Dazu ist ein spezielles Design des Dienstes erforderlich, so daß an Desktop-Clients ein UI-Objekt und an PDA-Clients ein XML-Stream (über eine TCP-Socketverbindung) geschickt wird.

Im Paketdiagramm (Abbildung 4.8) sieht man die Paketstruktur der Java-Applikation. Das Paket `common` beinhaltet Schnittstellen, die sowohl vom Client als auch vom Service benötigt werden. Das Paket `net.jini.lookup.ui.factory` wird zwar von der Implementierung der Service-UI-Spezifikation zur Verfügung gestellt (sowohl im Quell- als auch im Binärcode), allerdings wurden im Rahmen dieser Thesis einige Erweiterungen entwickelt, auf die zum Compilierzeitpunkt zugegriffen werden muß.

4.5.2 Dynamisches Modell

In diesem Abschnitt geht es um die Interaktion zwischen dem Desktop-Client bzw. dem PDA-Client und dem Jini-Dienst. Diese beiden Clients bieten die gleiche Benutzerschnittstelle zum Dienst an. Ein wesentlicher Unterschied ist allerdings, daß

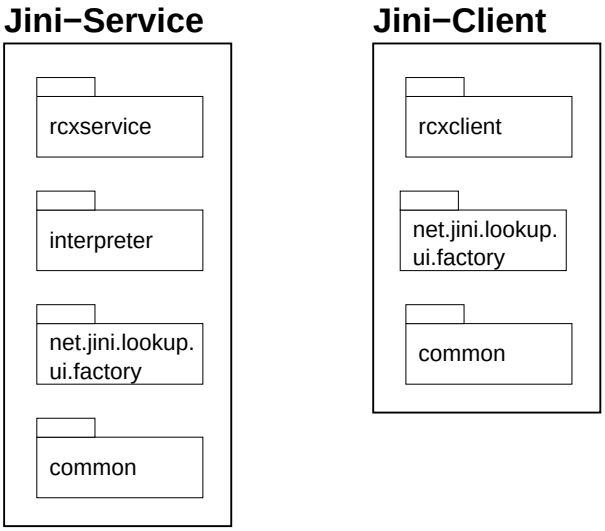


Abbildung 4.8: Paketdiagramm

der Desktop-Client sein UI über die Standard-Protokolle von Jini erhält, während der PDA-Client und der Jini-Dienst über ein proprietäres Protokoll kommunizieren.

Der Desktop-Client ist ein Jini-Client, der im Kontext einer J2SE-VM läuft und deshalb dynamisch Klassen nachladen kann. Also kann dieser Client sich über das Discovery-and-Join-Protokoll mit dem Jini-Netzwerk verbinden: Der Client schickt einen Multicast-Request in das Netz; der Lookup-Service antwortet und verschickt das Registrar-Objekt an den Client. Der Client sucht dann nach dem gewünschten Dienst und dessen Benutzerschnittstelle. Da in diesem Prototypen die Anwendung von XML-basierten Schnittstellenbeschreibungen gezeigt werden soll, instanziert die Benutzerschnittstellen-Fabrik die Schnittstelle nicht über eine Service-spezifische Klasse (etwa `RCXFrameFactory`) sondern über die XML-Beschreibung. Das heißt insbesondere, daß ein geeigneter Interpreter zur Verfügung gestellt werden muß, den der Client bei Bedarf von einem Web-Server laden muß. Abbildung 4.9 zeigt den genauen Ablauf.

Zuerst (1) fragt der Client beim Lookup-Server an, ob ein Dienst, der die Schnittstelle `JMLFrameFactory` bzw. die Schnittstelle `JMLSpeechFrameFactory` implementiert vorhanden ist. Da sich der Service zuvor beim LUS registriert hat, schickt der LUS das Proxy-Objekt des Services zum Client (2). Die übrigen benötigten Klassen (darunter auch der Interpreter) werden dann vom Webserver geladen (3). Über das Proxy-Objekt und die anderen Service-Klassen kann der Client den Lego-Roboter steuern (4). Dabei ist für den Client transparent, daß er nicht direkt mit dem Roboter kommuniziert, sondern mit einem Gerätetreiber (dem Jini-Dienst).

Beim PDA-Client sieht der Ablauf etwas anders aus (vgl. 4.10). Der PDA-Client kommuniziert weder mit dem Lookup-Service noch mit dem Webserver. Stattdessen verbindet sich dieser Client über Sockets direkt mit dem Service (1) und fordert sofort ein JML-UI an (2). Nachdem das UI geladen und interpretiert wurde (3) kom-

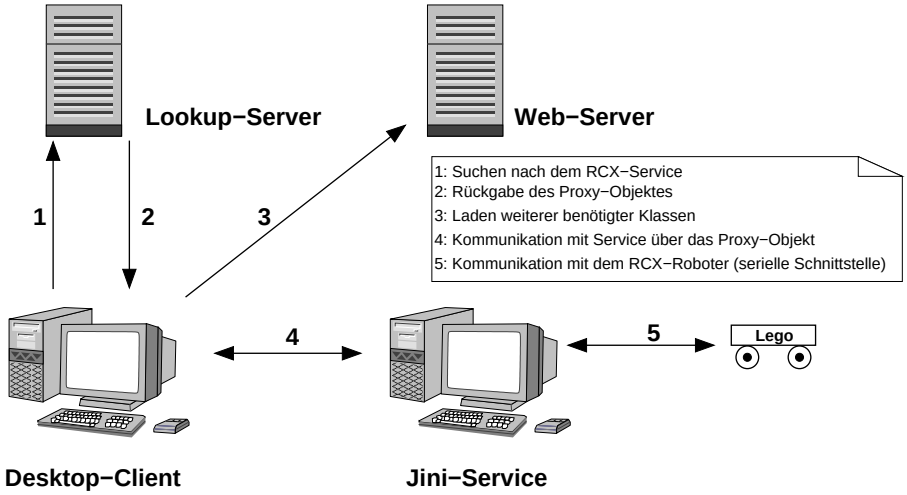


Abbildung 4.9: Laden einer Benutzerschnittstelle (Desktop-Client)

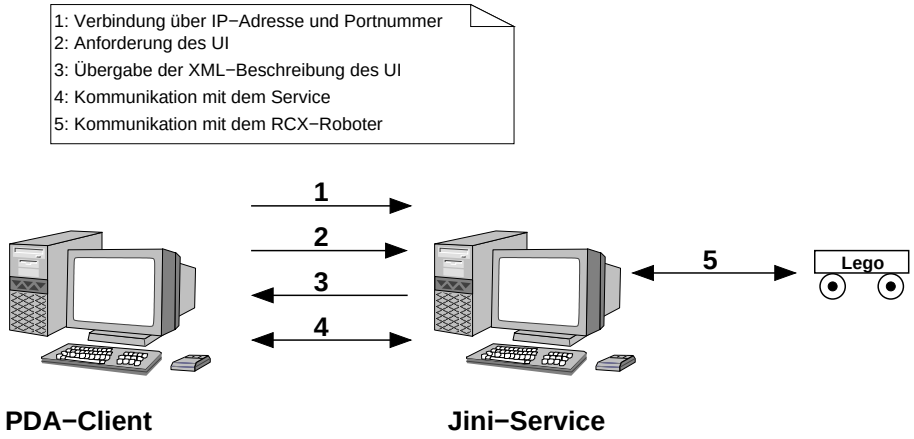


Abbildung 4.10: Laden einer Benutzerschnittstelle (PDA-Client)

munizieren Client und Service nun über Kommandos und Nachrichten miteinander (4). Diese Kommandos und Nachrichten sind Teil eines Protokolls, das speziell für diese Applikation entwickelt wurde. Details zu dem verwendeten Protokoll sind im Abschnitt 19 (Der PDA-Client) zu finden.

Die IP-Adresse bzw. der Name des Rechners, auf dem der Service läuft, muß clientseitig voreingestellt sein. Das ist ein großer Nachteil, da die Client-Applikation modifiziert werden muß, wenn der gewünschte Dienst auf einem anderen Rechner installiert wird. Eine mögliche Lösung wäre, einen Standard-Rechnernamen und eine Standard-Portadresse zu verwenden.

4.5.3 Benutzerschnittstelle

Die Benutzerschnittstelle für die Steuerung der Roboter soll möglichst einfach aufgebaut sein, aber dennoch eine befriedigende Funktionalität bieten. So soll es möglich sein, den Roboter vorwärts und rückwärts fahren zu lassen, eine Rechts- oder Linksdrehung vorzunehmen, den Roboter ein akustisches Signal ausgeben lassen und ihn schließlich auszuschalten[5].

Zunächst muß also eine abstrakte Beschreibung der Benutzerschnittstelle entwickelt werden. Dabei ist allerdings zu beachten, daß die Benutzerschnittstelle auch als Demonstration der Mächtigkeit der entwickelten XML-Sprache dient. Das UI sollte also mindestens zwei Dialoge besitzen und Daten dynamisch aus der Benutzerschnittstelle auslesen können (z.B. ein Textfeld, in das der Benutzer seinen Namen eingeben kann). Diese Anforderungen führen zu folgendem Quellcode:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE aml SYSTEM "aml.dtd">

<aml version="0.1">
<userInterface name="javaUI">
    <dialog class="frame" name="overview">
        <component class="label" name="overviewLabel"
            value="MINDSTORM CONTROL"/>
        <component class="label" name="userNameLabel"
            value="What's your name?"/>
        <component class="textfield" name="userName"
            value=""/>
        <component class="button" name="robotButton"
            value="Control Robot"/>
        <component class="label" name="authorLabel"
            value="written by Juergen Baier"/>
    </dialog>
    <dialog class="frame" name="robotControl">
        <table rows="5" cols="3">
            <row>
                <col></col>
                <col><component class="label" name="robotLabel"
                    value="ROBOT"/>
                </col>
                <col></col>
            </row>
            <row>
                <col></col>
                <col><component class="button" name="forwardButton"
                    value="Forward"/></col>
```

[5] Leider geht das Wiedereinschalten nicht, da ein Mindstorms-Roboter keinen «Stand-by»-Modus kennt.

```
            <col></col>
        </row>
        <row>
            <col><component class="button" name="leftButton"
                value="Left"/></col>
            <col><component class="button" name="stopButton"
                value="Stop"/></col>
            <col><component class="button" name="rightButton"
                value="Right"/></col>
        </row>
        <row>
            <col></col>
            <col><component class="button" name="backButton"
                value="Back"/></col>
            <col></col>
        </row>
        <row>
            <col><component class="button" name="beepButton"
                value="Beep"/></col>
            <col><component class="button" name="backToMainButton"
                value="MAIN"/></col>
            <col><component class="button" name="turnOffButton"
                value="Off"/></col>
        </row>
    </table>
</dialog>
</userInterface>

<behavior>
    <rule>
        <condition>
            <event class="ActionPerformed" comp-name="robotButton"/>
        </condition>
        <action><go dialog="robotControl" class="frame"/></action>
    </rule>
    <rule>
        <condition>
            <event class="ActionPerformed" comp-name="backToMainButton"/>
        </condition>
        <action><go dialog="overview" class="frame"/></action>
    </rule>
    <rule>
        <condition>
            <event class="ActionPerformed" comp-name="forwardButton"/>
        </condition>
        <action>
            <call method="forward">
```

```
78              <param name=""><value-of name="userName"/></param>
            </call>
80          <change-property comp-name="statusLine"
                value="Trying to call method 'forward'"/>
82      </action>
    </rule>
84  [...]
</behavior>
86 </aml>
```

Die Regeln in der Verhaltensbeschreibung sind relativ einfach. Beim Betätigen von zwei Buttons wechselt die Dialogansicht (<go>), anderen Buttons sind bestimmte Methodenaufrufe zugeordnet. Als Parameter für die Methodenaufrufe wird stets der Benutzername übergeben[6]. Nun muß die abstrakte Beschreibung in eine sprach- und plattformspezifische Beschreibung transformiert werden. Dazu wird das in XSLT geschriebene Transformationstool verwendet. Die generierte Beschreibung läßt sich sofort fehlerfrei interpretieren; eventuell ist noch etwas Nacharbeit nötig, falls einige Komponenten nicht optimal plaziert werden oder die Ergebnisdatei zu groß für den Palm ist (was zu einem `OutOfMemoryError` führt). Folgender Quellcode zeigt die wichtigsten Ausschnitte aus der J2ME/Palm-spezifischen XML-Beschreibung[7].

```
<?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE jml SYSTEM "jml.dtd">

4 <jml version="0.1">
   <userInterface>
6     <container class="Frame" name="javaUI">
        <events>
8         <eventHandler event="windowClosing" handler="exitForm"/>
        </events>
10      <layout class="BorderLayout"/>
        <container class="Panel" name="cardPanel">
12        <constraints>
            <constraint layoutClass="BorderLayout" value="">
14            <borderConstraints direction="CENTER"/>
            </constraint>
16        </constraints>
          <layout class="CardLayout"/>
18        <container class="Panel" name="overview">
            <constraints>
20            <constraint layoutClass="CardLayout" value="">
```

[6] Das macht natürlich wenig Sinn und dient lediglich der Demonstration eines Features des Interpreters: des dynamischen Auslesens von Text- und anderen Feldern.

[7] Die Beschreibung wurde wegen des begrenzten Speichers auf dem Palm nachträglich bearbeitet. Dabei wurde jedoch nur redundanter Code entfernt. Bei Buttons ist es z.B. nicht nötig, eine Schriftart anzugeben, wenn die voreingestellte Schriftart verwendet werden soll. Solche redundanten <properties> wurden dann auch entfernt.

```
                  <cardConstraints cardName="overview"
                                   cardString="OVERVIEWPANEL"/>
               </constraint>
            </constraints>
            <layout class="FlowLayout"/>
            [...]
            <component class="TextField" name="userName">
               <properties>
               [...]
               </properties>
               <constraints>
                  <constraint layoutClass="FlowLayout" value=""/>
               </constraints>
            </component>
            <component class="Button" name="robotButton">
               <properties>
                  <property name="label" type="String"
                     value="Control Robot"/>
               </properties>
               <events>
                  <eventHandler event="actionPerformed"
                     handler="robotButtonActionPerformed"/>
               </events>
               <constraints>
                  <constraint layoutClass="FlowLayout" value=""/>
               </constraints>
            </component>
            [...]
         </container>
         <container class="Panel" name="robotControl">
            <properties>
            [...]
            </properties>
            <constraints>
               <constraint layoutClass="CardLayout" value="">
                  <cardConstraints cardName="robotControl"
                                   cardString="ROBOTCONTROLPANEL"/>
               </constraint>
            </constraints>
            <layout class="GridLayout">
               <properties>
                  <property name="rows" value="5"/>
                  <property name="columns" value="3"/>
               </properties>
            </layout>
            [...]
            <component class="Button" name="forwardButton">
```

```
68                <properties>
                     <property name="label" type="String" value="Forward"/>
70                </properties>
                  <events>
72                   <eventHandler event="actionPerformed"
                                handler="forwardButtonActionPerformed"/>
74                </events>
                  <constraints>
76                   <constraint layoutClass="GridLayout" value=""/>
                  </constraints>
78             </component>
               [...]
80          </container>
         </container>
82    </container>
   </userInterface>
84 <behavior>
      <rule>
86       <condition>
            <event class="ActionPerformed" comp-name="robotButton"/>
88       </condition>
         <action>
90          <go dialog="robotControl" class="frame"/>
         </action>
92    </rule>
      <rule>
94       <condition>
            <event class="ActionPerformed" comp-name="backToMainButton"/>
96       </condition>
         <action>
98          <go dialog="overview" class="frame"/>
         </action>
100   </rule>
      <rule>
102      <condition>
            <event class="ActionPerformed" comp-name="forwardButton"/>
104      </condition>
         <action>
106         <call method="forward">
               <param name="">
108               <value-of name="userName"/>
               </param>
110         </call>
         </action>
112   </rule>
      [...]
114 </behavior>
```

Abbildung 4.11: Screenshots von Palm- und Desktopclient

```
</jml>
```

Die XML-Beschreibung für Desktop-Rechner ist identisch; allerdings wird hier das unveränderte Ergebnis der XSLT-Transformation verwendet.

Abbildung 4.11 zeigt die Screenshots des ersten Dialogs auf einem Desktop-Rechner (Windows NT, Java 2 Standard Edition) und des zweiten Dialogs auf dem Palm-Emulator mit der KVM von Sun und der kAWT-Bibliothek.

Die Bedienung ist einfach: Mit den Buttons «Forward», «Left», «Stop», «Right» und «Back» läßt sich der Lego-Roboter in die gewünschte Richtung steuern. Mit «Beep» wird ein Signal ausgegeben und mit «Off» wird der Roboter ausgeschaltet. Der «Main»-Button schaltet wieder zum Eingangsdialog zurück.

4.5.4 Der Jini-Service zur Robotersteuerung

Allgemeines

Einen Lego Mindstorms Roboter in ein Jini-Netzwerk einzubinden ist nicht einfach, da sich leider kein Java darauf installieren läßt[8]. Dennoch ist es möglich, einen Lego-Roboter über ein Netzwerk anzusprechen. Dazu muß lediglich ein Jini-Dienst implementiert werden, der mit dem Roboter über die serielle Schnittstelle (COM bzw. /dev/tty) kommuniziert. Der Jini-Dienst hat also die Rolle eines Gerätetreibers für RCX-Mikrocontroller. Um einen Lego Mindstorms Roboter über die serielle Schnittstelle anzusteuern sind zwei Pakete notwendig:

- Die Java Communications API und
- die RCX Java API.

Die Java Communications API für Microsoft Windows und Sun Solaris ist unter http://java.sun.com/products/javacomm/index.html zu finden; auch eine Linux-Portierung ist erhältlich (siehe http://www.frii.com/-jarvi/rxtx/. Die RCX Java API von David Laverde (unter http://www.escape.com/-dario/java/rcx/ erleichtert den Zugriff auf Mindstorms-Roboter. Die Programmierschnittstelle ist einfach zu handhaben und läßt sich leicht in eigene Programme einbauen. Nützlich sind noch die «RCX Internals» von Kekoa Proudfoot (im Netz unter http://graphics.stanford.edu/-kekoa/rcx/). Diese Website enthält neben Einzelheiten zur RCX Hardware auch eine Liste der RCX-Befehlscodes.

Design des Dienstes

Das Klassendiagramm in Abbildung 4.12 zeigt den Aufbau des Jini-Dienstes. Die Klasse `rcxservice.RCXServer` ist der «Entry Point» der Applikation. Hier werden nach dem Start zuerst einige Initialisierungen vorgenommen; außerdem registriert sich der Dienst an dieser Stelle beim Lookup-Service. Ist die Registrierung abgeschlossen, wird ein Objekt der Klasse `rcxservice.RCXPortImpl` instanziert. Diese Klasse ist für die Kommunikation von Clients mit dem Mindstorms-Roboter zuständig. Dazu wird ein Server-Socket kreiert, der Client-Anfragen und -Kommandos entgegennimmt und ausführt (pro Client wird ein neuer Thread gestartet).

Die Klasse `rcxservice.JMLRCXFrameFactory` ist für die Instanzierung der Benutzeroberfläche auf dem Client verantwortlich. Da dafür eine XML-Beschreibung interpretiert werden muß, wurde diese Klasse von `interpreter.Interpreter` abgeleitet.

[8] Was auch nicht weiter verwundern sollte, da der RCX-Mikroprozessor der Roboter ein 16-Bit-System mit 32k Speicher ist.

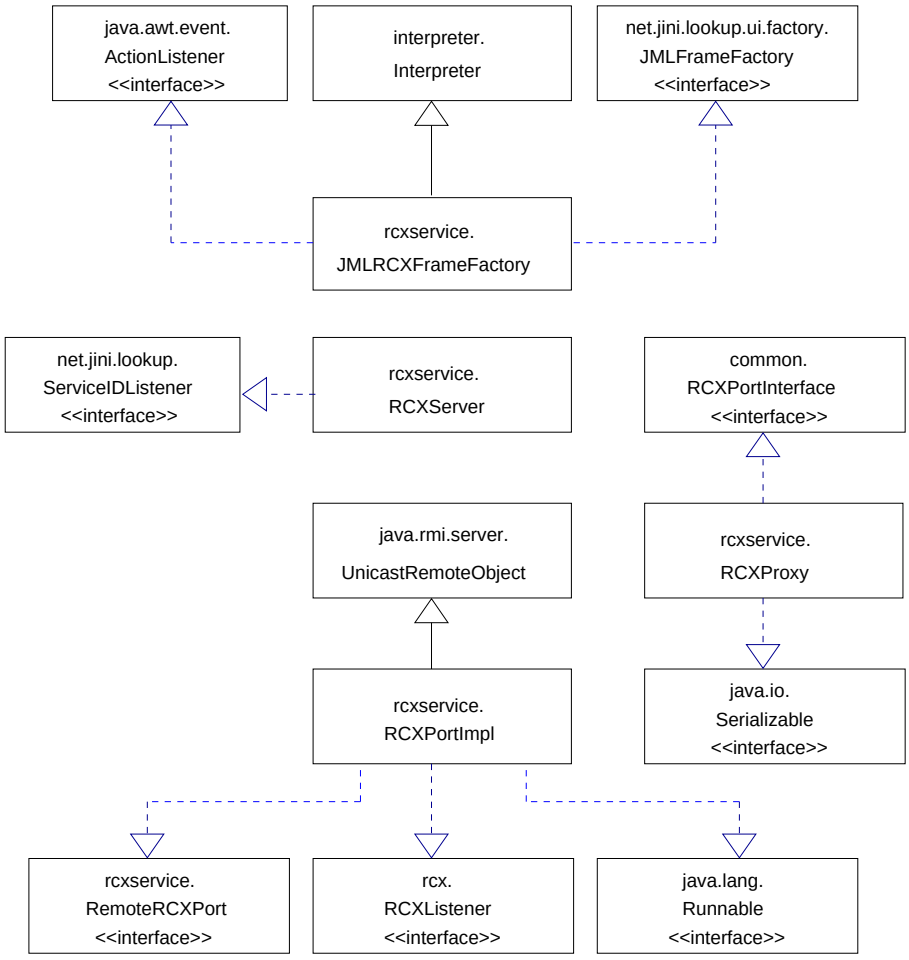


Abbildung 4.12: Klassendiagramm des Jini-Dienstes

Die Schnittstelle `JMLFrameFactory` wird implementiert, um den Typ von `JMLRCXFrameFactory` festzulegen. In diesem Fall erzeugt die Fabrik AWT-Benutzeroberflächen. Sollen Swing-Oberflächen erzeugt werden muß z.B. `JMLJFrameFactory` implementiert werden. Das `ActionListener`-Interface wird benötigt, um bestimmte Events (z.B. einen Buttonklick) zu verarbeiten.

API-Erweiterung der Service-UI-Spezifikation

Die Klasse `rcxservice.JMLRCXFrameFactory` wird nur für Jini-fähige Clients benötigt. Sie dient dazu, die Benutzerschnittstelle des Dienstes clientseitig zu instanzieren. Diese Fabrik-Klasse folgt den Regeln der Service-UI-Spezifikation, d.h. sie enthält eine Methode `getFrame()`, die vom Client aufgerufen wird, um die Benutzerschnittstelle anzuzeigen. Allerdings gibt es einen wesentlichen Unterschied zu typischen Service-UIImplementierungen. Üblicherweise instanziert die Fabrik eine in Binärform vorliegende Klasse, die ein hartcodiertes UI enthält. Da der Prototyp dieser Master Thesis aber vor allem beweisen soll, daß die XML-Beschreibung von Benutzerschnittstellen sinnvoll ist, geht die `JMLRCXFrameFactory` einen anderen Weg. Sie enthält nämlich einen Interpreter (genauer gesagt, sie ist von `interpreter.Interpreter` abgeleitet) und ist somit in der Lage, XML-UI-Beschreibungen zu interpretieren. Um diesen Ansatz konsistent zu der Service-UI-Spezifikation zu halten, wurde das API dieser Spezifikation um einige Schnittstellen erweitert (siehe Abbildung 4.13).

Die oberste Klasse in dieser Schnittstellenhierarchie (abgesehen von `Serializable`) ist `JMLFactory`. Diese Schnittstelle soll lediglich aussagen, daß Implementierungen in der Lage sind, die im Rahmen dieser Thesis entwickelten Schnittstellensprache (JML oder JavaML) zu interpretieren. Die nächste Spezialisierung (`JMLAWTFrameFactory` legt das verwendete Toolkit (AWT) fest. Eine Erweiterung der Hierarchie, um andere Toolkits (z.B. Swing) zu unterstützen ist sicher sinnvoll, wurde aber hier nicht implementiert. Von `JMLAWTFrameFactory` sind zwei Schnittstellen abgeleitet: `JMLFrameFactory` und `JMLSpeechFrameFactory`. Erstere Schnittstelle wird von `rcxservice.JMLRCXFrameFactory` implementiert und bietet eine Interpretation von XML-UI-Beschreibungen für das AWT-Toolkit mit einem Frame als Wurzelelement im GUI-Objektgraphen. Die zweite Schnittstelle, `JMLSpeechFrameFactory` wird im nächsten Abschnitt behandelt.

Robotersteuerung mit Sprache

Die Java Markup Language wurde primär im Hinblick auf grafische Benutzeroberflächen entwickelt. Eine Spracheingabe war ursprünglich nicht vorgesehen und ist auch nicht in der DTD enthalten. Für das Morpha-Projekt wurde in den Jini-Client mit der Java Speech API eine Spracheingabe integriert. Die Spracherkennung erfolgt über ViaVoice von IBM. Clientseitig müssen sowohl Java Speech als auch ViaVoice installiert sein, damit die Spracheingabe funktioniert. Abgesehen von dieser

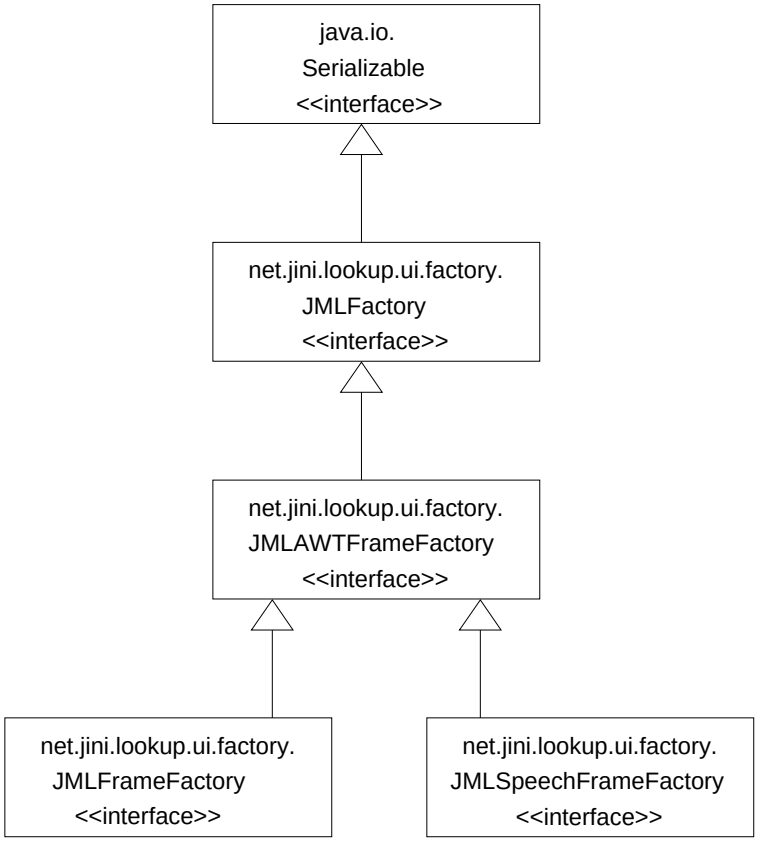


Abbildung 4.13: Klassendiagramm der Erweiterungen zur Service-UI-Spezifikation

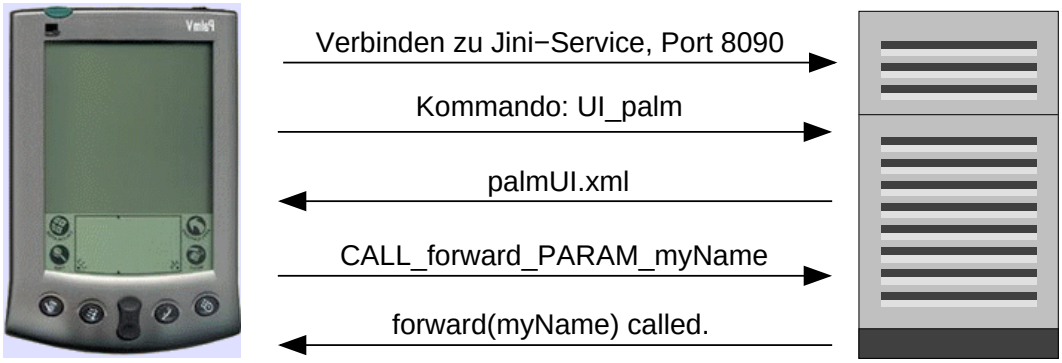


Abbildung 4.14: Kommunikation zwischen dem Jini-Service und dem PDA-Client

Installation sind clientseitig keinerlei Modifikationen notwendig. Alle notwendigen Erweiterungen wurden zunächst in die Klasse `JMLRCXFrameFactory` eingebaut und im Webserver aktualisiert. So ist die Robotersteuerung sowohl über Sprache als auch über das GUI möglich.

Aus Designgründen wurde allerdings das Service-UI-API um die Klasse `JMLSpeechRCXFrameFactory` erweitert. Damit ist `JMLRCXFrameFactory` ausschließlich für grafische Benutzeroberflächen zuständig und `JMLSpeechRCXFrameFactory` sowohl für das GUI als auch für das sprachbasierte UI.

Protokoll zur Kommunikation mit PDA-Clients

Da der Jini-Service nicht nur von Jini-Clients sondern auch von PDA-Clients genutzt werden soll, muß die Möglichkeit bestehen, den Dienst über eine einfache Socketverbindung in Anspruch zu nehmen. Das heißt für den Jini-Service, daß beim Start des Dienstes ein Server-Socket instanziert werden muß, der eingehende Kommandos annimmt. Da über die Socket-Schnittstelle UI-Beschreibungen angefordert werden und Methoden aufgerufen werden sollen, ist es nötig, ein eigenes Protokoll festzulegen. Da dieses Protokoll zunächst nur für den Prototyp dieser Master Thesis entwickelt wurde, werden nur wenige, eingeschränkte Funktionen zur Verfügung gestellt. Eine dieser Funktionen ist die Anforderung einer bestimmten Benutzerschnittstelle. Die Syntax zur Anforderung einer Benutzerschnittstelle ist `UI_<type>`.

Das Kommando `UI_` bewirkt, daß der Jini-Service eine bestimmte Benutzerschnittstelle lädt. Welche das ist, wird als Parameter `<type>` übergeben. Zur Zeit sind zwei verschiedene XML-Beschreibungen von Benutzerschnittstellen implementiert: für Desktop-Rechner und für Palm-PDAs. Will ein Client ein PDA-UI, fordert er `UI_palm` an; will er ein Desktop-UI, fordert er `UI_desktop` an. Der Jini-Service lädt dann die entsprechende XML-Datei und schickt sie über die Socket-Verbindung zum Client. Abbildung 4.14 zeigt den Ablauf der Kommunikation zwischen Jini-Service und Palm-Client.

Wenn der Client nicht über RMI verfügt, muß er Methodenaufrufe über ein proprietäres Protokoll über eine Socket-Verbindung zum Jini-Service schicken. Ein Protokoll für entfernte Methodenaufrufe zu entwickeln ist sehr aufwendig; vor allem wenn nicht nur elementare Datentypen als Parameter übergeben werden sollen. Deshalb wurde für den Prototyp der Master Thesis nur ein sehr eingeschränktes Protokoll entwickelt und implementiert. Die Syntax für entfernte Methodenaufrufe ist `CALL_<methodName><parameterList>`. Dabei ist `<methodName>` der Name der Methode, die im Kontext des Jini-Services aufgerufen werden soll und `<parameterList>` (wie der Name schon sagt) eine Parameterliste mit der Syntax `_PARAM_<value>`. Als `<value>` dürfen nur `String`-Werte verwendet werden[9]. Ein Aufruf der Methode `forward(String)` wird also mit `CALL_forward_PARAM_myString` vorgenommen und für den Aufruf von `forward(String, String)` ist das Kommando `CALL_forward_PARAM_myFirstString_PARAM_mySecondString` nötig. Der Rückgabewert (der über den Socket zum Client übertragen wird) ist stets in einem `StringBuffer` gespeichert. Wenn der Jini-Service ein bestimmtes `CALL`-Kommando erhalten hat, wird das Kommando geparst und versucht, über das Reflection-API die entsprechende Methode mit der gewünschten Anzahl von `String`-Parametern aufzurufen. Die Methoden, die so aufgerufen werden sollen, müssen aus dem Kontext des Jini-Dienstes aufrufbar sein. Wird keine Methode mit der gewünschten Anzahl von Parametern gefunden, wird eine Fehlermeldung zum Client zurückgegeben.

4.5.5 Die Client-Applikationen

Die beiden Client-Applikationen (der PDA-Client und der Desktop-Client) sind relativ einfach aufgebaut. Jeder der Clients besteht aus einer einzigen Klasse (`TestRCX` und `TestRCXPalm`). Der Desktop-Client verfügt über keinen eigenen JML-Interpreter, weil er die entsprechenden Klassen über das Netz nachladen kann. Dagegen muß der PDA-Client auf einen Interpreter zugreifen können, da hier kein Nachladen von Klassen möglich ist. Beide Clients wurden sehr generisch implementiert; sie enthalten kaum Applikationslogik. Während beim Desktop-Client die nachzuladenden Klassen die Kommunikation mit dem Server steuern erhält der PDA-Client alle notwendigen Informationen über die XML-Beschreibung.

[9] Ich habe mich dagegen entschieden, für den Prototypen `_PARAM_<type><value>` zu verwenden, also den Datentyp festzulegen, weil im Rahmen dieses Prototypen eigentlich überhaupt keine Parameterübergabe notwendig ist. Falls auch andere (einfache) Datentypen übergeben werden sollen, läßt sich das Protokoll jederzeit erweitern. Für ein «echtes» RPC-Protokoll bietet sich auf Palm-PDAs *RMILite* an, das als Teil des Berkeley-Ninja-Projektes entwickelt wird (siehe http://postpc.cs.berkeley.edu/rmilite/)

Fazit

In dieser Master Thesis wurden zwei XML-Sprachen zur Beschreibung von Benutzerschnittstellen entwickelt. Dabei wurde Wert auf die Praxistauglichkeit der Sprachen gelegt und nicht auf die Lückenlosigkeit der Dokumenttyp-Definitionen. Um die Brauchbarkeit des Konzeptes nachzuweisen, wurde ein Jini-Dienst zur Steuerung eines Lego Mindstorms Roboters entwickelt. Die Clients dieses Dienstes laden die zugehörige Benutzerschnittstelle dynamisch über das Netz. Die Zuordnung eines UI zu einem Jini-Dienst wird in der semi-offiziellen Service-UI-Spezifikation näher beschrieben. Diese Spezifikation wurde im Prototypen dieser Master Thesis ausgiebig verwendet und hat sich bewährt. Da PDA-Clients keine Java-Klassen dynamisch nachladen können, wurde ein eigenes Protokoll spezifiziert, welches das Laden eines JML-UI über eine Socketverbindung erlaubt. Damit wurde gezeigt, daß sich auch Clients, die nicht über für Jini notwendige Bibliotheken verfügen, in ein Jini-Netzwerk einbinden lassen. Leider hat sich gezeigt, daß sich wegen des geringen Speichers und der schwachen Prozessorleistung nur einfache Benutzerschnittstellen auf einem PDA-Client ausführen lassen. Für die Interpretation und Darstellung der Benutzerschnittstelle des Prototypen werden mehrere Minuten benötigt. Dagegen wird die gleiche Benutzerschnittstelle auf einem Desktop-Client (Pentium II, 333MHz, 256MB RAM) ohne Zeitverzögerung dargestellt.

Da während der Entwicklung des Prototypen das UI mehrfach neu geschrieben bzw. geändert wurde, läßt sich eine Aussage über den Nutzen XML-basierter Benutzerschnittstellen machen. Es hat sich gezeigt, daß auch bei einem kompletten Re-Design des UI nur ein sehr geringer Zeitaufwand notwendig ist, da entsprechende Änderungen in der AML-Beschreibung vorgenommen wurden und anschließend in JML transformiert wurden. In nahezu allen Fällen konnte der Interpreter die JML-Beschreibung richtig darstellen. Das bedeutet, daß sich mit dem in der Thesis formulierten Konzept viel Entwicklungszeit einsparen läßt. Zudem ist die XML-Beschreibung des UI zu Dokumentationszwecken sehr viel besser geeignet als Java-Quellcode bzw. der mit `javadoc` generierten HTML-Dokumentation. Interessant ist auch, daß mit diesem Konzept das Problem des dynamischen Nachladens von Klassen bei J2ME-Applikationen zumindest zum Teil gelöst wurde.

Eine sinnvolle Erweiterung der entwickelten XSLT-Module ist noch, statt AWT das Swing-Toolkit und die Waba-Programmiersprache zu unterstützen. Hilfreich ist auch ein Tool, das die XML-Beschreibungen der Forté-IDE nach JML transformiert. Da zu

erwarten ist, daß auch andere IDE-Hersteller in Zukunft UI-Beschreibungen in XML abspeichern, läßt sich JML als herstellerunabhängiges Zwischenformat einsetzen.

Literaturverzeichnis

[Cover2000] *The XML Cover Pages - WAP Wireless Markup Language Specification (WML)*
Robin Cover 2000
http://oasis.oasis-open.org/cover/wap-wml.html

[CSWL1999] *UPnP, Jini and Salutation - A look at some popular coordination frameworks for future networked devices*
California Software Laboratories (CSWL) 1999
http://www.cswl.com/whiteppr/tech/upnp.html

[Deleo2000] *A Demonstration of Jini Technology and the K Virtual Machine*
Michael Deleo (Sun Microsystems) 1999
Artikel über eine Demonstration auf der JavaOne 1999
http://developer.java.sun.com/developer/technicalArticles/jini/JavaTanks/Javatanks.html

[Day2000] *Java 2 Platform, Micro Edition for Mobile Devices*
Bill Day (Sun Microsystems) 2000
Folien zum Vortrag auf der JavaOne 2000
http://jsp.java.sun.com/javaone/javaone2000/pdfs/TS-575.pdf

[Gellersen2000] *Vorlesungsskriptum «Ubiquitous Computing», 2000*
Hans W. Gellersen 2000
Vorlesung im Wintersemester 2000/2001 an der Universität Karlsruhe (TH)
http://www.teco.edu/lehre/ubiq/

[Glade2000] *Glade Website*
Glade 2000
http://glade.pn.org/index.html

[HAVi2000] *Website des HAVi Konsortiums*
HAVi Organization 2000
http://www.havi.org

[IBM1999a] *IBM announces availability of NuOffice office system software for Lotus Notes Domino based on Salutation*
IBM 1999

Presseveröffentlichung
http://www-3.ibm.com/pvc/news/press_releases/nuoffice_0499.shtml

[IBM1999b] *IBM NuOffice*
IBM 1999
Werbebroschüre
http://www-3.ibm.com/pvc/tech/pdf/nuoffice.pdf

[Kay2000] *XSLT Programmer's Reference*
Michael Kay
Wrox Press Limited 2000
http://www.wrox.com

[Fischer2000] *Schnell und oberflächlich*
Thorsten Fischer
Artikel im Linux Magazin 09/2000
http://www.linux-magazin.de

[Morpha2000] *Morpha Flyer*
Projekt «Morpha» 2000
http://www.morpha.de

[Mozilla2000] *XPToolkit Project*
Mozilla.org 2000
http://www.mozilla.org/xpfe

[Newmarch2000] *Jan Newmarch's Guide to JINI Technologies Version 2.06*
Jini-Online-Tutorial
Jan Newmarch 2000
http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xm

[OaksWong2000] *Jini in a Nutshell*
Scott Oaks, Henry Wong
O'Reilly 2000
http://www.oreilly.com/catalog/jininut

[Ortner2000] *SelfWML*
WML-Einführung
Roland Ortner 2000
http://www.selfwml.f2s.com/intro.html

[Schulte96] *Entwurf und Implementation einer multimodalen Benutzerschnittstelle für eine CAD-Umgebung*
Diplomarbeit von Matthias Schulte
Universität Hamburg, Fachbereich Informatik 1996
http://nats-www.informatik.uni-hamburg.de/~jum/students/schulte_matthias/schulte.html

[Sing2000] *Professional Jini*
Sing Li
Wrox Press Limited 2000
http://www.wrox.com

[Sun1999a] *Jini Technology Architectural Overview*
Sun Microsystems 1999
http://www.sun.com/jini/whitepapers/architecture.html

[Sun1999b] *Philips, Sony, Sun Collaborate to Bridge HAVi and Jini Network Architectures*
Pressemitteilung Sun Microsystems 1999
http://www.sun.com/smi/Press/sunflash/9901/sunflash.990119.1.html

[Sun1999c] *Sony and Sun Join Forces To Link Digital Consumer Electronic Appliances Directly To Internet Content and Services*
Pressemitteilung Sun Microsystems 1999
http://www.sun.com/microelectronics/newsreleases/sunflash.991110.1.html

[Sun2000a] Jini Network Technology FAQs
Sun Microsystems 2000
http://www.sun.com/jini/faqs

[Sun2000b] *JAVA TECHNOLOGY ADOPTED AS THE STANDARD FOR A WIDE RANGE OF CONSUMER DEVICES ACROSS MULTIPLE MARKETS*
Pressemitteilung Sun Microsystems 2000
http://www.sun.com/smi/Press/sunflash/2000-01/sunflash.20000106.6.html

[Sun2000c] *Jini Technology Surrogate Architecture Specification*
Sun Microsystems 2000
http://developer.jini.org:80/exchange/projects/surrogate

[Sun2000d] *Connected, Limited Device Configuration*
Sun Microsystems 2000
http://java.sun.com/products/cldc/

[Tanenbaum1996] *Computer Networks, Third Edition*
Andrew S. Tanenbaum
Prentice-Hall International, Inc. 1996
http://www.cs.vu.nl/ãst

[UIML2000a] *UIML: An XML Language for Building Device-Independent User Interfaces*
Marc Abrams, Constantinos Phanouriou
Universal Interface Technologies, Inc. 2000
http://www.uiml.org

[UIML2000b] *User Interface Markup Language (UIML) Draft Specification Version 2.0a*
Constantinos Phanouriou
Universal Interface Technologies, Inc. 2000
http://www.uiml.org

[Venners2000] *Service UI Draft Specification Version 1.0*
Bill Venners
Artima Software 2000
http://www.artima.com/jini/serviceui/DraftSpec.html

[Vo98] *A Framework and Toolkit for the Construction of Multimodal Learning Interfaces*
Dissertation von Minh Tue Vo
Carnegie Mellon University, Computer Science Department 1998

[VoiceXML2k] *VoiceXML Specification Version 1.00*
VoiceXML Forum 2000
http://www.voicexml.org/specs_1.html

[VSYSa] *Hydepark - Hyper Distributed Environment for Personal Appliances*
Universität Hamburg, Computer Science Department, Distributed Systems Group 2000
http://vsys-www.informatik.uni-hamburg.de/ smueller/Hydepark/en/

[VSYSb] *XML and Jini - On Using XML and the «JAVA Border Service Architecture» to integrate mobile devices into the JAVA Intelligent Network Infrastructure*
Stefan Müller-Wilken, Daniel Hinz und Winfried Lamersdorf
Universität Hamburg, Computer Science Department, Distributed Systems Group 2000
http://vsys-www.informatik.uni-hamburg.de/publications/

[VSYSc] *Enhancing JINI to support non-JAVA appliances*
Stefan Müller-Wilken und Winfried Lamersdorf
Universität Hamburg, Computer Science Department, Distributed Systems Group 2000
http://vsys-www.informatik.uni-hamburg.de/publications/

[VSYSd] *On Integrating Mobile Devices into a Workflow Management Scenario*
Stefan Müller-Wilken, Daniel Hinz und Winfried Lamersdorf
Universität Hamburg, Computer Science Department, Distributed Systems Group 2000
http://vsys-www.informatik.uni-hamburg.de/publications/

[W3C1999a] *XSL Transformations (XSLT) Version 1.0*
W3C Recommendation vom 16. November 1999,

World Wide Web Consortium (W3C) 1999
http://www.w3.org/TR/xslt

[W3C1999b] *XML Path Language Specification (XPath) Version 1.0*
W3C Recommendation vom 16. November 1999,
World Wide Web Consortium (W3C) 1999
http://www.w3.org/TR/xpath

[WAPForum2000a] *Wireless Markup Language Specification Version 1.3*
WAP-Forum 2000
http://www1.wapforum.org/tech/documents/WAP-191-WML-20000219-a.pdf

[WAPForum2000b] *WMLScript Language Specification Version 1.2*
WAP-Forum 2000
http://www1.wapforum.org/tech/documents/WAP-193-WMLScript-20000324-a.pdf

[WAPForum2000c] *WMLScript Standard Libraries Specification Version 1.3*
WAP-Forum 2000
http://www1.wapforum.org/tech/documents/WAP-194-WMLScriptLibs-20000324-a.pdf

[WAPForum2000d] *Binary XML Content Format Specification*
WAP-Forum 2000
http://www1.wapforum.org/tech/documents/WAP-192-WBXML-20000306-a.pdf

[Wireless2000a] *An Introduction To VoiceXML*,
Wireless Developer Networks 2000
http://www.wirelessdevnet.com/training/voicexml/voicexmloverview.html

[Wireless2000b] *WML Tutorial*,
Wireless Developer Networks 2000
http://www.wirelessdevnet.com/training/WAP/WML5.html

[WirelessDevice2000] *Jini Wireless Device Project*
Jini Community 2000
http://developer.jini.org:80/exchange/projects/wirelessdevice/

[Wong2000] *Developing Embedded Applications Using Jini Connection Technology and Java 2 Platform, Micro Edition*
Hickmond Wong (Sun Microsystems) 2000
Folien zum Vortrag auf der JavaOne 2000
http://java.sun.com/people/hinkmond/conferences/javaone/

[Zvon2k] *XUL Reference*
Zvon.org 2000
http://zvon.org/mozilla/XUL-reference.html

Dokumenttyp-Definitionen

Application Markup Language - AML

Die «Application Markup Language» ist eine XML-basierte Sprache zur Beschreibung von Benutzerschnittstellen. Das folgende Listing zeigt die vollständige Dokumenttyp-Definition (DTD).

```
<!--
    Abstract Markup Language
    Version 0.1
    Author: Juergen Baier
-->

<!--
    Example for an AML document

<userInterface>
    <dialog>
        <component/>
        <component/>
    </dialog>
    <dialog>
        <table>
            <row>
                <col>
            </row>
        </table>
    </dialog>
</userInterface>

<behavior>
    <rule>
        <condition></condition>
        <action></action>
    </rule>
</behavior>

-->
```

```
32
<!--
34    Root element
-->
36 <!ELEMENT aml (userInterface,behavior)>
<!ATTLIST aml
38    version NMTOKEN #REQUIRED
>
40
<!--
42    User Interface.
-->
44 <!ELEMENT userInterface (dialog+)>
<!ATTLIST userInterface
46    name NMTOKEN #REQUIRED
>
48
<!--
50    <dialog> may have <component> or <table> subelements.
-->
52 <!ELEMENT dialog (component+|table)>
<!ATTLIST dialog
54    class (frame|popup) "frame"
      name NMTOKEN #REQUIRED
56 >

58 <!--
      <table> could be generalized to <layout class="table">
60 -->
<!ELEMENT table (row+)>
62 <!ATTLIST table
      rows   NMTOKEN #IMPLIED
64    cols   NMTOKEN #IMPLIED
>
66
<!ELEMENT row (col+)>
68
<!--
70    A <col> may have zero or more <component>s
-->
72 <!ELEMENT col (component*)>

74 <!ELEMENT component EMPTY>
<!ATTLIST component
76    class  NMTOKEN #REQUIRED
      name   NMTOKEN #REQUIRED
78    value  CDATA #IMPLIED
```

```
>

<!--
    Behavior.
-->
<!ELEMENT behavior (rule*)>

<!ELEMENT rule (condition, action+)>

<!ELEMENT condition (event|evaluate)>

<!ELEMENT action (change-property|call|go)+>

<!ELEMENT event EMPTY>
<!ATTLIST event
    class      NMTOKEN #IMPLIED
    comp-name  NMTOKEN #IMPLIED
>

<!ELEMENT go EMPTY>
<!ATTLIST go
    dialog NMTOKEN #REQUIRED
    class NMTOKEN #REQUIRED
>

<!ELEMENT evaluate (event,(if|if-not)+)>

<!ELEMENT call (param*)>
<!ATTLIST call
    method NMTOKEN #IMPLIED
>

<!ELEMENT param (value-of,value)>
<!ATTLIST param
    name CDATA #IMPLIED
>

<!ELEMENT value-of EMPTY>
<!ATTLIST value-of
    name NMTOKEN #IMPLIED
>

<!ELEMENT value (#CDATA)>

<!ELEMENT if EMPTY>
<!ATTLIST if
    comp-name  NMTOKEN #REQUIRED
```

```
    value      NMTOKEN #REQUIRED
>

<!ELEMENT if-not EMPTY>
<!ATTLIST if-not
    comp-name  NMTOKEN #REQUIRED
    value      NMTOKEN #REQUIRED
>

<!ELEMENT change-property EMPTY>
<!ATTLIST change-property
    comp-name  NMTOKEN #REQUIRED
    value      CDATA #REQUIRED
>
```

Java Markup Language - JML

Die «Java Markup Language» ermöglicht die Beschreibung von Java-Benutzeroberflächen in einem XML-Dokument. Folgendes Listing zeigt die zugehörige DTD.

```
<!--
    Java Markup Language
    Version 0.1
    Author: Juergen Baier
-->

<!ELEMENT jml (userInterface,behavior)>
<!ATTLIST jml version NMTOKEN #REQUIRED>

<!ELEMENT userInterface ( container ) >

<!ELEMENT container ( component | constraints | container | events
    | layout | properties )* >
<!ATTLIST container
    class NMTOKEN #REQUIRED
    name NMTOKEN #REQUIRED
>

<!ELEMENT component ( constraints | events | properties )* >
<!ATTLIST component
    class NMTOKEN #REQUIRED
    name NMTOKEN #REQUIRED
>

<!ELEMENT constraints ( constraint ) >
```

```
<!ELEMENT constraint ( borderConstraints | cardConstraints )* >
<!ATTLIST constraint
    layoutClass NMTOKEN #REQUIRED
    value CDATA #IMPLIED
>

<!ELEMENT events ( eventHandler ) >
<!ELEMENT eventHandler EMPTY >
<!ATTLIST eventHandler
    event NMTOKEN #REQUIRED
    handler NMTOKEN #REQUIRED
>

<!ELEMENT properties ( property )* >

<!ELEMENT property ( color )? >
<!ATTLIST property
    name NMTOKEN #REQUIRED
    type NMTOKEN #IMPLIED
    value CDATA #REQUIRED
>

<!ELEMENT color EMPTY >
<!--
    A color is identified by an ID or by the
    RGB values. ID should be a string ("white"),
    RGB values are numbers.
-->
<!ATTLIST color
    id CDATA #REQUIRED
    red CDATA #REQUIRED
    blue CDATA #REQUIRED
    green CDATA #REQUIRED
>

<!ELEMENT layout ( properties? ) >
<!ATTLIST layout class NMTOKEN #REQUIRED >

<!ELEMENT borderConstraints EMPTY >
<!ATTLIST borderConstraints direction NMTOKEN #REQUIRED >

<!ELEMENT cardConstraints EMPTY >
<!ATTLIST cardConstraints
    cardName NMTOKEN #REQUIRED
    cardString NMTOKEN #REQUIRED
>

```

```
<!--
    Behavior.
-->
<!ELEMENT behavior (rule*)>

<!ELEMENT rule (condition, action+)>

<!ELEMENT condition (event)>
<!ELEMENT action ((property-change|call|go)*, event?)>

<!ELEMENT event EMPTY>
<!ATTLIST event
    class      NMTOKEN #IMPLIED
    comp-name  NMTOKEN #IMPLIED
>

<!ELEMENT go EMPTY>
<!ATTLIST go
    dialog NMTOKEN #REQUIRED
>

<!ELEMENT evaluate (event,(if|if-not)+)>

<!ELEMENT call (param*)>
<!ATTLIST call
    method NMTOKEN #IMPLIED
>

<!ELEMENT param (value-of)>
<!ATTLIST param
    name CDATA #REQUIRED
>

<!ELEMENT value-of EMPTY>
<!ATTLIST value-of
    name NMTOKEN #IMPLIED
>

<!ELEMENT if EMPTY>
<!ATTLIST if
    comp-name  NMTOKEN #REQUIRED
    value      NMTOKEN #REQUIRED
>

<!ELEMENT if-not EMPTY>
<!ATTLIST if-not
    comp-name  NMTOKEN #REQUIRED
```

```
120     value      NMTOKEN #REQUIRED
>

122
<!ELEMENT change-property EMPTY>
124 <!ATTLIST change-property
    comp-name  NMTOKEN #REQUIRED
126     value      NMTOKEN #REQUIRED
>
```

Index